Rigorous Mathematics Language with Graphs

Alec Martin

May 12, 2020

1 Introduction

This paper introduces a language in which we can make rigorous mathematical statements by constructing particular graphs. The language shares the title of this paper, which we shorten to GRML. The goal of GRML is to create a concrete, standardized format for making mathematical statements. Doing this allows for, amonsg other things, the creation of computer applications which can understand, interpret, and manipulate mathematical statements directly by working with the graphs used to make said statements.

This is certainly not the first attempt to put math on computers. However, one aspect of GRML is the attempt to match as closely as possible the standard mathematical language already used by the general mathematics popluation. Existing mathematicians should be able to interpret between the graph and the intended statement with minimal esoteric knowledge of GRML itself or of formal languages in general.

1.1 A Note on Computer Languages

The general idea of a computer is to process information. Without getting into specific applications, a typical use of a computer, at any level, involves taking some information and processing it into different information. We would like to delineate this into three distinct steps and we use a calculator as a model.

First we have to start with some information. That is, we have some concept in our heads which we wish to work with. The first thing we have to do to get a computer involved is to encode this concept into a physical form so that it can be manipulated. On a calculator this consists of pushing the buttons.

Once the information exists in a physical form, we need to process it physically. This is the second step and it is what the computer does. The computer has some processing instructions and it follows them to manipulate the physical version of our information. On a calculator this is what the calculator does internally after we hit the buttons.

After the processing is done we are left with some physical data. We can now physically observe the data and interpret it into a concept which we understand. This is the third step. On a calculator this is when we look at the screen.

GRML is a language designed to make mathematical statements. The physical medium it uses is colored graphs which we describe in detail shortly. The three step process described above, as it applies to GRML, is this: we start with some mathematical statement. First we make a graph which encodes this mathematical statement, using the structures described in this paper. The second step is to compile this graph using the algorithms described later on in this paper. Third we interpret the graph as a mathematical statement, which is just the first step but in the other direction.

When talking about pure GRML, we do not want to change our graphs into new graphs. In this sense GRML is a passive language – the statements we make are only meant to be compiled, not executed. The only outcome of the three step process is that we know our statement succesfully compiles and therefore makes sense. GRML includes excplicitly what it means for a mathematical statement to make sense and what the statement states. GRML is intended to be a medium through which other entities, whether they be intelligent beings or computer applications, can communicate abstract mathematics.

2 The Graphs We Use

2.1 Terminology

In this paper we will be working with colored directed acyclic graphs. Essentially every concept we create will be encoded as some particular structure imposed on directed acyclic graphs with various degrees of coloring.

Our graphs consists of finitely many vertices called ***members*** and finitely many edges. By colors we mean identifiers such that any two colored objects either have the same color or not. In each graph we require that each member has a color which we call the ***type*** of the member and another color which we call the ***id*** of the member. No two members of a graph can share the same id. Each edge has a color as well, we call this the ***name*** of the edge. Each graph itself has a color which we call the ***UI*** (universal identifier) of the graph and we insist that no two different graphs can share the same UI. We universally fix one color which we call ***no color***. If a member is colored with no color then we say it has no type and if an edge has no color then we say it has no name. We insist that no graph can be given the UI no color.

We are not always interested in how the id of each member or the UI of each graph is assigned. The purpose of requiring them is so that we have a tangible datum which communicates information between different parts of different graphs. If we omit description of the ids of members, we implicitly assume that they have been assigned so that each member has a unique id. Likewise with UIs of graphs. Any time this information is relevant we will describe how so.

When we say $a \in G$ we mean that a is a member of G. The edges are oriented so that each edge has a ***parent*** member and a ***child*** member. We denote the parent memberf of edge N by $p(N)$ and the child member by $c(N)$.

For $a, b \in G$ if there is an edge N with $p(N) = a$ and $c(N) = b$ then we say a is a ***parent*** of b and that b is a ***child*** of a. We often insist that no two edges can have the same name and the same parent member in any graph. This ensures that parents can distinguish their children by name in graphs without nameless edges. Sometimes we will not require this feature, but unless explicitly stated we do require it.

We introduce an order called ***ancestry*** inductively on G as follows: for all $a, b \in G$, a is an ancestor of b if a is a parent of b or if a is a parent of an ancestor of b. If a is an ancestor of b we say b is a ***descendant*** of a. We insist acyclicity in that no member can be its own ancestor.

