ROSETTA PAPER 1: HOW CATEGORIES ARISE NATURALLY

D. E. STEVENSON

1. The Rosetta Project

The goal of the Rosetta Project is to develop a series of papers that act as Rosetta Stones for undergraduate students in understanding theoretical mathematics and theoretical computer science. The Rosetta papers are not research treatises; they are meant to promote self-study in advanced subjects. These papers are language intensive because the study of mathematics is language intensive. In fulfilling the Rosetta purpose, every attempt will be made to explain terminology especially when that terminology is not usually defined.

I assume that the reader has had a discrete mathematics course in which set theory was studied. In light of current curricula, I also assume introductory calculus. What will become apparent is that calculus plays almost no role in these papers. Computation is essentially an algebraic exercise.

The audience is computer scientists. Therefore, I will take a **constructive** viewpoint on definitions and proofs. This means that I am approaching any subject with the mindset of a programmer. The goal is to understand what the subject says about the two notions of programming: representation and algorithm.

The writing style for these papers is that of "just in time". That is, ideas are introduced as they naturally occur. Therefore, these papers are explanatory and not meant to be a research treatise — there are an ample number of such texts. The final section of the paper will be a concept map that properly describes the relationships among the terms. Words in **bold face** are concept words: these are words that play key roles in using knowledge of computation.

The Story. The *lingua franca* of mathematics and computer science is set theory. There are other ways to develop the material. Our goal is to use **category theory** to develop the original ideas. This development is aimed at students (and faculty) who understand the old way but want to understand and use the new.

We are interested in several intuitive **notions**. Notions in mathematics are not formally defined but instead convey a general concept of a desired feature. For example, the notion of proof is realized in many different ways. In fact, *realization* is itself a notion.

The story line for this paper is the development of the **vocabulary** of sets, categories, and computation. I retrace a standard development of terminology but concentrating on the where and why, not the how. I introduce an algebraic viewpoint through constructive concepts and by introducing category theory.

2. Sets Themselves

I will treat **sets** in a **naive** fashion; that is, I will address sets from a non-axiomatic view. A *set* is a collection of **elements**. Each set has a **membership** procedure that decides where or not an object is an element of the set. Following Errett Bishop, each set has an **equality** procedure.

Aside on definitions. Notice that I have a problem already because I need *procedures* to define *set* and I need *set* to define *procedure*. The general requirement for definitions is that they should form a tree. The way out of this is to separate definitions into to categories: theoretical and pre-theoretical. The term *procedure* is a pre-theoretical term. Every computer scientist has her/his own idea of what procedures should be — thus, *procedure* is a notion. □

3. Functions

Functions and relations describe relationships between elements of sets. In mathematical practice, functions are defined as sets of pairs while in constructive mathematical practice, functions are defined as procedures. Regardless of this difference, the three elements of any function definition is the **domain**, the **codomain**, and the **rule**. Normal mathematical practice denotes a function by $f : X \to Y$, where X is a set and is called the *domain*; Y is a set and called the *co-domain*; and f is the function being defined.

Aside on types. In computer science we would say that f has type $X \to Y$. The same concepts of type checking used in programming are applicable here: expressions must be correctly typed. □

There are many different ways in which **application** is denoted in the literature. Application is the term that means that we actually take an element from the domain and determine its co-domain point by "magically" using the function definition. While f is a function by itself (either a rule or a table or ...) application is denoted $f(x)$ or fx. The statement $fx = y$ means "if the function f is applied to the element x of the domain the result is y in the codomain." The same statement can also be written $x \vdash_f y$ or $xto^f y$. A clear intent of this notation is that x is a single member of the domain and y is a single member of the co-domain.

If $fXtoY$, then the **inverse** function would take Y to X: $f^{-1} : Y \to X$. Immediately, several problems arise.

3.1. Inverse Functions.

It is easy to describe the goal of the inverse: if f is a function then the inverse, denoted f^{-1} function "undoes" the effect of f. Suppose $X = \{1, 2\}$ and $Y = \{0\}$. Let $fXtoY$ map both 1 and 2 to 0. Then f^{-1} is not a function because 0 has *two* images. As a second example of an undesirable situation, let X be the same as above and $Z = \{a, b\}$. Now let $g : X \to Z$ with $g(1) = g(2) = a$. Now g^{-1} has another problem: there is no defined image for $g^{-1}(b)$. We are now faced with the issue of describing when a function has an inverse. To this end, several new terms are introduced: range, partial functions, total functions, injective, surjective, and bijective.

