UNIVERSITY OF GOTHENBURG

DEPARTMENT OF PHILOSOPHY,
LINGUISTICS AND THEORY OF SCIENCE

ON THE GRAMMAR OF PROOF

Warrick Macmillan

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

Abstract

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

Preface

Acknowledgements, etc.

Contents

2128 TODO Add section numbers to sections Fix figure titles

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.

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.[23]

We begin this discussion of the three relationships between three respective fields, mathematics, computer science, and logic. The aptly named trinity, shown in Figure 12, 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*.

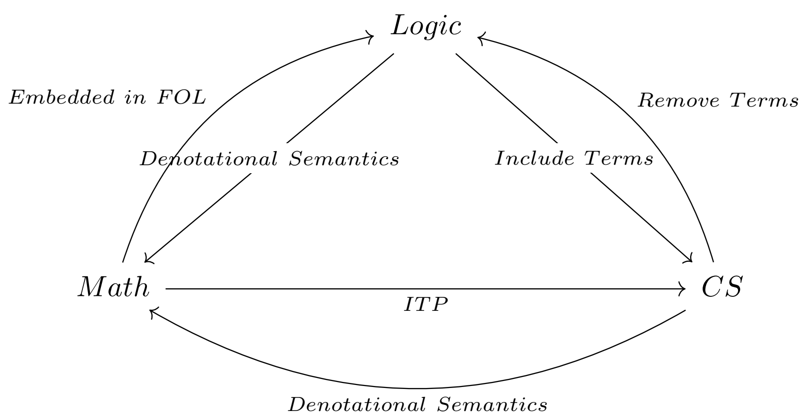

Figure 1: The Holy Trinity

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.

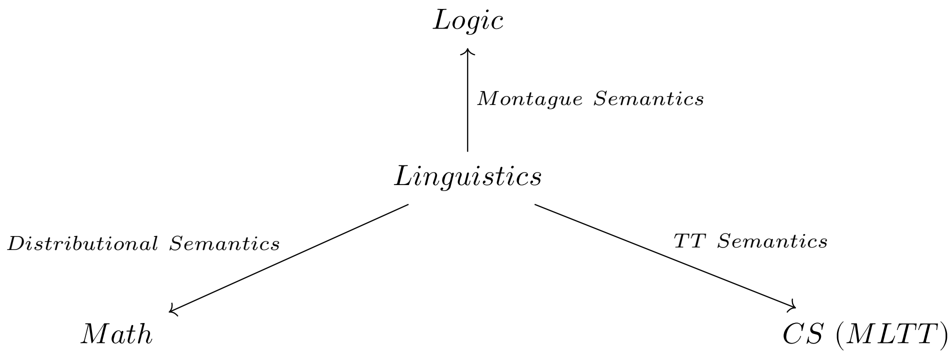

Figure 2: Formal Semantics

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 14, 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.

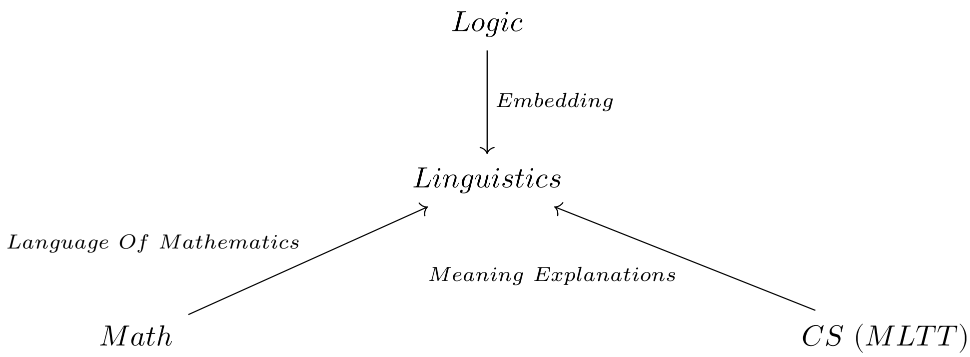

Figure 3: Interpretations of Natural Language

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.

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 $x, y \in G(Notethat$*xy*$ontheleftisformedusingthegroupoperationin$*G*$, whilsttheproduct$*ϕ(x) ϕ(y)*$isformedusingthegroupoperation$*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 [53].

