



DEPARTMENT OF PHILOSOPHY,  
LINGUISTICS AND THEORY OF SCIENCE

# ON THE GRAMMAR OF PROOF

**Warrick Macmillan**

---

Master's Thesis:	30 credits
Programme:	Master's Programme in Language Technology
Level:	Advanced level
Semester and year:	Fall, 2021
Supervisor	Aarne Ranta
Examiner	(name of the examiner)
Report number	(number will be provided by the administrators)
Keywords	Grammatical Framework, Natural Language Generation,

## **Abstract**

Brief summary of research question, background, method, results...

# Preface

Acknowledgements, etc.

# Contents

Introduction . . . . .	2
Beyond Computational Trinitarianism . . . . .	2
What is a Homomorphism? . . . . .	4
Perspectives . . . . .	9
Linguistic and Programming Language Abstractions . . . . .	9
Formalization and Informalization . . . . .	12
Syntactic Completeness and Semantic Adequacy . . . . .	13
What is a proof? . . . . .	15
References . . . . .	18
Appendices . . . . .	19

1440 Add section numbers to sections

# Introduction

The central concern of this thesis is the syntax of mathematics, programming languages, and their respective mutual influence, as conceived and practiced by mathematicians and computer scientists. From one vantage point, the role of syntax in mathematics may be regarded as a 2nd order concern, a topic for discussion during a Fika, an artifact of ad hoc development by the working mathematician whose real goals are producing genuine mathematical knowledge. For the programmers and computer scientists, syntax may be regarded as a matter of taste, with friendly debates recurring regarding the use of semicolons, brackets, and white space. Yet, when viewed through the lens of the propositions-as-types paradigm, these discussions intersect in new and interesting ways. When one introduces a third paradigm through which to analyze the use of syntax in mathematics and programming, namely linguistics, I propose what some may regard as superficial detail, indeed becomes a central paradigm raising many interesting and important questions.

## Beyond Computational Trinitarianism

The doctrine of computational trinitarianism holds that computation manifests itself in three forms: proofs of propositions, programs of a type, and mappings between structures. These three aspects give rise to three sects of worship: Logic, which gives primacy to proofs and propositions; Languages, which gives primacy to programs and types; Categories, which gives primacy to mappings and structures.[5]

We begin this discussion of the three relationships between three respective fields, mathematics, computer science, and logic. The aptly named trinity, shown in Figure 4, are related via both *formal* and *informal* methods. The propositions as types paradigm, for example, is a heuristic. Yet it also offers many examples of successful ideas translating between the domains. Alternatively, the interpretation of a Type Theory(TT) into a category theory is incredibly *formal*.

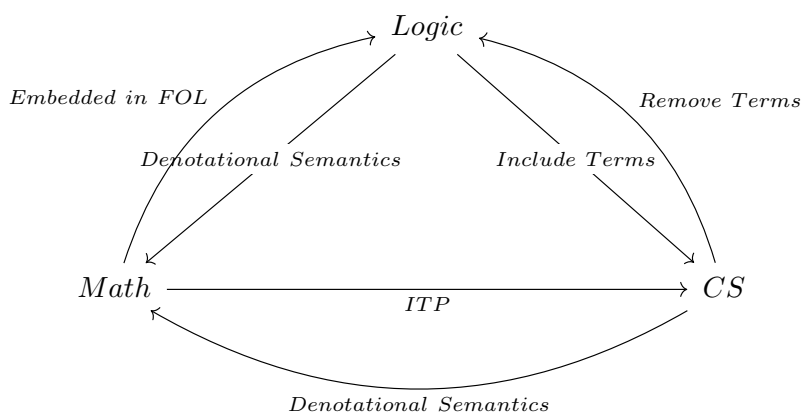


Figure 1: The Holy Trinity

We hope this thesis will help clarify another possible dimension in this diagram, that of Linguistics, and call it the “holy tetrahedron”. The different vertices also resemble religions in their own right, with communities convinced that they have a canonical perspective on foundations and the essence of mathematics. Questioning the holy trinity is an act of a heresy, and it is the goal of this thesis to be a bit heretical by including a much less well understood perspective which provides additional challenges and insights into the trinity.

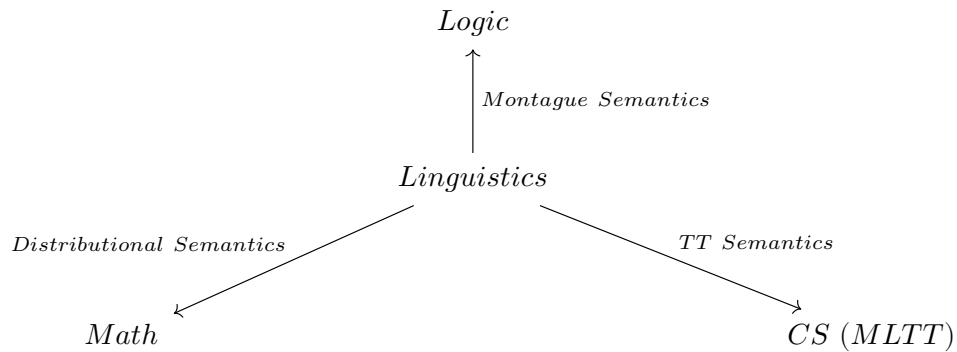


Figure 2: Formal Semantics

One may see how the trinity give rise to *formal* semantic interpretations of natural language in Figure 5. Semantics is just one possible linguistic phenomenon worth investigating in these domains, and could be replaced by other linguistic paradigms. This thesis is alternatively concerned with syntax.