Finally, there is the identity function denoted 1_X or just 1 if the set is clear. The identity function maps every element to itself as in $1(x) = x$.

3.2. Composition.

Let $f : X \to Y$ and $g : Y \to Z$ where X, Y and Z are any sets.

Aside on metavariables. Normal mathematical notation can be confusing due to a large number of **conventions** that have come about through normal practice. The above statement is an example of confusion over object language and metalanguage. X, Y and Z are not specific sets but rather denote any possible sets; in fact, the notation does not preclude the three sets from being the same. *Metalanguage* is the language used to talk *about* the *object language*. In our case, English is the metalanguage and sets are the object language. X is called a *metavariable* because it takes its values from the object language. □

So, let's start over. Let $f : X \to Y$ and $g : Y \to Z$ where X, Y and Z are any sets. The **composition** of f and g is again a function that takes elements of X to elementsof Z. The new function is denoted in two ways: $f; g$ or $g \circ f$. The first notation should make sense to computer scientists as it might be read "first apply f; then apply g". The more traditional form of $g \circ f$ is shorthand for $g(f(x))$. In some text, the latter form might be further abbreviated gf.

Discussion Aside on Functions and Composition. This theme of sets and functions recurs through out mathematics and computer science. I have a set of things that must be operated on. What are those things? How are they represented? What are the most fundamental things I need to worry about defining? The purpose of theory is to investigate these questions. □

4. Relations

Why do we restrict the concept of function to be one acting on a single element? There is not justifiable reason to not consider, for example, sets to sets. This line of thought represents how theoreticians think. One exercise is to look at the body of knowledge and find and remove unjustified restrictions. In the present case of functions, removing the restrictions requires adding two new concepts: the subset concept and the powerset concept.

If X is a set, then any set consisting of some, not necessarily all, elements of X is called a **subset** of X. Since we have required sets to have membership *functions* and *equality* functions, what must I do to define a subset? I must develop the two functions (membership, equality) that are usable for the subset. Here the approach is exactly the one programmers would adopt. What is the type of the membership function? In programming terminology it would be a function from the elements to "yes,no", the boolean set. Let's agree that the set for "yes, no" or "0,1" is denoted $Bool$. Now, the membership function is of type $X \to Bool$. If X provides the context (the "superset") for the subset function, then the subset function is a **restriction** of X's membership function.

On the other hand, the equality function is a function of two arguments with a co-domain of $Bool$. This is denoted $X \to X \to Bool$. This may seem strange to many programmers, but this notation is common in functional programming languages like `ML` and `Ocaml`. We will adopt it here. The subset equality function is also a restriction of X's equality function.

Aside on the word "sub". The use of *sub* as a prefix as in *subset* generally means a process similar to what we have done here. In "subbing", a new structure is formed by removing elements of a larger structure and maintaining the integrity of the other elements. □

So the power set comes from the idea of subsets: the powerset being defined as the set of all subsets, including the empty set $\emptyset$. Let us denote the power set by $\hat{X}$. Then a **relation** r between sets X and Y is just a mapping between subsets. In symbols, $r : \hat{X} \to \hat{Y}$.

Aside on mapping. The term mapping is meant to be a pre-theoretical notion. Mappings include functions and relations. □

We defined subsets and relations in order to generalize the concept of functions. Having done that the natural process is to now return to functions and ask what operations are there (composition and inverse) and how should these operations be extended. In the case of relations, the extension is natural.

5. Review of the Boolean Operations

The standard boolean operations on sets are union ($\cup$), intersection($\cap$), cartesian product ($\times$), complementation ( ) and subset ($\subset$). To understand the current literature in category theory and theoretical computer science, we need to add two more operations: disjoint union and exponentiation.

Disjoint union is simply a union, but the two sets are disjoint. Recall that two sets are disjoint if the do not have any common elements. The disjoint union of X and Y is denoted $X + Y$.