An ***elder*** is a member of the graph G which has no parents. Every graph has at least one elder. An ***eldest*** is a member e such that all other members of G are descendants of e. An eldest member, if it exists, is unique.

We are especially interested in graphs which have an eldest member. Any graph can be modified to contain an eldest member by inserting a member e and for each elder $g \in G$ inserting an edge N_g with parent e and child g. If a graph has an eldest member e then we call the children of e ***maximal***.

A member $g \in G$ has ***only one parent*** if there is precisely one edge N whose child member is g. A graph is called a ***tree*** if every member has either no parents or only one parent and if there is an eldest member.

One graph which we are particularly interested in consists of exactly one member and no edges. We call this the ***singleton*** graph. Any two singleton graphs are distinguishable by their UIs.

2.2 Graph Homomorphisms

Suppose G, G' are graphs and $f : G \to G'$ is a function. That is, f is a function from the members of G to those of G' and from the edges of G to those of G'. We call f a ***graph homomorphism*** if the following hold for all members $a, b \in G$ and all edges N in G with $p(N) = a, c(N) = b$:

- $p(f(N)) = f(a)$ and $c(f(N)) = f(b)$
- $f(a)$ has the same type as a
- $f(b)$ has the same type as b
- $f(N)$ has the same name as N

If G, G' are graphs, $f : G \to G'$ is a graph homomorphism, and $g \in G$, we say $f(g)$ is ***playing the role*** of g in G'.

3 Typed Graphs

This section is devoted to developing enough of the language of graphs to allow for mathematical grammar. A grammatically correct statement would be $x = y$ where x, y are integers. A grammatically incorrect statement would be $x = y$

where x is an integer and y is a topological space, or that $x =$ where x is an integer and we omit stating what x is supposed to be equal to.

The way to make statements as graphs is through the notion of typed graphs. We will define what a typed graph is, but the definition is recursive and we need some background structure first: we define *dictionary* then we go on to define *typed graph*. For a starting point, let G be a singleton graph with UI T and whose only member has type T. As we will see, such G are typed graphs. We call these ***genesis*** graphs.

A ***dictionary*** is a particular kind of graph. Terminology may change when referring to dictionaries, we may call the members ***definitions*** and the edges ***dependencies***, though we retain the member/edge terminology for now. Each member of a dictionary must be typed with the same color as the UI of some typed graph. If G is a typed graph with UI T and D is a dictionary with a member of type T then we say D contains the ***definition*** of G. If D contains the definition of G we may just say that D contains G or that D contains T.

Part of being a typed graph includes a notion of dependency. Specifically, if G is a typed graph then there is a well-defined finite set $\{T_i\}$ where each T_i is the UI of a typed graph upon which G depends. We say G is ***independent*** if this set is empty. As we will see, genesis graphs are independent.

If a dictionary D contains a member g of type G, meaning a member whose type is the same as the UI of the typed graph G, then for each color S upon which G depends there must be a member $g_S \in D$ of type S and an edge N_S whose child member is g_S and whose parent member is g. We call this requirement that dictionaries must be ***self-contained***. The condition that D is acyclic prevents us from making circular definitions. At this point we are not concerned with the names of the edges in a dictionary.

Suppose G is a graph, g is a non-eldest member in G, and the type of g is the UI of a typed graph $T \neq G$. As we will see, all typed graphs have an eldest member. We say g ***matches its type*** if there is a graph homomorphism from T to G such that g plays the role of the eldest member of T.

We are now ready to define typed graph. Suppose G is a graph. We say G is a ***typed graph*** if all of the following hold:

- G has an eldest member e
- The type of e is the UI of G
- Any child of e has only one parent, namely e
- Let X be all members of G which are not e. For $x \in X$ let T_x be the type of x. For all $x \in X$ T_x is not the UI of G
- For all $x \in X$ T_x is the UI of a typed graph
- For all $x \in X$ x matches its type
- There is a dictionary D_G such that for all $x \in X$ D_G contains a member of type T_x

We call the set $\{T_x : x \in X\}$ the set of types upon which G ***depends***.

3.1 Using Typed Graphs

Typed graphs are the mechanism for making mathematical statements. What we have seen so far has given us the power to declare any object definition. An ***object definition*** is context required to make a mathematical statement. Object definitions are different than property definitions. Observe the standard definition of a function,

> A ***function*** $f : A \to B$ where A and B are sets is a relation such that for all $a \in A$ there is a unique $b \in B$ with $f(a) = b$.