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 [21], and it should practically motivate work.

Practicalities aside, this work also attempts to offer a nuanced philosophical perspective on the matter by exploring why translation of mathematical language, despite it's seemingly structured form, is difficult. We note that the natural language definitions of monoid differ in form, but also in pragmatic content. How one expresses formalities in natural language is incredibly diverse, and Definition 4 as compared with the prior homomorphism definitions is particularly poignant in demonstrating this. These differ very much in nature to the Agda definitions - especially pragmatically. The differences between the Cubical Agda definitions may be loosely called pragmatic, in the sense that the choice of definitions may have downstream effects on readability, maintainability, modularity, and other considerations when trying to write good code, in a burgeoning area known as proof engineering.

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

Perspectives

> ...when it comes to understanding the power of mathematical language to guide our thought and help us reason well, formal mathematical languages like the ones used by interactive proof assistants provide informative models of informal mathematical language. The formal languages underlying foundational frameworks such as set theory and type theory were designed to provide an account of the correct rules of mathematical reasoning, and, as Gödel observed, they do a remarkably good job. But correctness isn't everything: we want our mathematical languages to enable us to reason efficiently and effectively as well. To that end, we need not just accounts as to what makes a mathematical argument correct, but also accounts of the structural features of our theorizing that help us manage mathematical complexity.[2]

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[48], 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 [9] and Frege [17]. 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

$$
\begin{tikzcd}
\mathit{Strings} \arrow[d, "\mathit{Lexical\ Analysis}"] \arrow[dd, bend right=60, "\mathit{Front\ End}"'] \\
\mathit{Lexemes} \arrow[d, "\mathit{Parsing}"] \\
\mathit{ASTs} \arrow[d, "\mathit{Type\ Checker}"] \\
\mathit{Typed\ ASTs} \arrow[d, bend right, "\mathit{Interpreter}"'] \arrow[d, phantom, "\cdots"] \arrow[d, bend left, "\mathit{Code\ Generator}"] \\
\mathit{Target\ Language}
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathit{Phonemes} \arrow[d, dashed, "\mathit{Morhphophonological}" description] \\
\mathit{Morphemes} \arrow[d, "\mathit{Parse}"] \\
\{\ \mathit{Syntactic\ Representation}\ \} \arrow[d, bend right, "\mathit{Montague}"'] \arrow[d, dashed, "\cdots" description] \arrow[d, bend left, "\mathit{Ranta}"] \\
\{\ \mathit{STLC},\ \ldots,\ \mathit{DTT}\ \} \arrow[d, dashed, "?" description] \\
\{\ \mathit{NearalEncoding}\ ,\ \ldots\ \mathit{Internal\ Language}\ \} \arrow[d, dashed, "?" description] \\
\mathit{Phonemes}
\end{tikzcd}
$$

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.

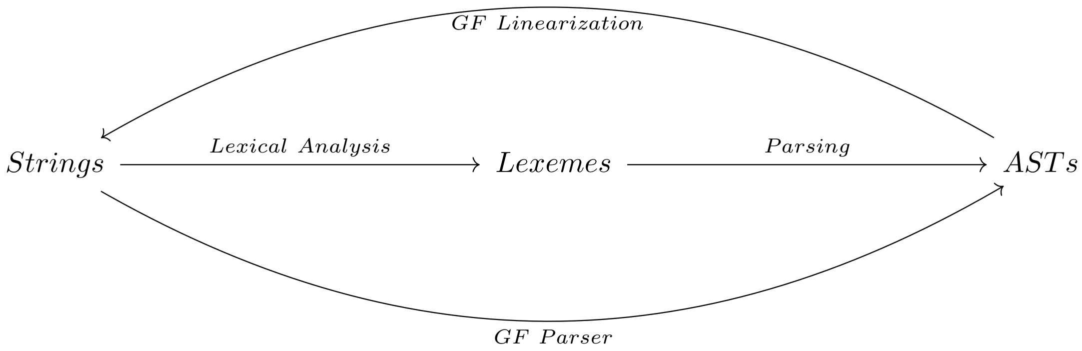

Figure 5: GF in a nutshell

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 [8]. 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.

