Haskell Lecture Notes

Zachary May

July 16, 2014

1 Introducing Haskell

- Haskell is a **statically-typed**, **non-strict**, **pure functional** programming language.

    - **Functional:** Conceptually, computation proceeds via the application of functions to parameters rather than by 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. These features are valuable when 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 only evaluates such an expression as needed.
    - **Statically-typed:** The type of every expression is known at compile time, preventing run-time errors caused by type incompatibilities. This feature prevents things like Java's `NullPointerException`, because a function that claims it returns a value of a specific type must live up to that promise. 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 :: Int -> [Int] -> Int
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 parameters: an `Int` and a list of `Int`s. It returns an `Int`. In general, the type after the last `->` is the return value, and the others are the parameters.

- Haskell would actually infer a more general type than what we see 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 parameters `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 notation for function application is lightweight: simple juxtaposition of terms. The parentheses here are for grouping only.

- 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 parameters 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 parameters, and its return type, `IO ()`, is quite interesting.

- Look back at the type `[Int]`. The list type `[]` is a **parameterized** type. That is, list itself is not a concrete type, but a **type constructor**. We need another type, the parameter, to create a concrete type. Similarly, `IO` is a type constructor.

- Here we instantiate `IO` with the type `()`, the **unit type** with only a single value, also written as `()`. The type `IO ()` represents an **I/O action** with no interesting 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. However, because Haskell evaluates expressions lazily, merely describing the value is not enough to force the run-time to 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`.

3 Basic Haskell Types

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

- Haskell offers basic primitive data types: [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, implemented in GHC as 31-bit ISO 10646 characters.
  - `Bool`: Boolean values (but see below; `Bool` is actually a composite type).

- Additionally, we have several composite data types:

  - 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.

- Programmer-defined 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 want a type to describe e-mail address values but do not want it to be interchangeable with **String**s in general, we can define a new type that simply tags a **String** value: **data EmailAddress = EmailAddress String**.
  - Wrapping the **String** in the **EmailAddress** wrapper comes with some amount of syntactic and run-time overhead each time we want to access the underlying **String**.
  - Instead we can use **newtype EmailAddress = EmailAddress String**. Haskell's type checker treats this type exactly like a type introduced with the **data** keyword, but the compiler generates code that disregards the wrapper, as though we had just used a type synonym.

- 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 infers the most general type of **moreNothing**, **[a]**. (**[a]** 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.
        - Also referred to as a Cartesian product.
        - Tuples are the archetypal product type.
        - The cardinality 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 $2 \times 2 = 4$ possible values.
    - An example:

```
1       data DimensionalValue =
2           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.
    - Following the `=` are the type's constructor definitions. The first identifier is the **data constructor**, followed by arbitrarily many field declarations.
    - By convention, if there is only one data constructor, it has the same name as the type itself.
    - 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.
        - Also referred to as a disjoint union.
        - Enumerations are the archetypal sum types
        - An example:

```
1       data Dimension = Seconds
2                      | Meters
3                      | Grams
```

- As before, `data` introduces a new type, here named `Dimension`.
- The vertical pipe | separates the various alternatives.
- Each alternative is given as a data constructor definition as described above. Here we define three simple data constructors, `Seconds`, `Meters`, and `Grams`.
- These identifiers can be used as literal values of the type `Dimension`.

- The cardinality 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:

```
1    data PlaneTicket
2        = PlaneTicket Section MealOption
3
4    data Section | Coach
5                 | Business
6                 | FirstClass
7
8    data MealOption = Regular
9                    | Vegetarian
10
11   data TravelDetails = Train
12                      | Automobile
13                      | 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`, `Business`, 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? $1 + 1 + 1 \times (3 \times 2) = 8$.

4 Polymorphic types

- Earlier, we described the list and tuple types in terms of other, unspec-ified 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 and 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 polymor-phism** seen in object-oriented programming.

- This example shows how we might implement our own linked-list and 2-tuple types:

```
1    data List a = Nil
2                | Cons a (List a)
3
4    data Pair a b = Pair a b
5
6    data UniformPair a = UniformPair 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. That is, the scope of a type variable is a single data definition.
- What is the difference between our definition of Pair and UniformPair? UniformPair is parametric in only one type variable so both of its elements must be of the same type.

- We see that `List` is a sum of products: A `List` containing elements of type `a` is either the empty list `Nil` or it is a value of type `a` followed by another `List` containing elements of type `a`. 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 a concrete type, and if applied to a type variable yields a polymorphic type.
    - Data constructors are applied to values to produce new values. For example, `Cons` must be applied to a value of type `a` and a value of type `List a` to produce a value of type `List a`. `Nil` is a **nullary data constructor** is already a value of type `List a` any more value
    - Analogously, type constructors are applied to types to produce new types. For example, `List` applied to the type `Integer` produces the type `List Integer`.

5   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.