Everything before the term "such that" is the object part of this definition. That is, the function f needs the sets A and B and the relation f as context before f can be said to be a function.

We say that the second half, after the "such that," is not object definition but behavior definition. Behavior is where mathematics gets interesting, objects are just for context.

When a typed graph G encodes a statement we often refer to the members of G as ***terms***. Specifically we refer to non-eldest members of G as terms. The eldest member is only used to turn our statement into a tangible GRML object, it does not in general interact with the content of the statement. The meaning of a statement is encoded in the terms, the statement itself is embodied in the eldest member.

Let's see how to use object definitions. Suppose S is a typed graph. Then we think of S as defining an object type through the contents of the graph S. If we are then constructing another statement L, we can add a term s to L of type S. We must perform a test now to check that s is valid, namely s must match its definition. If this test passes then we consider s to be an instance in L of the object type S. Type matching ensures grammatical consistency.

4 Encoding Behavior

We would like to be able to make statements which encode mathematical behavior. The typical way we add functionality to our language is by adding more layers of coloring to the components of our graphs or by requiring some particular extra piece of structure. This is intended to be universal and retroactive, so we commonly introduce a default value which we apply to everything covered so far.

The general process will be that each specific ability we introduce will be associated to a ***statement type***. Formally the *statement type* is a color we assign to each of our typed graphs. The default value is ***definition***, and a statement declared as a *definition* is interpreted the way described in section 3.1. We think of the material presented there as passive because it only allows us to make grammatically correct statements. We have not yet seen how to actively say anything with a statement, we can only build context.

The general idea is that if we want to describe some active behavior, we encode the action in a typed graph T and give T the statement type which corresponds to the type of action we want to perform. Then, if we are building a statement S and we want to refer to the behavior encoded in T, we construct a term $t \in S$ of type T. We look at the statement type of T and follow whatever algorithm was described when that statement type was introduced. This algorithm may include an extra verification test, and if this test passes then the algorithm performs in S the action we originally wanted to take. The action is then succesfully executed in S and if we need to refer to the execution of this action then we refer directly to the term $t \in S$.

The only action we are capable of at this point is defining grammatical rules and following them. This is because the only statement type we have seen so far is *definition*. We want to consider new statement types as extensions of previously understood statement types, with *definition* being the root of all of them. This gives us the ability to recover at least some of the intended meaning even if we encounter an unfamiliar statement type. It also means that evey statement must be grammatically correct, even if the focus is on more subtle behavior, because each statement type is a special case of *definition*.

4.1 Claims

The first action we would like to be able to perform is that of creating new terms given others. We already have the ability to create terms in our context, but sometimes we want to specify that a term exists because of some other terms. So far we only have the ability to state what a term is, not to specify that the term comes from the context. We call this ability making or invoking a claim and we use the statement type ***claim*** for this. If in the future we need to specify what type of claim, this current section is for existence claims.

Before we get to the process of constructing a claim T, we need to universally fix a genesis definition ***context***. That is, we define *context* as a type and we do not require any children for this type. With this we can insert terms of type *context* in any statement graph we construct.

Now we construct the typed graph T which is supposed to make the claim that some terms exist given some other terms. First we build the contextual terms into T. These play the role of what will be required to be given when the claim is invoked and we require that there be at least one such term. We call these context terms or given terms. Once we have completed the context terms, we add a term c of type *context* and make c the eldest term in T by inserting appropriate parent-child relationships. When we are finished c will no longer be the eldest but we need c to be an ancestor of every claimed term.

Note that the definition of *context* does not have any children because it is a genesis graph. However, c will have children in practice. The term c will still match its definition because having more children than required to match type is not a problem. From the view of *definition*, which is the starting point of all abilities in this language, c has no children to which we can refer by virtue of being type *context*. If we are working within T, we can refer to the children of

c because they are just terms like usual. From any other graph, however, we are not allowed to recognize that c has children in T because the definition of c describes no children.

This lack of ability is intentional. We do not want to allow too much meta-manipulation of our statements for fear of GRML-internal contradiction, opting instead to force ourselves to clearly state which level of logic we are working on. It will be perfectly valid to make statements and prove theorems about logic, even logic upon which GRML relies, but we cannot address directly the logical structure of GRML from inside a graph.