Finally, as in Figure 3, we can ask : how does the trinity embed into natural language? These are the most *informal* arrows of tetrahedron, or at least one reading of it. One can analyze mathematics using linguistic methods, or try to give a natural language justification of Intuitionistic Type Theory (ITT) using Martin-Löf’s meaning explanations.

In this work, we will see that there are multiple GF grammars which model some subset of each member of the trinity. Constructing these grammars, and asking how they can be used in applications for mathematicians, logicians, and computer scientists is an important practical and philosophical question. Therefore we hope this attempt at giving the language of mathematics, in particular how propositions and proofs are expressed and thought about in that language, a stronger foundation.

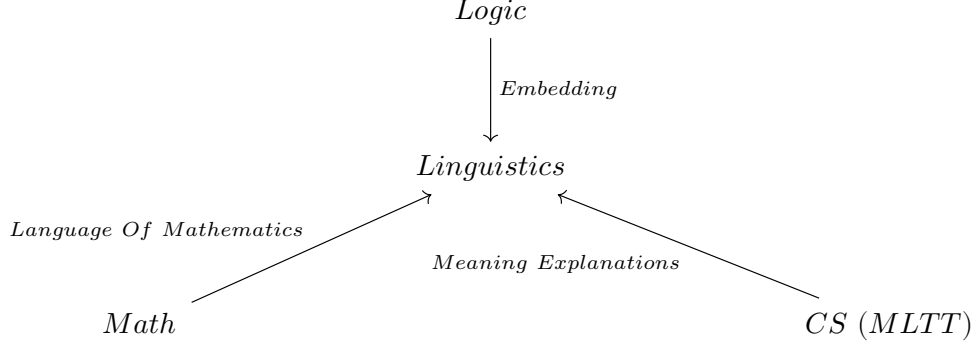


Figure 3: Interpretations of Natural Language

## What is a Homomorphism?

To get a feel for the syntactic paradigm we explore in this thesis, let us look at a basic mathematical example: that of a group homomorphism as expressed in by a variety of somewhat randomly sampled authors.

**Definition 1** *In mathematics, given two groups,  $(G, *)$  and  $(H, \cdot)$ , a group homomorphism from  $(G, *)$  to  $(H, \cdot)$  is a function  $h : G \rightarrow H$  such that for all  $u$  and  $v$  in  $G$  it holds that*

$$h(u * v) = h(u) \cdot h(v)$$

**Definition 2** *Let  $G = (G, \cdot)$  and  $G' = (G', *)$  be groups, and let  $\phi : G \rightarrow G'$  be a map between them. We call  $\phi$  a **homomorphism** if for every pair of elements  $g, h \in G$ , we have*

$$\phi(g * h) = \phi(g) \cdot \phi(h)$$

**Definition 3** *Let  $G, H$ , be groups. A map  $\phi : G \rightarrow H$  is called a group homomorphism if*

$$\phi(xy) = \phi(x)\phi(y) \text{ for all } x, y \in G$$

*(Note that  $xy$  on the left is formed using the group operation in  $G$ , whilst the product  $\phi(x)\phi(y)$  is formed using the group operation  $H$ .)*

**Definition 4** *Classically, a group is a monoid in which every element has an inverse (necessarily unique).*



We inquire the reader to pay attention to nuance and difference in presentation that is normally ignored or taken for granted by the fluent mathematician, ask which definitions feel better, and how the reader herself might present the definition differently.

If one want to distill the meaning of each of these presentations, there is a significant amount of subliminal interpretation happening very much analogous to our innate linguistic usage. The inverse and identity are discarded, even though they are necessary data when defining a group. The order of presentation of information is inconsistent, as well as the choice to use symbolic or natural language information. In Definition 3, the group operation is used implicitly, and its clarification a side remark.

Details aside, these all mean the same thing - don't they? This thesis seeks to provide an abstract framework to determine whether two linguistically nuanced presentations mean the same thing via their syntactic transformations. Obviously these meanings are not resolvable in any kind of absolute sense, but at least from a translational sense. These syntactic transformations come in two flavors : parsing and linearization, and are natively handled by a Logical Framework (LF) for specifying grammars : Grammatical Framework (GF).