```
1      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). Its 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:

```
1      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 parameter 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.

6 Ad-Hoc Polymorphism with Type Classes

- At the machine level, adding two integers is quite a different operation from adding two floating-point numbers. High-level languages, in an effort to hide this detail, **overload** the semantics of the addition operator to support these distinct operations using the same operator.

- In a strongly-typed language like Haskell, what might the type of `(+)` be?

  - In Haskell, infix binary operators are just syntactic sugar for functions of two parameters. When referring to binary operators outside of their normal infix notation, Haskell requires them to be surrounded by parentheses.

- `Int -> Int -> Int` or similar is insufficient, since the type of `(+)` needs to be general enough to describe adding together two operands of many different numeric types.

- `a -> a -> a` seems promising: two operands and a result, all of the same type. However, this type signature **unifies** with `TravelDetails -> TravelDetails -> TravelDetails` and addition of that type does not make sense.

- What happens if we ask the Haskell compiler to infer the type of `(+)`?

  - The most popular Haskell compiler, GHC, has a REPL interface called GHCi.
  - The command `:t` *`expression`* asks GHCi to infer the type of *`expression`*.
  - `:t (+)` yields the inferred type: `(Num a) => a -> a -> a`.

- `(Num a) => ...` is a **class constraint** on the type signature that follows the `=>`.

- Normally, a free type variable in a type signature can be unified with any type at all.

- A class constraint on a type variable restricts the types that it can be unified with to types that are **instances** of the named **type class**.

- So `(+) :: (Num a) => a -> a -> a` says that `(+)` is a function of two operands and result all of some type `a` where `a` is an instance of the type class `Num`.

- A type class acts a bit like a **interface** in object-oriented programming. It acts as a contract: any type that is an instance of a type class must implement certain methods to qualify.

- The terminology may be a bit confusing:

  - In an OO language, an object is an instance of a class which might implement an interface.
  - In Haskell, a value has a type which might be an instance of a type class.

7 Basic Typeclasses from The Haskell Prelude

Haskell offers quite a bit of functionality in its standard library. In particular, the **Prelude**, the set of type and function definitions imported automatically into every program, defines

One of the most basic typeclasses is `Eq`, consisting of types that implement an equality-testing operation. Here is how it is defined:

```
1    class Eq a where
2          (==), (/=)  ::  a -> a -> Bool
3
4          x /= y  = not (x == y)
5          x == y  = not (x /= y)
```

- The `class` keyword introduces a type class definition, followed by the name of the type class and a type variable that we will use in the description of the type class's interface. Think of this type variable as a formal parameter in a function definition.

- The `Eq` type class defines two required operations, equality and inequality. In this case, the two have exactly the same signature, so the type annotation is shared.

- If we asked Haskell to infer the type of `(==)`, what would we get? `(==) :: (Eq a) => a -> a -> Bool`.

- Then we see two equational function definitions. These are default implementations for `Eq`'s operations.

- This means we do not have to define both `(==)` and `(/=)`. Each is defined in terms of the other, so an implementation for one is enough. The compiler will complain if neither is implemented.

- Because we can define default implementations, type classes are actually more like **abstract classes** in OO languages.

Typeclasses can themselves have class constraints. Here is the definition of `Ord`, which describes operations available for totally ordered data types:

```
1  class Eq a => Ord a where
2      compare                :: a -> a -> Ordering
3      (<), (>=), (>), (<=) :: a -> a -> Bool
4      max                    :: a -> a -> a
5      min                    :: a -> a -> a
6
7  data Ordering = LT | EQ | GT
```

- Class constraints in a type annotation, as in line 1 above, require that the constrained type variable be an instance of the given typeclass.

- In this case, for a type to be an instance of `Ord`, it must also be an instance of `Eq`. It should be pretty clear why that is necessary.

- The full definition of `Ord` gives default implementations for all these operations so that an instance need only implement either `compare` or `(<=)`.

There are several other typeclasses worth mentioning:

- `Show` instances can be turned into a `String` representation with `show :: (Show a) => a -> String`.

- `Read` instances know how to undo the process and turn a string into a value.

- `Bounded` instances are types with a smallest and largest values, given as two polymorphic constants `minBound, maxBound :: (Bounded a) => a`.

- `Enum` instances are sequentially ordered types. Given a value in that sequence, we can use `succ, pred :: (Enum a) => a -> a` to get the next or previous value. We can use `enumFromTo :: (Enum a) => a -> a -> [a]` to get a list containing the elements in the sequence between a start value and an end value, inclusive.

Haskell's standard library also defines a hierarchy of numeric typeclasses.

- We saw that GHC would infer the type `(Num a) => a -> a -> a` for the `(+)` operator. That, along with `(*)`, `(-)` (the binary subtraction operator), `negate` (for unary negation), and a couple of others define the most basic interface for numeric types.

- `Fractional` extends `Num` with division in `(/)` and reciprocation in `recip`.

- `Floating` extends `Fractional` with real-valued logarithms, exponentiation, trigonometric functions and even `(Floating a) => pi :: a`, the polymorphic constant π.

The full numeric hierarchy is even richer and there is plenty of detail in the Prelude's typeclasses that we have glossed over. Full details are available in [1, section 6.4].

8 Creating New Typeclass Instances

Haskell typeclasses are **open**, meaning that we can define new instances of typeclasses defined in the Prelude or elsewhere.

Let's see how we can implement some of the Prelude's basic typeclasses for a simple type.

