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 1	Staffan Larsson
Examiner 2	Stergios Chatzikyriakidis
Report number	(number will be provided by the administrators)
Keywords	The Language of Mathematics, Type Theory, Grammatical Fram

Abstract

The notion of *formal proof* is a modern development, beginning with Frege's foundational studies in modern mathematical logic. Formal proofs have manifested more recently as verifiable computer programs written in programming languages like Agda, via the propositions-as-types interpretation of logical formulas. The notion of mathematical proof more generally, developed at least as far back as Euclid, may be viewed as a natural language argument which provides evidence for a proposition. Comparing notions of formal and natural language mathematics is of both significant practical and theoretical concern, and one means of comparison is seeking systematic ways of *translating* between them.

This thesis gives one possible mechanism of translation between mathematical vernacular and code via Grammatical Framework (GF), as a GF grammar can parse and linearize. It can therefore translate between natural and programming language utterances via a shared abstract syntax tree (AST). While grammars for programming languages are generally meant to be compact so-as to produce unique parses, natural language grammars must account for both a natural language's ambiguity and expressiveness - the fact that there are uncountable ways of saying "the same thing" makes it so interesting. Rectifying these opposing interests in a single grammar is therefore incredibly challenging.

I introduce dual notions of understanding and analyzing mathematical language. *Syntactic completeness* is a criteria for judging constructions which contain no errors and entirely encode an argument's subtlest details. *Semantically adequate* proofs and constructions are those which validate a claim to a fluent mathematician, but may require some implicit knowledge, explicitly defer arguments to the reader, or even contain errors. The grammars written for this thesis and prior to it are therefore able to compared on this spectrum. We demonstrate a syntactically complete grammar which can parse real Agda proofs but generates poor natural language, and compare it to a semantically adequate grammar which parses actual mathematics text, but generates ill-formed types and programs. A Haskell embedding of these grammars with intermediary transformations allows for at least a partial resolution of these competing interests.

To further elaborate this discord between syntactic completeness and semantic adequacy, we give parallel examples of mathematics text and Agda code, with an Agda formalization of parts of the Homotopy Type Theory (HoTT) book given to emphasize the needed for parallel corpus of programming and natural language data for future translation endeavors. Additionally, the differences between type theoretic, set theoretic, and logical language are explored throughout this work, because foundational attitudes create inherent frictions in the translation process. The insights gleaned from this work suggest new ways of analyzing and understanding the difference between formal and informal proofs.

Preface

Acknowledgements, etc.

Contents

1839 TODO

- Fix figure titles
- internal References
- Citations
- fix quotes to include author
- per inari, add more subsections
- fix codeword, term, etc. to more uniform way
- to emphasize or not, sem. adeq. , syntactic compl
- uniform way of referencing the grammars
- ask aarne about keywords in title

1 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 regarding 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.

1.1 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.*Robert Harper* [28]

We begin this discussion of the three relationships between three respective fields, mathematics, computer science, and logic. The aptly named trinity, shown in Figure 15, 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*.

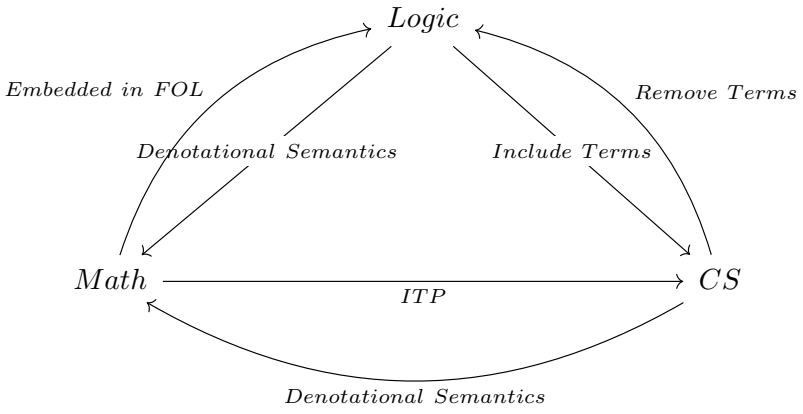

Figure 1: The Holy Trinity

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.

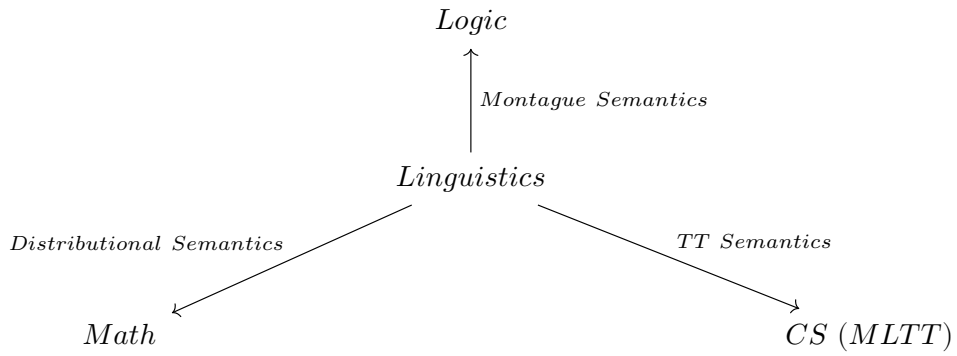

Figure 2: Formal Semantics

One may see how the trinity give rise to *formal* semantic interpretations of natural language in Figure 2. 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 17, 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.

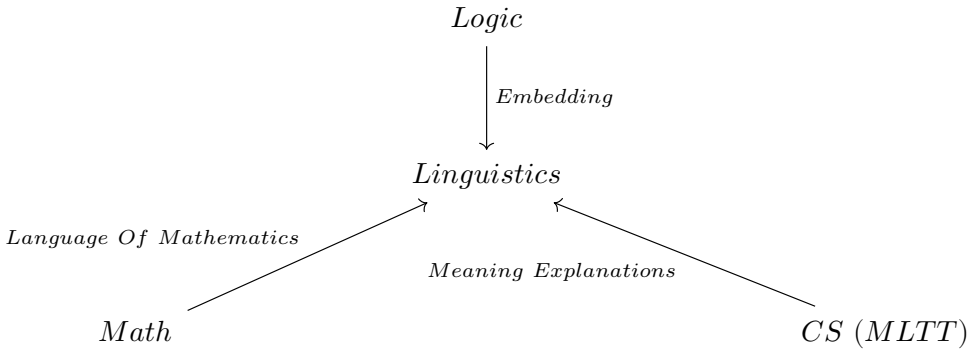

Figure 3: Interpretations of Natural Language

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.

1.2 What is a Homomorophism?

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* $h : G \to H$ *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* $\phi : G \to G'$ *be a map between them. We call* ϕ *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* $\phi : G \to H$ *is called a* group homomorphism *if*

$$\phi(xy) = \phi(x)\phi(y)$$

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 lingusitic ussage. The inverse and identity are discarded, even though they are necessary data when defning a group. The order of presentation of information is inconsistent, as well as the choice to use symbolic or natural language information. In Definition 7, 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 lingusitically nuanced

presenations 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 distinguising bewteen isGroupHom and GroupHom itself, and many other details might obscure what's going on.

We finally provide the current (May 2021) definition via Cubical Agda. One may witness a significant number of differences from the previous version : concrete syntax differenes 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 assintants as a tool for future thought. Indeed, this proof verification error was a key event that led to the Univalent Foundations Project [58].

While Agda and other programming languages are capable of encoding definitions, theorems, and proofs, they have so far seen little adoption. In some cases they have been 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 corallary of proof by contradction, are much more ameanable to formalization in systems with decideable 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 [26], 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.

2 Philosophical 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.[3]

2.1 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[45], gives one a way of comparing natural and programming languages. Type theoretical semantics it is concerned with the semantics of natural language in the logical tradition of Montague, who synthesized work in the shadows of Chomsky [12] and Frege [22]. 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 are 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 approximations, serves as an actual high altitude

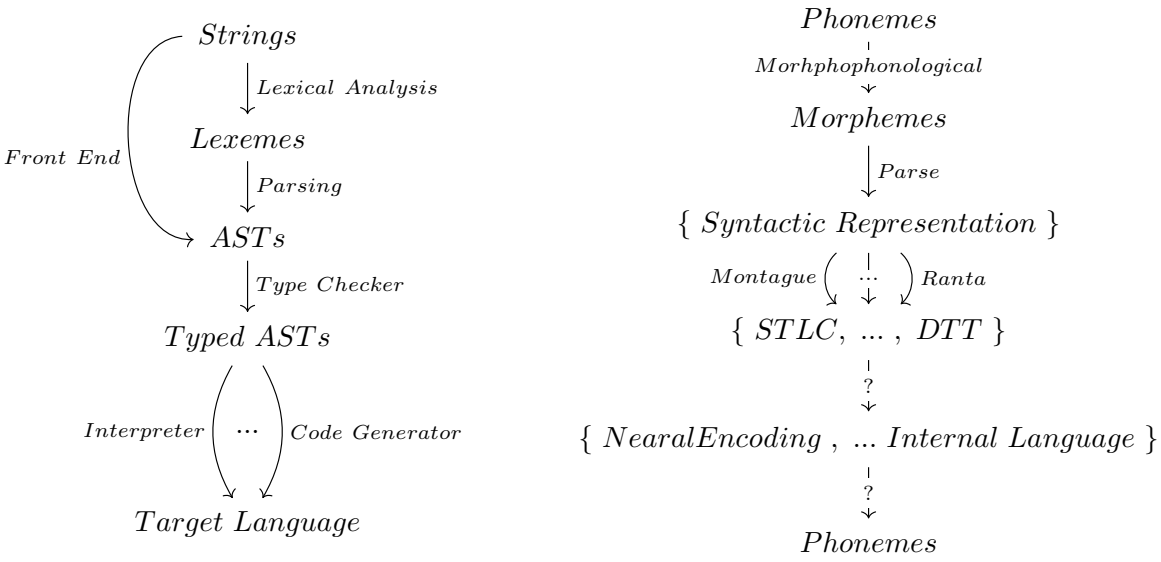

Figure 4: PL (left) and NL (right) Abstraction Ladders

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 comprehsive snapshot. For instance, we may consider intrinsic and extrensic abstractions that diverge from the idealized picture. In PL extrensic 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)" [1]. But this example is perhaps an outlier, because Agda's type-checker is so powerful that the design, implemenation, and use of Agda revolves around it, (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 stage one should connect to the Portable Grammar Format (PGF) embedding is nontrivial.

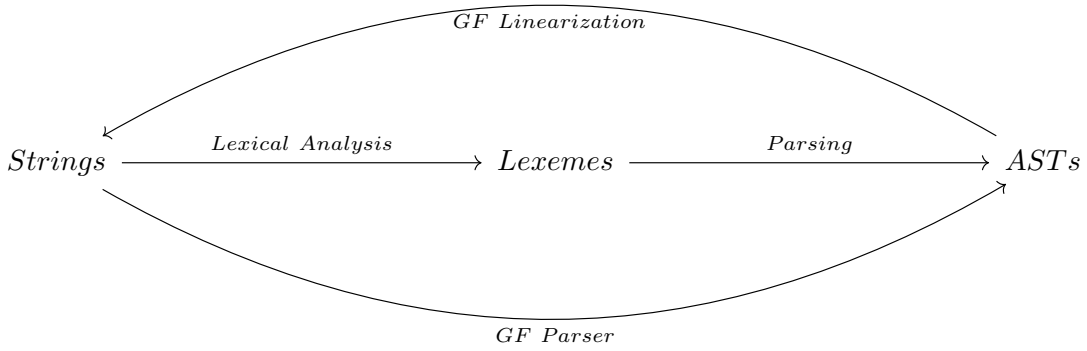

Figure 5: GF in a nutshell

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 a 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 specify and relatively expressive fragments of natural languages, via the composition of GF's parsing and linearization capabilities. It is in some sense the attempt to overlay the abstraction ladders at the syntactic level and semantic 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 (i-language) with the mechanism of externalization (generally speech) as proposed by Chomsky [11]. Extrinsic to the linguistic abstractions depicted, pragmatics is absent. . The point is to recognize their are stark differences between natural languages and programming languages which are even more apparent when one gets to certain abstractions. Classifying both programming languages as languages is best read as an incomplete (and even sometimes contradictory) metaphor, due to perceived similarities (of which their 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 make sense of it in a facile way.

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* and *informal* 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. One may entertain

the prospect of syntactically translated mathematics may 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. 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.

Historically, mathematics has been filtered of natural language artifacts, culminating in some sense with Frege's development of a formal proof. A mathematician often never sees a formal proof as it is treated in Logic and Type Theory. We hope this work helps with a new foundational mentality, whereby we try to bring natural language back into mathematics in a controlled way, or at least to bridge the gap between our technologies, specifically injecting ITPs into a mathematicians toolbox.

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 humans 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.

2.2 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, and working with types instead of sets. This often requires significant amounts of work. 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

Category	Formal Proof	Informal Proof
Audience	Agda (and Human)	Human
Translation	Compiler	Human
Objectivity	Objective	Subjective
Historical	20th Century	<= Euclid
Orientation	Syntax	Semantics
Inferability	Complete	Domain Expertise Necessary
Verification	PL Designer	Human
Ambiguity	Unambiguous	Ambiguous

Figure 6: Informal and Formal Proofs

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. In figure Figure 6 we offer a few dimensions of comparison.

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, on Virginia Woolf novel or Emily Dickinson poem from English to Mandarin would be absurd, 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 surpassing a mathematician at proof translation, it shouldn't deter those so inspired to try.

2.3 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*, says that a term either type-checks, or some natural language form can be deterministically transformed to a term that does type-check.

It 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? The saying "grammars leak", can be transposed to say (NL) "proofs leak" in that they are certain to contain omissions.

This problem of syntactically complete mathematics is certain to be infeasible in many cases - a mathematician might not be able to reconstruct the unstated syntactic details of a proof in an discipline outside her expertise, it is at worthy pursuit to ask why it is so difficult! Additionally, certain inferable details may also detract from the natural language reading rather than assist. Perhaps most importantly, 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 take issue with other 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.

An example of a syntactically complete but semantically inadequate statement which one of our grammars parses is "is it the case that the sum of 3 and the sum of 4 and 10 is prime and 9999 is odd". Alternatively, most mathematical proofs in most mathematical papers are most likely syntactically incomplete - anyone interested in formalizing a piece of mathematics from some arbitrary (even simple) resource will learn this the hard way.

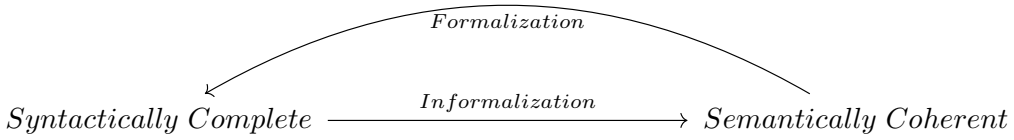

Figure 7: Formal and Informal Mathematics

We introduce these terms, syntactic completeness and semantic adequacy to highlight perspectives and insights that seems to underlie the biggest differences between informal and formal mathematics, as is show in Figure 7. We claim that mathematics, as done via 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 definitions is likely to be infeasible.

This perspective represents an observation and is 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 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 it is often orthogonal, as the term shadows the application of typing rules which enable its construction. 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, she has to reexamine parts of the proof reasoning in her head or perhaps, try to rebuild 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 [30] [44]. From our perspective, the use of tactics is an additional syntactic obfuscation of what a proof should look like from the mathematicians perspective - and it is important to attempt to remedy this is. Alecytron is one impressive development in giving Coq proofs more readability through a interactive back-end which shows the proof state, and offers other semantically appealing models like interactive graphics [43]. This kind of system could and should inspire other proof assistants to allow for experimentation with syntactic alternative to linear 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. 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.

2.4 What is a proof?

> A proof is what makes a judgment evident [37].

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 *objective evidence* 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, a we hope to elucidates 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 $h1$ and $h2$ agree that $p1$ 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 an 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 exterior 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. Alter-

natively, 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 two architects given the same engineering plans could produce two entirely different looking and functioning buildings. Oftentimes though, it is the architect who has the vision, and the engineers who end up implementing the architects art.

We also briefly explore the difference between the mathematician and the physicist. The physicist will often say under her breath to a class, “don’t tell anyone in the math department I’m doing this” when swapping an integral and a sum or other loose but effective tricks in her blackboard calculations. While there is an implicit assumption that there are theorems in analysis which may justify these calculations, it is not the physicist’s objective to be as rigorous as the mathematician. This is because the physicist is not using the mathematics as a syntactic mechanism to reflect the semantic domain of particles, energy, and other physical processes which the mathematics in physics serves to describe. The mathematician using Agda, needing to make syntactically complete arguments, needs to be obsessed with the details - whereas the “pen and paper” mathematician would need be reluctant to carry out all the excruciating syntactic details for sake of semantic clarity.

There isn’t a natural notion of equivalence between informal and formal proofs, but rather, loosely, some kind of adjunction between these two sets. We note the fact that the “acceptable” Natural language utterances aren’t inductively defined. This 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 can at least be an approximation of what a model may look like. It is our contention that the linguist interested in the language of mathematics should perhaps be seen as a scientist, whose point is to contribute basic ideas and insights from which the architects and engineers can use to inform their designs.

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. 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 proofs”, as was seen in the introduction. 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 agnosticism 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, 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 is 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.

2.5 What is a proof revisited

> Though philosophical discussion of visual thinking in mathematics has concentrated on its role in proof, visual thinking may be more valuable for discovery than proof [24]

As an addendum to asking such a presumably simple question in the previous section, we briefly address the one particular oversimplification which was made. We briefly touch on what isn't just syntactic about mathematics, namely so-called "Proofs without Words" [41] and other diagrammatic and visual reasoning tools generally. Because our work focuses on syntax, and is not generalized to other mathematical tools, we hope one considers this as well when pondering the language of mathematics.

The role of visualization in programming, logic, and mathematics generally offers an abundance of contrast to syntactically oriented alphanumeric alphabets, i.e. strings of symbols, which we discuss here. Although the trees in GF are visual, they are of intermediary form between strings in different languages, and therefore the type of syntax we're discussing here is strings, we hope a brief exploration of

alternatives for concrete syntax will be fruitful. Targeting latex via GF for instance, is a small step in this direction.

Graphical Programming languages facilitating diagrammatic programming are one instance of a nonlinear syntax which would prove tricky but possible to implement via GF. Additionally, Globular, which allows one to carry out higher categorical constructions via globular sets is an interesting case study for a graphical programming language which is designed for theorem proving [4]. Additionally, Alecytron supports basic data structure visualization, like red-black trees which carry semantic content less easy in a string based-setting [43].

Visualization are ubiquitous in contemporary mathematics, whether it be analytic functions, knots, diagram chases in category theory, and a myriad of other visual tools which both assist understanding and inform our syntactic descriptions. We find these languages appealing because of their focus on a different kind of internal semantic sensation. The diagrammatic languages for monoidal categories, for example, also allow interpretations of formal proofs via topological deformations, and they have given semantic representations to various graphical languages like circuit diagrams and petri nets [20].

We also note that, while programming languages whose visual syntax evaluates to strings, means that all diagrams can in some sense be encoded in more traditional syntax, this is only for the computers sake - the human may consume the diagram as an abstract entity other than a string. There are often words to describe, but not to give visual intuition to many of the mathematical ideas we grasp. There are also, famously blind mathematicians who work in topology, geometry, and analysis [29]. Bernard Morin, blinded at a young age, was a topologist who discovered the first eversion of a sphere by using clay models which were then diagrammatically transcribed by a colleague on the board. This is a remarkable use of mathematical tools PL researchers cannot yet handle, and warrants careful consideration of what the boundaries of proof assistants are capable of in terms of giving mathematicians more tangible constructions.

For if there is one message one should take away from this thesis, it is that there needs to be a coming to terms in the mathematics and TT communities, of the difference between *a proof*, both formal and informal, and the *the understanding of a proof*. The first is a mathematical judgment where one supplies evidence, via the form of a term that Agda can type-check and verify. A NL proof can be reviewed by a human. The understanding of a proof, however, is not done by anything but a human. And this internal understanding and processing of mathematical information, what I'll tongue-and-cheek call i-mathematics, with its externalization facilities being our main concerns in this thesis, requires much more work by future scholars.

