NOTES ON CATEGORY THEORY USING HASKELL

VAIBHAV KARVE

These notes were last updated August 9, 2018. They are notes taken from my reading of *Category Theory for Programmers* by *Bartosz Milewski*.

Contents

1. Introduction

(1) If $f : a \to b$ and $g : b \to c$ are morphisms in a category then so is $g \circ f : a \to c$, read as g after f. In Haskell, functions can be compased as follows:

```
> f :: A → B
> g :: B → C
> g . f --one way of writing composition
> g ∘ f -- another way
```

(2) By definition, in a category the composition should be associative and for every object in a category must have a corresponding identity morphism (wrt composition). The identity morphisms for each object in the category can be defined simultaneously by creating an identity-polymorphism-template. Such a polymorphism (a function that will work on all pbjects/types) can be written as:

```
> id :: a → a -- a denotes a type variable instead of a type
> id x = x
```

In Haskell, names of concrete types start with uppercase letters, while names of types start with lowercase letters. In Haskell, the body of a function is always an expression and can never be a statement. The result of a function is also always an expression.

(3) In Haskell, there is only one category – the category of types (this is an approximate statement as it ignores the existence of "bottoms").

(4) Types in Haskell are just a (finite or infinite) bunch (not necessarily sets) of values. Common types include `Integer`, `Bool`, `Char` and `String`. `String` is a synonym for list of `Char`.

```
> x :: Integer --this is how the type of a variable is declared.
```

(5) In order to account for potentially non-terminating functions, each type is extended by a special value called *bottom* denoted by $\_|\_$ or $\perp$. This value corresponds to a non-terminating computation. Hence, a function
```
> f :: Bool → Bool
```
might may return a value of `True`, `False` or $\perp$.

(6) Even runtime errors can be treated as bottoms in the type system. If one wishes to explicitly return $\perp$ as a value, one uses `undefined`:
```
> f:: Bool → Bool
> f = undefined --we don't need the x because bottom is not only a part of Bool, it is also a part of Bool
 -> Bool
```

(7) Functions that may return bottom are *partial*, as opposed to *total functions*.

(8) Thus, the category of Haskell types is not **Set**. It can be referred to as **Hask**.

(9) Language has two parts –
   (a) *Syntax* or *grammar* which talks about the rules of writing the language correctly.
   (b) *Semantics* which talks about what sentences in the language actually mean/do.

   Semantics of a program can be checked using either
   (a) *operational semantics*, which describes the mechanics of a program execution, and is harder to practically implement, or
   (b) *denotational semantics*, which is based on math, and is easier to implement.

(10) In denotational semantics, every programming construct is given its mathematical interpretation. Then, proving a property of a program becomes equivalent to proving a theorem.

(11) Factorial function in Haskell:
```
> fact n = prod [1..n]
```

(12) Eugino Moggi discovered that computational effect can be mapped to *monads*.

2. Types and Functions

(1) In programming languages, functions that always produce the same result given the same input and have no side effects are called *pure functions*. In Haskell, all functions are pure. Hence, Haskell is a pure functional language.

(2) It is easier to give pure functional languages denotational semantics and model them using category theory.

(3) Since types are sets, what is the type corresponding to the empty set? It is a type called `Void`. Functions that take `Void` can be defined but can never be called. Such a function is polymorphic in the return type as it doesn't matter what it returns (since it can never be called).
```
> absurd :: Void -> a --this is just a theoretical construct. Not actually implementable.
```
   Void is an empty or uninhabited type. However, since uninhabited types aren't really implemented in Haskell, `Void` and therefore `absurd` cannot be implemented as shown. One might be able to salvage the situation by considering bottom to be the only element in `Void`.

(4) The Curry-Howard theorem connects logic with types. According to this, the `Void` type represents falsity and the type of the function `absurd` corresponds to the statement "from falsity follows everything" or in Latin: *ex falso sequitur quodlibet*.

(5) The type for a single set is represented by the double parenthesis in Haskell: `()`. A function from this type can always be called but ut will always return a fixed value. In Haskell, a) the type, b) the

constructor for this type, and c) the only value corresponding to a singleton set are all represented by the same symbol `()`. This symbol is pronounced "unit".

(6) A function that acts on unit can be defined as:

```
> f23 :: () → Integer
> f23 () = 23 --always returns 23
```

This function can be called as:

```
> f23 ()
23
```

(7) In fact, `f23` is equivalent to the `Integer` element `23`. This way, we can replace explicit mentions to elements of a set with calls to functions that map unit into that set. In other words, functions from unit to any type `a` are in one-to-one correspondence with elements of that set `a`.

(8) What about functions with a return type of unit? For every set A, there is a unique function that maps A to unit, (this function maps each element to `()`). For example:

```
> fInt :: Integer → ()
fInt x = ()-- such a function just discards its argument
```

A cleaner way of doing the same thing would be:

```
> fInt :: Integer → ()
fInt _ = ()-- by using the wildcard character, we don't have to invent a name for a variable we are going
 to discard anyway
```

(9) This function that maps to unit is independent of the ot only the value, but even the type of the argument. Functions that can be implemented with the same formula for any type are called *parametrically polymorphic*.

```
> unit :: a → ()
> unit _ = ()
```

(10) The type corresponding to a two-element set in Haskell is called `Bool`. Since `Bool` already has an implementation in Haskell, one can define a new type called `Boolean` instead:

```
> data Boolean True|False --to be read as "Boolean is either True or False". This actually is bad because
 it redefines True to be of type Boolean rather than Bool.
```

(11) Pure functions from `Bool` just pick up two values from the target – one for `True` and one for False. Functions to `Bool` are *predicates*. For example, the predicate `isDigit` that acts on `Char`.

