The Haskell Programming Language as a Category

Kelvin Abrokwa-Johnson

December 2017

1 Introduction

This paper will be an introduction to Category Theory as it applies to functional programming. I will be exploring category theoretic ideas and their use in the Haskell programming language.

In general, Category Theory aims to abstract away the structures of individual objects in categories, relying more on the morphisms between them to explore the structures of categories as a whole. However, since this paper is aimed at the examination of a specific category, namely the **Hask** category of the Haskell programming language, we will dive into the details of the objects in **Hask**, especially in Section 4.

We begin with a high level overview of Cateogry Theory and some of its key features and definitions.

2 Category Theory

Category Theory is an abstraction of mathematical structures containing objects and relationships between those objects.

2.1 Definition

A category has the following:

1. A collection of objects
2. A collection of arrows, or morphisms, with objects as their source and target. An arrow f with source A and target B is denoted $f : A \to B$.
3. A composition operation $\circ$ such that given arrows $f : A \to B$ and $g : B \to C$, $g \circ f : A \to C$ is a composite arrow that satisfies the following associativity law:
   - Given arrows $f : A \to B, g : B \to C, h : C \to D$, $f \circ (g \circ h) = (f \circ g) \circ h$
4. For each object A, an identity arrow $id_A : A \to A$ such that for any arrow $f : A \to B$, $id_B \circ f = f = f \circ id_A$.

Example 2.1. *The category* **Set** *has sets as objects, functions as arrows. Function composition serves as arrow composition and identity functions are identity arrows.*

Example 2.2. *A partially ordered set or poset is a set S together with a binary relation $\leq$ satisfying the following conditions:*

- *$x \leq x$ for all $x \in S$ (reflexivity)*
- *if $x \leq y$ and $y \leq x$ then $x = y$ (antisymmetry)*
- *if $x \leq y$ and $y \leq z$ then $x \leq z$ (transitivity)*

Given two posets A and B, a function $f : A \to B$ is order-preserving or monotone if $x \leq y$ implies $f(x) \leq f(y)$ for $x, y \in A$.

The category **Poset** *has partially ordered sets as objects and order-preserving functions as arrows.*

Example 2.3. *A monoid $(M, \cdot, e)$ is a set with a binary operation $\cdot$ that is associative, that is given $x, y, z \in M$, $(x \cdot y) \cdot z = x \cdot (y \cdot z)$, and an identity element e such that $ex = x = xe$ for all $x \in M$. Let $(M_1, \cdot, e_1)$ and $(M_2, \cdot, e_2)$ be monoids. A monoid homomorphism is a function $f : M_1 \to M_2$ such that $f(e_1) = e_2$ and $f(xy) = f(x)f(y)$ for $x, y \in M_1$.*

The category **Mon** *has monoids as objects and monoid homomorphisms as arrows.*

Example 2.4. *A group G is a set with a binary operation, say $\cdot$, such that the set is closed under the binary operation, or group operation, the operation has associativity, there exists an identity e, and for each element x, an inverse x^{-1} such that $x \cdot x^{-1} = x^{-1} \cdot x = e$.*

The category **Grp** *has groups as objects and group homomorphisms as arrows.*

2.2 Monomorphisms, Epimorphisms, and Isomorphisms

Definition 2.1. *Let $\mathcal{C}$ be a category and let $f : B \to C$ be an arrow in $\mathcal{C}$, then, f is a **monomorphism** if for any arrows $g : A \to B, h : A \to B \in \mathcal{C}$, $f \circ g = f \circ h$ implies $g = h$. We say f is **monic** or f is a **monic**.*

Definition 2.2. *Let $\mathcal{C}$ be a category and let $f : A \to B$ be an arrow in $\mathcal{C}$, then, f is an **epimorphism** if for any arrows $g : B \to C, h : B \to C$, $g \circ f = h \circ f$ implies $g = h$. We say f is **epic** or f is an **epi**.*

Definition 2.3. *Let $\mathcal{C}$ be a category and let $f : A \to B$ be an arrow in $\mathcal{C}$, then, f is an **isomorphism** is f is **monic** and **epic**.*

We can observe examples of these arrows types in the category **Set**.