3 Technical Preliminaries

3.1 Martin-Löf Type Theory

3.1.1 Judgments

> With Kant, something important happened, namely, that the term judgement, Ger. Urteil, came to be used instead of proposition. *Per Martin-Löf* [37].

A central contribution of Per Martin-Löf in the development of type theory was the recognition of the centrality of judgments in logic. Many mathematicians aren't familiar with the spectrum of judgments available, and merely believe they are concerned with *the* notion of truth, namely *the truth* of a mathematical proposition or theorem. There are many judgments one can make which most mathematicians aren't aware of or at least never mention. Examples of both familiar and unfamiliar judgments include,

- A is true
- A is a proposition
- A is possible
- A is necessarily true
- A is true at time t

These judgments are understood not in the object language in which we state our propositions, possibilities, or probabilities, but as assertions in the metalanguage which require evidence for us to know and believe them. Most mathematicians may reach for their wallets if I come in and give a talk saying it is possible that the Riemann Hypothesis is true, partially because they already know that, and partially because it doesn't seem particularly interesting to say that something is possible, in the same way that a physicist may flinch if you say alchemy is possible. Most mathematicians, however, would agree that $P = NP$ is a proposition, and it is also possible, but isn't true.

For the logician these judgments may well be interesting because their may be logics in which the discussion of possibility or necessity is even more interesting than the discussion of truth. And for the type theorist interested in designing and building programming languages over many various logics, these judgments become a prime focus. The role of the type-checker in a programming language is to present evidence for, or decide the validity of the judgments. The four main judgments of type theory are given in natural language on the left and symbolically on the right :

- T is a type
- T and T' are equal types
- t is a term of type T
- t and t' are equal terms of type T

- $\vdash T$ type
- $\vdash T = T'$
- $\vdash t : T$
- $\vdash t = t' : T$

Frege's turnstile, $\vdash$, denotes a judgment. These judgments become much more interesting when we add the ability for them to be interpreted in a some context with judgment hypotheses. Given a series of judgments $J_1, ..., J_n$, denoted Γ, where J_i can depend on previously listed $J's$, we can make judgment J under the hypotheses, e.g. $J_1, ..., J_n \vdash J$. Often these hypotheses J_i, alternatively called *antecedents*, denote variables which may occur freely in the *consequent* judgment J. For instance, the antecedent, $x : \mathbb{R}$ occurs freely in the syntactic expression $\sin x$, a which is given meaning in the judgment $\vdash \sin x{:}\mathbb{R}$. We write our hypothetical judgement as follows :

$$x : \mathbb{R} \vdash \sin x : \mathbb{R}$$

3.1.2 Rules

Martin-Löf systematically used the four fundamental judgments in the proof theoretic style of Pragwitz and Gentzen. To this end, the intuitionistic formulation of the logical connectives just gives rules which admit an immediate computational interpretation. The main types of rules are type formation, introduction, elimination, and computation rules. The introduction rules for a type admit an induction principle derivable from that type's signature. Additionally, the β and η computation rules are derivable via the composition of introduction and elimination rules, which, if correctly formulated, should satisfy a relation known as harmony.

The fundamental notion of the lambda calculus, the function, is abstracted over a variable and returns a term of some type when applied to an argument which is subsequently reduced via the computational rules. Dependent Type Theory (DTT) generalizes this to allow the return type be parameterized by the variable being abstracted over. The dependent function forms the basis of the LF which underlies Agda and GF. Here is the formation rule :

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma, x : A \vdash B \text{ type}}{\Gamma \vdash \Pi x{:}A.B}$$

One reason why hypothetical judgments are so interesting is we can devise rules which allow us to translate from the metalanguage to the object language using lambda expressions. These play the role of a function in mathematics and implication in logic. More generally, this is a dependent type, representing the $\forall$ quantifier. Assuming from now on $\Gamma \vdash A$ type and $\Gamma, x : A \vdash B$ type, we present here the introduction rule for the most fundamental type in Agda, denoted `(x : A) -> B`.

$$\frac{\Gamma, x{:}A \vdash B \text{ type}}{\Gamma \vdash \lambda x.b{:}\Pi x{:}A.B}$$

Observe that the hypothetical judgment with $x : A$ in the hypothesis has been

reduced to the same hypothesis set below the line, with the lambda term and Pi type now accounting for the variable.

$$\frac{\Gamma \vdash f{:}\Pi x{:}A.B \qquad \Gamma \vdash a{:}A}{\Gamma \vdash f\,a{:}B[x := a]}$$

We briefly give the elimination rule for Pi, application, as well as the classic β and η computational equality judgments (which are actually rules, but it is standard to forego the premises):

$$\Gamma \vdash (\lambda x.b)\,a \equiv b[x := a]{:}B[x := a]$$

$$\Gamma \vdash (\lambda x.f)\,x \equiv f{:}\Pi x{:}A.B$$

Using this rule, we now see a typical judgment without hypothesis in a real analysis, $\vdash \lambda x.\sin x : \mathbb{R} \to \mathbb{R}$. This is normally expressed as follows (knowing full well that sin actually has to be approximated when saying what the computable function in the codomain is):

$$\begin{aligned} \sin &: \mathbb{R} \to \mathbb{R} \\ x &\mapsto \sin(x) \end{aligned}$$

Evaluating this function on 0, we see

$$\begin{aligned} (\lambda x.\sin x)\,0 &\equiv \sin 0 \\ &\equiv 0 \end{aligned}$$

While most mathematicians take this for granted, we hope this gives some insight into how computer scientists present functions. We recommend reading Martin-Löf's original papers [36] [38] to see all the rules elaborated in full detail, as well as his philosophical papers [37] [39] to understand type theory as it was conceived both practically and philosophically.

3.2 Propositions, Sets, and Types

> Treating propositions as types is definitely not in the way of thinking of ordinary mathematician, yet it is very close to what he actually does. *N. G. de Bruijn* [18].

While the rules of type theory have been well-articulated elsewhere, we provide briefly compare the syntax of mathematical constructions in FOL, one possible natural language use [49], and MLTT. From this vantage, these look like simple symbolic manipulations, and in some sense, one doesn't need a the expressive power of system like GF to parse these to the same form.

Additionally, it is worth comparing the type theoretic and natural language syntax with set theory, as is done in Figure 8 and Figure 9. Now we bear witness to some deeper cracks than were visible above. We note that the type theoretic syntax

is *the same* in both tables, whereas the set theoretic and logical syntax shares no overlap. This is because set theory and first order logic are distinct domains classically, whereas in MLTT, there is no distinguishing mathematical types from logical types - everything is a type.

FOL	MLTT	NL FOL	NL MLTT
$\forall\ x\ P(x)$	$\Pi x : \tau.\ P(x)$	$for\ all\ x,\ p$	$the\ product\ over\ x\ in\ p$
$\exists\ x\ P(x)$	$\Sigma x : \tau.\ P(x)$	$there\ exists\ an\ x\ such\ that\ p$	$there\ exists\ an\ x\ in\ \tau such\ that\ p$
$p\ \supset\ q$	$p\ \rightarrow\ q$	$if\ p\ then\ q$	$p\ to\ q$
$p\ \wedge\ q$	$p\ \times\ q$	$p\ and\ q$	$the\ product\ of\ p\ and\ q$
$p\ \vee\ q$	$p\ +\ q$	$p\ or\ q$	$the\ coproduct\ of\ p\ and\ q$
$\neg\ p$	$\neg\ p$	$it\ is\ not\ the\ case\ that\ p$	$not\ p$
$\top$	$\top$	$true$	top
$\bot$	$\bot$	$false$	$bottom$
$p\ =\ q$	$p\ \equiv\ q$	$p\ equals\ q$	$definitionally\ equal$

Figure 8: FOL vs MLTT

We show the Type and set comparisons in Figure 9. The basic types are sometimes simpler to work with because they are expressive enough to capture logical and set theoretic notions, but this also comes at a cost. The union of two sets simply gives a predicate over the members of the sets, whereas union and intersection types are often not considered "core" to type theory, with multiple possible ways of interpreting how to treat this set-theoretic concept. The behavior of subtypes and subsets, while related in some ways, also represents a semantic departure from sets and types. For example, while there can be a greatest type in some sub-typing schema, there is no notion of a top set. This is why we use the type theoretic NL syntax when there are question marks in the set theory column.

Set Theory	MLTT	NL Set Theory	NL MLTT
S	τ	$the\ set\ S$	$the\ type\ \tau$
$\mathbb{N}$	Nat	$the\ set\ of\ natural\ numbers$	$the\ type\ nat$
$S \times T$	$S \times T$	$the\ product\ of\ S\ and\ T$	$the\ product\ of\ S\ and\ T$
$S \rightarrow T$	$S \rightarrow T$	$the\ function\ \ S\ to\ T$	$p\ to\ q$
$\{x \vert P(x)\}$	$\Sigma x : \tau.\ P(x)$	$the\ set\ of\ x\ such\ that\ P$	$there\ exists\ an\ x\ in\ \tau such\ that\ p$
$\emptyset$	$\bot$	$the\ empty\ set$	$bottom$
?	$\top$	?	top
$S \cup T$	?	$the\ union\ of\ S\ and\ T$	?
$S \subset T$	$S <: T$	$S\ is\ a\ subset\ of\ T$	$S\ is\ a\ subtype\ of\ T$
?	U_1	?	$the\ second\ Universe$

Figure 9: Sets vs MLTT

We also note that pragmatically, type theorists often interchange the logical, set theoretic, and type theoretic lexicons when describing types. Because the types were developed to overcome shortcomings of set theory and classical logic, the lexicons of all three ended up being blended, and in some sense, the type theorist can substitute certain words that a classical mathematician wouldn't. Whereas

$p\ implies\ q$ and $function\ from\ X\ to\ Y$ are not to be mixed, the type theorist may in some sense default to either. Nonetheless, pragmatically speaking, one would never catch a type theorist saying $NatimpliesNat$ when expressing $Nat \rightarrow Nat$.

Terms become even messier, and this can be seen in just a small sample shown in Figure 10. In simple type theory, one distinguishes between types and terms at the syntactic level - this disappears in DTT. As will be seen later, the mixing of terms and types gives MLTT an incredible expressive power, but undoubtedly makes certain things very difficult as well. In set theory, everything is a set, so there is no distinguishing between elements of sets and sets even though practically they function very differently. Mathematicians only use sets because of their flexibility in so many ways, not because the axioms of set theory make a compelling case for sets being this kind of atomic form that makes up the mathematical universe. Category theorists have discovered vast generalizations of sets (where elements are arrows) which allow one to have the flexibility in a more structured and nuanced way, and the comparison with categories and types is much tighter than with sets. Regardless, mathematicians day to day work may not need all this general infrastructure.

In FOL, terms don't exist at all, and the proof rules themselves contain the necessary information to encode the proofs or constructions. The type theoretic terms somehow compress and encode the proof trees, of which, and in the case of ITPs nodes are displayed during the interactive type-checking phase.

Set Theory	MLTT	NL Set Theory	NL MLTT	Logic
$f(x) := p$	$\lambda x.p$	$f\ of\ x\ is\ p$	$lambda\ x,\ p$	$\supset -elim$
$f(p)$	fp	$fofp$	$the\ application\ of\ f\ to\ p$	$modus\ ponens$
(x, y)	(x, y)	$the\ pair\ of\ x\ and\ y$	$the\ pair\ of\ x\ and\ y$	$\wedge - i$
$\pi_{1,2}\ x$	$\pi_{1,2}\ x$	$the\ first\ projection\ of\ x$	$the\ first\ projection\ of\ x$	$\wedge - e$

Figure 10: Term syntax in Sets, Logic, and MLTT

We don't do all the constructors for type theory here for space, but note some interesting features:

- The disjoint union in set theory is actually defined using pairs - and therefore it doesn't have elimination forms other than those for the product. The disjoint union is also not nearly as ubiquitous, though.
- λ is a constructor for both the dependent and non-dependent function, so its use in either case will be type-checked by Agda, whereas it's natural language counterpart in real mathematics will have syntactic distinction.
- The projections for a Σ type behaves differently from the elimination principle for $\exists$, and this leads to incongruities in the natural language presentation.

Finally, we should note that there are many linguistic presentations mathematicians use for logical reasoning, i.e. the use of introduction and elimination rules. They certainly seem to use linguistic forms more when dealing with proofs, and symbolic notation for Sets, so the investigation of how these translate into type

theory is a source of future work. Whereas propositions make explicit all the relevant detail, and can be read by non-experts, proofs are incredibly diverse and will be incomprehensible to those without expertise.

A detailed analysis of this should be done if and when a proper translation corpus is built to account for some of the ways mathematicians articulate these rules, as well as when and how mathematicians discuss sets, symbolically and otherwise. To create translation with "real" natural language is likely not to be very effective or interesting without a lot of evidence about how mathematicians speak and write.

3.3 Agda

3.3.1 Overview

Agda is an attempt to faithfully formalize Martin-Löf's intensional type theory [36]. Referencing our previous distinction, one can think of Martin-Löf's original work as a specification, and Agda as one possible implementation.

Agda is a functionaly programming language which, through an interactive environment, allows one to iteratively apply rules and develop constructive mathematics. It's current incarnation, Agda2 (but just called Agda), was preceded by ALF, Cayenne, and Alfa, and the Agda1. On top of the basic MLTT, Agda incorporates dependent records, inductive definitions, pattern matching, a versatile module system, and a myriad of other bells and whistles which are of interest generally and in various states of development but not relevant to this work.

For our purposes, we will only look at what can in some sense be seen as the kernel of Agda. Developing a full-blown GF grammar to incorporate more advanced Agda features would require efforts beyond the scope of this work.

Agda's purpose is to manifest the propositions-as-types paradigm in a practical and useable programming language. And while there are still many reasons one may wish to use other programming languages, or just pen and paper to do her work, there is a sense of purity one gets when writing Agda code. There are many good resources for learning Agda [7] [57] [8] [60] so we'll only give a cursory overview of what is relevant for this thesis, with a particular emphasis on the syntax.

3.3.2 Agda Programming

To give a brief overview of the syntax Agda uses for judgements, namely `T : Set` means `T` is a type, `t : T` means a term `t` has type `T`, and `t = t'` means `t` is defined to be judgmentally equal to `t'`. Once one has made this equality judgement, agda can normalize the definitionally equal terms to the same normal form in downstream programs. Let's compare it these judgements to those keywords ubiquitous in mathematics, and show how those are represented in Agda directly below.

Formation rules, are given by the first line of the data declaration, followed by some number of constructors which correspond to the introduction forms of the

- Axiom
- Definition
- Lemma
- Theorem
- Proof
- Corollary
- Example

```
postulate -- Axiom
  axiom : A

definition : stuff → Set --Definition
definition s = definition-body

theorem : T -- Theorem Statement
theorem = proofNeedingLemma lemma -- Proof
  where
    lemma : L -- Lemma Statement
    lemma = proof

corollary : corollaryStuff → C
corollary coro-term = theorem coro-term

example : E -- Example Statement
example = proof
```

Figure 11: Mathematical Assertions and Agda Judgements

type being defined. Therefore, to define a type for Booleans, 𝔹, we present these rules both in the proof theoretic and Agda syntax.

$$\frac{}{\vdash \mathbb{B} : \text{type}}$$

$$\frac{}{\Gamma \vdash true : \mathbb{B}} \qquad \frac{}{\Gamma \vdash false : \mathbb{B}}$$

```
data 𝔹 : Set where -- formation rule
  true : 𝔹 -- introduction rule
  false : 𝔹
```

As the elimination forms are deriveable from the introduction rules, the computation rules can then be extracted by via the harmonious relationship between the introduction and elmination forms [42]. Agda's pattern matching is equivalent to the deriveable dependently typed elimination forms [14], and one can simply pattern match on a boolean, producing multiple lines for each constructor of the variable's type, to extract the classic recursion principle for Booleans.

When using Agda one is working interactively via holes in the emacs mode, and that once one plays around with it, one recognizes both the beauty and elegance in how Agda facilitates programming. We don't include the eqaulity rules as rules because they redundantly use the same premises as the typing judgment. Below we show the elimination and equality rules alongside the Agda version.

$$\frac{\Gamma \vdash A : \text{type} \quad \Gamma \vdash b : \mathbb{B} \quad \Gamma \vdash a1 : A \quad \Gamma \vdash a2 : A}{\Gamma \vdash boolrec\{a1; a2\}(b) : A}$$

$$\Gamma \vdash boolrec\{a1; a2\}(true) \equiv a1 : A$$

$$\Gamma \vdash boolrec\{a1; a2\}(false) \equiv a2 : A$$

```
if_then_else_ :
  {A : Set} → 𝔹 → A → A → A
if true then a1 else a2 = a1
if false then a1 else a2 = a2
```

The underscore denotes the placement of the arguement, as Agda allows mixfix operations. `if_then_else_` function allows for more nuanced syntacic features out of the box than most programming languages provide, like unicode. This is interesting from the *concrete syntax* perspective as the arguement placement, and symbolic expressiveness gives Agda a syntax more familiar to the mathematician. We also observe the use of parametric polymorphism, namely, that we can extract a member of some arbtitrary type `A` from a boolean value given two members of `A`.

This polymorphism allows one to implement simple programs like the two equivalent boolean negation function, `~-elimRule` and `~-patternMatch`. More interestingly, one can work with functionals, or higher order functions which take functions as arguements and return functions as well. We also notice in `functionalExample` below that one can work directly with lambda's if the typechecker infers a function type for a hole.

```
~-elimRule : 𝔹 → 𝔹
~-elimRule b = if b then false else true

~-patternMatch : 𝔹 → 𝔹
~-patternMatch true = false
~-patternMatch false = true

functionalNegation : 𝔹 → (𝔹 → 𝔹) → (𝔹 → 𝔹)
functionalNegation b f = if b then f else λ b' → f (~-patternMatch b')
```

This simple example leads us to one of the domains our subsequent grammars will describe, arithmetic. We show how to inductively define natural numbers in Agda, with the formation and introduction rules included beside for contrast.

$$\frac{}{\vdash \mathbb{N} : \text{type}}$$

$$\frac{}{\Gamma \vdash 0 : \mathbb{N}} \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash (sucn) : \mathbb{N}}$$

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
```

This is our first observation of a recursive type, whereby the pattern matching over $\mathbb{N}$ allows one to use an induction hypothesis over the subtree and gurantee termination when making recurive calls on the function being defined. We can define a recursion principle for $\mathbb{N}$, which essentially gives one the power to build iterators, i.e. for-loops. Again, we include the recursion rule elimination and equality rules for syntactic juxtaposition.

$$\frac{\Gamma \vdash X : \mathsf{type} \quad \Gamma \vdash n : \mathbb{N} \quad \Gamma \vdash e_0 : X \quad \Gamma, x : \mathbb{N}, y : X \vdash e_1 : X}{\Gamma \vdash natrec\{e\ x.y.e_1\}(n) : X}$$

$$\Gamma \vdash natrec\{e_0; x.y.e_1\}(n) \equiv e_0 : X$$

$$\Gamma \vdash natrec\{e_0; x.y.e_1\}(suc\ n) \equiv e_1[x := n, y := natrec\{e_0; x.y.e_1\}(n)] : X$$

```
natrec : {X : Set} → ℕ → X → (ℕ → X → X) → X
natrec zero e₀ e₁ = e₀
natrec (suc n) e₀ e₁ = e₁ n (natrec n e₀ e₁)
```

Since we are in a dependently typed setting, however, we prove theorems as well as write programs. Therefore, we can see this recursion principle as a special case of the induction principle, which is the classic proof by induction for natural numbers. One may notice that while the types are different, the programs `natrec` and `natind` are actually the same, up to α-equivalence. One can therefore, as a corollary, actually just include the type infomation and Agda can infer the speciliazation for you, as seen in `natrec'` below.

$$\frac{\Gamma, x : \mathbb{N} \vdash X : \mathsf{type} \quad \Gamma \vdash n : \mathbb{N} \quad \Gamma \vdash e_0 : X[x := 0] \quad \Gamma, y : \mathbb{N}, z : X[x := y] \vdash e_1 : X[x := suc\ y]}{\Gamma \vdash natind\{e_0,\ x.y.e_1\}(n) : X[x := n]}$$

$$\Gamma \vdash natinde_0; x.y.e_1\}(n) \equiv e_0 : X[x := 0]$$

$$\Gamma \vdash natinde_0; x.y.e_1\}(suc\ n) \equiv e_1[x := n, y := natind\{e_0; x.y.e_1\}(n)] : X[x := suc\ n]$$

```
natind : {X : ℕ → Set} → (n : ℕ) → X zero → ((n : ℕ) → X n → X (suc n)) → X n
natind zero base step = base
natind (suc n) base step = step n (natind n base step)

natrec' : {X : Set} → ℕ → X → (ℕ → X → X) → X
natrec' = natind
```

We will defer the details of using induction and recursion principles for later sections, when we actually give examples of pidgin proofs some of our grammars can handle. For now, the keen reader should try using Agda.

3.3.3 Formalizing The Twin Prime Conjecture

Inspired by Escardos's formalization of the twin primes conjecture [19], we intend to demonstrate that while formalizing mathematics can be rewarding, it can also create immense difficulties, especially if one wishes to do it in a way that prioritizes natural language. The conjecture is incredibly compact

Lemma 1 *There are infinitely many twin primes.*

Somebody reading for the first time might then pose the immediate question : what is a twin prime?

Definition 5 *A* twin prime *is a prime number that is either 2 less or 2 more than another prime number*

Below Escardo's code is reproduced.