3. Categories Great and Small

(1) The most trivial category is one with no objects and no morphisms.

(2) A *free category generated by a graph* is a category whose objects are the nodes of the graph and whose edges are formed by composable edge-chains of length ≥ 0.

(3) A set with a relation $\leq$ is a *preorder* if it satisfies:

- Reflexivity: $a \leq a$
- Transitivity/composability: $a \leq b$ and $b \leq c$ implies $a \leq c$
- Associativity of composition.

A preorder is a *partial order* if it also satisfies skew-symmetry: $a \leq b$ and $b \leq a$ implies $a = b$. If every pair of objects is necessarily related to each other by this partial order, then that is a *linear order* or *total order*.

(4) The set of morphisms from object a to b in a category C is called a *hom-set* and is written $C(a, b)$ or $\mathrm{Hom}_C(a, b)$.

(5) In categorical terms, a *preorder* is a category where there is at most one morphism going from any object a to any object b. Such a category is also called a *thin* category. Every hom-set in a preorder is either empty or a singleton set.

(6) A *partial order* is a preorder in which cycles of morphisms are forbidden.

(7) Sorting algorithms like quicksort, bubble sort and merge sort can only work on totally ordered sets. Partial orders can be sorted using topological sorts.

(8) A *monoid* is a set with a binary operation that is associative and that has one special element that behaves like a unit with respect to the operation. For example, natural numbers with zero, wrt addition. String concatenation is also a monoid with the unit element being the empty string.

(9) In Haskell, we can define a type class for monoids (note that a type class is like a template for several types that might belong to it).

```
class Monoid m where
    mempty  :: m -- the unit element in a monoid
    mappend :: m → m → m -- the binary operation in a monoid
```

It is the responsibility of the programmer to ensure that the properties of `mappend` and `mempty` are satisfied. These properties cannot be baked into the definition.

(10) The standard GHCI Prelude already implements `String` as an instance of a monoid as follows:

```
instance Monoid String where
    mempty  = ""
    mappend = (++) -- we use list concatenation operation because a string is jut
        a list of characters.
```

Note that in Haskell, an infix operator such as `++` can be converted into a two-argument function by writing it as `(++)`.

(11) A statement of equality of morphisms in the category **Hask** like `mappend = (++)` is not the same as a statement of the equality of values:
`> mappend s1 s2 = (++)s1 s2`. Even through they both work, only the first can be easily generalized to other categories. The latter is called an *extensional equality* or *point-wise equality*. The former is called a *point-free equality*.

(12) Monoids can also be thought of category-theoretically. A monoid is a single object category with a set of morphisms that follow appropriate rules of composition.

(13) It is a general rule that we can always extract a set from a single object category.

(14) For every categorical-monoid one can obtain a set-theoretic monoid. Suppose the categorical monoid is M. Then the set is $M(m, m)$ for the single object m in the category. The monoid's binary operation on this set is defined by $f + g = f \circ g$. Hence, the categorical and set-theoretic definitions of a monoid match up.

(15) There are categories in which morphisms don't form sets. If they do form sets they are *locally small categories*.

(16) Big takeaway – the composition of morphisms in M translates into monoidal product in the set $M(m, m)$.

4. Kleisli Categories

(1) Another way to think of their impure functions is that due to their side effects, they cannot be memoized. The aim in functional programming will be to transform impure functions into pure

functions. This can be done by "embellishing" the return types of a bunch of functions in order to piggyback some additional functionality in them.

(2) In the category of types and functions, we leave the types as objects but redefine morphisms to be embellished functions. For example, an embellished function from `Int` to `Bool` will actually have a return type of `(Bool, String)` where the string represents the log or the standard output message. However, note that we will still say that this is a function from `Int` to `Bool` by disregarding the embellishment.

(3) The price for embellishment is that simple composition doesn't work with embellished functions due to mismatch in the return types and input types. Recipe for composition in the category with embellished morphisms:
   (a) Execute the embellished function corresponding to the first morphism.
   (b) Extract the first component of the result pair and pass it to the embellished function corresponding to the second morphism.
   (c) Concatenate the second component (the string) of the first result and the second component (the string) of the second result.
   (d) Return a new pair combining the first component of the final result with the concatenated string.

(4) The embellished identity morphism will return the argument unchanged as the first component and will return an empty string as the second component. All that is needed for such embellishments in general is for the logger to be a monoid.

(5) To define the logger in Haskell, we define a new type:

```
> type Writer a = (a, String)--here we are defining a type alias
```

Here, the type `Writer` is parametrized by a type variable `a` and is equivalent to a pair of `a` and `String`. Then, embellished morphisms are functions of the type `a → Writer b` and composition itself can be defined as an infix operator (sometimes called *fish*):

```
> (>=>) :: (a → Writer b) → (b → Writer c) → (a → Writer c)
```

This is a function that itself takes in two functions as arguments and returns another function. The definition of the infix operator is:

```
-- [[This is returning a lambda function of one argument x. The lambda is
    indicated by the backslash \, to be thought of as an amputated λ.]]
m1 >=> m2 = \x →
    let (y, s1) = m1 x   -- ''let'' expression lets us declare auxilliary variables
        (z, s2) = m2 y
    in (z, s1 ++ s2)     -- [[the overall value of the let expression is specified
        in its ''in'' clause.]]
```

(6) The definition the identity morphism for this category can also be given

(7) To be continued...