Let A, B, C be sets and let $f : B \to C$ be an injective function, that is, for any $x, y \in B$, if $f(x) = f(y)$ then $x = y$. Now take functions $g : A \to B, h : A \to B$ such that $f \circ g = f \circ h$, that is, for any $a \in A$, $f(g(a)) = f(h(a))$. Since f is injective, $g(a) = h(a)$ for all elements in A, so $g = h$. Thus, injective functions are monomorphisms.

Now, let A, B, C be sets and let $f : A \to B$ be a surjective function, that is, for any $b \in B$ there exists some $a \in A$ such that $f(a) = b$. Now take functions $g : B \to C$ and $h : B \to C$ such that $g \circ f = h \circ f$. For any $b \in B$, $g(b) = g(f(a)) = h(f(a)) = h(b)$ where $f(a) = b$. So, $g = h$. Thus, surjective functions are epimorphisms.

From the definition and the above examples, it follows that bijective functions, functions that are injective and surjective, are isomorphisms since they are both monic and epic.

2.3 Functors

A functor is a structure preserving mapping between categories. We can think of a functor as two functions, the first function mapping objects of one category to the other, the second function mapping arrows of one category to another. Let $\mathcal{A}$ and B be categories, $Obj(\mathcal{A})$ and $Obj(B)$ be the objects in $\mathcal{A}$ and $\mathcal{B}$ respectively, and $Arrow(\mathcal{A})$ and $Arrow(\mathcal{B})$ be the arrows in $\mathcal{A}$ and $\mathcal{B}$ respectively, a functor $F : \mathcal{A} \to \mathcal{B}$ is two functions

- $F : Obj(\mathcal{A}) \to Obj(\mathcal{B})$
- $F : Arrow(\mathcal{A}) \to Arrow(\mathcal{B})$ such that
  - if i_a is an identity arrow for some object in $a \in Obj(\mathcal{A})$ and $F(a) = b \in Obj(\mathcal{B})$, then $F(i_a) = i_b$
  - given objects $A, B, C \in Object(\mathcal{A})$ and arrows $f : A \to B, g : B \to C \in Arrow(\mathcal{A})$, $F(g \circ f) = F(g) \circ F(f) \in Arrow(\mathcal{B})$

From this definition, we can see that we can construct categories **Cat** with category as objects and functors as arrows. Note, however that **Cat** can only contain *small categories* (a category $\mathcal{C}$ is small if $Obj(\mathcal{C})$ and $Arrow(\mathcal{C})$ are sets, otherwise it is large). **Cat** itself is a large category, so it does not contain itself.
Let us look at some examples:

Definition 2.4. *An* ***endofunctor*** *is an functor that maps a category onto itself.*

Example 2.5. *The* ***identity functor*** *is an endofunctor that maps each object and arrow in a category to itself.*

Example 2.6. *Let F : **Mon** $\to$ **Set** be a functor from the category of monoids to the category of sets. If F maps a monoid $(M, \cdot, e)$ to the set M, the underlying set of the monoid, and monoid homomorphisms to their underlying set functions, we call it a* ***forgetful functor*** *because it "forgets" aspects of the monoid structure in the mapping.*

It is worth mentioning (if only for fun) that we may construct a category of functors, so-called **functor categories**, with functors as objects and *natural transformations* as morphisms. Let C, D be categories and let $F, G : C \to D$ be functors. A natural transformation $\eta : F \to G$ is a collection of maps $\eta : F(C) \to G(C)$ such that for any $x, y \in C$, $f \in Arrow(C)$ the following diagram commutes:

$$\begin{tikzcd}
F(x) \arrow[r, "F(f)"] \arrow[d, "\eta_x"'] & F(y) \arrow[d, "\eta_y"] \\
G(x) \arrow[r, "G(f)"] & G(y)
\end{tikzcd}$$

2.4 Duals

Let $\mathcal{C}$ be a category. The dual category $dual(\mathcal{C})$ has, as objects, the objects of $\mathcal{C}$ and, as arrows, the arrows of $\mathcal{C}$ reversed. For example, let A and B be objects in $\mathcal{C}$, and $f : A \to B$ be an arrow in $\mathcal{C}$, the $f^{op} : B \to A$ is an arrow in the category $dual(\mathcal{C})$.