```
1  data Section = Coach
2                | Business
3                | FirstClass
4
5  instance Eq Section where
6      FirstClass == FirstClass = True
7      Business   == Business   = True
8      Coach      == Coach      = True
9      _          == _          = False
10
11 instance Ord  Section where
12     x            <= y            | x == y = True
13     Coach        <= _                     = True
14     Business     <= Coach                 = False
15     Business     <= FirstClass            = True
16     FirstClass   <= _                     = False
17
18 instance  Show  Section  where
19     show Coach      = "Coach"
20     show Business   = "Business"
21     show FirstClass = "FirstClass"
```

- An instance declaration begins with the keyword `instance`, followed by equational definitions for the various functions defined for the class.

- The definition of (`==`) for `Eq` is straightforward. We define the function in four cases. In the first three equations, we enumerate the cases where values could be considered equal and the last equation is a catch-all: the underscore character matches any value, so any case not matched by the first three equations will get caught by the fourth and will return `False`.

- We define an `Ord` instance by enumerating the ways in which `Section` values can be ordered. The first equation uses a **guard**: `x` and `y` will match any values, but the match is only successful if the **guard expression** evaluates to `True`.

- The `Show` instance is trivial: we simpliy define a string value to return for each of `Section`'s three data constructors.

We can imagine that defining instances for these typeclasses would be quite similar for any algebraic data type. It seems trivial to automatically construct an `Eq` instance for any simple sum type. Furthermore, because of the recursive nature of algebraic data types, it would be easy to extend that idea to arbitrary sum-of-products types.

```
1  data S = P1 T1_1 ...  T1_K1
2         | P2 T2_1 ...  T2_K2
3            ...
4         | PN TN_1 ...  TN_KN
5
6  instance Eq S where
7      P1 u1_1 ... u1_k1 == P1 v1_1 ... v1_k1 = u1_1 == v1_1 && .. && u1_k1
8      P2 u2_1 ... u2_k2 == P2 v2_1 ... v2_k2 = u2_1 == v1_1 && .. && u2_k2
9      ..
10     PN un_1 ... uN_kN == PN vN_1 ... v1_kN = uN_1 == vN_N && .. && uN_kN
11     _ == _ = False
```

- This is pseudocode for the general form of an `Eq` instance for a sum of N constructors that are each a product of K_N values.

- In words, two `S` values are equal if their data constructors are equal and each pair of constituent values are equal.

In fact, Haskell offers the ability to **derive** typeclass instances, and not just for `Eq`.

```
1  data Section = Coach
2               | Business
3               | FirstClass
4               derving (Eq, Ord, Show)
```

- The `deriving` keyword instructs the compiler to automatically derive instances for the typeclasses that follow.

- The Haskell 98 standard can derive instances for `Eq`, `Ord`, `Enum`, `Bounded`, `Show`, and `Read`.

- In automatically derived instances of `Ord`, `Enum`, and `Bounded`, the order of declaration of the data constructors is used. So our derived `Ord` instance for `Section` still returns `True` for `Coach <= FirstClass`.

Because the derived definitions are recursive, we might not always be able to derive instances when the constituents of product types do not support the operations we need:

```
1  data Section = Coach
2               | Business
3               | FirstClass BeverageOption
4               derving (Eq, Ord, Show)
5
6  data BeverageOption = Wine
7                      | Beer
8                      | Soda
```

- Here, we cannot automatically derive an `Eq` instance for `Section` because `BeverageOption` is not an instance of `Eq`. We cannot determine if two values using the `FirstClass` constructor are equal because we have no way of checking two `BeverageOption` values for equality.

- Similarly, we could not derive an `Ord` instance since we have no notion of ordering on `BeverageOption`s.

- In this case, adding a `deriving` clause to our definition of the `BeverageOption` type would resolve the problem.

9 Maybe, Lists, and The Functor Typeclass

Now let us consider a type defined in the Haskell Prelude, `Maybe`:

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

- `Maybe` is the Haskell version of the **option type**. It offers us a way to represent a value that might not exist. `Nothing` is the null value, and the `Just` constructor wraps an actual value.

- For example, we might want a function that parses an integer value from a string to have the return type `Maybe Integer`, since the parse might fail.

- Compare this to Java's type system where `null` is a possible value for any reference type. `null` is a `Person` even though `null` does not respond to any of `Person`'s methods–or any methods at all!

- In Haskell, on the other hand, a function that claims to return a `Person` always returns a full-fledged `Person` (barring exceptional failure) and a function that sometimes returns a `null`-like value must declare that in its type, e.g., `String -> Maybe Integer`.

The safety we get when we use an option type is nice, but it comes with some inconvenience: If I have a value of type `Maybe Integer`, how do I add five to it? In general, how do I unwrap a value of the form `Just x` to get at `x`? It is not difficult in principle:

```
1  parseInteger :: String -> Maybe Integer
2  # Implemented elsewhere
3
4  example1 :: String -> Integer
5  example1 str = case parseString str of
6                   Just x  -> x + 5
7                   Nothing -> 0
8
9  example2 :: String -> Maybe Integer
10 example2 str = case parseString str of
11                   Just x  -> Just (x + 5)
12                   Nothing -> Nothing
```

- However, this code has some shortcomings:

- In `example1`, we are making an assumption about how to handle the error case (returning zero if the parse failed) that is now interwoven with the independent process of adding five.
- Both examples repeat the code to test both the `Just` and `Nothing` case. Moreover, if we used `Maybe` frequently (which is encouraged), this would start to get rather annoying.

We will reject `example1` because we really would like to maintain the orthogonality of dealing with `Maybe` values and our actual operation. However we can use higher-order functions to factor out repetition we see in analyzing `Maybe`s.

```
1  applyToMaybe :: (a -> b) -> (Maybe a) -> (Maybe b)
2  applyToMaybe f Nothing  = Nothing
3  applyToMaybe f (Just x) = Just (f x)
```

- `applyToMaybe` factors out handling the `Nothing` and `Just` cases of `Maybe` values.
- We also get some vocabulary for lifting normal function application into the world of `Maybe` values.
- In our add five example, we can now just use `applyToMaybe (+5) $ parseString str`

Here is another example. We have seen Haskell's basic, homogeneous list type. It offers us a way to represent a collection of zero or more values of some type.

We have also seen the function `map ::  (a -> b) -> [a] -> [b]` that applies a function to each element in a list and returns the results collected into a new list.

```
1  map :: (a -> b) -> [a] -> [b]
2  map f []     = []
3  map f (x:xs) = f x : (map f xs)
```

If we compare `applyToMaybe` and `map`, we see some important similarities:

- Both functions have an empty case and a case where one or more values are unwrapped, a function applied, and the result(s) wrapped back up.

- If we ignore the special case of Haskell's list type syntax, the functions have analogous types of the form `(a -> b) -> f a -> f b`

9.1 The Functor Typeclass

In fact, this pattern is codified in Haskell with the `Functor` type class:

```
1  class Functor f where
2      fmap :: (a -> b) -> f a -> f b
```

- A `Functor` instance is always a polymorphic data type with a single type parameter.

- At one level we can think of `Functor`s as simply mappable containers.

- At another level, we can think of them as values in some sort of context, where `fmap` lifts function application into that new context.

9.1.1 Functor Instances

Here are some instances of the `Functor` type class:

- `Maybe`

  - We can think of `Maybe a` as a context representing a value of type `a` with the possibility of failure.
  - In this context, we can think of `fmap` as creating new functions that know how to propagate these failure states.

- **Lists, i.e., `[]`**

  - The implementation of `fmap` for lists is literally just the Prelude's `map` function.

- From the context perspective, we can think of lists as non-deterministic values; i.e., the result of applying the function (* 2) to the non-deterministic value that might be one of [1, 2, 3] would be the non-deterministic value that might be one of [2, 4, 6].

- Tree

  - Mapping over a collection makes sense for trees, but what might Tree represent from the perspective of values in a context?
  - Interestingly, while mapping over elements in a set seems reasonable enough, Haskell's Set type cannot be directly declared an instance of Functor. Because Set is implemented via balanced binary trees, it has an Ord constraint on the types it can contain. This extra constraint is incompatible with the general Functor definition; we would need fmap to have the type (Ord a, Ord b) => (a -> b) -> f a -> f b.
  - Map, however can be a Functor. Rather, maps with keys of type k, i.e., Map k can be a Functor. Haskell Maps are represented using balanced binary trees over the key type k, so there is still an Ord constraint, Functor cares about the type of the values, not the type of the keys.

- ((->) e)

  - This type looks a big strange. Haskell's syntax does not allow it, but read this type as (e ->).
  - Concretely, the type of the fmap implementation here would be (a -> b) -> (e -> a) -> (e -> b)
  - [5] describes ((->) e) as "a (possibly infinite) set of values of a, indexed by values of e," or "a context in which a value of type e is available to be consulted in a read-only fashion."
  - If we have a predicate isOdd :: Int -> Bool, and fmap it over a function length :: String -> Int that returns the length of its parameter, we get a new function of type String -¿ Bool that returns whether or not the String's length is odd.
  - From the context perspective, fmap isOdd takes us from Ints indexed by Strings to Bools indexed by Strings.

9.1.2 Functor Laws

For the Haskell type system, anything that implements `fmap :: (a -> b) -> f a -> f b` is perfectly suitable as an instance of `Functor`. However, the concept of functors come to us from the branch of mathematics called category theory, where functors must satisfy certain laws. In Haskell terms:

```
1       fmap id == id
2 fmap (g . h) == (fmap g) . (fmap h)
```

- Mapping the identify function over the contents of a `Functor` just gives back the original `Functor`.
- `fmap` distributes over function composition.
- Ultimately, these two laws just mean that a well-behaved `Functor` instance only operates on the contents of the `Functor`, leaving its structure unchanged.

Consider this badly-behaved instance definition for lists taken from [5].

```
1 instance [] where
2     fmap g []      = []
3     fmap g (x:xs) = g x : g x : fmap g xs
```

- This implementation of `fmap` duplicates all the output values: `fmap (+1) [1, 2, 3]` returns `[2, 2, 3, 3, 4, 4]`.
- The first law is broken because `fmap id [1, 2, 3]` returns `[1, 1, 2, 2, 3, 3]` rather than `[1, 2, 3]`.
- The second law is broken because `fmap ((+1) . (*2)) [1,2,3]` returns `[3, 3, 5, 5, 7, 7]` rather than `[3, 5, 7]`.

Although Haskell's type system is quite powerful, it is in general undecideable whether a `Functor` instance satisfies the two laws described above, so that requirement cannot be checked at compile time. Since other Haskell code in

the standard libraries and elsewhere will expect new `Functor` instances to be well-behaved, it is the responsibility of the programmer to prove, at least to their own satisfaction, that their implementation satisfies those laws. We will look at more typeclasses later on, each with their own laws, and this caveat applies to them as well.

10 Applicative Functors

When we looked at the `Functor` type class, we saw in `fmap` a way for us to lift normal functions into the domain of computational contexts, the `Functor` instances, where they can operate on values in those contexts.

But recall that in Haskell, functions are themselves first-class values. So how do we use a function that is itself in a computational context? This question is answered by the concept of applicative functors, realized in Haskell with the `Applicative` type class.

Consider this scenario. We are given an `Integer` with the possibility that it might not actually be there, i.e., `Maybe Integer`. We are also given a function to apply to that value, again with the possibility that it might not actually be there, i.e., `Maybe (Integer -> Integer)`. Of course, since either the function or the parameter might be `Nothing`, the return value needs to be able to propagate this possibility of failure. How would we accomplish that?

```
1  maybeApplyToMaybe :: Maybe (Integer -> Integer) -> Maybe Integer -> Maybe
2  maybeApplyToMaybe Nothing _ = Nothing
3  maybeApplyToMaybe _ Nothing = Nothing
4  maybeApplyToMaybe (Just f) (Just x) = Just \$ f x
```

There are three cases:

- Line 2: When we get no function to apply, the parameter does not matter: the result is `Nothing`.
- Line 3: When we get no parameter to apply the function to, the function does not matter: the result is `Nothing`.

- Line 4: When we actually get a function and an parameter, we can unwrap them from their `Maybe` wrapper, apply the function to the parameter, and return the result wrapped back up.

What happens if we want to use a function with two or more parameter in this way? We do not actually have to write any more code: Haskell's default of curried functions gives us native partial application of functions and `maybeApplyToMaybe` gets that for free.

Say we have a function `add ::  Integer -> Integer -> Integer`. The expression `add 5` has type `Integer -> Integer`.

If we look back at line 4 of the previous example, we apply our unwrapped function to the unwrapped parameter, and return the result wrapped back up. Since the partial application `add 5` returns a function of type `Integer -> Integer`, `maybeApplyToMaybe (Just add) (Just 5)` returns a value of type `Maybe (Integer -> Integer)`.

10.1   The `Applicative` Type Class

We saw in the previous section that `map` for lists and `applyToMaybe` shared a common pattern, which led us to the general `Functor` type class. The same idea works here, and the resulting type class is called `Applicative`, short for **applicative functor**.