We now show yet another definition of a group homomorphism formalized in the Agda programming language:

```
isGroupHom : (G : Group {ℓ}) (H : Group {ℓ'}) (f : ( G ) → ( H )) → Type _
isGroupHom G H f = (x y : ( G )) → f (x G.+ y) ≡ (f x H.+ f y) where
  module G = GroupStr (snd G)
  module H = GroupStr (snd H)

record GroupHom (G : Group {ℓ}) (H : Group {ℓ'}) : Type (ℓ-max ℓ ℓ') where
  constructor grouphom

  field
    fun : ( G ) → ( H )
    isHom : isGroupHom G H fun
```

This actually *was* the Cubical Agda implementation of a group homomorphism sometime around the end of 2020. We see that, while a mathematician might be able to infer the meaning of some of the syntax, the use of levels, the distinguishing between `isGroupHom` and `GroupHom` itself, and many other details might obscure what's going on.

We provide the current, as of May 2021, definition via Cubical Agda. One may witness a significant number of differences from the previous version : concrete syntax differences via changes in camel case, new uses of `Group` vs `GroupStr`, as well as, most significantly, the identity and inverse preservation data not appearing as corollaries, but part of the definition. Additionally, we had to refactor the

commented lines to those shown below to be compatible with our outdated version of cubical. These changes would not just be interesting to look at from the author of the libraries's perspective, but also syntactically.

```

record IsGroupHom {A : Type ℓ} {B : Type ℓ'}
  (M : GroupStr A) (f : A → B) (N : GroupStr B)
  : Type (ℓ-max ℓ ℓ')
  where

  -- Shorter qualified names
  private
    module M = GroupStr M
    module N = GroupStr N

  field
    pres· : (x y : A) → f (M._+_ x y) ≡ (N._+_ (f x) (f y))
    pres1 : f M.0g ≡ N.0g
    presinv : (x : A) → f (M._ x) ≡ N._ (f x)
    -- pres· : (x y : A) → f (x M.· y) ≡ f x N.· f y
    -- pres1 : f M.1g ≡ N.1g
    -- presinv : (x : A) → f (M.inv x) ≡ N.inv (f x)

  GroupHom' : (G : Group {ℓ}) (H : Group {ℓ'}) → Type (ℓ-max ℓ ℓ')
  -- GroupHom' : (G : Group ℓ) (H : Group ℓ') → Type (ℓ-max ℓ ℓ')
  GroupHom' G H = Σ[ f ∈ (G .fst → H .fst) ] IsGroupHom (G .snd) f (H .snd)

```

While the last two definitions may carry degree of comprehension to a programmer or mathematician not exposed to Agda, it is certainly comprehensible to a computer : that is, it typechecks on a computer where Cubical Agda is installed. While GF is designed for multilingual syntactic transformations and is targeted for natural language translation, it's underlying theory is largely based on ideas from the compiler communities. A cousin of the BNF Converter (BNFC), GF is fully capable of parsing programming languages like Agda! And while the Agda definitions are just another concrete syntactic presentation of a group homomorphism, they are distinct from the natural language presentations above in that the colors indicate it has indeed type checked.

While this example may not exemplify the power of Agda's type-checker, it is of considerable interest to many. The type-checker has merely assured us that `GroupHom(')` are well-formed types - not that we have a canonical representation of a group homomorphism. The type-checker is much more useful than is immediately evident: it delegates the work of verifying that a proof is correct, that is, the work of judging whether a term has a type, to the computer. While it's of practical concern is immediate to any exploited grad student grading papers late on a Sunday night, its theoretical concern has led to many recent developments in modern mathematics. Thomas Hales solution to the Kepler Conjecture was seen as unverifiable by those reviewing it, and this led to Hales outsourcing the verification to

Interactive Theorem Provers (ITPs) HOL Light and Isabelle. This computer delegated verification phase led to many minor corrections in the original proof which were never spotted due to human oversight.

Fields medalist Vladimir Voevodsky had the experience of being told one day his proof of the Milnor conjecture was fatally flawed. Although the leak in the proof was patched, this experience of temporarily believing much of his life's work invalidated led him to investigate proof assistants as a tool for future thought. Indeed, this proof verification error was a key event that led to the Univalent Foundations Project [11].

While Agda and other programming languages are capable of encoding definitions, theorems, and proofs, they have so far seen little adoption, and in some cases treated with suspicion and scorn by many mathematicians. This isn't entirely unfounded : it's a lot of work to learn how to use Agda or Coq, software updates may cause proofs to break, and the inevitable imperfections we humans are prone to instilled in these tools . Besides, Martin-Löf Type Theory, the constructive foundational project which underlies these proof assistants, is often misunderstood by those who dogmatically accept the law of the excluded middle as the word of God.

It should be noted, the constructivist rejects neither the law of the excluded middle, nor ZFC. She merely observes them, and admits their handiness in certain citations. Excluded middle is indeed a helpful tool as many mathematicians may attest. The contention is that it should be avoided whenever possible - proofs which don't rely on it, or it's corollary of proof by contradiction, are much more amenable to formalization in systems with decidable type checking. And ZFC, while serving the mathematicians of the early 20th century, is lacking when it comes to the higher dimensional structure of  $n$ -categories and infinity groupoids.