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.

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 other pedantic details. Whether one asks that semantic adequacy means some kind of sociological consensus among those with relevant expertise, or a more relaxed criterion that some expert herself understands the argument, a dubious perspective in scientific circles, semantic adequacy should appease at least one and potentially more mathematicians.

$$\begin{tikzcd}
Syntactically\ Complete \arrow[r, "Informalization"] & Semantically\ Coherent \arrow[l, bend right, "Formalization"']
\end{tikzcd}$$

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 [26] [40]. 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 [38]. 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.

What is a proof?

> A proof is what makes a judgment evident [32].

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. Alternatively, during the informalization process, two different authors may suppress different details, or phrase a given utterance entirely differently, possibly leading to two different, but possibly similar proofs. Extrapolating our analogy, the same 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.

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 [19]

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" [36] 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 [3]. Additionally, Alecytron supports basic data structure visualization, like red-black trees which carry semantic content less easy in a string based-setting [38].

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 [15].

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 [25]. 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.

Preliminaries

Martin-Löf Type Theory

Judgments

> With Kant, something important happened, namely, that the term judgement, Ger. Urteil, came to be used instead of proposition [32].

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 :

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

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 [24] [33] to see all the rules elaborated in full detail, as well as his philosophical papers [32] [34] to understand type theory as it was conceived both practically and philosophically.

Propositions, Sets, and Types

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 [44], 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
$\forall\ x\ P(x)$	$\Pi x : \tau.\ P(x)$	$for\ all\ x,\ p$	$the\ product\ over\ x\ in\ p$
$\exists\ x\ P(x)$	$\Sigma x : \tau.\ P(x)$	$there\ exists\ an\ x\ such\ that\ p$	$there\ exists\ an\ x\ in\ \tau such\ that\ p$
$p\ \supset\ q$	$p\ \to\ q$	$if\ p\ then\ q$	$p\ to\ q$
$p\ \wedge\ q$	$p\ \times\ q$	$p\ and\ q$	$the\ product\ of\ p\ and\ q$
$p\ \vee\ q$	$p\ +\ q$	$p\ or\ q$	$the\ coproduct\ of\ p\ and\ q$
$\neg\ p$	$\neg\ p$	$it\ is\ not\ the\ case\ that\ p$	$not\ p$
$\top$	$\top$	$true$	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	$the\ set\ of\ natural\ numbers$	$the\ type\ nat$
$S \times T$	$S \times T$	$the\ product\ of\ S\ and\ T$	$the\ product\ of\ S\ and\ T$
$S \to T$	$S \to T$	$the\ function\ \ S\ to\ T$	$p\ to\ q$
$\{x \mid P(x)\}$	$\Sigma x : \tau.\ P(x)$	$the\ set\ of\ x\ such\ that\ P$	$there\ exists\ an\ x\ in\ \tau such\ that\ p$
$\emptyset$	$\bot$	$the\ empty\ set$	$bottom$
?	$\top$	?	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 $Nat\ \to\ Nat$.

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.

Agda

Overview

Agda is an attempt to faithfully formalize Martin-Löf's intensional type theory [24]. 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 [5] [52] [6] [54] so we'll only give a cursory overview of what is relevant for this thesis, with a particular emphasis on the syntax.

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

- 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

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

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

```
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 [37]. Agda's pattern matching is equivalent to the deriveable dependently typed elimination forms [10], 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.

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

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

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

```
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 syntactic 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 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 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.

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

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

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

```
natrec : {X : Set} → ℕ → X → (ℕ → X → X) → X
```

natrec zero e_0 e_1 = e_0
natrec (suc n) e_0 e_1 = e_1 n (natrec n e_0 e_1)

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.

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

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

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

natind : $\{X : \mathbb{N} \to \text{Set}\} \to (n : \mathbb{N}) \to X$ zero $\to ((n : \mathbb{N}) \to X\ n \to X$ (suc n)) $\to X\ n$
natind zero *base step* = *base*
natind (suc n) *base step* = *step* n (natind n *base step*)

natrec' : $\{X : \text{Set}\} \to \mathbb{N} \to X \to (\mathbb{N} \to X \to X) \to X$
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.