```
isPrime : ℕ → Set
isPrime n =
  (n ≥ 2) ×
  ((x y : ℕ) → x * y ≡ n → (x ≡ 1) + (x ≡ n))

twinPrimeConjecture : Set
twinPrimeConjecture = (n : ℕ) → Σ[ p ∈ ℕ ] (p ≥ n)
  × isPrime p
  × isPrime (p + 2)
```

We note there are some both subtle and big differences, between the natural language claim. First, twin prime is defined implicitly via a product expression, ×. Additionally, the "either 2 less or 2 more" clause is oringially read as being interpreted as having "2 more". This reading ignores the symmetry of products, however, and both "p or (p + 2)" could be interpreted as the twin prime. This phenomenon makes translation highly nontrivial; however, we will later see that PGF is capable of adding a semantic layer where the theorem can be evaluated during the translation. Finally, this theorem doesn't say what it is to be infinite in general, because such a definition would require a proving a bijection with the real numbers. In this case however, we can rely on the order of the natural numbers, to simply state what it means to have infinitely many primes.

Despite the beauty of this, mathematicians always look for alternative, more general ways of stating things. Generalizing the notion of a twin prime is a prime gap. And then one immediately has to ask what is a prime gap?

Definition 6 *A* twin prime *is a prime that has a prime gap of two.*

Definition 7 *A* prime gap *is the difference between two successive prime numbers.*

Now we're stuck, at least if you want to scour the internet for the definition of "two successive prime numbers". That is because any mathematician will take for granted what it means, and it would be considered a waste of time and space to include something *everyone* alternatively knows. Agda, however, must know in order to typecheck. Below we offer a presentation which suits Agda's needs, and matches the number theorists presentation of twin prime.

```
isSuccessivePrime : (p p' : ℕ) → isPrime p → isPrime p' → Set
isSuccessivePrime p p' x x₁ =
  (p'' : ℕ) → (isPrime p'') →
```

```
    p ≤ p' → p ≤ p'' → p' ≤ p''
primeGap :
  (p p' : ℕ) (pIsPrime : isPrime p) (p'IsPrime : isPrime p') →
  (isSuccessivePrime p p' pIsPrime p'IsPrime) →
  ℕ
primeGap p p' pIsPrime p'IsPrime p'-is-after-p = p - p'

twinPrime : (p : ℕ) → Set
twinPrime p =
  (pIsPrime : isPrime p) (p' : ℕ) (p'IsPrime : isPrime p')
  (p'-is-after-p : isSuccessivePrime p p' pIsPrime p'IsPrime) →
  (primeGap p p' pIsPrime p'IsPrime p'-is-after-p) ≡ 2

twinPrimeConjecture' : Set
twinPrimeConjecture' = (n : ℕ) → Σ[ p ∈ ℕ ] (p ≥ n)
  × twinPrime p
```

We see that `isSuccessivePrime` captures this meaning, interpreting "successive" as the type of suprema in the prime number ordering. We also see that all the primality proofs must be given explicitly.

The term `primeGap` then has to reference this successive prime data, even though most of it is discarded and unused in the actual program returning a number. One could keep these unused arguements around via extra record fields, to anticipate future programs calling `primeGap`, but ultimately the developer has to decide what is relevant. A GF translation would ideally be kept as simple as possible. We also use propositional equality here, which is another departure from classical mathematics, as will be elaborated later.

Finally, {twinPrime is a specialized version of `primeGap` to 2. "has a prime gap of two" needs to be interpreted "whose prime gap is equal to two", and writing a GF grammar capable of disambiguating *has* in mathematics generally is likely impossible. One can also uncurry much of the above code to make it more readable, which we include in the appendix 9.1.

While working on this example, I tried to prove that 2 is prime in Agda with this defintion. It turned out to be nontrivial. When I told this to an analyst (in the mathematical sense) he remarked that couldn't possibly be the case because it's something which a simple algorithm can compute (or generate). This exchange was incredibly stimulating, for the mathematian didn't know about the *propositions as types* principle, and was simply taking for granted his internal computational capacity to confuse it for proof, especially in a constructive setting. He also seemed perplexed that anyone would find it interesting to prove that 2 is prime. The proof that 2 is prime, via Agda's standard libary, is done via reflection - a way of quoting a term into in abstract syntax tree and then performing some kind of metacomputation. While elegant, this obviously requires a lot of machinery, none of which would be easy to communicate to a mathematician who doesn't know much about coding. As is hopefully revealed by this discussion, seemingly trivial things, when treated by the type theorist or linguist, can become wonderful areas of exploration.

4 Previous Work

There is a story that at some point in the 1980s, Göran Sundholm and Per Martin-Löf were sitting at a dinner table, dicsussing various questions of interest to the respective scholars, and Sundholm presented Martin-Löf with the problem of Donkey Sentences in natural language semantics, those analogous 'Every man who owns a donkey beats it'. This had been puzzling to those in the Montague tradition, whereby higher order logic didn't provide facile ways of interpreting these sentences. Martin-Löf apparently then, using his dependent type constructors, provided an interpretation of the donkey sentence on the back of the napkin. This is perhaps the genesis of dependent type theory in natural language semantics. The research program was thereafter taken up by Martin-Löf's student Aarne Ranta [45], bled into the development of GF, and has now in some sense led to this current work.

The prior exploration of these interleaving subjects is vast, and we can only sample the available literature here. Indeed, there are so many approaches that this work should be seen in a small (but important) case in the context of a deep and broad literature [31]. Acquiring expertise in such a breadth of work is outside the scope of this thesis. Our approach, using GF ASTs as a basis language for Mathematics and the logic the mathematical objects are described in, is both distinct but has many roots and interconnections with the remaining literature. The success of finding a suitable language for mathematics will obviously require a comparative analysis of the strengths and weaknesses in the goals in such a vast bibliography. How the GF approach compares with this long merits careful consideration and future work.

It will function of our purpose, constrained by the limited scope of this work, to focus on a few important resources.

4.1 Ranta

The initial considerations of Ranta were both oriented towards the language of mathematics [46], as well as purely linguistic concerns [45]. In the treatise, Ranta explored not just the many avenues to describe NL semantic phenomena with Dependent Types, but, after concentrating on a linguistic analysis, he also proposed a primitive way of parsing and sugaring these dependently typed interpretations of utterances into the strings themselves - introducing the common nouns as types idea which has been since seen great interest from both type theoretic and linguistic communities [34]. Therefore, if we interpret the set of men and the set of donkeys as types, e.g. we judge $\vdash man$:type and $\vdash donkey$:type where type really denotes a universe, and ditransitive verbs "owns" and "beats" as predicates, or dependent types over the CN types, i.e. $\vdash owns : man \to donkey \to$ type we can interpret the sentence "every man who owns a donkey beats it" in DTT via the following judgment :

$$\Pi z : (\Sigma x : man.\ \Sigma y : donkey.\ owns(x, y)).\ beats(\pi_1 z, \pi_1(\pi_2 z))$$

We note that the natural language quantifiers, which were largely the subject of Montague's original investigations [40], find a natural interpretation as the dependent product and sum types, Π and Σ, respectively. As type theory is constructive, and requires explicit witnesses for claims, we admit the behavior following semantic interpretation : given a man m, a donkey d and evidence $m - owns - d$ that the man owns the donkey, we can supply, via the term of the above type applied to our own tripple $(m, d, m - owns - d)$, evidence that the man beats the donkey, $beats(m, d)$ via pi_1 and pi_2, the projections, or Σ eliminators.

In the final chapter of [45], $SugaringandParsing$, Ranta explores the explicit relation, and of translation between the above logical form and the string, where he presents a GF predecessor in the Alfa proof assistant, itself a predecessor of Agda. To accomplish this translation he introduces an intermediary , a functional phrase structure tree, which later becomes the basis for GFs abstract syntax. What is referred to as "sugaring" later changes to "linearization".

Soon thereafter, GF became a fully realized vision, with better and more expressive parsing algorithms [33] developed in Göteborg allowed for sugaring that can largely accommodate morphological features of the target natural language [21], the translation between the functional phrase structure (ASTs) and strings [47].

Interestingly, the functions that were called $ambiguation : MLTT \to \{Phrase\ Structure\}$ and $interpretation : \{PhraseStructure\} \to MLTT$ were absorbed into GF by providing dependently typed ASTs, which allows GF not just to parse syntactic strings, but only parse semantically well formed, or meaningful strings. Although this feature was in some sense the genesis that allowed GF to implement the lingusitic ideas from the book [50], it has remained relatively limited in terms of actual GF programmers using it in their day to day work. Nonetheless, it was intriguing enough to investigate briefly during the course of this work as one can implement a programming language grammar that only accepts well typed programs, at least as far as they can be encoded via GF's dependent types [35]. Although GF isn't designed with TypeChecking in mind explicity, it would be very interesting to apply GF dependent types in the more advanced programming languages to filter parses of meaningless strings.

While the semantics of natural language in MLTT is relevant historically, it is not the focus of this thesis. Its relevance comes from the fact that all these ideas were circulating in the same circles - that is, Ranta's writings on the language of mathematics, his approach to NL semantics, and their confluence among other things, with the development of GF. This led to the development of a natural language layer to Alfa [27], which in some sense can be seen as a direct predecessor to this work. In some sense, the scope of work seeks to recapitulate what was already done in 1998 - but this was prior to both GF's completion, and Alfa's hard fork to Agda.

4.2 Mohan Ganesalingam

> there is a considerable gap between what mathematicians claim is true and what they believe, and this mismatch causes a number of serious linguistic problems

Perhaps the most substantial analysis of the linguistic perspective on written mathematics comes from Ganesalingam [23]. Not only does he pick up and reexamine much of Ranta's early work, but he develops a whole theory for how to understand with the language mathematics from a formal point of view, additionally working with many questions about the foundation of mathematics. His model which is developed early in the treatise and is referenced throughout uses Discourse Representation Theory [32], to capture anaphoric use of variables. While he is interested in analyzing language, our goal is to translate, because the meaning of an expression is contained in its set of formalizations, so our project should be thought of as more of a way to implement the linguistic features of language rather than Ganesalingam's work analyzing the infrastructure of natural language mathematics.

Gangesalingem draws insightful, nuanced conclusions from compelling examples. Nonetheless, this subject is somewhat restricted to a specific linguistic tradition and framing and modern, textual mathematics. Therefore, we hope to (i) contrast our GF implementation point of view and (ii) offer some perspectives on his work.

He remarks that mathematicians believe "insufficiently precise" mathematical sentences are would be results from a failure to into logic. This is much more true from the Agda developers perspective, than the mathematicians. It is likely there are many mathematicians who assume small mistakes may go by the reviewers unchecked, as are the reviewers. However, the Brunei number offers a counterexample even the computer scientist has to come to terms with - because it's based off pen and paper work which hasn't terminated in Agda [9]. And while this one example which may see resolution, one may construct other which won't, and it is speculative to think what mathematics is formalizable.

Gangesalingem also articulates "mathematics has a normative notion of what its content should look like; there is no analogue in natural languages." While this is certainly true in *local* cases surrounding a given mathematical community , there are also many disputes - the Brouwer school is one example, but our prior discussion of visual proofs also offers another counterexample. Additionally, the "GF perspective" presented here is meant to disrupt the notion of normativity, by suggesting that concrete syntax can reflect deep differences in content beyond just its appearance. Escher's prints, alternatively uniquely mirror both mathematics and art - they are constructions using rules from formal systems, but are appreciated by a general audience.

He also discusses the important distinction between formal (which he focuses on) and informal modes in mathematics, with the informal representing the text which is a kind of commentary which is assumed to be inexpressible in logic. GF, fortunately can actually accommodate both if one considers only natural language

translation in the informal case. This is interesting because one would need extend a “formal grammar” with the general natural language content needed to include the informal.

He says symbols serve to “abbreviate material”, and “occur inside textual mathematics”. While his discourse records can deal with symbols, in GF, overloading of symbols can cause overgeneration. For example certain words like ”is” and ”are” can easily be interpreted as equality, equivalence, or isomorphism depending on the context.

One of Ganesalingam’s original contributions, is the notion of adaptivity : “Mathematical language expands as more mathematics is encountered”. He references someones various stages of coming to terms with concepts in mathematics and their generalization in somebodies head. For instance, one can define the concept of the n squared as n^2 of two as ”n*n”, which are definitionally equal in Agda if one is careful about how one defines addition, multiplication, and exponentiation. Writing grammars, on has to cater the language to the audience, for example, which details does one leave out when generating natural language proofs?

Mathematical variables, it is also noticed, can be treated anaphorically. From the PL persepctive they are just expressions. Creating a suitable translation from textual math to formal languages accounting for anaphora with GF proves to be exceedingly tricky, as can be seen in the HoTT grammar below.

4.2.1 Pragmatics in mathematics

Ganesalingam makes one observation which is particularly pertinent to our analysis and understanding of mathematical language, which is that of pragmatics content. The point warranted both a rebuttal [55] and an additional response by Ranta [53]. Ganesalingam says “mathematics does not exhibit any pragmatic phenomena: the meaning of a mathematical sentence is merely its compositionally determined semantic content, and nothing more. ”. We explore these fascinating dialogues here, adjoining our own take in the context of this project.

San Mauro et al. disagree with this conception, stating mathematicians may rely “on rhetorical figures, and speak metaphorically or even ironically”, and that mathematicians may forego literal meaning if considered fruitful. The authors then give two technical examples of pragmatic phenomena where pragmatics is explicitly exhibited, but we elect to give our own example relevant for our position on the matter.

We look ask what is the difference in meaning between lemma, proof, and corollary. While there is a syntactic distinction between `Lemma` and `Theorem` in Coq, Agda which resembles Haskell rather than a theorem prover at a first glance, sees no distinction as seen in Figure 11. The words carry semantic weight : *lemma* for concepts preceding theorems and *corollaries* for concepts applying theorems. The interpretation of the meaning when a lemma or corollary is called a carry pragmatic content in that the author has to decide how to judge the content by its importance, and relation of them to the *theorems* in some kind of natural ways.

Inferring how to judge a keyword seems impossible for a machine, especially since critical results are perhaps misnamed the Yoneda Lemma is just one of many examples.

Ranta categorizes pragmatic phenomena in 5 ways : speech acts, context, speaker's meaning, efficient communication, and the *wastebasket*. He asserts that the disagreement is really a matter of how coarsely pragmatics is interpreted by the authors - Ganesalingam applies a very fine filter in his study of mathematical language, whereas the coarser filter applied by San Mauro et al. allows for many more pragmatics phenomena to be captured, and that the "wastebasket" category is really the application of this filter. Ranta shows that both Speech Acts and Context are pragmatic phenomena treated in Ganesalingam's work and speaker's meaning and efficient communication are in covered by San Mauro et al., and that the authors disagreement arises less about the content itself and how it is analyzed, but rather whether the analysis should be classified as pragmatic or semantic.

Our Grammars give us tools to work with the speaker's meaning of a mathematical utterance by a translation into syntactically complete Agda judgment (assuming it type-checks). Dually, efficient communication is the goal of producing a semantically adequate grammar. The task of creating a grammar which satisfies both is obviously the most difficult task before future grammar writers. We therefore hope that the modeling of natural language mathematics via the grammars presented will give insights into how understanding of all five pragmatic phenomena are necessary for good grammatical translations between CNLs and formal languages. For the CNLs to really be "natural", one must be able to infer and incorporate the pragmatic phenomena discussed here, and indeed much more.

Ganesalingam points out that "a disparity between the way we think about mathematical objects and the way they are formally defined causes our linguistic theories to make incorrect predictions." This constraint on our theoretical understanding of language, and the practical implications yield a bleak outlook. Nevertheless, mathematical objects developing over time is natural, the more and deeper we dig into the ground, the more we develop refinements of what kind of tools we are using, develop better iterations of the same tools (or possibly entirely new ones) as well as knowledge about the soil in which we are digging.

4.3 Other authors

> QED is the very tentative title of a project to build a computer system that effectively represents all important mathematical knowledge and techniques. [25]

The ambition of the QED Manifesto, with formalization and informalization of mathematics being a subset, is probably impossible. The myriad attempts at formalization and informalization are too much to compress here - a survey and comparison of these ideas is unfortunately unavailable. We recount some of them briefly.

The Naproche project (Natural language Proof Checking) is a CNL for studying the language of mathematics by using Proof Representation Structures, a mutated form of Discourse Representation Structures [16]. A central goal of Naproche is to develop a controlled natural language (CNL), based off FOL, for mathematics texts. It parses a theorem from the CNL into fully formal statement, and then comes with a proof checking back-end to allow verification, where it uses an Automated Theorem Prover (ATP) to check for correctness. While the language is quite "natural looking", it doesn't offer the same linguistic flexibility as our GF approach and aspirations.

Mizar is a system attempting to be a formal language,which mathematicians can use to express their results, and a database [54]. It is based off Tarski-Grothendeick set theory, and allows for correctness checking of articles. It was originally developed concurrent to Martin-Löf's work in 1973, and so much of the interest in types instead of sets couldn't be anticipated. The focus Mizar on syntax resembling mathematics was pioneering, nonetheless, it uses clumsy references and looks unreadable to those without expertise. Mizar has a journal devoted to results in it, *Formalized Mathematics*, and offers a large library of known results. Additionally, it has inspired iterations for other vernacular proof assistants, like Isabelle's Intelligible semi-automated reasoning (Isar) extension [62].

Subsequently, in [61], the authors take a corpus of parallel Mizar proofs natural language proofs with latex, and seek to *autoformalize* natural language text with the intention of, in the future, further elaboration into an ITP. This work uses traditional language models from the machine learning community, and analyze the results. They were able to see some results, but nothing that as of yet can be foreseen to general use. Interestingly, a type elaboration mechanism in some of their models was shown to bolster results.

Formalization seems more feasible with machine learning methods than informalization , partially because tactics like "hammer" in Coq for example, are capable of some fairly large proofs [17] . Nonetheless, for the Agda developer this isn't yet very relevant, and it's debatable whether it would even be desirable. Voevodsky, for example, was apparently skeptical of the usefulness of automated theorem proving for much of mathematics, as are many mathematicians (although this is certainly changing).

The Boxer system, a CCG parser [6] which allows English text translation into FOL. However, it is not always correct, and dealing with the language of mathematics will present obstacles.

In [15] the authors test the informalization. Despite working with Coq, the the authors poignantly distinguish between proof scripts, sequences of tactics, and proof objects, and focus on natural deduction proofs. Since Coq is equipped with notions of Set, Type, and Prop, their methods make distinguishing between these possibly easier. This work only focuses on linearization of trees, and GF's pretty printer is likely superior to any NL generation techniques because of help from the Resource Grammar Library (RGL). The complexity of the system also made it untenable for larger proofs - nonetheless, it serves as an important prelude to

many of the subsequent GF developments in this area.

There are many other examples worth exploring in the natural language and theorem prover boundary. It should be noted that GF's role in this space is primitive, but it does offer the advantage of providing interface for natural languages and programming languages. We also hope other PL developers will and use and develop tools like the Grammatical Logical Inference Framework (GLIF), which uses GF as a front-end for the Meta-Meta-Theory framework [56]. With many approaches are not mentioned here, we a hungry reader should evaluate these many sources with respect to this work.

5 Grammatical Framework

5.1 Thinking about GF

A grammar specification in GF is actually just an abstract syntax. With an abstract syntax specified, one can then define various linearization rules which compositionally evaluate to strings. An Abstract Syntax Tree (AST) may then be linearized to various strings admitted by different concrete syntaxes. Conversely, given a string admitted by the language being defined, GF's powerful parser will generate all the ASTs which linearize to that tree.

When defining a GF pipeline, one has to merely to construct an abstract syntax file and a concrete syntax file such that they are coherent. In the abstract, one specifies the *semantics* of the domain one wants to translate over, which is ironic, because we normally associate abstract syntax with *just syntax*. However, because GF was intended for implementing the natural language phenomena, the types of semantic categories (or sorts) can grow much bigger than is desirable in a programming language, where minimalism is generally favored. The *foods grammar* is the *hello world* of GF, and should be referred to for those interested in example of how the abstract syntax serves as a semantic space in non-formal NL applications [48].

Let us revisit the "tetrahedral doctrine", now restricting our attention to the subset of linguistics which GF occupies. We first examine how GF fits into the trinity, as seen in Figure 12. Immediately, GF abstract syntax with dependent types can just be seen as an implementation of MLTT with the added bonus of a parser. Additionally, GF is a relatively tame Type Theory, and therefore it would be easy to construct a model in a general purpose programming language, like Agda. Embeddings of GF already exist in Coq [cite FraCoq], Haskell [cite pgf], and MMT [cite mmt], These applications allow one to use GF's parser so that a GF AST may be transformed into some kind of inductively defined tree these languages all support. Future work could involve modeling GF in Agda would allow one to prove things about GF meta-theorems about soundness and termination, or perhaps statements about specific grammars, such as one being unambiguous.

From the logical side, we note that GF's parser specification was done using inference rules [cite krasimir]. Given the coupling of Context-Free Grammars (CFGs) and operads (also known as multicategories) [cite lambek, etc], one could use much more advanced mathematical machinery to articulate and understand GF. We sketch this briefly below [refer].

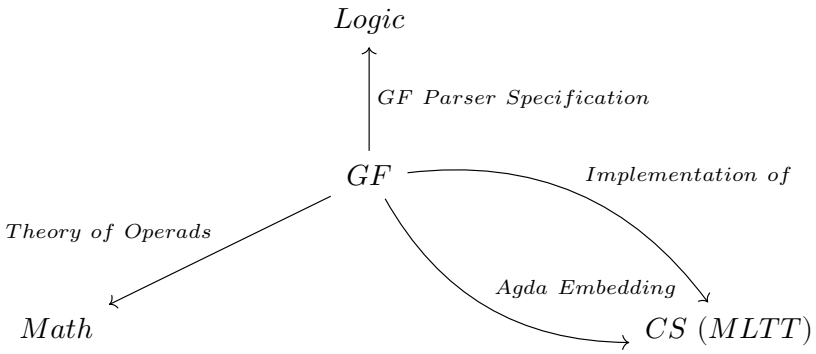

Figure 12: Models of GF

One can additionally model these domains in GF, which is obviously the main focus of this work. In Figure 13, we see that there are 3 grammars which give allow one to translate in these domains. Ranta's grammar from CADE 2011, built a propositional framework with a core grammar extended with other categories to capture syntactic nuance. Ranta's grammar from the Stockholm University mathematics seminar in 2014 took verbatim text from a publication of Peter Aczel and sought to show that all the syntactic nuance by constructing a grammar capable of NL translation. Finally, our work takes a BNFC grammar for a real programming language cubicaltt [cite], GFifies it, producing an unambiguous grammar.

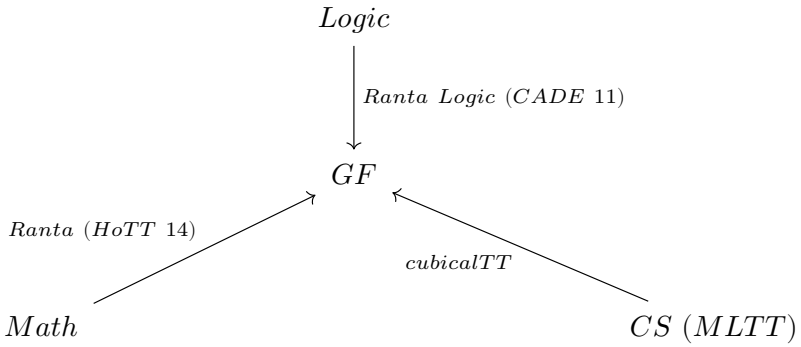

Figure 13: Trinitarian Grammars

While these three grammars offer the most poignant points of comparison between the computational, logical, and mathematical phenomena they attempt to capture, we also note that there were many other smaller grammars developed during the course of this work to supplement and experiment with various ideas presented. Importantly, the "Trinitarian Grammars" do not only model these different domains, but they each do so in a unique way, making compromises and capturing various linguistic and formal phenomena. The phenomena should be seen on a spectrum of *semantic adequacy* and *syntactic completeness*, as in autoreffig:G3 .

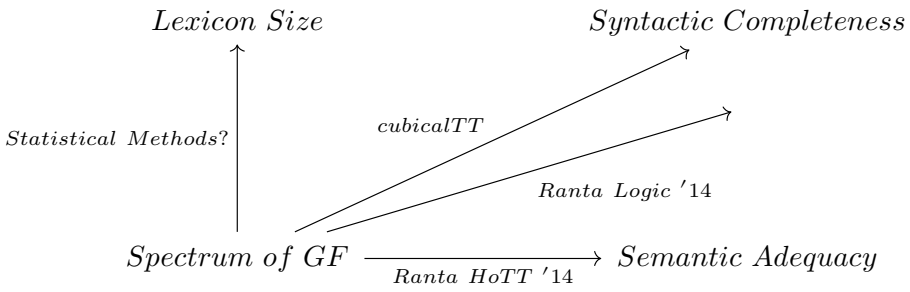

Figure 14: The Grammatical Dimension

The cubicalTT grammar, seeking syntacitic completeness, only has a pidgin English syntax, and therefore is only capable of parsing a programming language. Ranta's HoTT grammar on the other hand, while capable of presenting a quasi-logical form, would require extensive refactoring in order to transform the ASTs to something that resembles the ASTs of a programming language. The Logic grammar, which produces logically coherent and linguistically nuanced expressions, does not yet cover proofs, and therefore would require additional extensions to actually express anything a computer might understand, or, alternatively, theorems capable of impressing a mathematician. Finally, we note that large-scale coverage of linguistic phenomena for any of these grammars will additionally need to incorporate statistical methods in some way.

Before providing perspectives on the grammar design process, it is alo When designing grammars, the foremost question one should ask A few remarks on designing GF grammars should be noted as well.

The PMCFG class of languages is still quite tame when compared with, for instance, Turing complete languages. Thus, the 'abstract' and 'concrete' coupling tight, the evaluation is quite simple, and the programs tend to write themselves once the correct types are chosen. This is not to say GF programming is easier than in other languages, because often there are unforeseen constraints that the programmer must get used to, limiting the palette available when writing code. These constraints allow for fast parsing, but greatly limit the types of programs one often thinks of writing.

5.2 A Brief Introduction to GF

GF is a very powerful yet simple system. While learning the basics may not be to difficult for the experienced programmer, GF requires the programmer to work with, in some sense, an incredibly stiff set of constraints compared to general purpose languages, and therefore its lack of expressiveness requires a different way of thinking about programming.

The two functions displayed in Figure 5, $Parse : \{Strings\} \rightarrow \{\{ASTs\}\}$ and $Linearize : \{ASTs\} \rightarrow \{Strings\}$, obey the important property that :

$$\forall s \in \{Strings\} \forall x \in (Parse(s)), Linearize(x) \equiv s$$

This seems somewhat natural from the programmers perspective. The limitation on ASTs to linearize uniquely is actually a benefit, because it saves the user having to make a choice about a translation (although, again, a statistical mechanism could alleviate this constraint). We also want our translations to be well-behaved mathematically, i.e. composing $Linearize$ and $Parse$ ad infinitum should presumably not diverge.

GF captures languages more expressive than Chomsky's original CFG [cite] but is still remains decidable, with parsing in polynomial time. Which polynomial depends on the grammar [cite krasimir]. It comes equipped with 6 basic judgments:

- Abstract : 'cat, fun'
- Concrete : 'lincat, lin, param'
- Auxiliary : 'oper'

There are two judgments in an abstract file, for categories and named functions defined over those categories, namely `cat` and `fun`. The categories are just (succinct) names, and while GF allows dependent types, e.g. categories which are parameterized over other categories and thereby allow for more fine-grained semantic distinctions. We will leave these details aside, but do note that GF's dependent types can be used to implement a programming language which only parses well-typed terms (and can actually compute with them using auxiliary declarations).

In a simply typed programming language we can choose categories, for variables, types and expressions, or what might `Var`, `Typ`, and `Exp` respectively. One can then define the functions for the simply typed lambda calculus extended with natural numbers, known as Gödel's T.