What these theorem provers give the mathematician is confidence that her work is correct, and even more importantly, that the work which she takes for granted and references in her work is also correct. The task before us is then one of religious conversion. And one doesn't undertake a conversion by simply by preaching. Foundational details aside, this thesis is meant to provide a blueprint for the syntactic reformation that must take place.

We don't insist a mathematician relinquish the beautiful language she has come to love in expressing her ideas. Rather, it asks her to make a hypothetical compromise for the time being, and use a Controlled Natural Language (CNL) to develop her work. In exchange she'll get the confidence that Agda provides. Not only that, she'll be able to search through a library, to see who else has possibly already postulated and proved her conjecture. A version of this grandiose vision is explored in The Formal Abstracts Project [4], and it should practically motivate work.

Practicalities aside, this work also attempts to offer a nuanced philosophical perspective on the matter by exploring why translation of mathematical language, despite it's seemingly structured form, is difficult. We note that the natural language definitions of monoid differ in form, but also in pragmatic content. How one expresses formalities in natural language is incredibly diverse, and Definition 4

as compared with the prior homomorphism definitions is particularly poignant in demonstrating this. These differ very much in nature to the Agda definitions - especially pragmatically. The differences between the Cubical Agda definitions may be loosely called pragmatic, in the sense that the choice of definitions may have downstream effects on readability, maintainability, modularity, and other considerations when trying to write good code, in a burgeoning area known as proof engineering.

A pragmatic treatment of the language of mathematics is the golden egg if one wishes to articulate the nuance in how the notions proposition, proof, and judgment are understood by humans. Nonetheless, this problem is just now seeing attention. We hope that the treatment of syntax in this thesis, while a long ways away from giving a pragmatic account of mathematics, will help pave the way there.

## Perspectives

...when it comes to understanding the power of mathematical language to guide our thought and help us reason well, formal mathematical languages like the ones used by interactive proof assistants provide informative models of informal mathematical language. The formal languages underlying foundational frameworks such as set theory and type theory were designed to provide an account of the correct rules of mathematical reasoning, and, as Gödel observed, they do a remarkably good job. But correctness isn't everything: we want our mathematical languages to enable us to reason efficiently and effectively as well. To that end, we need not just accounts as to what makes a mathematical argument correct, but also accounts of the structural features of our theorizing that help us manage mathematical complexity.[1]

### Linguistic and Programming Language Abstractions

The key development of this thesis is to explore the formal and informal distinction of presenting mathematics as understood by mathematicians and computer scientists by means of rule-based, syntax oriented machine translation.

Computational linguistics, particularly those in the tradition of type theoretical semantics[10], gives one a way of comparing natural and programming languages. Type theoretical semantics is concerned with the semantics of natural language in the logical tradition of Montague, who synthesized work in the shadows of Chomsky [2] and Frege [3]. This work ended up inspiring the GF system, a side effect of which was to realize that machine translation was possible as a side effect of this abstracted view of natural language semantics. Indeed, one such description of GF is that it is a compiler tool applied to domain specific machine translation. We may compare the “compiler view” of PLs and the “linguistics view” of NLs, and interpolate this comparison to other general phenomenon in the respective domains.

We will reference these programming language and linguistic abstraction ladders, and after viewing Figure 4, the reader should examine this comparison with her own knowledge and expertise in mind. These respective ladders are perhaps the most important lens one should keep in mind while reading this thesis. Importantly, we should observe that the PL dimension, the left diagram, represents synthetic processes, those which we design, make decisions about, and describe formally. Alternatively, the NL abstractions on the right represent analytic observations. They are therefore subject to different, in some ways orthogonal, constraints.

The linguistic abstractions are subject to empirical observations and constraints, and this diagram only serves as an atlas for the different abstractions and relations between these abstractions, which may be subject to modifications depending on the linguist or philosopher investigating such matters. The PL abstractions as represented, while also an approximation, serves as an actual high altitude