2.5 Universal Constructions

Since category theory often treats objects as black boxes, it is advantageous to define constructions using arrows as much as possible. In this section, we do so for a number of *universal constructions.*

Definition 2.5. *An* ***initial object*** *in a category is an object 0 such that for each object X in the category, there exists a unique arrow $! : 0 \to X$.*

Definition 2.6. *A* ***terminal object*** *in a category is an object 1 such that for each object X in the category, there exists a unique arrow $! : X \to 1$.*

Proposition The initial object is unique up to isomorphism.

Proof. Two objects are isomorphic if there exists an isomorphism, or an invertible morphism between them. Let 0_1 and 0_2 be initial objects, then must be a unique arrow from $!_1 : 0_1 \to 0_2$ and a unique arrow $!_2 : 0_2 \to 0_1$. $!_1 \circ !_2 = id_{0_1}$ and $!_2 \to !_1 = id_{0_2}$, so 0_1 and 0_2 are isomorphic. □

Terminal objects are also unique up to isomorphism. The proof for this is the same as the proof for the uniqueness of the initial object, but with the arrows reversed. This is because, the terminal object is the dual of the initial object.

Example 2.7. *The initial object in* **Poset** *is the least element.*

Example 2.8. *The initial object in* **Set** *is the empty set ϕ. For each object A in* **Set***, there is a unique function $! : \emptyset \to A$ which characterizes the elements of A.*

Example 2.9. *The terminal object in* **Poset** *is the maximal element.*

Example 2.10. *The terminal object in* **Set** *is any set with a single element.*

Definition 2.7. *Let A and B be objects in a category. An object $A \times B$ with two projection arrows $f : A \times B \to A$ and $g : A \times B \to B$ is a* ***product*** *of A and B if for any object Z with arrows $p : Z \to A$ and $q : Z \to B$ there exists a unique arrow $\alpha : Z \to A \times B$ such that $p \circ \alpha = f$ and $q \circ \alpha = g$.*

The essence of the above definition is that we are not able to *factor* the arrows f and g for the product $(A \times B, f, g)$.

3 Programming in Haskell

The Haskell programming language came about in the late 1980s at a time of immense excitement about the functional programming paradigm. Its creation was motivated by a need for a standard non-strict (or lazy) purely functional language that can be used for both education/research and for building real world applications. It was named after the mathematician Haskell Curry, whose work in combinatorial logic was essential to the development of functional programming.

In this section, I give a brief overview of Haskell's syntax and semantics as it relates to code examples in this text. It is assumed that the reader has at least a basic understanding of programming.

3.1 Primitive Types

Haskell has the following basic primitive data types:

- Boolean: an enumeration with values True and False
- Number: there are several classes of numbers in Haskell
  - Int: a bounded integer that depends on a machine's architecture
  - Integer: an unbounded integer
  - Float: a single precision floating point number (think decimal)
  - Double: a double precision floating point number

- Character: an enumeration of Unicode characters
- List: an algebraic data type (more specifically, a disjoint union) that can either be an empty collection `Nil` or a Cons (construction) of a value of the parameterized type and a List (this represents the head and the remainder of the list, a common convention in functional programming languages such as Lisp). We discuss lists in depth in section 4.2.2.
- String: a List of characters
- Tuples: a collection items. Unlike `List`, though, the types of objects in a tuple don't have to be homogenous. The size of a tuple is fixed once it is defined.
- Unit: A zero sized tuple used as the null value
- Function: like `List`, this type is parameterized by other types as well, namely the parameter types and return type of the function. Functions in Haskell are much like functions in any language but since Haskell is a functional language, they are *first class*, meaning they can be treated as values (they can be passed to other functions as arguments and returned from functions as well).

3.2 Functions, Types, and Typeclasses

Haskell is a statically typed language. This entails that the type of every variable and function, indeed every expression, in a Haskell program is known at compile time. As we will see in the next section, this type system is the basis for thinking about Haskell category theoretically.