```
cat
  Typ ; Exp ; Var ;
fun
  Tarr : Typ -> Typ -> Typ ;
  Tnat : Typ ;

  Evar : Var -> Exp ;
  Elam : Var -> Typ -> Exp -> Exp ;
  Eapp : Exp -> Exp -> Exp ;

  Ezer : Exp ;
  Esuc : Exp -> Exp ;
  Enatrec : Exp -> Exp -> Exp ->  Exp ;

  X : Var ;
  Y : Var ;
  F : Var ;
```

```
  IntV : Int -> Var ;
```

So far we have specified how to form expressions : types built out of possibly higher order functions between natural numbers, and expressions built out of lambda and natural number terms. The variables are kept as a separate syntactic category, and integers, `Int`, are predefined via GF's internals and simply allow one to parse numeric expressions. One may then define a functional which takes a function over the natural numbers and returns that function applied to 1 - the AST for this expression is :

```
Elam
    F
    Tarr
        Tnat Tnat
      Eapp
        Evar
            F
        Evar
            IntV
                1
```

Dual to the abstract syntax there are parallel judgments when defining a concrete syntax in GF, `lincat` and `lin` corresponding to `cat` and `fun`, respectively. Wher the AST is the specification, the concrete form is its implementation in a given lanaguage. The `lincat` serves to give *linearization types* which are quite simply either strings, records (or products which support sub-typing and named fields), or tables (or coproducts) which can make choices when computing with arbitrarily named parameters, which are naturally isomorphic to the sets of some finite cardinality. The tables are actually derivable from the records and their projections, which is how PGF is defined internally, but they are so fundamental to GF programming and expressiveness that they merit syntactic distincion. The `lin` is a term which matches the type signature of the `fun` with which it shares a name. The `lincat` constrains the concrete types of the arguments, and therefore subjects the GF user to how they are used.

If we assume we are just working with strings, then we can simply define the functions as recursively concatenating `++` strings. The lambda function for pidgin English then has, as its linearization form as follows :

```
lin
  Elam v t e = "function taking" ++ v ++ "in" ++ t ++ "to" ++ e ;
```

Once all the relevant functions are giving correct linearizations, one can now parse and linearize to the abstract syntax tree above the to string "function taking f in the natural numbers to the natural numbers to apply f to 1". This is clearly unnatural for a variety of reasons, but it's an approximation of what a computer scientist

might say. Suppose instead, we choose to linearize this same expression to a pidgin expression modeled off Haskell's syntax, "
(f : nat -> nat) -> f 1". We should notice the absence of parentheses for application suggest something more subtle is happening with the linearization process, for normally programming languages use fixity declarations to avoid lispy looking code. Here are the linearization functions which allow for linearization from the above AST :

```
lincat
  Typ = TermPrec ;
  Exp = TermPrec ;
lin
  Elam v t e =
      mkPrec 0 ("\\" ++ parenth (v ++ ":" ++ usePrec 0 t) ++ "-
>" ++ usePrec 0 e) ;
  Eapp = infixl 2 "" ;
```

Where did `TermPrec`, `infixl`, `parenth`, `mkPrec`, and `usePrec` come from? These are all functions defined in the RGL. We show a few of them below, thereby introducing the final, main GF judgments `param` and `oper` for parameters and operators.

```
param
  Bool = True | False ;
oper
  TermPrec : Type = {s : Str ; p : Prec} ;
  usePrec : Prec -> TermPrec -> Str = \p,x ->
    case lessPrec x.p p of {
      True => parenth x.s ;
      False => parenthOpt x.s
    } ;
  parenth : Str -> Str = \s -> "(" ++ s ++ ")" ;
  parenthOpt : Str -> Str = \s -> variants {s ; "(" ++ s ++ ")"} ;
```

Parameters in GF, to a first approximation, are simply data types of unary constructors with finite cardinality. Operators, on the other hand, encode the logic of GF linearization rules. They are an unnecessary part of the language because they don't introduce new logical content, but they do allow one to abstract the function bodies of `lin`'s so that one may keep the actual linearization rules looking clean. Since GF also support `oper` overloading, one can often get away with often deceptively sleek looking linearizations, and this is a key feature of the RGL. The variants is one of the ways to encode multiple linearizations forms for a given tree, so here, for example, we're breaking the nice property from above.

This more or less resembles a typical programming language, with very little deviation from what when would expect specifying something in twelf. Nonetheless, because this is both meant to somehow capture the logical form in addition to

the surface appearance of a language, the separation of concerns leaves the user with an important decision to make regarding how one couples the linear and abstract syntaxes. There are in some sense two extremes one can take to get a well performing GF grammar.

Suppose you have a page of text from some random source of length l, and you take it as an exercise to build a GF grammar which translates it. The first extreme approach you could take would be to give each word in the text to a unique category, a unique function for each category bearing the word's name, along with a single really function with l arguments for the whole sequence of words in the text. One could then verbatim copy the words as just strings with their corresponding names in the concrete syntax. This overfitted grammar would fail : it wouldn't scale to other languages, wouldn't cover any texts other than the one given to it, and wouldn't be at all informative. Alternatively, one could create a grammar of a two categories c and s with two functions, $f_0 : c$ and $f_1 : c \to s$, whereby c would be given n fields, each strings, with the string given at position i in f_0 matching $word_i$ from the text. f_1 would merely concatenate it all. This grammar would be similarly degenerate, despite also parsing the page of text.

This seemingly silly example highlights the most blatant tension the GF grammar writer will face : how to balance syntactic and semantic content of the grammar in between the concrete and the abstract syntax. It is also highly relevant as concerns the domain of translation, for a programming language with minimal syntax and the mathematicians language in expressing her ideas are on vastly different sides of this issue.

We claim that syntactically complete grammars are much more easily dealt with simple abstract syntax. However, to take allow a syntactically complete grammar to capture semantic nuance and neutrality then humans requires immensely more work on the concrete side. Semantically adequate grammars on the other hand, require significantly more attention on the abstract side, because semantically meaningful expressions often don't generalize - each part of an expressions exhibits unique behaviors which can't be abstracted to apply to other parts of the expression. Therefore, producing a syntactically complete expressions which doesn't overgenerate parses also requires a lot work from the grammar writer.

We hope the subsequent examples will illuminate this tension. The problem with treating a syntactically oriented domain like type theory with and a semantically oriented one like mathematics with the same abstract syntax poses very serious problems, but also highlights the power of other features of GF, like the RGL [cite] and Haskell embedding PGF [cite].

The GF RGL is a very robust library for parsing grammatically coherent language. It exists for many different natural languages with a core abstract syntax shared by all of them. The API allows one to easily construct, sentence level phrases once the lexicon has been defined, which are also greatly facilitated by the API.

PGF, is an embedding of a GF abstract syntax into Haskell, where the categories are given "shadow types", so that one can build turn an abstract syntax into (a

possibly massive) Generalized Algebraic Data Type (GADT) `Tree` with kind `* -> *` where all the functions serve as constructors. If function `h` returns category `c`, the Haskell constructor `Gh` returns `Tree c`.

The PGF API also allows for the Haskell user to call the parse and linearization functions, so that once the grammar is built, one can use Haskell as an interface with the outside world. While GF originally was conceived as allowing computation with ASTs, using a semantic computation judgment `def`, this has approach has largely been overshadowed by Haskell. Once a grammar is embedded in Haskell, one can use general recursion, monads, and all other types of bells and whistles produced by the functional programming community to compute with the embedded ASTs.

We note that this further muddies the water of what syntax and semantics refer to in the GF lexicon. Although a GF abstract syntax somehow represents the programmers idealized semantic domain, once embedded in PGF the trees now may represent syntactic objects to be evaluated or transformed to some other semantic domain which may or may not eventually be linked back to a GF linearization.

These are all the main ingredients a GF user will hopefully need to understand the grammars hereby elaborated, and hopefully these examples will showcase the full potential of GF for the problem of mathematical translations.

6 Prior GF Formalizations

Prior to the grammars explored thesis, Ranta produced two main results [49] [51]. These are incredibly important precedents in this approach to proof translation, and serve as important comparative work for which this work responds.

6.1 CADE 2011

In [49], Ranta designed a grammar which allowed for predicate logic with a domain specific lexicon supporting mathematical theories , say geometry or arithmetic, on top of the logic. The syntax was both meant to be relatively complete, so that typical logical utterances of interest could be accommodated, as well as support relatively non-trivial linguistic nuance like lists of terms, predicates, and propositions, in-situ and bounded quantification, like other ways of constructing more syntactically nuanced predicates. The more interesting syntactic details captured in this work was by means of an extended grammar on top of the core. The bidirectional transformation between the core and extended grammars via a PGF also show the viability and necessity of using more expressive programming languages (Haskell) when doing thorough translations.

Lists are natural to humans - this is reflected in our language. The RGL supports listing the sentences, noun phrases, and other grammatical categories. One can then use PGF to unroll the lists into binary operators, or alternatively transform them in the opposite direction. , we first mention that GF natively supports list categories, the judgment `cat [C] {n}` can be desugared to