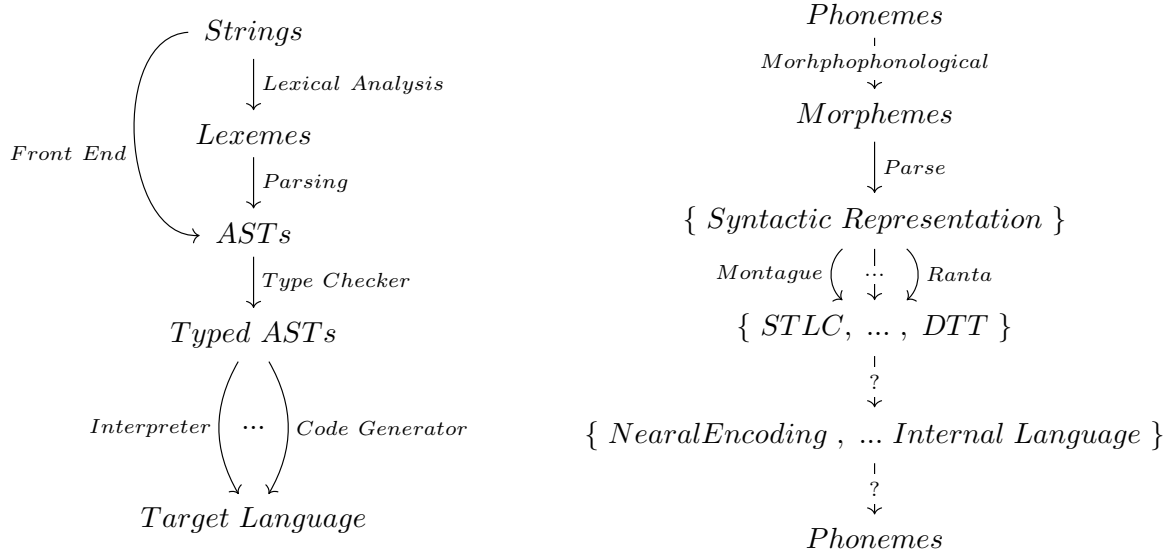


Figure 4: PL (left) and NL (right) Abstraction Ladders

blueprint for the design of programming languages. While the devil is in the details and this view is greatly simplified, the representation of PL design is unlikely to create angst in the computer science communities. The linguistic abstractions are at the intersection of many fascinating debates between linguists, and there is certainly nothing close to any type of consensus among linguists which linguistic abstractions, as well as their hierarchical arrangement, are more practically useful, theoretically compelling, or empirically testable.

There are also many relevant concerns not addressed in either abstraction chain that are necessary to give a more comprehensive snapshot. For instance, we may consider intrinsic and extrinsic abstractions that diverge from the idealized picture. In PL extrinsic domain, we can inquire about

- systems with multiple interactive programming language
- how the programming languages behave with respect to given programs
- embedding programming languages into one another

Alternatively, intrinsic to a given PL, there picture is also not so clear. Agda, for example, requires the evaluation of terms during typechecking. It is implemented with 4.5 different stages between the syntax written by the programmers and the “fully reflected Abstract Syntax Tree (AST)”. But this example is perhaps an outlier, because Agda’s typechecker is so powerful that the design, implemenation, and use of Agda revolves around its type-checker, (which, ironically, is already called during the parsing phase). It is not anticipated that floating point computation, for instance, would ever be considered when implementing new features of Agda, at least not for the foreseeable future. Indeed, the ways Agda represents ASTs were an obstacle encountered doing this work, because deciding which parsing

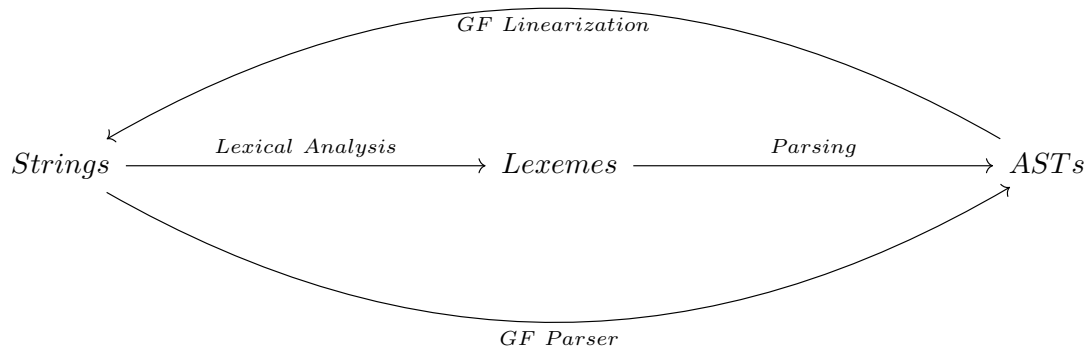


Figure 5: GF in a nutshell

stage one should connect to the Portable Grammar Format (PGF) embedding is nontrivial.

Let's zoom in a little and observe the so-called front-end part of the compiler. Displayed in Figure 5 is the highest possible overview of GF. This is deceptively simple depiction of such a powerful and intricate system.

What makes GF so compelling is its ability to translate between inductively defined languages that type theorists define and relatively expressive fragments of natural languages, via the composition GF's parsing and linearization capabilities. It is in some sense the attempt to overlay the abstraction ladders that led to this development.

For natural language, some intrinsic properties might take place, if one chooses, at the neurological level, where one somehow can contrast the internal language, mechanism of externalization (generally speech), as proposed by Chomsky. Extrinsically to the linguistic abstractions depicted, pragmatics is absent, but of incredible interest in the field generally.

The point is to recognize there are stark differences between natural languages and programming languages which are even more apparent when one gets to certain abstractions. Classifying both programming languages and natural languages as languages is best read as an incomplete (and even sometimes contradictory) metaphor, due to perceived similarities (of which there are ample).