Formalizing The Twin Prime Conjecture

Inspired by Escardos's formalization of the twin primes conjecture [14], 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.

```
--TODO ADD to the appendix
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[ pr@(p , pIsPrime) ∈ prime ]
  Σ[ pr’@(p’ , p’IsPrime) ∈ prime ]
  Σ[ pr-after-pr’ ∈ isSuccessivePrime’ pr pr’ ]
    (p ≥ n)
  × twinPrimes (pr , pr’ , pr-after-pr’)
```

While working on this example, I tried to prove that 2 is prime in Agda, which 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. As is hopefully revealed by this discussion, seemingly trivial things, when treated by the type theorist or linguist, can become wonderful areas of exploration.

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 [48], 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 [27]. 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.

Ranta

The initial considerations of Ranta were both oriented towards the language of mathematics [41], as well as purely linguistic concerns [48]. 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 [30]. 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 [35], 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 [48], $SugaringandParsing$, 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 [29] developed in Göteborg allowed for sugaring that can largely accommodate morphological features of the target natural language [16], the translation between the functional phrase structure (ASTs) and strings [42].

Interestingly, the functions that were called $ambiguation : MLTT \rightarrow \{Phrase\ Structure\}$ and $interpretation : \{PhraseStructure\} \rightarrow MLTT$ 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 [45], 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 [31]. 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 [22], 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.

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 [18]. 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 [28], 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 [7]. 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.

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 [50] and an additional response by Ranta [47]. 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.

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. [20]

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 [12]. 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 [49]. 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 [56].

Subsequently, in [55], 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 [13] . 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 [4] which allows English text translation into FOL. However, it is not always correct, and dealing with the language of mathematics will present obstacles.

In [11] 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 [51]. With many approaches are not mentioned here, we a hungry reader should evaluate these many sources with respect to this work.

Grammatical Framework

A Brief Introduction to 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 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 NL applications [43].

Consider a language L we want to represent, and we come up with a model that we build as a set of categories and functions over those categories. Let $Cat(L)$, denote the categories. Also suppose we define functions such that, given an ordered list $x_1, ..., x_n; y \in Cat(L)$ we define a set of functions, $Fun_L(x_1, ..., x_n; y)$ defined over the categories. In gf, a function can be denoted something like $\phi : x_1 \to ... \to x_n$. We may compose these based off their arities. So, given a function $\psi \in Fun_L(y_1, , ..., y_n; z)$, functions $\phi_1, ...\phi_n$ such that $\phi_i \in Fun_L(x_{i,1}, ..., x_{i,m}; y_i)$ we can plug these functions in together, or nest them such that

$$\psi \circ (\phi_1, ..., \phi_n) :\to (x_{i,j}) \to (y_i) \to Z$$

This is how abstract syntax trees are formed. It is also worth noting that they obey an associativity property, namely that

$$\begin{aligned} &\theta \circ (\psi_1 \circ (\phi_{1,1}, ..., \phi_{1,k_1}), ..., \psi_n \circ (\phi_{n,1}, ..., \phi_{n,k_n})) \\ =&(\theta \circ \psi_1, ...\psi_n) \circ (\phi_{1,1}, ..., \phi_{1,k_1}, ..., \phi_{n,1}, ..., \phi_{n,k_n}) \end{aligned}$$

This means that trees in GF are invariant as to how they are built - we can build a tree from the leaves to the root or vice versa.

Example : consider the arithmetic grammar of exponentiation, multiplication, and addition defined over a single category of natural number expressions, $\mathbb{N}$.

$$\_\hat{}\_ : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$$

$$\_ * \_ : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$$

$$\_ + \_ : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$$

We can think of constructing the trees by partial application, i.e.,

$(\lambda x.2\hat{}x) : \mathbb{N} - > \mathbb{N}$

Lets try see the constructions yielding the string $(1+2)\hat{}(3*4)$.