Now that we have the context terms covered, we insert terms into T which we would like to follow from the context. These are called our claimed terms. We insist that no claimed term can be a parent of c if we want T to be a *claim*, though we can allow it if we only want T to be a *definition*. Claimed terms are are distinguished as claims because they are not descendants of c. Once we have inserted all the claimed terms, we cap off T with an eldest member to make T into a typed graph and we set the statement type of T to *claim*.

Formally, a *claim* is a typed graph T with statement type *claim* such that there is one and only one term $c \in T$ of type *context*. Moreover, c must be maximal and must have at least one child. We interpret T by treating any descendant of c is as context and any other term (except c) as claimed. If we want to read T as a traditional statement with traditional quantifiers, we read the descendents of c first, each with the $\forall$ quantifier, then we read the claimed terms and give each the $\exists$ quantifier.

Now that we have seen how to make claims, let's see how to invoke them. Suppose that T is a *claim* graph and that we are building the statement graph S. Suppose we have terms in S which match the context part of T. We wish to use T to construct new terms in S and to do so in a way which records that these new terms follow from the context.

Invoking the claim in T consists of simply inserting a term t of type T in our statement graph S. Of course, to do this, we need to already have terms in S which play the role of the claimed terms. How we distinguish justified-by-claim terms from the rest requires some retroactive structural addition to typed graphs.

Every term in every typed graph is given another layer of coloring called ***existence***. The possible values for *existence* are ***given*** and ***justified***, with the default being *given*. We are not free to change *existence* to *justified* at will; this would defeat the purpose of proofs. Instead *justified* comes about from the following algorithm.

Before we give the algorithm we first require every typed graph S carry a graph P called the existence graph. There must be a 1-1 correspondence between members of P and members of S. If $a, b \in P$ with a a descendant of b in P, it must be that b is *justified* in S and we interpret the relationship as that the justification of b relies on a.

No statement can have a term with an *existence* of *justified* without some corresponding reliance in P. If we add a feature to GRML which results in a graph being considered valid with a *justified* term which has no other terms on which it relies, then we will have broken GRML. There are no absolute truths in GRML – all existence is relative.

Suppose $a, b \in P$ and a is a child of b. The name of this parent-child edge must be the id of a term $t \in S$ where t is a term of statement type *claim*. That is, the graph T which defines the type of t must have statement type *claim* (or some extension of *claim*). In this way the existence graph records which terms are justified, which claims were used, which terms invoked these claims, and in each invocation which terms were given as context. The requirement that P be acyclic ensures that no term can justify itself.

Now that we have the necessary structure of existence graphs, we can describe the algorithm for invoking a *claim*. Suppose T is a *claim*, S is a statement, and we have already constructed the terms in S which we want to use as context in the claim we are invoking. The next thing we do is insert terms to S which we want to be justified by this invocation. Then we add t and set its children appropriately.

At this point we first check that t matches its type T as a *definition*. Once this is verified, we need to account for the actual claim by manipulating the proof graph. Suppose C is the set of terms in S which play the roles of context terms in T and let E be the terms in S which play the roles of claimed terms. For every $c \in C$ and $e \in E$, we make e a parent of c in P with the child name t.

If doing this introduces a cycle in P then we stop and the test failed. If P remains acyclic then the test passed and we have succesfully invoked the claim. We now set the *existence* of every term in E to be *justified*.

4.2 Testables

Another useful behavior we want to have available is boolean truth. Suppose we have a definition D and a term d of type D. If we are considering D as just a definition, we cannot refer to d as being true or false. The only thing we can say about d is what its children are, i.e. in what context d resides. It is very common in math, however, to treat d as being true or false. Many arguments are made by splitting into cases based on whether d is true or false and analyzing each case separately. Without this we would not be able to prove by contradiction, and proof by contradiction, when carefully executed, is the fundamental tool we use in proving theorems. We introduce this behavior of allowing us to consider whether d is true or false at all through the statement type ***testable***.

We do not want to blindly make every concept testable. That adds unnecessary complication and does not even make sense with the rest of the language. If we define Set with a genesis graph, a natural approach which we see in practice in the appendix, that means sets need no context to exist. Essentially, that means the phrase "Let X be a set" makes sense on its own. We could also read that as "X is a set" where this is the introduction of X and is the first thing we state. If Set were testable, that would mean the phrase "X is a set" could be either true or false. Truth is not really a problem because that is in some sense what we meant anyway, but what does it mean for this statement to be false? The only thing we know about X is that X is a set, and if that statement is false then what is X? We cannot go the route of proper classes because we explicitly defined sets without reference to them, and even if we went that route we could ask the same question of them.