Nonetheless, the point of this thesis is to take a crack at that exact question : how can one compare programming and natural languages, in the sense that a natural language, when restricted to a small enough (and presumably well-behaved) domain, behaves as a programming language. Simultaneously, we probe the topic of Natural Language Generation (NLG). Given a logic or type system with some theory inside (say arithmetic over the naturals), how do we not just find a natural language representation which interprets our expressions, but also does so in a way that is linguistically coherent in a sense that a competent speaker can read such an expression and make sense of it.

The specific linguistic domain we focus on, that of mathematics, is a particular sweet spot at the intersection of these natural and formal language spaces. It should be noted that this problem, that of translating between formal (in a PL or logic) and informal (in linguistic sense) mathematics as stated, is both vague and difficult. It is difficult in both the practical sense, that it may be either of infeasible complexity or even perhaps undecidable. But it is also difficult in the philosophical sense, that it is a question which one may entertain arguments which a priori may deflate its effectiveness or meaningfulness. Like all collective human endeavors, mathematics is a historical construction - that is, its conventions, notations, understanding, methodologies, and means of publication and distribution have all been in a constant flux. While in some sense the mathematics today can be seen today as a much refined version of whatever the Greeks or Egyptians were doing, there is no consensus on what mathematics is, how it is to be done, and most relevant for this treatise, how it is to be expressed.

We present a sketch of the difference of this so-called formal/informal distinction. Mathematics, that is mathematical constructions like numbers and geometrical figures, arose out of ad-hoc needs as human cultures grew and evolved over the millennia. Indeed, just like many of the most interesting human developments of which there is a sparsely documented record until relatively recently, it is likely to remain a mystery what the long historical arc of mathematics could have looked like in the context of human evolution. And while mathematical intuitions precede mathematical constructions (the spherical planet precedes the human use of a ruler compass construction to generate a circle), we should take it as a starting point that mathematics arises naturally out of our linguistic capacity. This may very well not be the case, or at least not universally so, but it is impossible to imagine humans developing mathematical constructions elaborating anything particularly general without linguistic faculties. Despite whatever empirical or philosophical dispute one takes with this linguistic view of mathematical abilities, we seek to make a first order approximation of our linguistic view for the sake of this work. The discussion around mathematics relation to linguistics generally, regardless of the stance one takes, should benefit from this work.

## Formalization and Informalization

Formalization is the process of taking an informal piece of natural language mathematics, embedding it in into a theorem prover, constructing a model, , working with types instead of sets. This often requires significant amounts of work, and if the proof is big enough, warrants the title *proof engineering*. We note some interesting artifacts about a piece of mathematics being formalized:

- it may be formalized differently by two different people in many different ways
- it may have to be modified, to include hidden lemmas, to correct of an error, or other bureaucratic obstacles
- it may not type check, and only be presumed hypothetically to be 'a correct formalization' given evidence



Informalization, on the other hand is a process of taking a piece formal syntax, and turning it into a natural language utterance, along with commentary motivating and or relating it to other mathematics. It is a clarification of the meaning of a piece of code, suppressing certain details and sometimes redundantly reiterating other details.

Mathematicians working in either direction know this is a respectable task, often leading to new methods, abstractions, and research altogether. And just as any type of machine translation, rule-based or statistical, for a Virginia Woolf novel or Emily Dickinson poem from English to Mandarin would be entirely laughable, so to would the pretense that the methods we explore here using GF could actually match the competence of mathematicians translating work between a computer a book. Despite the futility of beating a mathematician at proof translation, it shouldn't deter those so inspired to try.

## Syntactic Completeness and Semantic Adequacy

The GF pipeline, that of bidirectional translation through an intermediary abstract syntax representation, has, two fundamental criteria that must be assessed for one to judge the success of an approach over both formalization and informalization.

The first criterion mentioned above, which we'll call *syntactic completeness*, asks the following : given an utterance or natural language expression that a mathematician might understand, does the GF grammar emit a well-formed syntactic expression in the target logic or programming language? [Example?]

This problem is certain to be infeasible in many cases - a mathematician might not be able to reconstruct the unstated syntactic details of a proof in a discipline outside her expertise, it is at worthy pursuit to ask why it is so difficult! Additionally, one does not know a priori that the generated expression in the logic has its intended meaning, other than through some meta verification procedure.

Conversely, given a well formed syntactic expression in, for instance, Agda, one can ask if the resulting English expression generated by GF is *semantically adequate*. This notion of semantic adequacy is also delicate, as mathematicians themselves may dispute, for instance, the proof of a given proposition or the correct definition of some notion. However, if it is doubtful that there would be many mathematicians who would not understand some standard theorem statement and proof in an arbitrary introductory analysis text, even if one may dispute it's presentation, clarity, pedagogy, or other pedantic details. Whether one asks that semantic adequacy means some kind of sociological consensus among those with relevant expertise, or a more relaxed criterion that some expert herself understands the argument, a dubious perspective in scientific circles, semantic adequacy should appease at least one and potentially more mathematicians.