The following is an example of a function definition.

```
addOne x = x + 1
```

It takes one argument, adds 1 to it, and then returns the incremented value. The name of the function, `addOne` comes first, followed by a space separated list of its parameters. The last expression on the right side of the = is the function's return value. In this case it is `x + 1`.

Note that `addOne` is quite generic. It does not specify any types (can you add 1 to `"hello"`, for example). Fortunately, the Haskell type interpreter is sophisticated enough to determine that you can only add 1 to types in the `Num` typeclass.

To invoke a function, write the function name followed by its parameter list separated by spaces:

```
>>> addOne 1
2
>>> addOne 1.5
2.5
```

Since the function is generic, it works for both ints and floats. Note, however that the implicit function signature is

```
addOne :: Num a => a -> a
```

Which can be read as "`addOne` takes a type `a` of the `Num` typeclass and returns type `a`". That is, the input type is always the output type (int yields int and float yields float). The `::` symbol is used for type definitions.

This brings us to another important feature of Haskell: typeclasses. A typeclass can be thought of as a class of types, a collection of types that implement a some interface. Lets obeserve, for example, the `Ord` typeclass. This typeclass contains types that have some sort of ordering. That is, types that can be applied to functions `f` with the signature

```
f :: (Ord a) => a -> a -> Bool
```

These functions include operators such as $>, <, \geq$, and $\leq$. Mathematically speaking, these are sets that equipped with some binary relation, $\geq$, with logical reflexivity, antisymmetry, and transitivity —the same structure as posets!
Note that we can restrict this by explicitly stating the type before the function definition:

```
addOneInt :: Int -> Int
addOneInt x = x + 1
```

Now `addOneInt` only works for inputs of type `Int` and returns an `Int`.

4 The Category Hask

In this section we will examine Haskell as a category named **Hask**.

4.1 Objects and Arrows

In the category **Hask**, types are the objects. In fact, we can think of a type in Haskell as a set in the mathematical sense. For example, the unbounded type `Integer` represents the set of integers $\mathbb{Z}$. In this vane, we can think of an instance of a type, for example

```
x = 1 :: Integer
```

as an element in the set, that is $1 \in \mathbb{Z}$. The above declaration is essentially selecting an element of the set and storing it in the variable `x`.

In the category **Hask**, functions are the arrows (note that since functions are first class in Haskell, that is they can be treated as regular values, they are also objects, think the set of functions in the category **Set**).

4.2 Functors

Type constructors are the most typical example functors in Haskell. They are functions that take a type as an argument and return another type. Note that since we are mapping types to types, type constructors in **Hask** are *endofunctors.*

As we will see in the following examples, type constructors are usually parameterized by other types.

Functors belong to the `Functor` class, defined as follows:

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

This means they must implement the `fmap` function, discussed in the following example.

4.2.1 Maybe

Like in any other language, not all computations in Haskell are successful. Errors may occur based on certain inputs. Take the following function definition:

```
succChar :: Char -> Char
succChar 'a' = 'b'
succChar 'b' = 'c'
```

The purpose of `succChar` seems to be to take either `'a'` or `'b'` as an argument and return the successive character. It is a partial function on `Char`s. However, since the function signature indicates that it takes an element of the `Char` type, we may freely call `succChar 'c'`, which would lead to an exception.

To solve this problem, we introduce the `Maybe` functor. Maybe is a type constructor that maps a type `a` to a type `Maybe a` defined as

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

That is, values of the `Maybe a` type can either be `Nothing` (null) or values of the `a` type, represented by `Just a`. So let us improve `succChar` by using `Maybe`

```
succChar :: Char -> Maybe Char
succChar 'a' = Just 'b'
succChar 'b' = Just 'c'
succChar x = Nothing
```

Now, for all values that aren't `'a'` or `'b'`, `succChar` returns `Nothing`. Since `succChar` now returns `Maybe Char`, the compiler will enforce handling of the `Nothing` case by the caller. And just like that, no more exceptions!

