Category Theory in Haskell

Kenneth Watkins

June 5, 2022

1 Introduction

In the Typescript world we currently don't have a clear source for how Typescript and its ecosystem surrounding functional programming relate to the categorical concepts. The current efforts in the community have been largely been comprised of limiited documentation surrounding functional programming libraries and in partial blog post. This leads to a large barrier of entry for developers coming into the ecosystem either new or from another functional language. By Using a subset of Category Theory concepts as an abstract model we can create a generalized model that can be implemented into typescript.

For an abstract model to be useful it would have to be communicable accross programming languages of different types meaning that concepts from the model would need to be recognizable in multiple languages modern functional languages. For the Typescript implementation to be practical inside typescripts type system it would need to be able to use existing typescript ecosystem constructs where possible, be low boilerplate, and performant in both the front and backend ecosystems where it is used.

This document assumes the reader is a beginner with Haskell and Typescript. And presumes no mathematical knowledge outside of basic logic, algebra, and functions.

1.1 Goals

This document is broken into several parts

The first part seeks to first define an abstract model that a practical implementation can be based on. We choose to avoid typescript entirely in

this section and turn to Haskell as the language of choice due to the existing documentation and libraries that define category theory concepts already.

The second part seeks to define an implementation and identify the caveats between the typescript representation and the Haskell representation of the model

The third part is to be determined but its focus will be on communication between typescript and haskell, translating domain types to the model, and migration schemes for existing representations of concepts.

The forth part seeks to define existing functional architectures in terms of the abstract model as well as provide basic example implementations and possible improvements

2 Part 1 - Category Theory

A Category consist of two things

- Objects
- Arrows - Morphisms

Category Theory serves as an interpretation for the foundation of mathematics much like Set Theory and Type Theory. The key focus is on the the composition of mathematical stuctures. These structures are called objects. Again the focus is on the composition of the objects which is captured with the notion of an arrow (formally a morphism) that points to objects.

So a Category is made up of objects and arrows (morphism) for which the follow properties hold true.

- Objects must have a an arrow originating from itself to itself as the Identity morphism. This arrow serves as a unit of composition such that when composed with any Arrow that either starts at A or ends at A it gives back the same arrow. So if f is an arrow f . (idA) = f
- Given an object A, B, C and two arrows, one from A to B and B to C there must exist a third arrow from A to C.
- Given 3 arrows f, g, h, the they must be associative h . (g . f) = (h . g) . f = h . g . f

3 Haskell Types and the Category of Set

A type is a set of values. As an example the Haskell type Bool is a two element set of value True and False. Sets can be finite or infinite. x :: Integer is saying x is an element of Integer. The category of sets is called Set. Its special because we can peak in at its objects.

In the category of set objects and morphisms are defined as...

- Objects are sets
- functions are morphisms

Intutions we can gain because we can peek into objects in the category of Set

- empty set has no elements
- there are special one element sets
- functions map elements of one set to elements of another set
- functions can map two elements to one but not one element to two
- there exist an identity function that maps each element of a set to itself

In Haskell the category of haskell types and functions is referred to as Hask. By forgetting the bottom hask is the Category Set

Because of currying functions with multiple type arguments have two interpretations as well. a -¿ a -¿ a can be interpreted as a function that takes multiple arguments and the last value is the return type. or it can be interpreted as a -¿ (a -¿ a) function that takes an argument and returns a function requring another argument.

Set	Haskell Type	Typescript Type	Description
Empty Set	Void	never	Type not inhibited by any values
Singleton Set	() “Unit”	void	Type has one value that always exist
Two Element Set	Bool	boolean	true or false functions to this type are called predicates

3.1 Examples of Categories

Category	Objects	Morphisms	Description
Empty	None	None	A category without objects or morphisms
Set	Sets	Functions	A category in which we can peek into the objects
Graph (Free Category)	Nodes that represent objects	all possible chains of composable edges	A category for Free construction
Preorder (Thin Category)	objects	relationship between objects where objects are less than or equal	only one morphism between two objects
Partial Order	objects	relationship between objects where objects are less than or equal and that if a ¡= b and b ¡= a then a must be the same as b	only one morphism between two objects
Linear/Total Order	objects	relationship between objects where objects are less than or equal and that if a ¡= b and b ¡= a then a must be the same as b. Finally it states any two objects are in relation to one another	only one morphism between two objects

Free Construction is the process of completing a given structure by extending it with the minimum number of items to satisfy its laws. Graphs are the free constructor for the Free category. chains of length zero serve as the identity morphism in the free category.

Thin categories are those where there is at most one morphism going from any object to any other object.

3.2 Hom-Set

Hom-set is the set of morphisms from object a to object b in the category of C written as C(a,b) or HomC(a,b)

Every hom-set is either empty, a singleton, or a preorder.

Preorders can have cycles where as partial orders cannot

Sorting algorithms like quicksort, bubble sort, merge sort, only work on total orders. Parital orders use topological sorts

3.3 Haskell Typeclasses as Subcategories

In the category of C a subcategory of C is a subcollection of the objects of C and a subcollection of the morphism of C which map only between objects in the subcollection of the subcategory

Subcategories can be represented as typeclasses in haskell and have two interpretations depending on the amount of type parameters

- A single parameter typeclass can be interpreted as describing the members of a set of types
- Multi-parameter typeclasses is interpreted as a relation on types

To defined a collection of types that are members of a particular class A type signature with a universally quantified type variable constrained by the typeclass type class can be used

Example type signatures and their interpretation

- Eq a =¿ a; the collections of types that are members of the class Eq. Members of Eq are a subcollection of objects in the Category of Hask
- (Eq a, Eq b) =¿ (a -¿ b) is a subcollection of the morphisms in Hask mapping between objects in the subcollection of objects defined in EQ

3.4 Monoids

Monoids are a mathematical concept found across different branches of mathematics.

In set theory a monoid is a set equiped with a binary function that is associative and a unit element. An example is addition on the set of integers is equiped with the pesudo function (+) :: a -¿ a -¿ a.

Monoid M = a set with a unit e and binary operator

a,b,c elements of M a . b element M (a . b) . c = a . (b . c) e . a = a . e = a

In category theory a monoid is a one object category with a set of morphisms that follow the rules of composition.

An example. Given a single object category C, morphism in this Category can only originate from and to the object.

C = object is c C = morphism c -¿ c

identity = 1c : c -¿ c f . g Elements of Morph(C) (f . g) . h = f . (g . h) 1c . f = f . 1c = f composition = f g c

This definition is to the above set definition are informally equiveleant the important factor is we're no longer talking about functions but morphism.

We can always extract a set from a single object category. The set extraced is the set of morphism.

Category M single object category single object m is element of M hom-set is M(m,m) binary operator is the monoidal product of of two set-elements is the element corresponding to the composition of the corresponding morphisms.

In the Hask (Set) we can define a subcategory definition for the Monoid using a typeclass.

class Monoid m where mempty :: m mappend :: m -¿ m -¿ m