We introduce these terms, syntactic completeness and semantic adequacy, to highlight a perspective and insight that seems to underlie the biggest differences between informal and formal mathematics. We claim that mathematics, as done on

a theorem prover, is a syntax oriented endeavor, whereas mathematics, as practiced by mathematicians, prioritizes semantic understanding. Developing a system which is able to formalize and informalize utterances which preserve syntactic completeness and semantic adequacy, respectively, is probably infeasible. Even introducing objective criteria to really judge these terms is likely to be infeasible.

This perspective represents an observation, and not intended to judge whether the syntactic or semantic perspective on mathematics is better - there is a dialectical phenomena between the two. Let's highlight some advantages both provide, and try to distinguish more precisely what a syntactic and semantic perspective may be.

When the Agda user builds her proof, she is outsourcing much of the bookkeeping to the type-checker. This isn't purely a mechanical process though, she often does have to think, how her definitions will interact with downstream programs, as well as whether they are even sensible to begin with (i.e. does this have a proof). The syntactic side is expressly clear from the readers perspective as well. If Agda proofs were semantically coherent, one would only need to look at code, with perhaps a few occasional remarks about various intentions and conclusions, to understand the mathematics being expressed. Yet, papers are often written exclusively in Latex, where Agda proofs have had to be reverse engineered, preserving only semantic details and forsaking syntactic nuance. Oftentimes the code is kept in the appendix so as to provide a complete syntactic blueprint. But the act of writing an Agda proof and reading them are often orthogonal, as the term somehow shadows the application of typing rules which enable its construction. In some sense, the construction of the proof is entirely engaged with the types, whereas the human witness of a large term is either lost as to why it fulfills the typing judgment, they have to reexamine parts of the proof reasoning in their head or perhaps, again, try to build it interactively with Agda's help.

Even in cases where Agda code is included in a paper, it is most often the types which are emphasized and produced. Complex proof terms are seldom to be read on their own terms. The natural language description and commentary is still largely necessary to convey whatever results, regardless if the Agda code is self-contained. And while literate Agda is some type of bridge, it is still the commentary which in some sense unfolds the code and ultimately makes the Agda code legible.

This is particularly pronounced in the Coq programming language, where proof terms are built using Ltac, which can be seen as some kind of imperative syntactic metaprogramming over the core language, Gallina. The user rarely sees the internal proof tree that one becomes familiar with in Agda. The tactics are not typed, often feel very adhoc, and tacticals, sequences of tactics, may carry very little semantic value (or even possibly muddy one's understanding when reading proofs with unknown tactics). Indeed, since Ltac isn't itself typed, it often descends into the sorrows of so-called untyped languages (which are really uni-typed), and there are recent attempts to change this [6] [9]. From our perspective, the use of tactics is an additional syntactic obfuscation of what a proof should look like from the mathematicians perspective - and attempt to remedy this is. Alecyrton is one impressive development in giving Coq proofs more readability through a interac-

tive back-end which shows the proof state, and offers other semantically appealing models like interactive graphics [8]. This kind of system could and should inspire other proof assistants to allow for experimentation with syntactic alternative to code.

Tactics obviously have their uses, and sometimes enhance high level proof understanding, as tactics like *ring* or *omega* often save the reader overhead of parsing pedantic and uninformative details. And for certain proofs, especially those involving many hundreds of cases, the metaprogramming facilities actually give one exclusive advantages not offered to the classical mathematician using pen and paper. Nonetheless, the dependent type theorist's dream that all mathematicians begin using theorem provers in their everyday work is largely just a dream, and with relatively little mainstream adoption by mathematicians, the future is all but clear.

Mathematicians may indeed like some of the facilities theorem provers provide, but ultimately, they may not see that as the "essence" of what they are doing. What is this essence? We will try to shine a small light on perhaps the most fundamental question in mathematics.

## What is a proof?

A proof is what makes a judgment evident [7].

The proofs of Agda, and any programming language supporting proof development, are *formal proofs*. Formal proofs have no holes, and while there may very well be bugs in the underlying technologies supporting these proofs, formal proofs are seen as some kind of immutable form of data. One could say they provide [?] for judgments, which themselves are objective entities when encoded on a computer. What we call formal proofs might provide a science fiction writer an interesting thought experiment as regards communicating mathematics with an alien species incapable of understanding our language otherwise. Formal proofs, however, certainly don't appease all mathematicians writing for other mathematicians.

Mathematics, and the act of proving theorems, according to Brouwer is a social process. And because social processes between humans involve our linguistic faculties, as we hope to elucidate what a proof with a simplified description. Suppose we have two humans,  $h_1$  and  $h_2$ . If  $h_1$  claims to have a proof  $p_1$ , and elaborates it to  $p_2$  who claims she can either verify  $p_1$  or reproduce and re-articulate it via  $p'_1$ , such that  $h_1$  and  $h_2$  agree that  $p_1$  and  $p'_1$  are equivalent, then they have discovered some mathematics. In fact, in this guise mathematics, can be viewed as a science, even if in fact it is constructed instead of discovered.