So, we have seen how `Maybe` maps types, or objects in the category **Hask**. However, we know that to define a functor, we must also define its mapping between arrows. In Haskell, this is done by defining the `fmap` function. For `Maybe`, we can define `fmap` as the following:

```
fmap :: (a -> b) -> (Maybe a -> Maybe b)
fmap _ Nothing = Nothing
fmap f (Just x) = Just (f x)
```

In words, `fmap` takes as an argument function mapping some type `a` to some type `b` and returns a function `fmap f` that takes a `Maybe a` and returns a `Maybe b`. When the input is `Nothing`, `fmap f` returns `Nothing`. When the input is a `Just a` it simply applies `f` and rewraps it in `Just`. In functional programming, this is called *lifting* the function.

4.2.2 List

A `List a` is a type that represents a collection of elements of type `a`. It is defined as:

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

A `List a` can either be empty (`Nil`) or constructed (`Cons`) from an element of `a` and a `List a`. This recursive definition `List` types is quite common in functional programming languages. In this way, a list declared as

```
[0, 1, 2, 3, 4]
```

is internally represented by

```
Cons 0 (Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil))))
```

A way to define the mapping of arrows from types to collections of types is to have the new arrow be defined as an application of the old arrow to each individual element of the collection. So, for example, let F be an endofunctor on the category **Set** and let $A, B \in \mathbf{Set}$ such that for any element $A \in \mathbf{Set}$, $FA = \mathcal{P}(A)$ and let $f : A \to B$. Then, we can define $Ff = \{f(x) \in B, f(y) \in B, \ldots : x, y \in A\}$ for $\{x, y, \ldots\} \in FA$. To implement this, we can define `fmap` on the List functor as follows:

```
fmap :: (a -> b) -> (List a -> List b)
fmap _ Nil = Nil
fmap f (Cons x t) = Cons f(x) (fmap f t)
```

The above definition makes use of Haskell's pattern matching feature. The argument to `fmap f` is deconstructed into its parts, a `Cons` of the first first element of the list, `x :: a`, and the remainder of the list, `t :: List a`. `fmap f` essentially recursively unrolls the list, applying `f` to each element as it goes. When it reaches `Nil`, it will stop.

4.3 Universal Contructions in Hask

Undefined types in Haskell are *terminal objects*. For the following type definition

```
Type Terminal
```

the only member of this type is `undefined`, so any function with the signature

```
toUndef :: a -> Terminal
```

can only be defined as

```
toUndef _ = undefined
```

That is, there is only one arrow from any object (type) to `T`, and it is `toUndef`.

All types in Haskell contain the polymorphic member `undefined`, so it is impossible to construct a type with no members, so, in **Hask** there are no objects with a unique arrow to every other object. Thus, **Hask** does not have initial objects.

Products manifest themselves in the form of *product types* such as tuples. For example, take the following data type definition:

```
data Student = Student (String, Int)
```

The `Student` type represents a record for a student (perhaps from a database), the first field can be the name of the student as a string, the second field can be their identification number. This data type is the product `String` $\times$ `Int` with projection arrows `f : String x Int -> String` being

```
f (name, id) = name
```

and `g: String x Int` being

```
g (name, id) = id
```

In Haskell, tuples, the generic pair type, come equipped with these projection function. They are `fst` and `snd` respectively (in GHCI, the most popular implementation of Haskell).

5 Conclusion

I have demonstrated the construction of Haskell as a category. This ensures that programs using the builtin structures can be provably correct. Indeed, thinking category theoretically while programming forces a programmer to think abstractly about the structures that comprise their programs. These ideas are easily lifted from the creation of programming languages into the realm of programs themselves, and interfaces to programs, to take it one step further. We may embed a category within Haskell, and once we prove that all of its objects and arrows check the right boxes, we can ensure that our program will always be exactly as expected.

References

[1] Bartosz Milewski *Category Theory for Programmers.* http://bartoszmilewski.com, 2014.

[2] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* The MIT Press, Cambridge Massachusetts, 1991.

[3] David E. Rydeheard, Rod M. Bustall. *Computational Category Theory.* Prentice Hall, 1988.

[4] Ion Bucur, Aristide Deleanu. *Introduction to the Theory of Categories and Functors.* John Wiley and Sons Ltd., 1968.

[5] Miran Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide.* No Starch Press, San Francisco, California, 2011.

[6] R. F. C. Walters. *Categories and Computer Science.* Cambridge University Press, Cambridge, 1991.