6. Category Theory — First Look

Set theory arose from the realization that when we think, we may start with concrete objects but then we want to think about *all* chairs, or *all* unicorns. Mathematics and computer science evolve. As we solve one problem, others appear. If one wants to look at the history of the development of ideas in the formal sciences (mathematics and computer science), we find that ideas are discovered and explored primarily as the result of some sort of problem in the current state of affairs. This observation is not uniquely mine; Thomas Kuhn and Karl Popper made a similar observations about science.

In practice, sets are not as interesting as the concept of certain sets coupled with certain operations. For example, sets and the boolean operators; functions and composition; data structures

and algorithms; etc. One way to justify **category theory** is that it mimics practice. That is, categories are about **objects** and their changes by **morphisms**. Since morphisms are a type of function, we are naturally interested in composition.

Before I develop the formal ideas of a category, it is important to look at *why* we will do what we will do.

7. Strings

The word *structure* in the mathematical context usually means a set that has operations and relations associated with it plush axioms that describe how the system is supposed to work. `class`es in object oriented systems are similar, but lack the logical framework. That framework is presented by the programmer but it is not explicated in the code (a huge failing on the part of computer science, in my opinion). One structure that is universally known by both mathematics and computer science is the structure surrounding *strings*. In mathematics, strings are an example of a monoid.

Suppose I have a type called `string`. What can I say about the structure? What can I say about the logical rules surrounding strings?

7.1. **Alphabets And All That.** In programming, the first exposure to strings is through ASCII strings. Later on, we learn strings can come from other sources. We would naturally call the set of ASCII characters an *alphabet* (a set). The elements of the set we call *characters*, but they're still elements of a set. What is a string? Actually, here we get into a bit of type trouble. A string is a sequence of characters. In programming, we treat sequences as vectors. For our purposes, it is best not to think of strings at this level but only at the sequence level. Notationally, if s is a string then the individual characters can be named

$$s = c_1 c_2 \dots c_n$$

where each c_i is a character in the alphabet. We will also adopt the convention that the subscript indicate position in the string.

I purposely did not start the count at zero so that I can discuss the *empty string*. Can a string have zero characters? Yes, as every programmer knows. The **convention** is that we start counting from one so as to avoid ambiguity. But we need to somehow denote the fact that the string is empty. I will adopt the convention (there are many such) that Λ is the empty string.

One important property of ASCII strings is that they can be sorted. The relation used to sort strings is a **partial ordering**. It turns out that the partial ordering is the lynch pin of computation.

7.2. **Length as the First Morphism.** The next question is, Is a one character string the same as one character? Again, as every programmer knows, they are not. We already have a good notation for indicating type, the colon. We will adopt it in this discussion so that `s : string` is read "s is of type *string*".

A natural operation on strings is to find its length. The type of this is $length : string \to \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers $\{0, 1, \dots\}$. From programming we can guess that $length(\Lambda)$ is zero.

Functions like *length* have a special name: morphisms. The word *morph* means "shape" or "body"; in mathematics it means *structure*. *length* is a function that transforms the string into the number while preserving a property: the partial order is maintained. If $s, t : strings$ and $s \leq t$ then $length(s) \leq length(t)$ (using the usual definition of $\leq$).

7.3. **Concatenation.** Now, what operations come to mind for strings? Concatenation. If we do not worry about string length, we can define concatenation by the following. Let $s, t : \mathit{string}$ (both s aned t are strings) with $s = s_1 \dots s_m$ and $t = t_1 \dots t_n$. Then st is a string with $st = s_1 \dots s_m t_1 \dots t_n$.

Concatenation obeys three axioms (laws):

Closure If $r, s : \mathit{strings}$ then rs is a string.
Associativity If $r, s, t : \mathit{strings}$, then $r(st) = (rs)t$.
Unit For any $s : \mathit{string}$, $\Lambda s = s = s\Lambda$.

Every string can be broken down into three parts, $r = stu$, even though one or more might be empty and the decomposition is not unique. Obvious definitions concerning *segment*s come about from this. A *segment* is any consequetive letters in a string so r, s, t and u are segments. In this example s is an *initial* segment and u is a *terminal* segment.