An apt comparison is to see the mathematician is architect, whereas the computer scientist responsible for formalizing the mathematics is a civil engineer. The mathematics is the building which, like all human endeavors, is created via resources and labor of many people. The role of the architect is to envision the facade, the ex-

terior layer directly perceived by others, giving a building its character, purpose, and function. The engineer is on the other hand, tasked with assuring the building gets built, doesn't collapse, and functions with many implicit features which the user of the building may not notice : the running water, insulation, and electricity. Whereas the architect is responsible for the building's *specification*, the engineer is tasked with its *implementation*.

We claim informal proofs are specifications and formal proofs are implementations. Additionally, via the propositions-as-types interpretation, one may see a logic as a specification and a PL as an implementation of a given logic, often with multiple ways of assigning terms to a given type. Therefore, one may see the mathematician ambiently developing a theorem in classical first order logic as providing a specification of a proposition in that language, whereas a given implementation of that theorem in Agda could be viewed as a model construction of some NL fragment, where truth in the model would correspond to termination of type-checking. Alternatively, during the informalization process, two different authors may suppress different details, or phrase a given utterance entirely differently, possibly leading to two different, but possibly similar proofs. Extrapolating our analogy, the same way, two architects hypothetically, given the same engineering plans, could produce two entirely different looking buildings.

There isn't a natural notion of equivalence between informal and formal proofs, but rather, loosely, some kind of adjunction in the categorical sense between these two sets. We note the fact that the "acceptable" Natural language utterances aren't inductively defined precludes us from actually constructing a canonical mathematical model of formal/informal relationship, but we most certainly believe that if the GF perspective of translation is used, there at least an approximation of what a model may look like.

Mathematicians seek model independence in their results (i.e., they don't need a direct encoding of Fermat's last theorem in set theory in order to trust its validity). This is one possible reason why there is so much reluctance to adopt proof assistant, because the implementation of a result in Coq, Agda, or HOL4 may lead to many permutations of the same result, each presumably representing the same piece of knowledge with. It's also noted a proof doesn't obey the same universality that it does when it's on paper or verbalized - that Agda 2.6.2, and its standard library, when updated in the future, may break current developments, and proofs. While this is a unanimous problem with all software, we believe the GF approach offers at least a vision of not only linguistic, but also foundation independence with respect to mathematics.

This thesis examines not just a practical problem, but touches many deep issues in some space in the intersection of the foundations, both practical and philosophical, of mathematics, logic, computer science, and their relations studied via linguistic formalisms. These subjects, and their various relations, are the subject of countless hours of work and consideration by many great minds. We barely scratches the surface of a few of these developments, but it nonetheless, it is hoped, provides a nontrivial perspective at many important issues.

Recapitulating much of what was said, we hope that the following questions may have a new perspective :

- what are mathematical objects ?
- how do their encodings in different foundational formalisms affect their interpretations ?
- how does mathematics develop as a social process ?
- how does what mathematics is and how it is done rely on given technologies of a given historical era. ?

While various branches of linguistics have seen rapid evolution due to, in large part, their adoption of mathematical tools, the dual application of linguistic tools to mathematics is quite sparse and open terrain. We hope the reader can walk away with an new appreciation to some of these questions and topics after reading this. These nuances we will not explore here, but shall be further elaborated in the future and and more importantly, hopefully inspire other readers to respond accordingly.

Although not given in specific detail, the view of what mathematics is, in both a philosophical and mathematical sense, as well as from the view of what a foundational perspective, requires deep consideration in its relation to linguistics. And while this work is perhaps just a finer grain of sandpaper on an incomplete and primordial marble sculpture, it is hoped that the sculptor's own reflection is a little bit more clear after we polish it here.

## References

- [1] Avigad, J. (2015). Mathematics and language.
- [2] Chomsky, N. (2009). *Syntactic Structures*. De Gruyter Mouton.
- [3] Frege, G. (1879). *Begrisschrift*. Halle.
- [4] Hales, T. (2019). An argument for controlled natural languages in mathematics.
- [5] Harper, R. (2011). The holy trinity.
- [6] Kaiser, J.-O., Ziliani, B., Krebbers, R., Régis-Gianas, Y., & Dreyer, D. (2018). Mtac2: Typed tactics for backward reasoning in coq. *Proc. ACM Program. Lang.*, 2(ICFP).
- [7] Martin-Löf, P. (1996). On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1), 11–60.
- [8] Pit-Claudel, C. (2020). Untangling mechanized proofs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2020 (pp. 155–174). New York, NY, USA: Association for Computing Machinery.
- [9] Pédrot, P.-M. (2021). Ltac2 : Tactical warfare.
- [10] Ranta, A. & Ranta, D. (1994). *Type-theoretical Grammar*. Indices (Claredon). Clarendon Press.
- [11] The Univalent foundations program & Institute for advanced study (Princeton, N. (2013). *Homotopy Type Theory: Univalent Foundations of Mathematics*.

## Appendices