We can either (i) construct this as the exponent of two fully formed expressions, namely a sum and a product applied to some numbers, or we can first apply the exponent to the two binary functions, yielding a quaternary function

$$\begin{aligned} &\lambda x, y.x\hat{}y) \\ ((\lambda x, y.x+y)12) & \\ ((\lambda x, y.x*y)34) & \\ &\mapsto(\lambda x, y.x\hat{}y) \\ (1+2) & \\ (3*4)) & \end{aligned}$$

(1 + 2) ^ (3 * 4)

((λx,y. x ^ y) (λx,y. x + y) (λx,y. x * y)) 1 2 3 4

((λx,y. x + y) ^ (λx,y. x * y)) 1 2 3 4 ((λx,y. x + y) ^ (λx,y. x * y)) 1 2 3 4

(1 + 2) ^ (3 * 4)

and then say (λx. 2 ^ x) (1 + 3) * (4 + 5) = (λx. 2 ^ x) (1 + 3) * (4 + 5)

$(\lambda x.2 \wedge x) : \mathbb{N} - > \mathbb{N}$

and then apply it to a complex arguement, say (1 + 3) * (4 + 5) (λx. 2 $^{x}) : Nat - > Nat$

where

λy : Pow y 1 : Nat -> Nat

(times (plus 2 3) (plus 4 5)) (Pow ∘(1,times)) : Nat -> Nat -> Nat

(plus 2 3) (plus 4 5)

can either be

$2^(1 + 3) * (4 + 5)$

The two functions displayed in, Figure 5. If we can loosely call String the set of strings freely generated osome acan be

for now given a single linear presentation C^{AST} , where

AST$_L String_L 0 denote the sets GF AST s and Strings in the languages generated by the rules of L' s abstract syntax ar$

$$Parse : String- > AST$$
$$Linearize : AST- > String$$

with the important property that given a string s,

forall x in (Parse s), Linearize x == s

And given an AST a, we can Parse . Linearize a belongs to AST

Now we should explore why the linearizations are interesting. In part, this is because they have arisen from the role of grammars have played in the intersection and interaction between computer science and linguistics at least since Chomsky in the 50s, and they have different understandings and utilities in the respective disciplines. These two discplines converge in GF, which allows us to talk about natural languages (NLs) from programming languages (PLs) perspective. GF captures languages more expressive than Chomsky's Context Free Grammars (CFGs) but is still decideable with parsing in (cubic?) polynomial time, which means it still is quite domain specific and leaves a lot to be desired as far as turing complete languages or those capable of general recursion are concerned.

Theoretical Overview

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 which 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 ;
  EnatrecLam : Exp -> Exp -> Exp ->  Exp ;