Aside on definitions. When initially studying a subject, it is difficult to tell exactly what is important and what is not. It turns out that strings will play a big role in introducing categories and the issues of initiality and finality are illustrated by these definitions. □

It turns out that this structure is very common and has a special name of **monoid**. In other words, strings are a realization of the structures known as monoids. Therefore, studying monoids tells us much about many structures, including strings.

7.4. **Free Monoid.** Let A be the ASCII character set. What are all the strings that I can build from A? I would start with the empty string Λ then add the set of strings of one-character long, then the set of two-character strings, etc. How long would this go on? I don't really know. But at any point in time, I would know what length I was on. If I can build such sets and the process does not terminate, then we call the length **countable infinite**. On the other hand, there are sets much bigger than this set of all strings: the number of real numbers in the interval $[0, 1]$ is an example.

We can symbolize the process by writing

$$A^* = \Lambda \cup A \cup A^2 \cup A^3 \cup \dots,$$

where $A^{n+1} = A^n A$. The elements of A^n are n-sequences of elements from A. If we adopt the convention $A^0 = \Lambda$, then we can compactly write

$$\begin{aligned} A^* &= A^0 \cup A^1 \cup A^2 \cup A^3 \cup \dots, \\ &= \bigcup_{i=0}^{\infty} A^i \end{aligned}$$

Such a construction generates a **free** structure from the base. So, in this exact case, A generates all strings and these strings have a monoidal structure. Therefore, A^* is the **free monoid** of strings generated by the base A.

Aside on the term free. The term *free* refers to the fact that there are no rules about the formation of the strings. Not all strings are free: the English words derivable from the Roman alphabet are decidedly rule bound — otherwise, we would not be able to write spell checkers. □

Discussion Aside on strings. Strings represent a system of values that every programmer is familar with. In the next section on monoids, it is important that the reader see that everything stated there is something that can be programmed about strings. □

8. Monoids

We know what a monoid is from strings, but let's collect the formal definitions. Let M be a set with one binary operator, say $\times$. The system $(M, \times)$ is a monoid if the system is closed, associative, and has a unit. In an abuse of language, we conventionally identity $(M, \times)$ by just the set M.

8.1. **Submonoids.** If M is a monoid and $A \subseteq M$ then A is a *submonoid* if the system $(A, \times)$ obeys the monoidal axioms when the $\times$ function is restricted to the elements of A.

Consider again the free monoid A^* with base A. Fact (I do not provide a proof at this point) is, A^* is a submonoid of M and A^* is the least such submonoid of M that contains A. In other words, A^* is a *closure* of A. Closures represent the concept of **convergence** or **limit**.

The basic property that closures enjoy is the following. Let $f : \Sigma \to M$ and M be a monoid. Then there is a unique morphic extension for $\Sigma \to M$. This is the fundamental idea explored in the theory of computing machines like Turing machines.

8.2. **Examples.** In the section on functions, we defined functions and relations. The theoritician first response to these definitions is, How are these related?

It would appear that the set of all relations must be the most general collection of "maps." From this, it is clear that partial functions are less general because the map individuals to individuals but not necessearily does every co-domain point serve as an image. Total functions do require that everything have an image, so they are less general than partial functions. Yet more specific are functions that have inverses.

Since the set of all relations is a monoid, the various categories are submonoids.

9. Semigroups

The last algebraic structure to mention is the **semigroup**. Semigroups are sets with one binary operator but they do not require a unit.

10. Partitions and Equivalences

A relation $r : X \to X$ is called an **equivalence relation** if it satisfies three conditions. For most readers, you probably learned these conditions as follows. Write $x_1\ x_2$ to stand for "x_1 is related to x_2." Then the three rules are given as follows:

Reflexivity	For all x, $x\ x$.
Symmetry	For all x_1, x_2, if $x_1\ x_2$ then $x_2\ x_1$.
Transitivity	For all x_1, x_2, x_3, if $x_1\ x_2$ and $x_2\ x_3$ then $x_1\ x_3$.

The original idea is that of standard equality on numbers.