```
cat ListC ;
fun BaseC : C -> ... -> C -> ListC ; -- n C 's
fun ConsC : C -> ListC -> ListC
```

As a case study for this grammar, the proposition $\forall x(Nat(x) \supset Even(x) \vee Odd(x))$ can be given a maximized and minimized version. The tree representing the *syntactically complete* phrase "for all natural numbers x, x is even or x is odd" would be minimized to a tree which linearizes to the *semantically adequate* phrase "every natural number is even or odd".

We see that our criteria of semantic adequacy and syntactic completeness can both occur in the same grammar, with the different subsets related not by a direct GF translation but a PGF level transformation. Problematically, this syntactically complete phrase produces four ASTs, with the "or" and "forall" competing for precedence. Where PGF may only give one translation to the extended syntax, this doesn't give the user of the grammar confidence that her phrase was correctly interpreted.

In the opposite direction, the desugaring of a logically "informal" statement into something less linguistically idiomatic is also accomplished. Ranta claims "Finding extended syntax equivalents for core syntax trees is trickier than the opposite

direction". While this may be true for this particular grammar, we argue that this may not hold generally. Dealing with these ambiguities must be solved first and foremost to satisfy the PL designer who only accepts unambiguous parses. For instance, the gf shell shows "the sum of the sum of x and y and z is equal to the sum of x and the sum of y and z" giving 32 unique parses. Ranta also outlines the mapping, $[\![-]\!] : Core \to Extended$, which should hypothetically return a set of extended sentences for a more comprehensive grammar.

- Flattening a list $x \textit{ and } y \textit{ and } z \mapsto x,\ y \textit{ and } z$
- Aggregation $x \textit{ is even or } x \textit{ is odd} \mapsto x \textit{ is even or odd}$
- In-situ quantification
  $\forall\ n\ \in Nat,\ x \textit{ is even or } x \textit{ is odd} \mapsto \textit{every } Nat \textit{ is even or}$
- Negation $\textit{it is not that case that } x \textit{ is even} \mapsto \textit{is not even}$
- Reflexivitazion $x \textit{ is equal to } x \mapsto x \textit{ is equal to itself}$
- Modification $x \textit{ is a number and } x \textit{ is even} \mapsto x \textit{ is an even number}$

Scaling this to cover more phenomena, such as those from [cite ganesalingam], will pose challenges. Extending this work in general without very sophisticated statistical methods is impossible because mathematicians will speak uniquely, and so choosing how to extend a grammar that covers the multiplicity of ways of saying "the same thing" will require many choices and a significant corpus of examples. Efficient communication, is a pragmatic feature which this only begins to barely address. The most interesting linguistic phenomena covered by this grammar, In-situ quantification, has been at the heart of the Montague tradition.

In some sense, this grammar serves as a case study for what this thesis is trying to do. However, we note that the core logic only supports propositions without proofs - it is not a type theory with terms. Additionally, the domain of arithmetic is an important case study, but scaling this grammar (or any other, for that matter) to allow for *semantic adequacy* of real mathematics is still far away, or as Ranta concedes, "it seems that text generation involves undecidable optimization problems that have no ultimate automatic solution." It would be interesting to further extend this grammar with both terms and an Agda-like concrete syntax.

6.1.1 An Additional PGF Grammar

One of the difficulties encountered in this work was reverse engineering Ranta's code - the large size of a grammar and declarative nature of the code makes it incredibly difficult to isolate individual features one may wish to understand. This is true for both GF and PGF, and therefore a lot of work went into filtering the grammars to understand behaviors of individual components of interest. Careful usage of the GF module system may allow one to look at "subgrammars" for some circumstances, but there is not proper methodology to extract a sub-grammar and therefore it was found that writing a grammar from scratch was often the easiest way to do this. Grammars can be written compositionally (adding new categories and functions, refactoring linearization types, etc.) but decomposing them is not a compositional process.

We wrote a smaller version [cite mycode] of, just focused on propositional logic, but with the added interest of not just translating between Trees, but also allowing Haskell computation and evaluation of expressions. Although this exercise was in some ways a digression from the language of proofs, it also highlighted many interesting problems.

We begin with an example : the idea was to create a PGF layer for the evaluation of propositional expressions to their Boolean values, and then create a question answering system which gave different types of answers - the binary valued answer, the most verbose possible answer, and the answer which was deemed the most semantically adequate, `Simple`, `Verbose`, and `Compressed`, respectively. The system is capable of the following :

```
is it the case that if the sum of 3 , 4 and 5 is prime , odd and even then 4
  is prime and even

  Simple : yes .
 Verbose : yes . if the sum of 3 and the sum of 4 and 5 is prime and the sum
   of 3 and the sum of 4 and 5 is odd and the sum of 3 and the sum of 4 and
    5 is even then 4 is prime and 4 is even .
 Compressed : yes . if the sum of 3 , 4 and 5 is prime , odd and even then
    4 is prime and even .
```

The extended grammar in this case only had lists of propositions and predicates, and so it was much simpler than [cite logic]. GF list categories are then transformed into Haskell lists via PGF, so the syntactic sugar for a GF list is actually functionally tied to its external behavior as well. The functions for our discussion are:

```
  IsNumProp : NumPred -> Object -> Prop ;
  LstNumPred : Conj -> [NumPred] -> NumPred ;
  LstProp : Conj  -> [Prop] -> Prop ;
```

Note that a numerical predicate, `NumPred`, represents, for instance, primality. In order for our pipeline to answer the question, we had to not only do transform trees, $[\![-]\!] : \{pgfAST\} \to \{pgfAST\}$, but also evaluate them in more classical domains $[\![-]\!] : \{pgfAST\} \to \mathbb{N}$ for the arithmetic objects and $[\![-]\!] : \{pgfAST\} \to \mathbb{B}$, `evalProp`, for the propositions.

The extension adds more complex cases to cover when evaluating propistions, because a normal “propositional evaluator” doesn’t have to deal with lists. For the most part, this evaluation is able to just apply boolean semantics to the *canonical* propositional constructors, like `GNot`. However, a bug that was subtle and difficult to find appeared, thereby forcing us to dig deep inside GIsNumProp, preventing an easy solution to what would otherwise be a simple example of denotational semantics.

```
evalProp :: GProp -> Bool
evalProp p = case p of
  ...
  GNot p -> not (evalProp p)
  ...
  GIsNumProp (GLstNumProp c (GListNumPred (x : xs))) obj ->
    let xo = evalProp (GIsNumProp x obj)
     xso = evalProp (GIsNumProp (GLstNumProp c (GListNumPred (xs))) obj) in
    case c of
      GAnd -> (&&) xo xso
      GOr -> (||) xo xso
  ...
```

While this case is still relatively simple, an even more expressive abstract syntax may yield many more subtle obstacles, which is the reason it's so hard to understand PGF helper functions by just trying to read the code. The more semantic content one incorporates into the GF grammar, the larger the PGF GADT, which leads to many more cases when evaluating these trees.

There were many obstructions in engineering this relatively simple example, particularly when it came to writing test cases. For the naive way to test with GF is to translate, and the linearization and parsing functions don't give the programmer many degrees of freedom. ASTs are not objects amenable to human intuition, which makes it problematic because understanding the transformations of them constantly requires parsing and linearizing to see their "behavior". While some work has been done to allow testing of GF grammars for NL applications [cite inari], the specific domain of formal languages in GF requires a more refined notion of testing because they should be testable relative to some model with well behaved mathematical properties. Debugging something in the pipeline $String \to GADT \to GADT \to String$ for a large scale grammar without a testing methodology for each intermediate state is surely to be avoided.

Unfortunately, there is no published work on using Quickcheck [cite hughes] with PGF. The bugs in this grammar were discovered via the input and output *appearance* of strings. Often, no string would be returned after a small change, and discovering the source (abstract, concrete, or PGF) was excruciating. In one case, a bug was discovered that was presumed to be from the PGF evaluator, but was then back-traced to Ranta's grammar from which the code had been refactored. The sentence which broke our pipeline from core to extended, "4 is prime , 5 is even and if 6 is odd then 7 is even", would be easily generated (or at least its AST) by quickcheck.

An important observation that was made during this development : that theorems should be the source of inspirations for deciding which PGF transformations should take place. For instance, one could define $odd : \mathbb{N} \to Set$, $prime : \mathbb{N} \to Set$ and prove that $\forall n \in \mathbb{N}.\ n > 2 \times prime\ n \implies odd\ n$. We can use this theorem as a source of translation, and in fact encode a PGF rule that transforms anything of the form "n is prime and n is odd" to "n is prime", subject to the condition that $n \neq 2$. One could then take a whole set of theorems from predicate calculus and encode them

as Haskell functions which simplify the expressions to a minified expression with the same meaning, up to some notion of equivalence. The verbose "if a then b and if a then c, can be more canonically read as "if a then b and c". The application of these theorems as evaluation functions in Haskell could help give our QA example more informative and direct answers.

We hope this intricate look at a fairly simple grammar highlights some very serious considerations one should make when writing a PGF embedded grammar. These include : how does the semantic space the grammar seeks to approximate effects the PGF translation, how testing formal grammars is non-trivial but necessary future work, and finally, how information (in this case theorems) from the domain of approximation can shape and inspire the PGF transformations during the translation process.

7 Natural Number Proofs

We now explore the "main goal" of this work : proofs in GF. We commence with perhaps the most natural kind of proof one would expect, those over the inductively defined natural numbers. As a proposed foundational alternative to mathematics, dependent type theories allow types to depend on terms and therefore allow propositions which include terms to be encoded as types.

In the simple type theory example, we included *types* and *expressions* as distinct syntactic categories, whereby the linearization of a type can't possibly call the linearization of a term. We now experiment with a small dependently typed programming language with only Π-types. The big difference for such a simple fragment like natural numbers in the dependent setting is the fact that the recursion principle becomes an induction principle. The types of a sub-expression being evaluated with a recursive call may depend on the values being computing. Extra work is required in implementing type-checkers for dependent language because they have to deal with a much more sensitive and computationally expensive notion of type.

A dependent type theorist will assert that every time mathematicians use a notion like $\mathbb{R}^n$, they are implicitly quantifying over the natural numbers, namely n, and therefore are referring to a parameterized type, not a *set*. There are many more elaborate examples of dependency in mathematics, but because this notation is ubiquitous, we note that the type theorist would not be satisfied with many expressions from real analysis, because they assert things about $\mathbb{R}^n$ all the time without ever proving anything by induction over the numbers. Perhaps this seems pedantic, but it highlights a large gap between the type-theorists syntactic approach to mathematics and the mathematicians focus on the domain semantics of her field of interest.

Delaying a more in depth discussion of equality [ref later section], we here assert that one proves equality in Agda by finding something that is *irrefutably equal* to itself, where the notion of irrefutably is in some sense gave birth to subject matter of higher type theory. Taking this for granted, we begin by looking at one of the simplest natural numbers proof's : that addition is associative.

7.1 Associativity of Natural Numbers

We define addition in Agda by recursion on the first argument, and notice that the sum of two natural numbers is always a natural number. Therefore, agda has the capacity to always compute the sum of two given natural numbers, via the defining equations, and indeed $2 + 2 = 4$ is irrefutably true. Additionally, we know that 0 added to a number is always that number.

```
_+_ : ℕ → ℕ → ℕ
zero + n = n
suc x + n = suc (x + n)

2+2=4 : 2 + 2 ≡ 4
2+2=4 = refl
```

We now present the type which encodes the proposition which says some number 0 plus some number is propositionally equal to that number. Agda is able to compute evidence for this proposition via the definition of addition, and therefore just reflexivly know that number is equal to itself. Yet, the novice Agda programmer will run into the quagmire that the proposition that any number added to 0 is not definitionally equal to n, i.e. that the defining equations don't give an automatic way of universally validating this fact about the second arguement. We're stuck.

```
0+n=n : ∀ (n : ℕ) → 0 + n ≡ n
0+n=n n = refl

3+0=n : 3 + 0 ≡ 3
3+0=n = refl

n+0=n : ∀ (n : ℕ) → n + 0 ≡ n
n+0=n = roadblock
```

Instead, one must use induction, which we show here by pattern matching. We use an auxiliary lemma ap which essentially says, from a classical sense, that equality is well defined with respect to function application (or that all functions are well defined). Then we can simply apply the successor function to the induction hypothesis which manifests as a simple recursive call. This proof is actually, verabatim, the same as the associativity proof - which gives us one perspective that suggests, at least sometimes, types can be even more expressive than programs in Agda.

```
ap : (f : A → B) → a ≡ a' → f a ≡ f a'
ap f refl = refl

n+0=n' : ∀ (n : ℕ) → n + 0 ≡ n
n+0=n' zero = refl
n+0=n' (suc n) = ap suc (n+0=n' n)

associativity-plus : (n m p : ℕ) → ((n + m) + p) ≡ (n + (m + p))
associativity-plus zero m p = refl
associativity-plus (suc n) m p = ap suc (associativity-plus n m p)
```

To construct a GF grammar which includes both the simple types as well as those which may depend on a variable of some other type, one simply gets rid of the syntactic distinction, whereby everything is just in Exp. We show the dependent function along with its introduction and elimination forms, noting that we include *telescopes* as syntactic sugar to not have to repeat λ or Π expressions. Telescopes are lists of types which may depend on earlier variables defined in the same telescope.

```
fun
  Pi : [Tele] -> Exp -> Exp ;  -- type
  Fun : Exp -> Exp -> Exp ;
  Lam : [Tele] -> Exp -> Exp ; --term
  App : Exp -> Exp -> Exp ;
  TeleC : [Var] -> Exp -> Tele ;
```

This grammar actually allows us to prove the above right-identity and associativity laws. Before we look at the natural language proof generated by this code, we first look at an idealized version, which is reproduced from [cite software foujndations].

```
Theorem: For any n, m and p,
  n + (m + p) = (n + m) + p.
Proof: By induction on n.
  First, suppose n = 0. We must show that
    0 + (m + p) = (0 + m) + p.
  This follows directly from the definition of +.
  Next, suppose n = S n', where
    n' + (m + p) = (n' + m) + p.
  We must now show that
    (S n') + (m + p) = ((S n') + m) + p.
  By the definition of +, this follows from
    S (n' + (m + p)) = S ((n' + m) + p),
  which is immediate from the induction hypothesis. Qed.
```

While overly pedantic relative to a mathematicians preferred conciseness, this illustrates a proof which is both syntactically complete and semantically adequate. Let's compare this proof with our idealized Agda reconstruction, using the induction principle (as given earlier with the arguements commuted).

```
associativity-plus-ind' : (n m p : ℕ) → ((n + m) + p) ≡ (n + (m + p))
associativity-plus-ind' n m p = natind baseCase (λ n₁ ih → simpl n₁ (indCase n₁ ih)) n
  where
    baseCase : (zero + m + p) ≡ (zero + (m + p))
    baseCase = refl
    indCase : (n' : ℕ) → (n' + m + p) ≡ (n' + (m + p)) →
                  suc (n' + m + p) ≡ suc (n' + (m + p))
    indCase = (λ n' x → ap suc x )
    simpl : (n' : ℕ) -- we must now show that
            → suc (n' + m + p) ≡ suc (n' + (m + p))
            → (suc n' + m + p) ≡ (suc n' + (m + p))
    simpl n' x = x
```

This proof, aligned with with the text so-as to allow for idealized translation, is actually overly complicated and unnessary for the Agda programmer. For the proof state is maintained interactively, the definitional equalities are normalized via the typechecker, and therefore the base case and inductive case can be simplified considerably once the *motive* is known [cite mcbride]. Fortunately, Agda's pattern matching is powerful enough to infer the motive, so that one can generally pay attention to "high level details" generally. We see a "more readable" rewrting below:

```
associativity-plus-ind : (m n p : ℕ) → ((m + n) + p) ≡ (m + (n + p))
associativity-plus-ind m n p =
```

```
    natind {λ n' → (n' + n) + p ≡ n' + (n + p)} baseCase indCase m
    where
      baseCase = refl
      indCase = λ (n' : ℕ) (x : n' + n + p ≡ n' + (n + p)) → ap suc x
```

Finally, taking a "desguared" version of the Agda proof term, as presented in our grammar, we can can reconstruct the lambda term which would, in an idealized world, match the software foundations proof.

```
p -lang=LHask "
  \\ ( n m p : nat ) ->
  natind
    (\\ (n' : nat) ->
      ((plus n' (plus m p)) == (plus (plus n' m) p)))
    refl
    ( \\ ( n' : nat ) ->
     \\ (x : ((plus n' (plus m p)) == (plus (plus n' m) p)))
      -> ap suc x )
    n" | l
------------
  function taking n , m p in the natural numbers
  to
  We proceed by induction over n .
  We therefore wish to prove : function taking n',
    in the natural numbers to apply apply plus to
    n' to apply apply plus to m to p is equal
    to apply apply plus to apply apply plus to n'
    to m to p .
  In the base case, suppose m equals zero.
    we know this by reflexivity .
  In the inductive case,
    suppose m is the successor.
    Then one has one has function taking n' ,
      in the natural numbers to function
      taking x , in apply apply plus to n'
      to apply apply plus to m to p is equal to
      apply apply plus to apply apply plus to n'
      to n' to p to apply ap to the successor
      of x.
```

This is by all accounts horrendous, nonetheless it does contain enough information to say it is syntactically complete. There are a few points which make this proof non-trivial to translate.

First, as is obvious, there is little support for punctuation and proof structure - the indentations were added by hand. The semantic distinction is left to the type-checker in the dependently typed language, so the syntactic distinctions have been

discharged into a single `Exp` category, and therefore, terms like 0, a noun, and the whole proof term above (multiple sentences) are both compressed into the same box. This poses a huge issue for the GF developer wishing to utilize the RGL, whereby these grammatical categories (and therefore linearization types) are distinct, but our abstract syntax offers an incredibly course view of the PL syntax. This can be achieved by creating many fields in the records of the `lincat` for `Exp`, one for each syntactic category. Then one may have parameters, with which to match them on and determine how they are expressed as syntactic categories. This has been done for a Digital Grammars client looking to produce natural language for a code base, but unfortunately the code is not publically avaialable [cite aarne]. We note that this error may become increasingly difficult the more syntax one covers, and generalizing it to full scale mathematics texts with the myriad syntactic uses of different types of mathematical terms seems intractible (although Ganesalignam came up with a different theoretical notion "type" to cover grammartical artificats in textual mathematics [cite ganes]).

Second, the application function, which is so common it gets the syntactic distinction of being whitespace, does not have the same luxury in the natural language setting. This is because the typechecker is responsible for determining if the function is applied to the right number of arguements, and we have chosen a *shallow embedding* in our programming language, whereby plus is a variable name and not a binary function. This can also be reconciled at the concrete level, as was demonstrated with a relatively simple example [cite mycode /home-/wmacmil/gf/Precedences/ExpFormal.gf]. Nonetheless, to add this layer of complextiy to the linearization seems unneccessary, and it would be simpler to resolve this by somehow matching the arguement structure of the agda function to some deeply embedded addition function, `Plus : Exp -> Exp -> Exp`. Ranta, for example, does this in the HoTT code [refer ahead].

Finally, we should point out an error that makes demonstrates the failure of this to parse a semantically adequate phrase : "apply ap to the successor of x" is incorrect. The `successor` is actually an arguement of `ap`, and isnt applied to x directly on the final line. This is because `Suc : Exp -> Exp` was deeply embedded into GF, and the η-expanded version should be substituted to correct for the error (which will make it even more unreadable). Alternatively, one could include all permutation forms of an expression's type signature up to eta expansion (depending on the number of both implicit and explicit arguements), but this could make the code both overgenerate and also make it significantly more complex to implement. These are relatively simple obstacles for the PL developer where the desugaring sends η-equivalent expressions to some normal form, so that the programmer can be somewhat flexible. The freedom of natural language, however, creates numerous obstacles for the GF developer. These are often nontrivial to identify and reconcile, especially when one layers the complexity of multiple natural language features covered by the same grammar.

We hope this demonstrates the obstacles one faces when even playing with simple grammars. Unfortunately, the repairs needed to generate both Agda code and natural language proofs for this specific grammar were beyond the scope of what time allowed, and we hope this can be picked up by someone else soon.

7.2 What is Equality?

> ... the univalence axiom validates the common, but formally unjustified, practice of identifying isomorphic objects. [cite hottbook]

Mathematicians, and most people generally, have an intuition for equality, that of an identification between two pieces of information which intuitively must be the same thing, i.e. $2 + 2 = 4$. The philosophically inclined might ask about identification generally. We showcase different notions of identifying things in mathematics, logic, and type theory :

- Equivalence of propositions
- Equality of sets
- Equality of members of sets
- Isomorphism of structures
- Equality of terms
- Equality of types

While there are notions of equality, sameness, or identification outside of these formal domains, we don't dare take a philosophical stab at these notions here. Earlier, we saw two notions of equality in type theory, the judgmental equality in our introduction to MLTT, and the propositional equality which was used in the twin prime conjecture. Judgmental equality is the means of computing, for instance, that $2 + 2 = 4$, for there is no way of proving this other than appealing to the definition of addition. Propositional equality, on the other hand, is actually a type. It is defined as follows in Agda, with an accompanying natural language definition from [cite hottbook] :