We avoid this confusion by not making the genesis definition Set testable. If Set is just a definition, it does not make sense to ask about the validity of the phrase "X is a set." That phrase is neither true nor false, we interpret it as only existing in its context. Being a genesis graph there is no context, so we interpret "X is a set" as merely being a statement and we prefer to phrase it "Let X be a set" to avoid the temptation of thinking it can be tested. In general we do not want genesis graphs to be *testable* because falsehood does not make sense without context.

An example of something we do want to be testable is the concept of emptiness of a set: if X is a set then X is empty or not. We can imagine performing a test to decide if X is empty or not if somebody gives us the set X. In this case the test is to look at X and say whether it is empty or not. We want to be assured that this test will always give a result. Moreover we want this test to be exclusive and well-defined in that X should not be both empty and nonempty at the same time and, if X is fixed, so should be the emptiness of X. This behavior is precisely what we mean by *testable* and when we declare set emptiness to be testable we are assuming this behavior.

Let us construct this example of Empty Set. We start with a genesis definition Set. We then define Empty Set to have a child, named *this*, which is a Set, by making Empty Set the appropriate typed graph. We then set the statement type of Empty Set to *testable*.

The only requirement for a typed graph to pass before being marked as *testable* is that it is not a genesis graph. There is no requirement for a term of type *testable* in addition to *definition*, the only requirement is that terms match their definitions.

4.2.1 Contradiction

One of the most powerful tools available to a mathematician is contradiction. Without it we would be unable to prove any but the most simple claims. However, contradiction is also potentially dangerous. In fact the primary reason we consider contradiction is so we can avoid it in practice.

We introduce how we handle contradiction in this language. We universally fix a genesis graph called ***contradiction***. This defines the type *contradiction*, so according to our rules we can arbitrarily insert terms of type *contradiction* into our statements. However, recall we default terms to the existence level of *given*, which we can also treat as assumed. While we are free in mathematics to assume a contradiction at any point, doing so typically destroys any meaning we are trying to convey. We are only interested in when we can prove a contradiction in some context, meaning when a term of type *contradiction* is *justified*.

As a rule, we limit our statements to have at most one term of type *contradiction*, and we do not want such a term unless it is *justified* or about to be *justified*. This is more of a style rule than a fundamental requirement but it helps keep things clear.

We want to interpret contradiction as being equivalent to simultaneous truth and falsehood. To do this, we introduce a rule for when contradictions can be *justified*. Suppose S is the statement we are currently constructing and T is some testable type. If we have two terms $t, f \in S$, both of type T, and both with exactly the same children, but t is true and f is false, then we *justify* a contradiction term c. In the existence graph c is made to be a parent of both t and f, with corresponding child names *true* and *false*. If a statement graph S has a *justified* member of type *contradiction* then S is said to be a contradiction or S is said to be in contradiction.

4.3 Implications

The next type of claim we see is the implication, which is an extension of the previously described *claim*. An implication has three components: context, assumed, and claimed. If we want to look at an *implication* as a *claim*, we consider the context and assumed portions as the (larger) context component. The claimed component plays the same role in both interpretations. What makes an implication different from a claim is that every term in the assumed or claimed sections must be of type *testable*.

Formally we universally fix a genesis graph ***assumption***. An ***implication*** is a typed graph S with precisely one term c of type *context* and one term a of type *assumption*. We require c to be a child of a and a to be maximal. Any descendant of c is called a context term, any descendant of a which is not a context term is called an assumed term, and any other term is called a claimed term.

The distinction between context and assumed is more than just that assumed terms must be testable. We may choose to put some testable terms in the context portion even if they could be placed in the assumed portion. The distinction allows us fine control over things like the proof graph, which we have not seen yet, and how to take the contrapositive of an implication.

Before describing how invoking an implication works we need to decribe proof graphs. Every statement S must carry a graph P called the proof graph. The idea is similar to the existence graph, but now we consider only *testable* terms.

Every member p of P corresponds to a *testable* term in S. We insist p be designated as *true*or *false*, we call this the truth value of p. Any time $p, q \in P$ with p a parent of q, the child name must be the id of a term in S of type *implication*.