In our world, we want to be more focused on functions and functional representation, so we want to rewrite these rules. Let $r : X \to X$. The three rules above translate as follows:

Reflexivity	$1_X \subseteq r$.
Symmetry	$r^{-1} \subseteq r$
Transitivity	$rr \subseteq r$

It takes a bit of work to see that the second set of rules is the same as the first. Reflexivity is easy since $1(x) = x$. Symmetry takes a bit of thinking but if $(x_1, x_2) \in r$ then (x_2, x_1) is the inverse. rr is a simple metaphor for transitivity. One more translation is necessary: *implication* in the logical world is *subset* in the functional world.

Equivalence relations have the effect of defining **partitions** on the set by simply placing two elements in the same partition if they are equivalent. The partitions are disjoint. Just as important, there are just enough partitions to **cover** the set. That is, the union of all partitions X under the relation r is the set X.

The relationship between X and r when r is an equivalence relation is denoted X/r and is called the quotient of X mod r. The functional eqivalent is called the *natural factorization mapping*, such as $\pi : X \to X/r$, which sends the element $x \in X$ to its partition.

11. Kernels and Factorization

Consider our old friend $f : X \to Y$, a function from X to Y. Now let's consider the inverse f^{-1}. The inverse sends elements of the co-domain Y to *subsets* of the domain X. Take all the non-empty subsets. Define the set of these non-empty subset as the *kernel* of f and denote this as $ker\ f$.

We say that $ker\ f$ *factorizes* the function f in the following way. One can consider that f has two parts: one part sends inputs to the partition and then the entire partition is given its final value by a second function g. In other words $f = g \circ \pi$. In pictures

$$X \xrightarrow{\pi} X/ker\ f \xrightarrow{g} Y$$

This trick is often used to reduce the number of elements needed in memory.

12. Congruences

How do equivalences play out in monoids? Let M be a monoid. r is said to be a **congruence relation** in M if

$$r(m_1) \times r(m_2) \subseteq r(m_1 m_2)$$

for all $m_1,\ m_2 \in M$. This condition guarantees that the equivalence relation works out.

This fact can be used to define how $\times$ works in X/r. Suppose $f : M \to M'$ is a morphism of monoids; similarly, with $\pi : M \to ker\ f$. We can reuse the relation above to

$$M \xrightarrow{\pi} M/ker\ f \xrightarrow{g} M'$$

$ker\ f$ is a congruence.

13. Categories

Aside on on definition of category. In true abstractive tradition, I'm going to define a category as an *entity* with two *projections*: If $\mathcal{A}$ is a category then $\mathsf{obj}\mathcal{A}$ is the class of objects and $\mathsf{arr}\mathcal{A}$ is the set of morphisms. This makes the *internal structure* completely invisible. □

We have seen enough to motivate the introduction of categories. I now follow the more-or-less standard definition of categories. A *category* $CatA$ is a structure that has two type of data:

(1) Objects. A class called $\mathsf{obj}\mathcal{A}$, the elements of which are called objects.
(2) Arrows or Morphisms. A function which to everyh ordered pair (A, B) of objects of the category assignes a set $\mathcal{A}(A, B)$; the elements $f \in \mathcal{A}(A, B)$ are called *arrows* or *morophisms* from A to B.
(3) Composition among arrows.
(4) Associativity among arrows.
(5) For every pair of objects in $\mathcal{A}$, there exists an arrow

$$1_A : A \to A$$

Similarly, there can be *subcategories*. *Full subcategories* are obtained from the parent category by restricting the objects.

The definition of conditions for a category includes the work *class*. For most students, *class* is probably synonymous with *set*. The term has a techical meaning in Von Neuman-Gödel-Bernays set theory. A class can be much bigger than a set; it can be larger than the set of all sets. Therefore, terminology is introduced for *small categories*, which have sets for their objects. *Large categories* are classes.

As an example of categories, we provide the following:

- $\mathcal{S}et$.
- The usual litany with explanation.

Categories can be based on most mathematical concepts. For example, We could call $\mathcal{R}el$ the category of relations on sets. Here the objects are sets and the arrows are all the relations $r : X \to Y$ and the composition as the usual composition on relations. We can now see that $\mathcal{T}Fun$, the class of total functions, is a subcategory of $\mathcal{R}el$, but not a full subcategory.

442 R. C. Edwards Hall Department of Computer Science, Clemson University, PO Box 341906, Clemson, SC 29634-1906

E-mail address: steve@cs.clemson.edu