```
data _≡'_ {A : Set} : (a b : A) → Set where
  r : (a : A) → a ≡' a
```

Definition 8 *The formation rule says that given a type $A : \mathcal{U}$ and two elements $a, b : A$, we can form the type $(a =_A b) : \mathcal{U}$ in the same universe. The basic way to construct an element of $a = b$ is to know that a and b are the same. Thus, the introduction rule is a dependent function*

$$\mathsf{refl} : \prod_{a:A} (a =_A a)$$

*called **reflexivity**, which says that every element of A is equal to itself (in a specified way). We regard refl_a as being the constant path at the point a.*

The astute might ask, what does it mean to "construct an element of $a = b$"? For the mathematician use to thinking in terms of sets $\{a = b \mid a, b \in \mathbb{N}\}$ isn't a well-defined notion. Due to its use of the axiom of extensionality, the set theoretic notion of equality is, no suprise, extensional. This means that sets are identified

when they have the same elements, and equality is therefore external to the notion of set. To inhabit a type means to provide evidence for that inhabitation. The reflexivity constructor is therefore a means of providing evidence of an equality. This evidence approach is disctinctly constructive, and a big reason why classical and constructive mathematics, especially when treated in an intuitionistic type theory suitable for a programming language implementation, are such different beasts.

In Martin-Löf Type Theory, there are two fundamental notions of equality, propositional and definitional. While propositional equality is inductively defined (as above) as a type which may have possibly more than one inhabitant, definitional equality, denoted $- \equiv -$ and perhaps more aptly named computational equality, is familiarly what most people think of as equality. Namely, two terms which compute to the same canonical form are computationally equal. In intensional type theory, propositional equality is a weaker notion than computational equality : all propositionally equal terms are computationally equal. However, computational equality does not imply propistional equality - if it does, then one enters into the space of extensional type theory.

Prior to the homotopical interpretation of identity types, debates about extensional and intensional type theories centred around two features or bugs : extensional type theory sacrificed decideable type checking, while intensional type theories required extra beauracracy when dealing with equality in proofs. One approach in intensional type theories treated types as setoids, therefore leading to so-called “Setoid Hell”. These debates reflected Martin-Löf’s flip-flopping on the issue. His seminal 1979 Constructive Mathematics and Computer Programming, which took an extensional view, was soon betrayed by lectures he gave soon thereafter in Padova in 1980. Martin-Löf was a born again intensional type theorist. These Padova lectures were later published in the ”Bibliopolis Book”, and went on to inspire the European (and Gothenburg in particular) approach to implementing proof assitants, whereas the extensionalists were primarily eminating from Robert Constable’s group at Cornell.

This tension has now been at least partially resolved, or at the very least clarified, by an insight Voevodsky was apparently most proud of : the introduction of h-levels. We’ll delegate these details to more advanced references, it is mentioned here to indicate that extensional type theory was really “set theory” in disguise, in that it collapses the higher path structure of identity types. The work over the past 10 years has elucidated the intensional and extensional positions. HoTT, by allowing higher paths, is unashamedly intentional, and admits a collapse into the extensional universe if so desired. We now the examine the structure induced by this propositional equality.

Definition: A type A is contractible, if there is $a : A$, called the center of contraction, such that for all $x : A$, $a = x$.

Figure 15: Rendered Latex

```
isContr ( A : Set ) : Set = ( a : A ) ( * ) ( ( x : A ) -> Id ( a ) ( x ) )
```

$$\begin{aligned}&\mathsf{isContr} : (A : \mathsf{Set}) \to \mathsf{Set}\\&\mathsf{isContr}\ A = \Sigma\ A\ \lambda\ a \to (x : A) \to (a \equiv x)\end{aligned}$$

Figure 16: Contractibility

7.3 Ranta's HoTT Grammar

In 2014, Ranta gave an unpublished talk at the Stockholm Mathematics Seminar [51]. Fortunately the code is available, although many of the design choices aren't documented in the grammar. This project aimed to provide a translation like the one desired in our current work, but it took a real piece of mathematics text as the main influence on the design of the abstract syntax.

This work took a page of text from Peter Aczel's book which more or less goes over standard HoTT definitions and theorems. The grammar allows the translation of the latex document in English to the same document in French, and to a pidgin logical language. The central motivation of this grammar was to capture entirely "real" natural language mathematics, i.e. that which was written for the mathematician. Therefore, it isn't reminiscent of the slender abstract syntax the type theorist adores, and sacrificed "syntactic completeness" for "semantic adequacy". This means that the abstract syntax is much larger and very expressive, but it no longer becomes easy to reason about and additionally quite ad-hoc. Another defect is that this grammar overgenerates, so producing a unique parse from the PL side would require a significant amount of refactoring. Nonetheless, it is presumably possible to carve a subset of the GF HoTT abstract file to accommodate an Agda program, but one encounters rocks as soon as one begins to dig.

In **??** one can see different syntactic presentations of a notion of *contractability*, that a space is deformable into a single point, or that a Type is actually inhabited by a unique term. Some rendered latex taken verbatim from Ranta's test code, compared with the translated pidgin logic code (after refactoring of Ranta's linearization scheme) and an Agda program. We see that it was fairly easy to get the notation for our cubicalTT grammar [ref cubicaltt]. When parsing the logical form, unfortunately, the grammar is incredibly ambiguous.

To extend this grammar to accommodate a chapter worth of material, let alone a book, will not just require extending the lexicon, but encountering other syntactic phenomena that will further be difficult to compress when defining Agda's concrete syntax. This demonstrates that to design a grammar prioritizing *semantic adequacy* and subsequently trying to incorporate *syntactic completeness* becomes a very difficult problem. Depending on the application of the grammar, the em-

Definition: A map $f : A \to B$ is an equivalence, if for all $y : B$, its fiber, $\{x : A \mid fx = y\}$, is contractible. We write $A \simeq B$, if there is an equivalence $A \to B$.

```
Equivalence ( f : A -> B ) : Set =
  ( y : B ) -> ( isContr ( fiber it ) ) ; ; ;
  fiber it : Set = ( x : A ) ( * ) ( Id ( f ( x ) ) ( y ) )
```

```
Equivalence : (A B : Set) → (f : A → B) → Set
Equivalence A B f = ∀ (y : B) → isContr (fiber' y)
  where
    fiber' : (y : B) → Set
    fiber' y = Σ A (λ x → y ≡ f x)
```

Figure 17: Contractibility

phasis on this axis is most assuredly a choice one should consider up front.

The next grammar, taking an actual programming language parser in Backus-Naur Form Converter (BNFC), GFifying it, and trying to use the abstract syntax to model natural language, gives in some sense a dual challenge, where the abstract syntax remains simple as in our dependently typed grammar, but its linearizations become increasingly complex, especially when generating natural language.

7.4 cubicalTT Grammar

Cubical type theories arose out of the desire to give a complete computational interpretation to HoTT, whereby nonviolence would become a theorem rather than an axiom [13]. The utility of this is that canonicity, the property of an expression having a irreducible normal form, is satisfied for all expressions. Univalence, by introducing a type without computational behavior, means that the constructivist using Agda will be able to define terms which don't normalize.

The origin of cubical, looking beyond simplicial models of type theory to cubical categories instead [5], gave a blueprint for a totally new type theory which natively supports proving functional extensionality, which is a especially important for mathematicians. The ideas of cubical became the origin for a new series of proof assistants, cubical [cite https://github.com/simhu/cubical] and cubicaltt [cite https://github.com/mortberg/cubicaltt], and Cubical Agda [59], as well as other in originating from Robert Constables disciples in the NuPrl tradition [cite redprl, redtt, jonprl]. cubicalTT, which was relatively complete, had an unambiguous BNFC grammar, more or less represents a kernel of Agda with cubical primitives. This final grammar, which we don't as cubicalTT, took the actual cubicalTT grammar and GFified the subset which is in the intersection with vanilla Agda. Extending our GF version to include cubical primitives would facilitate the extension of the work to Cubical Agda, and we hope future endeavors will go in this direction. Cubical Agda supports Higher Inductive Types natively and is capable of all types of new constructions [cite stuff] not mentioned in the HoTT book, but is also incredibly experimental, with large changes to the standard library constantly underway

as in [refer intro].

Our grammar for vanilla dependent Π-types [refer earlier section] was actually a subset of the current cubicalTT abstract syntax. We give a brief sketch of the algorithm to go between a BNFC grammar and a GF grammar. BNFC essentially combines the abstract and concrete syntax, enabling a hierarchy of numbered expressions `ExpN` to minimize use of parentheses. So, given m names and choosing $Name_i$, with the accompanying rule :

$$Name_i.\ ReturnCat_{i_n} ::= s_i^0\ C_{i_0}^0\ ...\ C_{i_{n-1}}^{n-1}\ s_i^n\ ;$$

where string s_j^i may be empty and the k in the i_k^{th} subscript represents the precedence number of a category. These precedences are indicated with a `Coercions N` keyword in BNFC. We can produce the following in GF.

$$cat\ Name_i \bigcap\{ReturnCat_i, C^0, ..., C^{n-1}\}\ ;$$
$$fun\ Name_i : C^0 \to ... \to C^{n-1} \to ReturnCat_i$$
$$lincat \bigcap\{ReturnCat_i, C^0, ..., C^{n-1}\}\ ; = TermPrec$$
$$lin\ Name_i\ c^0\ ...\ c^n = mkPrec(i_n, (s_i^0 \texttt{++} usePrec(i_0{+}1, c^0) \texttt{++} ... \texttt{++} usePrec(i_{n-1}{+}1, c^{n-1}) \texttt{++} s_i^n));$$

where $c^j \in C^j\ \forall i, j$, and `usePrec` and `mkPrec` come from the RGL, as seen earlier. We also note that some `lincat` might actually just be strings (or something else), for it is only when a precedence is observed that the `TermPrec` is applicable. The use of `usePrec` is only applicable when i_k isn't empty. Additionally, this doesn't account for the fact that already some categories may have been witnessed in which case we want to intersect over the whole set of rules at once. We reiterate the examples from the simply typed lambda calculus. The BNFC code results in the GF code immediately below.

```
--BNFC
Lam. Exp  ::= "\\" [PTele] "->" Exp ;
Fun. Exp1 ::= Exp2 "->" Exp1 ;
-- GF
cat Exp ; PTele ;
fun
  Lam : [PTele] -> Exp -> Exp ;
  Fun : Exp -> Exp -> Exp ;
lincat Exp = TermPrec ; [PTele] = Str ;
lin
  Lam pt e = mkPrec 0 ("\\" ++ pt ++ "->" ++ usePrec 0 e) ;
  Fun = mkPrec 1 (usePrec 2 x ++ "->" ++ usePrec 1 y) ;
```

This more or less elaborates exactly how to implement a programming language

with unambiguous parsing in GF. There is also a simple means of translating lists, including BNFC's `separator` and `terminator` keywords during the linearization process. Finally, there is a custom `token` keyword, and this is perhaps the most important feature absent in GF. Because BNFC generates Haskell code reminiscent of the PGF embedding, it would also be possible to translate the trees directly, if parsing complexity with GF was found to be slower than BNFC.

Most interestingly is to look at what is absent in BNFC, namely, the ability to add records and paremeters into the linearization types generally, although these GF features are implicitly used to encode precedence. For one could add unique categories in GF $Exp_1, ..., Exp_n$, but this would clutter the abstract syntax with information which isn't *semantically* relevant. And while the Haskell code generated by BNFC for cubicalTT is sent through a resolver to the *actual* abstract syntax used by the type-checker and evaluator, the fact that it parses the concrete syntax into an appropriate intermediary form is enough for our purposes.

We give the full grammar, including examples, in the appendix 9.2.

7.4.1 Difficulties

While our grammar certainly supports a real programming language syntax, modulo a few quarks, linearizing to a CNL for mathematics was not implemented due to time constraints, and the difficulties already encountered for an even simpler programming language 7.1, namely that types and terms in dependent type theory can be of just about any grammatical category, where we list a few :

- nouns, "zero"
- adjectives, "prime"
- verbs, "add"
- verb phrase, "apply the function to the subset of..."
- sentence, "if x is odd, then y is even"
- paragraph or more, "suppose x. then by y we know z. hence, w. but the v gives additionally gives us..."

In [52], the authors, generating human readable natural language from specifications, used a word type with many different fields for different grammatical categories (with the same grammatical categories sometimes accounting for multiple fields), in addition to symbolic fields. While deemed successful by the client, it would be interesting to apply this methodology to cubicalTT the grammar, and see how it scales once one begins to add more of Agda's capabilities. Their system also involved other components, like haskell transformations, and it is uncertain how these specific approaches would also allow for the generation of more *semantically adequate* language.

Other issues encountered in this grammar were Agda's pattern matching, whereby arguments are arranged in a matrix, as opposed to explicit cases, or *splits*. While cubicalTT allows syntax like

```
equalNat : nat -> nat -> bool = split
    zero -> split@ ( nat -> bool ) with
      zero  -> true
      suc n -> false
    suc m -> split@ ( nat -> bool ) with
      zero  -> false
      suc n -> equalNat m n
```

The problem is that when linearizing a split, one cannot know how many further splits will take place, and so going from this form to the more "readable" Agda code below is outside of GF's linearization capabilities - although a proof of this fact would require advanced mathematical capabilities.

```
equalNat : nat → nat → bool
equalNat zero zero = true ;
equalNat zero (suc n2) = false ;
equalNat (suc n1) zero = false ;
equalNat (suc n1) (suc n2) = equalNat n1 n2
```

One could instead just introduce a new form of declarations in the abstract syntax so-as to allow for `equalNat`, but this would require more Haskell overhead to allow for the correct AST transformations.

The way lists are dealt with in natural language vs. programming languages present obstacles, because the RGL's support for lists require certain numbers of categories in the end node, e.g. `cat[2]`, whereas our Agda grammar may instead have `cat[1]` or `cat[0]` for the same category, thereby require overloading of categories for the two linearization spaces, or alternatively adding more complexity to the linearization categories.

While presented succinctly here, these obstacles were legitimate difficulties which obliged us to test them on smaller grammars to isolate the phenomena trying to be overcome.

7.4.2 More advanced Agda features

Our grammar realistically covers just a small kernel of Agda's features and syntax. Agda supports much more, both in terms of syntactic sugar and semantically interesting. Aside from telescopes, other syntactic sugar features of Agda include unicode support, do notation, idiom brackets, generalized variable declarations, and more. While require significant work to extend the cubicalTT grammar with these, it is doubtful these kinds of features offer significant theoretical challenges in terms of translation to natural language.

From the semantic side, however, Agda offers many features which extend just the kernel of the Π, Σ, and recursive data type definitions which form the basis of

any dependent type theory. These include universes, sized types, modules, overloading for more ad-hoc polymorphism, proof by reflection, a sort system, higher inductive types (thanks to cubical), and many more things visible in the Agda documentation [cite agda docs]. Additionally, it has more traditional PL features, like the ability to perform side effects or call Haskell functions. Adding any one of these not only adds overhead to the parser, but would require lots of thought in terms of how to these features manifest in natural language for mathematicians (and programmers). Additionally, these features make the metatheory of Agda much more expensive to understand, in addition to the practical implications of introducing bugs in its implementation.

Mathematics on the other hand, doesn't often introduce more advanced "semantic machinery" like those listed, at least not in a way that is explicitly designed. Perhaps idioms and conventions change, as well as generalizations, i.e. Category Theory, offering ways of presenting ideas more succinctly, but these are merely reflected in the presentation, not in the underlying logical formalism. The linguistic evolution of mathematics additionally reflects some kind of meta-changes, but not in a coherent way that is yet understood. For many mathematicians are largely interested in proving theorems and solving problems specific to some domain, and many mathematicians are unfamiliar with logic as a discipline as a whole, let alone type theory.

The resolution of these meta-ideas from both the type theoretic and mathematical perspectives is what makes this problem of translation so philosophically intriguing, as well as intractable. We hope these observations might offer some light when trying to examine any one of these deep and undeveloped problems.

7.5 Comparing the Grammars

To conclude this section, we compare the Ranta's HoTT grammar and our cubicalTT grammar. We hope that doing so offers some final insights into how to approach the problem of syntactic completeness and semantic adequacy.

cubicalTT is in some sense takes expressions as its epicenter, whereby declarations, branches, telescopes, where expressions, and lists offer syntactic sugar so that it becomes a minimally readable programming language. It is a synthetic approach to writing a grammar, whereby one has an a priori idea of what an expression syntactically should be, with the most important feature being that it is inductively generated. It is not really concerned with semantics per-se, because this is the job of the type-checker and evaluator.

HoTT, on the other hand, analyses real text, and decisions about the grammar are made posterior to observing phenomena. The grammar makes distinction between `Formulas`, namely expressions with symbolic support for latex, `Framework` which allows one to construct natural language sentences, and a `HottLexicon`. This grammar, while having some inductive notion of what an expression is, puts the bulk of work in producing valid sentences in `Framework`.

```
cat
  Paragraph ;        -- definition, theorem, etc
  Definition ;       -- definition of a new concept
  Assumption ;       -- assumption in a proof  -- let ...
   [Assumption]{1} ;   -- list of assumptions in one sentence --
let ... and ...
  Conclusion ;       -- conclusion in a proof -- thus P
   Prop ;                -- proposition (sentence or formula) --
A is contractible
  Sort ;             -- set, type, etc corresponding to a common noun
 Ind ;          -- individual, element, corresponding to a singular term
  Fun ;              -- function with individual value
  Pred ;             -- predicate: function with proposition value --
contractible
  [Ind] ;           -- list of individual expressions   -- 1, 2 and 3
 UnivPhrase ;      -- universal noun phrase        -- for all x,y : A
  ConclusionPhrase ; -- conclusion word               -- hence
  Label ;           -- name/number of definition, theorem, etc -- Id-
induction
  Title ;            -- title for theorem, definition, etc
```

The distinction between individuals, propositions, sorts, functions, and predicates also allows more nuance, but delegates the work of deciding what category a term represents much more difficult, which makes the possibility of having some algorithm infer the right category much more difficult. The expressions can be embedded into any of these categories. Additionally, we see that the universal phrase, the notion of a Π-type, merits semantic distinction in this grammar, with unique functions being assigned for all the (observed) ways of saying it - this is the case with existential statements as well.

```
  plainUnivPhrase   : [Var] -> Sort -> UnivPhrase ;  -- for x, y : A
  eachUnivPhrase    : [Var] -> Sort -> UnivPhrase ;  -- for each x,y : A
  allUnivPhrase     : [Var] -> Sort -> UnivPhrase ;  -- for all x,y : A
  ifUnivPhrase      : [Var] -> Sort -> UnivPhrase ;  -- if x,y : A
  if_thenUnivPhrase : [Var] -> Sort -> UnivPhrase ;  -- if x,y : A then
```

One caveat is that set comprehensions are treated as expressions, whereas existential phrases are propositions, even though to the Agda programmer they are the same thing. This differences obviously arises in the fact that expressions are meant to be symbolic in this grammar, whereas functions taking `Exp` arguments generally return things of grammatical categories with possibly auxiliary data, i.e.