```
1  class Functor f => Applicative f where
2      pure  :: a -> f a
3      (<*>) :: f (a -> b) -> f a -> f b
```

The `Applicative` type class has its own class constraint: every instance of `f` of `Applicative` must also be an instance of `Functor`. In fact, most of the standard library's `Functor` instances are also `Applicatives` as well.

Importantly, though, `Applicative` and its functions are not included in the Prelude and must be imported manually from the `Control.Applicative` module.

10.2 Applicative's Functions

If we look back at `maybeApplyToMaybe :: Maybe (Integer -> Integer) -> Maybe Integer -> Maybe Integer`, we see simply a monomorphic instance of the more general, polymorphic type of (`<*>`). In fact, Haskell would have inferred a more general type for `maybeApplyToMaybe`: `Maybe (a -> b) -> Maybe a -> Maybe b` and our implementation is essentially the standard library's implementation of (`<*>`) for `Maybe`.

That means we can scrap our 17 character function name and rewrite `maybeApplyToMaybe (Just (+5)) (Just 2)` as `Just (+5) <*> Just 2`.

In general, (`<*>`) is the function (used infix like an operator) that takes a function in some `Applicative` context `f` and a value in the same context and handles the plumbing of unwrapping the function and the value, applying the function to the value, and returning the result wrapped back up in the `f` context.

We have not said much about the other half of the `Applicative` class, but it is quite simple and the type is very telling. What makes sense for the type `a -> f a`?

We are getting a value of any type and returning a value of that type in the context described by `f`. Without knowing anything about the type `a` of the parameter, we cannot modify it in any way. So, from the type alone, we can surmise that `pure` probably injects a value into the context described by `f` in some default way.

What might `Maybe`'s implementation of `pure` look like? It is literally just `Just`!

10.3 Building a Better `fmap`

It might seem that all we really get from `Applicative` is a way to factor out the unwrapping required to apply a function in a context to a value in a context, but there is more here.

Consider the operator `(<$>) :: (a -> b) -> f a -> f b` provided by the `Control.Applicative` module. It that takes a function, injects it into `f` with `pure`, and then uses `(<*>)` to apply it to a value in `f`.

The type of `(<$>)` should look familiar: it is the same type as `fmap`. In fact, assuming both `Functor` and `Applicative` laws (which we will see in a moment) the two are synonyms: `g <$> pure x == fmap g $ pure x`.

What we really get out of `Applicative` is a better version of `fmap`.

Suppose we called `fmap (+) (Just 3)`. This is not an error, we will just partially apply `(+)` to the wrapped value `3` and get back basically `Just (3+)`, a function value inside a `Maybe` context, exactly where we were at the beginning of this section.

Now that we have seen how `Applicative`s work, we have the tools needed to finish fmapping a function with two parameters: `fmap (+) (Just 3) <*> Just 5`, in `Applicative` terms: `pure (+) <*> Just 3 <*> Just 5`, or even more idiomatically: `(+) <$> Just 3 <*> Just 5`.

From a practical perspective, we could write a function that attempted to build a `PlaneTicket` value based on optional `Section` and `MealOption` values:

```
1  createPlaneTicket
2      :: Maybe Section
3      -> Maybe MealOption
4      -> Maybe PlaneTicket
5  createPlaneTicket section meal
6      = PlaneTicket <\$> section <*> meal
```

`createPlaneTicket` uses the language of `Applicative` to succinctly lifts the `PlaneTicket` data constructor into the `Maybe` context where the `MealOption`s and `Section`s might not exist.

In fact, the type that Haskell would actually infer for `createPlaneTicket` is `(Applicative f) => f Section -> f MealOption -> f PlaneTicket` and would work for any `Applicative`, such as lists, which we will take a look at shortly.

10.4 Applicative **Laws**

There are four laws that `Applicative` instances should follow, with the same
motivations and caveats described when we discussed the `Functor` laws.

- **Identity**: `pure id <*> v == v`
- **Homomorphism**: `pure g <*> pure x == pure (g x)`
- **Interchange**: `g <*> pure x == pure ($ x) <*> g`
- **Composition**: `g <*> (h <*> k) == pure (.)  <*> g <*> h <*> k`
- **Functor Instance**: `fmap g x == pure g <*> x`

The **identity law** can be thought of as putting an upper bound on what can
actually happen inside the plumbing of the `Applicative` implementation. If
that plumbing does anything that fails to preserve identity, it is not a proper
`Applicative`.

The intuition behind the **homomorphism law** is that these operations are
just lifting function application into `Applicative` contexts. If we have a
function `f` and a value `x`, inject each into the context via `pure` and apply
them via (`<*>`), we should get the same thing as if we had injected `f x` into
the context directly.

The **interchange law** is a bit tricky. To start, remember that the (`$`)
operator is just function application with a very low precedence. So (`$ y`)
is a function that takes a function and applies it to `y`. What the interchange
law is trying to express is that the order in which we evaluate the function
and its parameter should not matter in a proper `Applicative` instance.

We can think of the **composition law** as formalizing an associative property
for (¡*¿) in terms of Haskell's standard function composition operator (`.`).

Finally, the **functor instance law** describes how an `Applicative` instance
should behave relative to its `Functor` instance and is required for the equiv-
alence between (`<$>`) and `fmap` to hold for an `Applicative` instance.

[5][section 4.2] offers a bit more detail on the `Applicative` laws.

10.5 `Applicative` **and Lists**

When we looked at `Functor`s, our two canonical examples were `Maybe` and lists, but we have not really mentioned lists yet in this section. The problem is not that lists are not `Applicative`s, but that there are two perfectly reasonable ways to implement the `Applicative` instance for lists!

Lists are a context that support zero or more values. So suppose we had a list of functions and wanted to apply them (in the `Applicative` sense) to some values also in a list context:

```
1  [(+1), (*2), (^3)] <*> [4, 5, 6]
```

We could certainly interpret this as pair-wise application, applying `(+1)` to `4`, `(*2)` to `5`, etc.

However, recall that we could view lists not just as a container of zero or more values but as a kind of non-deterministic value where `[4, 5, 6]` represents a value that might be any one of those numbers. In this interpretation, it might make more sense to do apply each function from the left-hand list to each value value in the right-hand list.

The result is a list containing the possible values when a non-deterministic function is applied to a non-deterministic value, yielding a total of 9 possible values in this example.

In fact, the Haskell library's `Applicative` instance for lists uses the latter interpretation. The implementation looks like this:

```
1  instance Applicative [] where
2      pure x    = [x]
3      gs <*> xs = [ g x | g <- gs, x <- xs ]
```

- `pure` injects a value into the list context by creating a singleton list containing that value.
- `(<*>)` applies each function from the left-hand list to each value in the right-hand list as discussed. It does so via Haskell's **list comprehen-**

sion syntax.

What about the pair-wise version of `Applicative` for lists? Due to language constraints, the list type cannot have two implementations for the same type class. Instead, Haskell offers type called `ZipList` that wraps a normal list but offers a different `Applicative` instance:

```
1  newtype ZipList a = ZipList { getZipList :: [a] }
2
3  instance Applicative ZipList where
4      pure x = repeat x
5      (ZipList gs) <*> (ZipList xs) = ZipList (zipWith (\$) gs xs)
```

- The `newtype` keyword defines a type synonym that is checked at compile time but discarded so there is no run-time overhead. A `ZipList` can never be used as a normal list, but there is no additional overhead. Record syntax is used here to automatically create a function `getZipList` to translate normal lists to `ZipList`s.

- `zipWith` takes two lists and applies a function pair-wise to the elements of those lists. In this case the function is (`$`), which we have seen previously. So the `ZipList` instance of `Applicative` is implementing pair-wise application.

- Because `zipWith` truncates the result to the length of the shorter of its two parameter, it makes sense for `pure` to inject values into the `ZipList` context by creating an infinite list via `repeat`.

- If we used the same `pure` implementation as normal lists, `pure g <*> [1..]` == `[g 1]` and the functor instance law no longer holds.

10.6   Summary

In this section we have looked at applicative functors and Haskell's `Applicative` type class. We have seen how `Applicative` offers an abstraction for applying functions even when the functions themselves were wrapped up in a context just as `fmap` allowed us to apply bare functions to values in a context.

In the next section we will discuss the `Monad` type class and see how it further extends the notion of computational contexts that we have built up via `Functor` and `Applicative`.

11   Monads

Consider the following code in a Java-like language:

```
1  String emailDomain = user.getContactInfo()
2                           .getEmailAddress()
3                           .getDomain();
```

We have several domain objects:

- `User`, which has method `getContactInfo()` that returns the user's contact information with the type

- `ContactInfo`, which as a method `getEmailAddress()` that returns the associated email address with type

- `EmailAddress`, which has a method `getDomain()` which returns the the domain portion of that email address as a `String`.

Now suppose that a `User`'s `ContactInfo` is optional so that `getContactInfo()` might return `null`. Likewise, a `ContactInfo` record's `EmailAddress` is optional so that `getEmailAddress()` might return `null`.

That makes the above code snippet dangerous. The `getEmailAddress()` and `getDomain()` calls could be performed on null references, causing a `NullPointerException`. If uncaught, the program crashes.

We could try this:

```
1  String emailDomain;
2  ContactInfo contactInfo;
3  EmailAddress emailAddress;
4
```

```
 5 | contactInfo = user.getContactInfo();
 6 |
 7 | if ( contactInfo != null )
 8 | {
 9 |     emailAddress = contactInfo.getEmailAddress();
10 |
11 |     if ( emailAddress != null )
12 |     {
13 |         emailDomain = emailAddress.getDomain();
14 |     }
15 |     else
16 |     {
17 |         emailDomain = null;
18 |     }
19 | }
20 | else
21 | {
22 |     emailDomain = null;
23 | }
```

We have managed to propagate possible `null` values through the chain of method calls, but at the cost of a great deal of boilerplate code.

Of course, we have already seen how Haskell's `Maybe` type lets us encode the possibility of `null`-like values explicitly. Suppose we have analagous Haskell types and functions. The above code translates to something like:

```
 1 | userEmailDomain :: User -> Maybe String
 2 | userEmailDomain user = case getContactInfo user of
 3 |     Nothing         -> Nothing
 4 |     Just contactInfo -> case getEmailAddress contactInfo of
 5 |         Nothing    -> Nothing
 6 |         Just email -> getDomain email
```

Although we are now explicit about the possibility of a null result, we have not really addressed the issue of boilerplate. If we had to chain together even more calls, our code would quickly stair-step right off the screen.

We can see a pattern, however. When we apply `getContactInfo` to `user`, we do a pattern match. If we got an actual value (the `Just` case), we take that

value and pass it on to the `getEmailAddress` call. In the `null` case, though, we short-circuit the chain of function calls and just return `Nothing`. The same strategy is used when we try to pass the result of `getEmailAddress` into `getDomain`.

We can generalize this pattern:

```
1  chainMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
2  Nothing  `chainMaybe` f = Nothing
3  (Just x) `chainMaybe` f = fx
```

Note: Here we juse backticks around the function name to cause Haskell treat the function like an infix operator, not unlike using parentheses around an infix operator causes Haskell to treat it like a regular, prefix function.

Having factored out the pattern matching to handle both cases, we can rewrite our stair-stepped chain of function calls:

```
1  userEmailDomain :: User -> Maybe String
2  userEmailDomain user =
3  (Just user) `chainMaybe` getContactInfo
4              `chainMaybe` getEmailAddress
5              `chainMaybe` getDomain
```

Now our code is as readable as the original Java chain of method calls, but with the `null`-safety of the clunky second attempt.

11.1   The `Monad` Type Class

Not surprisingly, Haskell offers a type class to describe this same pattern in polymorphic terms:

```
1  class Monad m where
2      (>>=)  :: m a -> (a -> m b) -> m b
3      (>>)   :: m a -> m b -> m b
4      return :: a -> m a
5      fail   :: String -> m a
```

```
6
7      m >> k = m >>= \_ -> k
```

- `(>>=)` is the monadic chaining operator. It is just a polymorphic, infix version of `chainMaybe`.

- `(>>)` can be thought of as a chaining operator where the value passed into the right-hand function is simply ignored. It is given a default implementation in terms of `(>>=)` on line 5.

- `return` is the `Monad` class's general method for injecting a value into a monadic wrapper. Code that uses the `Monad` interface cannot use specific constructors like `Just` on line 3 of our final `userEmailDomain` implementation, so `Monad` instances implement `return` to define the behavior. If `return` sounds familiar, it is because it is actually identical to `pure` from `Applicative`. We will discuss this further in a moment.

- `fail` offers a way for `Monad` instances to short-circuit evaluation when a computation has failed. Use of `fail` is discouraged in general because the `IO` monad implements `fail` by raising a fatal error.

11.2 `Monad` **and** `Applicative`

We mentioned before that `return` and `pure` were basically identical. In fact, conceptually, `Applicative` is a superclass of `Monad`.

- Applicative is actually a superclass of Monad
- But only very recently has that relationship been codified in the actual definition of Monad
- (what version?)

11.3 The `Maybe` Monad

- Walk through instance definitions

11.4   The List Monad

- Lists as non-deterministic values -¿ List monad as non-deterministic computations
- Walk through instance definitions (concatMap)
- Describe how userEmailDomain behaves if written as generic monadic operation and applied in list monad.

11.5   `do`-Notation

- Describe do-notation syntax, syntactic sugar
- Left-arrow binding
- Desugaring do-notation

11.6   The `IO` Monad

- Describe IO monad as mechanism for defining impure computations in a pure language
- IO values aren't "flagged" values, but lazy descriptions of computations that go out and get values from the outside world when finally executed.
- Examples

References

[1] Simon Peyton Jones, et al., *Haskell 98 Language and Libraries: The Revised Report*, 2002.

[2] Paul Hudak, et al., "A History of Haskell: Being Lazy With Class", 2007.

[3] R.M. Burstall, D.B. MacQueen, D.T. Sannella, "Hope: An Experimental Applicative Language", 1980.

[4] Ralf Hinze, Simon Peyton Jones, "Derivable Type Classes", *Proceedings of the Fourth Haskell Workshop*, 227–236, 2000

[5] Brent Yorgey, "The Typeclassopedia", *The Monad.Reader*, 13, 17-68, 2009

[6] GHC Documentation, , 2014