```
lincat
  Sort = SortExp ;
  Fun  = FunExp ;
  Ind  = IndExp ;
```

```
  Prop = S ;
oper
  SortExp = {cn : CN ; postname : Str ; isSymbolic : Bool} ;
  IndExp = {s : NP ; isSymbolic : Bool} ;
  FunExp = {s : CN ; isSymbolic : Bool} ;
```

Ranta clearly chose to treat prioritize semantic adequacy by placing the grammatical categories at the forefront. This was not an error, as Peter Azcel wrote this mixing notations from set theory, type theory, first order logic, and homotopy theory presented in a Latex document. For as much as the type theorist insists on his or her exclusive use of types, the written language tradition is still tied to the logical and set theoretic tradition of presenting mathematics - and this constraint arises in a more expressive abstract syntax.

This includes document structure categories, `Title`, `Label`, `Paragraph`, `Definition`, `Conclusion`, etc. While these may resembling a module system in ways, they also reflect a different semantic sense than Agda's module system, which gives the programmer greater control of handling software complexity. `ConclusionPhrase` reflects what Agda's typechecker infers and is displayed to the user, and is therefore redundant from the programmers perspective.

Another observation of the grammar is that the certain notions come with more semantic information that the type-checker would be able to infer, so for instance, `fiberExp` is a binary function, as opposed to the cubicalTT grammar which treats it as a variable and explicitly uses the application operator for each of its arguments, leading to the application hell observed in our earlier work.

Despite the complexity of the abstract syntax relative to cubicalTT, it is remarkable that Ranta was able to capture the entire text with a few days of labor. Expertise in GF, however, reveals itself through trial, error, and patience. Despite the success, we hypothesize that extending it to longer lengths of text would very difficult for anyone without deep knowledge of GF and type theory generally. The ease of extending cubicalTT to cover more text, , despite its limitations regarding language generation, pose a dual problem of extending the concrete syntax each time a new grammatical "feature" is discovered.

We have taken the text parsed by Ranta's HoTT grammar and implemented both an Agda representation which type-checks as well as the cubicalTT syntax in the appendix 9.3.

7.5.1 Ideas for resolution

Based off these comparisons, we now propose a road-map for future investigations of how to build a "master grammar", which should ideally seek to do at least the following:

- Allow for expressive natural language - maximize *semantic adequacy*

- Enable parsing of a real programming language - ensure *syntactic completeness*
- Allow GF developers to expand the grammar in a compositional, modular, safe, reliable, and methodologically precise way
- Enable long-term integration of the grammars into practical tools for mathematicians and computer scientists

We therefore believe there is a set of principles one can follow to achieve these goals : namely, start with a small, syntactically precise core, and extend it based off the needs of either the programmer or the mathematician.

Let's suppose that our hypothetical "core" should consist of a desugared type theory with Π, Σ, and Equality types, with their respective introduction and elimination forms, inductive definitions and a means of case analysis, and declarations for building types and terms. We could then *extend* this with telescopes for syntactic sugar, where and let bindings to allow for local definitions, and modules to allow for the basic needs of a suitable programming language - and we'd essentially have the cubicalTT grammar. One thing to be emphasized is that the extension should already map to the core. As was noted when we discussed Haskell transformations for logic 6.1, the mapping $[\![-]\!] : Extended \to Core$ can follow relatively conventional techniques.

This can then be extended again to include more nuance that a particular Agda programmer might desire : unicode support, universes, Agda-style pattern matching, cubical primitives (although this *fundamentally* changes the underlying type theory), higher inductive types, and many other possible add-ons. While important to capture more and more, it should be noted that creating a GF grammar capable of parsing all of Agda would be overkill, and working with Agda's existing parser would probably be preferred at some point if for no other reason than that the myriad of features would create a grammar that would no longer presumable be feasible for natural language generation.

Once the grammar for the logical framework has been established, the grammar writer would then have the lexical data, specific to the domain being modeled - our two case studies previous being natural numbers propositions and notions from homotopy type theory. This presents the challenge of how "deep" does one wish to embed the domain into GF. For our cubicalTT parser, we chose the shallowest possible embedding, whereby every term was just a *variable* with no semantic distinction. In the grammar for QA, we chose the deepest possible embedding, with `Nat` being a distinguished category, not just a function. While this is convenient for the example, it was only so because we could coerce the builtin number to it, i.e. Int -> Nat. Unless one intends to use GF's dependent types, this deep embedding is likely unnecessary, and in some sense creates to much semantic space in the grammar.

The "in-between" depth is to include `Nat` as an expression, whereby the zero, successor, and induction principle, included as functions, retain their arities from the actual programming language, but don't actually specify what types of expressions work for them - this work is delegated to the type-checker. While this has the bene-

fit of disallowing the "application hell" we saw in our unreadable natural language proof, it also requires what we'll passively call "arity inference", and therefore some components of the type-checker would be needed to scale this depth of embedding functions into Haskell. Additionally, the use of the phrase "successor function" to refer to η-expanded form in contrast to "the successor of" reveals the deep difficulties of how to delegate unique linguistic forms to all the possible arity assignments, something that a programming language can infer automatically based of a term's use.

Once the grammar has been extended *enough* such that it satisfies the programmers needs, the even more difficult aspect of extending it to satisfying the mathematician would come into play. One could have categories for things like sets and propositions, as Ranta does in HoTT. These extensions of the $\Pi, \Sigma, \equiv$ core, if given a Haskell embedding, could be made such that any proposition in the extended langauge would be mapped to its type in the core language. Therefore, in Ranta's code, the existential propositions and set comprehension syntax could be evaluated, via Haskell, to the Σ type in the core grammar, thereby allowing for at least a hypothetical translation from semantically sound utterances to syntactically complete ones. While this sounds good, we offer a conterexample, namely the definition of a left coset in group theory, $gH = \{gh{:}h \in H\}\ \forall g \in G$, because $h \in H$ is a judgment and not a type. Indeed, quotients, subsets, and subgroups in type theory must be treated differently than their set-theoretic counterparts whereby the sets must be given encodings which break the propositions as sets mentality that we're used to [63]. Additionally, in the reverse direction, taking a Σ type and generating a natural language utterance which may be of the type, set, or one the many propositional flavors would require some pragmatic knowledge that our system is not capable of handling.

Our proposal is to build a core, *syntactically complete* grammar similar to cubicalTT and extend it to a *semantically adequate* which follows Ranta's HoTT grammar methodology. It should be based of the condition that they are coherent in a sense that the extended grammar can be compositionally evaluated to the core, via a Haskell function, which obviously borrows the approach taken from the CADE logic grammar. This proposal is based off extensive trial and error with these respective grammars, witnessing their respective strengths and deficiencies, and weaving these strengths as cogently as is possible.

Typechecker's Role One of the difficult things in dealing with all of this is coming to terms with the strengths and weaknesses of a purely, or at least mostly syntactical approach to translation from machine to natural language. One of the central pieces of a programming language missing from this approach is the role of the typechecker. In dependently typed languages like Agda, where the typechecker evaluates programs in types to a canonical form, this is especially acute - for the typechecker tells you when a proof is valid. For the mathematician, a proof may be valid even if it doesn't type-check, because they can more or less account for many details which ensure that an argument is articulated honestly and truthfully - and while there may be small errors, presumptions, or holes in a syntactic proof, this ultimately doesn't detract from the semantic ideas being portrayed and

perceived.

The coherence of a semantically adquate and syntactically complete object via some intermediary form like an AST, sadly doesn't seem feasible, without some kind of verification procedure in between. We imagine that a semantically adequate proof, may generate through this idealized system an Agda proof with holes, for instance, which could either be filled in with tactics or with help through Agda's interactive proof development system.

We know that despite its intractability, this should be a long term goal for whoever continues this project. Even if there's not an equivalence between syntactically complete and semantically adequate objects, it is feasible that one can come up with ways of approximating one inside the other's domain, and we believe the power of dependent type theories may give us one way of achieving this approximation.

8 Conclusion

> Concrete syntax is in some sense where programming language theory meets psychology. *Robert Harper*

There are two major problems in the reformulation of mathematics via typed languages which underlay interactive proofs assistants that fall under the scope of this thesis tried to grapple with.

The first is how to make a dependently typed programming language capable of formulating proposition and proof more amenable to mathematicians with the goal of improving semantic adequacy. The second seeks to ask how to facilitate the formalization of mathematics, whether that be translation of theorem statements, proofs, or giving the mathematicians a template for using CNLs that are more suited to their tastes and also capable of providing tractable data in various applications.

Progress in either of these directions will only be realized and through significant time, labor, expertise, and most vitally, *original thinking* through collaborative efforts. It is uncertain what the role of parsers, abstract syntax trees, linearization schemes, and other components of the GF ecosystem will have on these efforts, but we hope that our efforts have shown that its very much an open plane of exploration.

Our work has perhaps only made a small contribution to these incredibly difficult problems, yet, we hope that compiling various ideas across many different fields have at least given some philosophical clarification as to why the problems are so difficult. Additionally, we hope our proposal to the GF means of approaching these problems through the analysis and comparison different grammars gives legitimation and evidence that there's a feasibility of actually applying these ideas to solve real problems, or at the very least, asking important questions that may influence other methods. For there is no doubt a role to play for statistical methods in tackling these problems as well, How these data-based methods along with the rule-based techniques is also incredibly speculative and merits careful consideration and research.

Our contributions, partially original and partially extrapolated from others, are the following :

- Introduce notions of *syntactic completeness* and *semantic adequacy*, so-as to allowing understanding a piece of mathematics based off its degree of formality as well as clarity of presentation
- Recognition that the GF approach is limited, especially as regards pragmatic concerns and other philosophical stances about mathematics, like the role of visualization in mathematical thinking and presentation
- Offer explicit comparisons, through examples of mathematics in a textual form and a type theoretic presentation
- The developments of new GF grammars for analyzing this problem

- The first comparison of all known GF grammars in this domain with respect to *syntactic completeness* and *semantic adequacy*
- The development of an Agda library which mirrors the HoTT book so that future work can seek a possible “large-scale” translation case study

It has been remarked that the bigger grammars gets, the more it begins to resemble a domain specific RGL [2]. We advocate to actually produce a “formal language RGL”, whereby many of the ideas advocated and observed in this work, like document structure, latex (and symbolic support generally), custom lexical classes (like in BNFC), and many other features not witnessed in the small case studies undertaken here should be accounted for. Therefore, the grammar writer’s time could be better spent either focusing on the scaling of programming language features or the actual linguistic analysis of mathematics text - thereby making a more natural CNLs.

Despite the promise of various topics discussed here like Cubical Agda, the Formal Abstracts Project , and the use of Lean in mathematics education [10], we don’t foresee a convergence of type theorists and mathematicians, even though belief the holy trinity would compel us to think so. GF as a PL paradigm, applied to this problem, gives us a stark contrast of how different these two approaches to mathematical language. For the grammars of proof are insufficient to capture the complexity and nuance about the language of proof, so much of which has yet to captured in an existing linguistic framework. Grammars for propositions and definitions offer a much more limited and seemingly feasible problem, especially with GF largely because mathematicians write them with the explicit intention of being comprehensible and unambiguous. We hope this works serves the mathematical community in achieving some of these goals.

8.1 The Mathematical Library of Babel

The Library of Babel [**?**] was a profound mirror held up to the human species as regards our comprehension of the world through language. It reflected our inability to grasp and reconcile our own finitude. The infinite stack of shelves, containing every book with every permutation of letters from the Hebrew alphabet, leaves the humans who inhabit the closed space in a state of discord as regards their failures to navigate and interpret the myriad texts.

The Library most certainly contains all mathematical statements, with all possible foundations of mathematics, theorems, proofs of those theorems, in all the possible syntactic presentations. In addition, it contains a catalogue documenting the mathematical constructions, and how these constructions can be encoded in the multiplicity of foundational systems. If there is a master GF grammar for translating all of the mathematics in *The Library*, it certainly contains the source code for that as well.

Unfortunately, the library also contains all the erroneous proofs, whether they be lexical errors or a reference to flawed lemma somewhere much deeper in the library. There are certainly proofs of the Riemann conjecture, its negation, and its

undecidability. When perceives mathematics through the lens of human language generally, we must acknowledge that mathematical content, constructions, and discoveries, are not developments that come by chance, through sifting through bags of words until some gemstone gleams through the noise. Humans have to produce mathematical constructions through hard labor, sweat, and tears. More importantly we create mathematics through dialogue, laughter, and occasionally even dreams.

To imbue the sentences of mathematics which we see on paper or in the terminal with meaning we have some kind of internal mental mechanism that is at play with our other mental faculties : our motor system and sensory capabilities generally. We don’t merely derive formulas by computing, but we distill ideas in our linguistic capacity to some kind of unambiguous kernel. The view of mathematics, that is just some subset of *The Library*, waiting to be discovered or verified by a machine, is an incredibly misinformed and myopic view of the subject. That mathematics is a human endeavor, complete with all our lust, flaws, and ingenuity should be more clear after contemplating how difficult it is to construct a grammar of proof.

References

[1] Abel, A. (2020). Personal correspondance.

[2] Angelov, K. (2021). Meet the grammarian: Why or do we need to write application grammars? GF Summer School.

[3] Avigad, J. (2015). Mathematics and language.

[4] Bar, K., Kissinger, A., & Vicary, J. (2016). Globular: an online proof assistant for higher-dimensional rewriting.

[5] Bezem, M., Coquand, T., & Huber, S. (2017). The univalence axiom in cubical sets.

[6] Bos, J., Clark, S., Steedman, M., Curran, J. R., & Hockenmaier, J. (2004). Wide-coverage semantic representations from a CCG parser. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics* (pp. 1240–1246). Geneva, Switzerland: COLING.

[7] Bove, A. & Dybjer, P. (2009). *Dependent Types at Work*, (pp. 57–99). Springer Berlin Heidelberg: Berlin, Heidelberg.

[8] Bove, A., Dybjer, P., & Norell, U. (2009). A brief overview of agda – a functional language with dependent types. In S. Berghofer, T. Nipkow, C. Urban, & M. Wenzel (Eds.), *Theorem Proving in Higher Order Logics* (pp. 73–78). Berlin, Heidelberg: Springer Berlin Heidelberg.

[9] Brunerie, G. (2016). On the homotopy groups of spheres in homotopy type theory.

[10] Buzzard, K. (2020). When will computers prove theorems?

[11] Chomsky, N. (1995). *The Minimalist Program*. MIT Press.

[12] Chomsky, N. (2009). *Syntactic Structures*. De Gruyter Mouton.

[13] Cohen, C., Coquand, T., Huber, S., & Mörtberg, A. (2015). Cubical Type Theory: a constructive interpretation of the univalence axiom. In *21st International Conference on Types for Proofs and Programs*, number 69 in 21st International Conference on Types for Proofs and Programs (pp. 262). Tallinn, Estonia: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[14] Coquand, T. (1992). Pattern matching with dependent types.

[15] Coscoy, Y., Kahn, G., & Théry, L. (1995). Extracting text from proofs. In M. Dezani-Ciancaglini & G. Plotkin (Eds.), *Typed Lambda Calculi and Applications* (pp. 109–123). Berlin, Heidelberg: Springer Berlin Heidelberg.

[16] Cramer, M., Fisseni, B., Koepke, P., Kühlwein, D., Schröder, B., & Veldman, J. (2009). The naproche project controlled natural language proof checking of mathematical texts. In *International Workshop on Controlled Natural Language* (pp. 170–186).: Springer.

[17] Czajka, Ł. & Kaliszyk, C. (2018). Hammer for coq: Automation for dependent type theory. *Journal of automated reasoning*, 61(1), 423–453.

[18] de Bruijn, N. G. (1983). *AUTOMATH, a Language for Mathematics*, (pp. 159–200). Springer Berlin Heidelberg: Berlin, Heidelberg.

[19] Escardó, M. H. (2020). Introduction to univalent foundations of mathematics with agda.

[20] Fong, B. (2016). The algebra of open and interconnected systems.

[21] Forsberg, M. & Ranta, A. (2004). Functional morphology. *SIGPLAN Not.*, 39(9), 213–223.

[22] Frege, G. (1879). *Begrisschrift*. Halle.

[23] Ganesalingam, M. (2013). The language of mathematics. In *The language of mathematics* (pp. 17–38). Springer.

[24] Giaquinto, M. (2020). The Epistemology of Visual Thinking in Mathematics. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Spring 2020 edition.

[25] Gödel, K. (1994). The qed manifesto. *Lecture Notes in Artificial Intelligence*, 814, 238–251.

[26] Hales, T. (2019). An argument for controlled natural languages in mathematics.

[27] Hallgren, T. & Ranta, A. (2000). An extensible proof text editor. In *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning*, LPAR'00 (pp. 70–84). Berlin, Heidelberg: Springer-Verlag.

[28] Harper, R. (2011). The holy trinity.

[29] Jackson, A. (2002). The world of blind mathematicians.

[30] 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).

[31] Kaliszyk, C. & Rabe, F. (2020). A survey of languages for formalizing mathematics. In C. Benzmüller & B. Miller (Eds.), *Intelligent Computer Mathematics* (pp. 138–156). Cham: Springer International Publishing.

[32] Kamp, H., Van Genabith, J., & Reyle, U. (2011). Discourse representation theory. In *Handbook of philosophical logic* (pp. 125–394). Springer.

[33] Ljunglöf, P. (2004). Expressivity and complexity of the grammatical framework.

[34] Luo, Z. (2012). Common nouns as types. In D. Béchet & A. Dikovsky (Eds.), *Logical Aspects of Computational Linguistics* (pp. 173–185). Berlin, Heidelberg: Springer Berlin Heidelberg.

[35] Macmillan, W. (2020). *GF-Typechecker*. https://github.com/wmacmil/GF_Typechecker.

[36] Martin-Löf, P. (1984). *Intuitionistic type theory*, volume 1 of *Studies in proof theory*. Bibliopolis.

[37] 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.

[38] Martin-Lof, P. & Lozinski, Z. A. (1984). Constructive mathematics and computer programming [and discussion]. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 312(1522), 501–518.

[39] Martin-Löf, P. (1987). Truth of a proposition, evidence of a judgement, validity of a proof. *Synthese*, 73(3), 407–420.

[40] Montague, R. (1973). *The Proper Treatment of Quantification in Ordinary English*, (pp. 221–242). Springer Netherlands: Dordrecht.

[41] Nelsen, R. (1993). *Proofs Without Words*.

[42] Pfenning, F. (2009). Lecture notes on harmony.

[43] 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.

[44] Pédrot, P.-M. (2021). Ltac2 : Tactical warfare.

[45] Ranta, A. (1994a). *Type-theoretical Grammar*. Indices (Claredon). Clarendon Press.

[46] Ranta, A. (1994b). Type theory and the informal language of mathematics. In H. Barendregt & T. Nipkow (Eds.), *Types for Proofs and Programs* (pp. 352–365). Berlin, Heidelberg: Springer Berlin Heidelberg.

[47] RANTA, A. (2004). Grammatical framework. *Journal of Functional Programming*, 14(2), 145–189.

[48] Ranta, A. (2011a). *Grammatical framework: Programming with multilingual grammars*, volume 173. CSLI Publications, Center for the Study of Language and Information Stanford.

[49] Ranta, A. (2011b). Translating between language and logic: what is easy and what is difficult. In *International Conference on Automated Deduction* (pp. 5–25).: Springer.

[50] Ranta, A. (2013). *gf-contrib/typetheory*. https://github.com/GrammaticalFramework/gf-contrib/tree/master/typetheory.

[51] Ranta, A. (2014). Translating homotopy type theory in grammatical framework.

[52] Ranta, A. (2019). *From machine readable Z to engineer and manager readable English* . www.grammaticalframework.org/~aarne/final-dg-zed.pdf.

[53] Ranta, A. (2020). Some remarks on pragmatics in the language of mathematics: Comments to the paper "at least one black sheep: Pragmatics and mathematical language" by luca san mauro, marco ruffino and giorgio venturi. *Journal of Pragmatics*, 160, 120–122.

[54] Rudnicki, P. (1992). An overview of the mizar project. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs* (pp. 311–330).

[55] Ruffino, M., San Mauro, L., & Venturi, G. (2020). At least one black sheep: Pragmatics and mathematical language. *Journal of Pragmatics*, 160, 114–119.

[56] Schaefer, J. F. & Kohlhase, M. (2020). The glif system: A framework for inference-based natural-language understanding.

[57] Stump, A. (2016). *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan Claypool.

[58] The Univalent foundations program & Institute for advanced study (Princeton, N. (2013). *Homotopy Type Theory: Univalent Foundations of Mathematics*.

[59] Vezzosi, A., Mörtberg, A., & Abel, A. (2019). Cubical agda: A dependently typed programming language with univalence and higher inductive types. *Proc. ACM Program. Lang.*, 3(ICFP).

[60] Wadler, P., Kokke, W., & Siek, J. G. (2020). *Programming Language Foundations in Agda*.

[61] Wang, Q., Brown, C., Kaliszyk, C., & Urban, J. (2020). Exploration of neural machine translation in autoformalization of mathematics in mizar. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020 (pp. 85–98). New York, NY, USA: Association for Computing Machinery.

[62] Wenzel, M. et al. (2004). The isabelle/isar reference manual.

[63] Zipperer, A. (2016). *A formalization of elementary group theory in the proof assistant Lean*. PhD thesis, Master's thesis, Carnegie Mellon University.

9 Appendix

9.1 Twin Primes Conjecture Revisited

We now give the dependent uncurring from the functions from **??** We note that this perhaps is a bit more linguistically natural, because we can refer to definitions of a prime number, sucessive prime numbers, etc. We leave it to the reader to decide which presentation would be better suited for translation.

```
prime = Σ[ p ∈ ℕ ] isPrime p

isSuccessivePrime : prime → prime → Set
isSuccessivePrime (p , pIsPrime) (p' , p'IsPrime) =
  ((p'' , p''IsPrime) : prime) →
  p ≤ p' → p ≤ p'' → p' ≤ p''

successivePrimes =
  Σ[ p ∈ prime ] Σ[ p' ∈ prime ] isSuccessivePrime p p'

primeGap : successivePrimes → ℕ
primeGap ((p , pIsPrime) , (p' , p'IsPrime) , p'-is-after-px) = p - p'

nth-pletPrimes : successivePrimes → ℕ → Set
nth-pletPrimes (p , p' , p'-is-after-p) n =
  primeGap (p , p' , p'-is-after-p) ≡ n

twinPrimes : successivePrimes → Set
twinPrimes sucPrimes = nth-pletPrimes sucPrimes 2

twinPrimeConjecture : Set
twinPrimeConjecture = (n : ℕ) →
  Σ[ sprs@((p , p'), p'-after-p) ∈ successivePrimes ]
    (p ≥ n)
  × twinPrimes sprs
```

9.2 cubicalTT

```
abstract Exp = {

flags startcat = Decl ;
    -- note, cubical tt doesn't support inductive families, and therefore the datat

cat
  Comment ;
  Module  ;
  AIdent ;
  Imp ; --imports, add later
  Decl ;
  Exp ;
```

```
  ExpWhere ;
  Tele ;
  Branch ;
  PTele ;
  Label ;
  [AIdent]{0} ; -- "x y z"
  [Decl]{1} ;
  [Tele]{0} ;
  [Branch]{1} ;
  [Label]{1} ;
  [PTele]{1} ;
  -- [Exp]{1};

fun

  DeclDef : AIdent -> [Tele] -> Exp -> ExpWhere -> Decl ;
  -- foo ( b : bool ) : bool = b
  DeclData : AIdent -> [Tele] -> [Label] -> Decl ;
  -- data nat : Set where zero | suc ( n : nat )
  DeclSplit : AIdent -> [Tele] -> Exp -> [Branch] -> Decl ;
  -- caseBool ( x : Set ) ( y z : x ) : bool -> Set = split false -
> y || true -> z
  DeclUndef : AIdent -> [Tele] -> Exp -> Decl ;
   -- funExt ( a : Set ) ( b : a -> Set ) ( f g : ( x : a ) -
> b x ) ( p : ( x : a ) -> ( b x ) ( f x ) == ( g x ) ) : ( ( y : a ) -
> b y ) f == g = undefined

  Where : Exp -> [Decl] -> ExpWhere ;
  -- foo ( b : bool ) : bool =
  -- f b where f : bool -> bool = negb
  NoWhere : Exp -> ExpWhere ;
  -- foo ( b : bool ) : bool =
  -- b

  Split : Exp -> [Branch] -> Exp ;
  --split@ ( nat -> bool ) with zero  -> true || suc n -> false
  Let : [Decl] -> Exp -> Exp ;
  -- foo ( b : bool ) : bool =
  -- let f : bool -> bool = negb in f b
  Lam : [PTele] -> Exp -> Exp ;
  -- \\ ( x : bool ) -> negb x
  -- todo, allow implicit typing
  Fun : Exp -> Exp -> Exp ;
  -- Set -> Set
  -- Set -> ( b : bool ) -> ( x : Set ) -> ( f x )
  Pi    : [PTele] -> Exp -> Exp ;
  --( f : bool -> Set ) -> ( b : bool ) -> ( x : Set ) -> ( f x )
  -- ( f : bool -> Set ) ( b : bool ) ( x : Set ) -> ( f x )
  Sigma : [PTele] -> Exp -> Exp ;
```

```
  -- ( f : bool -> Set ) ( b : bool ) ( x : Set ) * ( f x )
  App : Exp -> Exp -> Exp ;
  -- proj1 ( x , y )
  Id : Exp -> Exp -> Exp -> Exp ;
  -- Set bool == nat
  IdJ : Exp -> Exp -> Exp -> Exp -> Exp -> Exp ;
  -- J e d x y p
  Fst : Exp -> Exp ; -- "proj1 x"
  Snd : Exp -> Exp ; -- "proj2 x"
  -- Pair : Exp -> [Exp] -> Exp ;
  Pair : Exp -> Exp -> Exp ;
  -- ( x , y )
  Var : AIdent -> Exp ;
  -- x
  Univ : Exp ;
  -- Set
  Refl : AIdent ; -- Exp ;
  -- refl
  --Hole : HoleIdent -> Exp ; -- need to add holes

  OBranch :  AIdent -> [AIdent] -> ExpWhere -> Branch ;
   -- suc m -> split@ ( nat -> bool ) with zero -> false || suc n -
> equalNat m n
  -- for splits

  OLabel : AIdent -> [Tele] -> Label ;
  -- suc ( n : nat )
  -- fora data types

  -- construct telescope
  TeleC : AIdent -> [AIdent] -> Exp -> Tele ;
  -- "( f g : ( x : a ) -> b x )"
   -- ( a : Set ) ( b : ( a ) -> ( Set ) ) ( f g : ( x : a ) -
> ( ( b ) ( x ) ) ) ( p : ( x : a ) -> ( ( ( b ) ( x ) ) ( ( f ) ( x ) ) == ( ( g ) ( x ) ) ) )

  -- why does gr with this fail so epically?
  PTeleC : Exp -> Exp -> PTele ;
  -- ( x : Set ) -- ( y : x -> Set )" -- ( x : f y z )"

  --everything below this is just strings

  Foo : AIdent ;
 A , B , C , D , E , F , G , H , I , J , K , L , M , N , O , P , Q , R , S , T , U , V , W , X
  True , False , Bool : AIdent ;
  NegB : AIdent ;
  CaseBool : AIdent ;
  IndBool : AIdent ;
  FunExt : AIdent ;
  Nat : AIdent ;
```

```
  Zero : AIdent ;
  Suc : AIdent ;
  EqualNat : AIdent ;
  Unit : AIdent ;
  Top : AIdent ;
  Contr : AIdent ;
  Fiber : AIdent ;
  IsEquiv : AIdent ;
  IdIsEquiv : AIdent ;
  IdFun : AIdent ;
  ContrSingl : AIdent ;
  Equiv : AIdent ;
  EqToIso : AIdent ;
  Ybar : AIdent ;
  IdFib : AIdent ;
  Identity : AIdent ;
  Lemma0 : AIdent ;
}

concrete ExpCubicalTT of Exp = open Prelude, FormalTwo in {

lincat
  Comment,
  Module ,
  AIdent,
  Imp,
  Decl ,
  ExpWhere,
  Tele,
  Branch ,
  PTele,
  Label,
    -- = Str ;
  [AIdent],
  [Decl] ,
  -- [Exp],
  [Tele],
  [Branch] ,
  [PTele],
  [Label]
    -- = {hd,tl : Str} ;
    = Str ;
  Exp = TermPrec ;

lin

  DeclDef a lt e ew = a ++ lt ++ ":" ++ usePrec 0 e ++ "=" ++ ew ;
  DeclData a t d = "data" ++ a ++ t ++ ": Set where" ++ d ;
 DeclSplit ai lt e lb = ai ++ lt ++ ":" ++ usePrec 0 e ++ "= split" ++ lb ;
```

```
  DeclUndef a lt e = a ++ lt ++ ":" ++ usePrec 0 e ++ "= undefined" ; -
- postulate in agda

  Where e ld = usePrec 0 e ++ "where" ++ ld ;
  NoWhere e = usePrec 0 e ;

  Let ld e = mkPrec 0 ("let" ++ ld ++ "in" ++ (usePrec 0 e)) ;
  Split e lb = mkPrec 0 ("split@" ++ usePrec 0 e ++ "with" ++ lb) ;
  Lam pt e = mkPrec 0 ("\\" ++ pt ++ "->" ++ usePrec 0 e) ;
  Fun = infixr 1 "->" ; -- A -> Set
  Pi pt e = mkPrec 1 (pt ++ "->" ++ usePrec 1 e) ;
  Sigma pt e = mkPrec 1 (pt ++ "*" ++ usePrec 1 e) ;
  App = infixl 2 "" ;
 Id e1 e2 e3 = mkPrec 3 (usePrec 4 e1 ++ usePrec 4 e2 ++ "==" ++ usePrec 3 e3) ;
-- for an explicit vs implicit use of parameters, may have to use expressions as recc
 IdJ e1 e2 e3 e4 e5 = mkPrec 3 ("J" ++ usePrec 4 e1 ++ usePrec 4 e2 ++ usePrec 4 e3 ++
  Fst e = mkPrec 4 ("fst" ++ usePrec 4 e) ;
  Snd e = mkPrec 4 ("snd" ++ usePrec 4 e) ;
 Pair e1 e2 = mkPrec 5 ("(" ++ usePrec 0 e1 ++ "," ++ usePrec 0 e2 ++ ")") ;
  Var a = constant a ;
  Univ = constant "Set" ;
  Refl = "refl" ; -- constant "refl" ;

  BaseAIdent = "" ;
  ConsAIdent x xs = x ++ xs ;

  -- [Decl] only used in ExpWhere
  BaseDecl x = x ;
  ConsDecl x xs = x ++ "^" ++ xs ;

  -- maybe accomodate so split on empty type just gives ()
  -- BaseBranch = "" ;
  BaseBranch x = x ;
  -- ConsBranch x xs = x ++ "\n" ++ xs ;
  ConsBranch x xs = x ++ "||" ++ xs ;

  -- for data constructors
  BaseLabel x = x ;
  ConsLabel x xs = x ++ "|" ++ xs ;

  BasePTele x = x ;
  ConsPTele x xs = x ++ xs ;

  BaseTele = "" ;
  ConsTele x xs = x ++ xs ;

  OBranch a la ew = a ++ la ++ "->" ++ ew ;
  TeleC a la e = "(" ++ a ++ la ++ ":" ++ usePrec 0 e ++ ")" ;
  PTeleC e1 e2 = "(" ++ top e1 ++ ":" ++ top e2 ++ ")" ;
```

```
OLabel a lt = a ++ lt ;

--object language syntax, all variables for now
Bool = "bool" ;
True = "true" ;
False = "false" ;
CaseBool = "caseBool" ;
IndBool = "indBool" ;
FunExt = "funExt" ;
Nat = "nat" ;
Zero = "zero" ;
Suc = "suc" ;
EqualNat = "equalNat" ;
Unit = "unit" ;
Top = "top" ;
Foo = "foo" ;
A = "a" ;
B = "b" ;
C = "c" ;
D = "d" ;
E = "e" ;
F = "f" ;
G = "g" ;
H = "h" ;
I = "i" ;
J = "j" ;
K = "k" ;
L = "l" ;
M = "m" ;
N = "n" ;
O = "o" ;
P = "p" ;
Q = "q" ;
R = "r" ;
S = "s" ;
T = "t" ;
U = "u" ;
V = "v" ;
W = "w" ;
X = "x" ;
Y = "y" ;
Z = "z" ;
NegB = "negb" ;
-- everything below is for contractible proofs
Contr = "isContr" ;
Fiber = "fiber" ;
IsEquiv = "isEquiv" ;
IdIsEquiv = "idIsEquiv" ;
```

```
  IdFun = "idfun" ;
  ContrSingl = "contrSingl" ;
  Equiv = "equiv" ;
  EqToIso = "eqToIso" ;
  Identity = "id" ;
  Ybar = "ybar"  ;
  IdFib = "idFib"  ;
  Lemma0 = "lemma0" ;
}
```

The resource FormalTwo.gf merely substitutes more precedences than Formal.gf from the RGL, in the ideal case that we could scale the grammar to include larger and more complicated fixity information.

```
resource FormalTwo = open Prelude in {

----Everything the same up until the definition of Prec in Formal.gf


    Prec : PType = Predef.Ints 9 ;

    highest = 9 ;

    lessPrec : Prec -> Prec -> Bool = \p,q ->
      case <<p,q> : Prec * Prec> of {
       <3,9> | <2,9> | <4,9> | <5,9> | <6,9> | <7,9> | <8,9> => True ;
        <3,8> | <2,8> | <4,8> | <5,8> | <6,8> | <7,8> => True ;
        <3,7> | <2,7> | <4,7> | <5,7> | <6,7> => True ;
        <3,6> | <2,6> | <4,6> | <5,6> => True ;
        <3,5> | <2,5> | <4,5> => True ;
        <3,4> | <2,3> | <2,4> => True ;
        <1,1> | <1,0> | <0,0> => False ;
        <1,_> | <0,_>         => True ;
        _ => False
      } ;

    nextPrec : Prec -> Prec = \p -> case <p : Prec> of {
      9 => 9 ;
      n => Predef.plus n 1
      } ;
```

9.3 Hott and cubicalTT Grammars

```
id : A → A
id = λ z → z

iscontr : (A : Set) → Set
iscontr A = Σ A λ a → (x : A) → (a ≡ x)
```

```
fiber : (A B : Set) (f : A -> B) (y : B) → Set
fiber A B f y = Σ A (λ x → y ≡ f x)

isEquiv : (A B : Set) → (f : A → B) → Set
isEquiv A B f = (y : B) → iscontr (fiber A B f y)

isEquiv' : (A B : Set) → (f : A → B) → Set
isEquiv' A B f = ∀ (y : B) → iscontr (fiber' y)
  where
    fiber' : (y : B) → Set
    fiber' y = Σ A (λ x → y ≡ f x)

-- proof from Aarne
idIsEquiv' : (A : Set) → isEquiv A A (id {A})
idIsEquiv' A y = ybar , help
  where
    fib' : Set -- {y : A}
    fib' = fiber A A id y
    ybar : fib'
    ybar = y , r
    help : (x : fib') → _≡_ {Σ A (_≡_ y)} ybar x
    help = λ {(a , r) → r}

equiv : ( a b : Set ) → Set
equiv a b = Σ (a → b) λ f → isEquiv a b f

equivId : (x : Set) → equiv x x
equivId x = id , (idIsEquiv' x)

eqToIso : ( a b : Set ) → _≡_ {Set} a b → equiv a b
eqToIso a .a r = equivId a
```

Compared with the latex code

Definition: A type A is contractible, if there is $a : A$, called the center of contraction, such that for all $x : A$, $a = x$.
Definition: A map $f : A \to B$ is an equivalence, if for all $y : B$, its fiber, $\{x : A \mid fx = y\}$, is contractible. We write $A \simeq B$, if there is an equivalence $A \to B$.
Lemma: For each type A, the identity map, $1_A := \lambda_{x:A} x : A \to A$, is an equivalence.
Proof: For each $y : A$, let $\{y\}_A := \{x : A \mid x = y\}$ be its fiber with respect to 1_A and let $\bar{y} := (y, r_A y) : \{y\}_A$. As for all $y : A$, $(y, r_A y) = y$, we may apply Id-induction on y, $x : A$ and $z : (x = y)$ to get that

$$(x, z) = y$$

. Hence, for $y : A$, we may apply Σ -elimination on $u : \{y\}_A$ to get that $u = y$, so that $\{y\}_A$ is contractible. Thus, $1_A : A \to A$ is an equivalence. □
Corollary: If U is a type universe, then, for $X, Y : U$,

$$(*) X = Y \to X \simeq Y$$

.
Proof: We may apply the lemma to get that for $X : U$, $X \simeq X$. Hence, we may apply Id-induction on $X, Y : U$ to get that $(*)$. □
Definition: A type universe U is univalent, if for $X, Y : U$, the map $E_{X,Y} : X = Y \to X \simeq Y$ in $(*)$ is an equivalence.

cubicalTT parses the following. We note an idealization : that agda supports ananymous pattern matching, so `\\ ( ( b , refl )` would not typecheck in the original cubicalTT. Additionally, the reflexivity constructor is only present when the identity is inductively defined, as it is a primitive in cubical type theories.

```
id ( a : Set ) : a -> a = \\ ( b : a ) -> b
isContr ( a : Set ) : Set = ( b : a ) * ( x : a ) -> a b == x
fiber ( a b  : Set ) ( f : a -> b ) ( y : b )  : Set
  = ( x : a ) * ( x : a ) -> b y == ( f x )
isEquiv ( a b  : Set ) ( f : a -> b )   : Set
  = ( y : b ) -> isContr ( fiber a b f y )
  where fiber ( a b  : Set ) ( f : a -> b ) ( y : b )  : Set
    = ( x : a ) * ( x : a ) -> b y == ( f x )
equiv ( a b : Set ) : Set = ( f : a -> b ) * isEquiv a b f

idIsEquiv ( a : Set ) : isEquiv a a ( id a ) =  ( ybar , lemma0 )
  where
    idFib : Set = fiber a a id y
    ^ ybar : idFib = ( y , refl )
    ^ lemma0 ( x : idFib ) : ( ( p ) ybar == x )
      = \\ ( ( b , refl ) : ( c : a ) * ( a c == c ) ) -> refl

idIsEquiv ( x : Set ) : equiv x x = ( id , idIsEquiv x )
eqToIso ( a b : Set ) : ( Set a == b ) -> equiv a b
  = split refl -> idIsEquiv a
```

```
Exp>
* DeclDef
    * Contr
      ConsTele
        * TeleC
            * A
              BaseAIdent
              Univ
          BaseTele
      Univ
      NoWhere
        * Sigma
            * BasePTele
                * PTeleC
                    * Var
                        * B
                      Var
                        * A
              Pi
                * BasePTele
                    * PTeleC
                        * Var
                            * X
                          Var
                            * A
                  Id
                    * Var
                        * A
                      Var
                        * B
                      Var
                        * X
```

```
* PredDefinition
    * type_Sort
      A_Var
      contractible_Pred
      ExistCalledProp
        * a_Var
          ExpSort
            * VarExp
                * A_Var
          FunInd
           * centre_of_contraction_Fun
          ForAllProp
            * allUnivPhrase
                * BaseVar
                    * x_Var
                  ExpSort
                    * VarExp
                        * A_Var
              ExpProp
                * DollarMathEnv
                  equalExp
                    * VarExp
                        * a_Var
                      VarExp
                        * x_Var
```

Figure 18: Mathematical Assertions and Agda Judgements

We compare two abstract syntax trees side by side to show that they have quite different structures,

What we notice :

todo : refactor to have the final sections side-by-side, do a more ”thorough analysis of the text fragment above” namely - look at the redundancy, the intro of identity local to a definition (often having more than one proposition in a proposition) the failure in some instances to provide relevant info, etc.

also, refactor to have the sigma proof here

Exp> * DeclDef * IdIsEquiv ConsTele * TeleC * X BaseAIdent Univ BaseTele App * App * Var * Equiv Var * X Var * X NoWhere * Pair * Var * Identity App * Var *

```
* DeclSplit
    * EqToIso
      ConsTele
        * TeleC
            * A
              ConsAIdent
                * B
                  BaseAIdent
              Univ
          BaseTele
      Fun
        * Id
            * Univ
              Var
                * A
              Var
                * B
          App
            * App
                * Var
                    * Equiv
                  Var
                    * A
              Var
                * B
      BaseBranch
        * OBranch
            * Refl
              BaseAIdent
              NoWhere
                * App
                    * Var
                        * IdIsEquiv
                      Var
                        * A
```

```
3 PropParagraph
    * NoConclusionPhrase
      ForAllProp
        * if_thenUnivPhrase
            * BaseVar
                * U_Var
              type_universe_Sort
          ForAllProp
            * plainUnivPhrase
                * ConsVar
                    * X_Var
                      BaseVar
                        * Y_Var
                  ExpSort
                    * VarExp
                        * U_Var
              LabelledExpProp
                * DisplayMathEnv
                  StarLabel
                  mapExp
                    * equalExp
                        * VarExp
                            * X_Var
                          VarExp
                            * Y_Var
                      equivalenceExp
                        * VarExp
                            * X_Var
                          VarExp
                            * Y_Var
```

```
4 ConclusionParagraph
    1 NoConclusionPhrase
      ApplyLabelConclusion
        * the_lemma_Label
          BaseInd
          ForAllProp
            * plainUnivPhrase
                * BaseVar
                    * X_Var
                  ExpSort
                    * VarExp
                        * U_Var
              ExpProp
                * DollarMathEnv
                  equivalenceExp
                    * VarExp
                        * X_Var
                      VarExp
                        * X_Var
    2 henceConclusionPhrase
      ApplyLabelConclusion
        * id_induction_Label
          ConsInd
            * FunInd
                * ExpFun
                    * TypedExp
                        * ConsExp
```

IdIsEquiv Var * X

9.4 HoTT Agda Corpus