  X : Var ;
  Y : Var ;
  F : Var ;
  IntV : Int -> Var ;
```

So far we specified how to form expressions, namely types built out of 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, there are parallel judgments when defining a concrete syntax in GF, `lincat` and `lin` corresponding to `cat` and `fun`, respectively. Whereas 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. 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 intolerable for a variety of reasons, but it’s an approximation of what a computer scientist might say. If instead, we want to linearize this same expression to a pidgin programming language string modeled off Haskell, “
(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 lisp 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 with finite cardinality of unary constructors. 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

Let's look at the

We can now discuss

One can now form expressions like the following :

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

Consider a language L, for now given a single linear presentation C^L , where AST$_L String_L 0denotethesetsGFASTsandStringsinthelanguagesgeneratedbytherulesofL'sabstractsyntaxar$

GF has basic judgments

- The Resource Grammar Library (RGL)
- A way of interfacing GF with Haskell and transforming ASTs externally
- The Portable Grammar Format (PGF)
- Module System and directory structure

We can should note that computa

given a string s , perhaps a phrase map Linearize (Parse s)

is understood as the set of translations of a phrase of one language to possibly grammatical phrases in the other languages connected by a mutual abstract syntax. So we could denote these L$^English, L^Bantu, L^SwedishforL^Englsih_AmericanvsL^Englsih_EnglishorL$

One could also further elaborate these L$^English_0 L^English_1 tovaryingdegreesofgranularity, likeL^Englis$

L$_0 < L_1 -> L_English_0 < L_English_1$

But this would be similair to a set of expressions translateable between programming languages, like Logic$^English, Logic^Latex, Logic^Agda, Logic^Coq, etc$

where one could extend the

Logic$_Core^EnglishLogic_Extended^English$

whereas in the PL domain

Logic$_{C}ore^{A}gdaLogic_{E}xtended^{A}gdamaycollapseatcertainpoints, butalsoinsomewaysextendbeyondourLangu$

or Mathematics = Logic + domain specific Mathematics$^{E}nglishMathematics^{A}gda$

where we could have further refinements, via, for instance, the module system, the concrete and linear designs like

Mathematics$^{E}nglishiI--Somethingabout$

The Functor (in the module sense familiar to ML programmers)

break down to different classifications of

– Something about

The Functor (in the module sense familiar to ML programmers)

break down to different classifications of

The indexes here, while seemingly arbitrary,

One could also further elaborate these L$^{E}nglish_{0}L^{E}nglish_{1}tovaryingdegreesofgranularity, likeL^{E}nglis$

because Chomsky may something like ”I was stoked” and Partee may only say something analogous ”I was really excited” or whatever the individual nuances come with how speakers say what they mean with different surface syntax, and also

Given a set of categories, we define the functions □: (c$_{1},...,c_{n})->c_{t}overthecategorieswhichwillservec$

Some preliminary observations and considerations

There are many ways to skin a cat. While in some sense GF offers the user a limited palette with which to paint, she nonetheless has quite a bit of flexibility in her decision making when designing a GF grammar for a specific domain or application. These decisions are not binary, but rest in the spectrum of of considerations varying between :

* immediate usability and long term sustainability * prototyping and production readiness * dependency on external resources liable to change * research or application oriented * sensitivity and vulnerability to errors * scalability and maintainability

Many answers to where on the spectrum a Grammar lies will only become clear a posteriori to code being written, which often contradicts prior decisions which had been made and requires significant efforts to refactor. General best practices that apply to all programming languages, like effective use of modularity, can and should be applied by the GF programmer out of the box, whereas the strong type system also promotes a degree of rigidity with which the programmer is forced to stay in a certain safety boundary. Nonetheless, a grammar is in some sense a really large program, which brings a whole series of difficulties.

When designing a GF grammar for a given application, the most immediate question that will come to mind is separation of concerns as regards the spectrum of

[Abstract <-> Concrete] syntax

Have your cake and eat it ?

A grammar for basic arithmetic

Abstract Judgments The core syntax of GF is quite simple. The abstract syntax specification, denoted mathematically above as $_{,}andinGFas`Arith.gf`isgivenby$:

```
abstract Arith = { ... }
```

Please note all GF files end with the ‘.gf‘ file extension. More detailed information about abstract, concrete, modules, etc. relevant for GF is specified internal to a ‘*.gf‘ file

The abstract specification is simple, and reveals GF’s power and elegance. The two primary abstract judgments are :

1. ‘cat‘ : denoting a syntactic category
2. . ‘fun‘ : denoting a n-ary function over categories. This is essentially a labeled context-free rewrite rule with (non-)terminal string information suppressed

While there are more advanced abstract judgments, for instance ‘def‘ allows one to incorporate semantic information, discussion of these will be deferred to other resources. These core judgments have different interpretations in the natural and formal language settings. Let’s see the spine of the ‘Arith‘ language, where we merely want to be able to write expressions like ‘(3 + 4) * 5‘ in a myriad of concrete syntaxes.

```
cat Exp ;
```

```
fun Add : Exp -> Exp -> Exp ; Mul : Exp -> Exp -> Exp ; EInt : Int -> Exp ;
```

To represent this abstractly, we merely have two binary operators, labeled ‘Add‘ and ‘Mul‘, whose intended interpretation is just the operator names, and the ‘EInt‘ function which coerces a predefined ‘Int‘, natively supported numerical strings ‘”0”,”1”,”2”,...‘ into arithmetic expressions. We can now generate our first abstract syntax tree, corresponding to the above expression, ‘Mul (Add (EInt 3) (EInt 4)) (EInt 5)‘, more readable perhaps with the tree structure expanded :

```
Mul Add EInt 3 EInt 4 EInt 5
```

The trees nodes are just the function names, and the leaves, while denoted above as numbers, are actually function names for the built-in numeric strings which happen to be linearized to the same piece of syntax, i.e. ‘linearize 3 == 3‘, where the left-hand 3 has type ‘Int‘ and the right-hand 3 has type ‘Str‘. GF has support for very few, but important categories. These are ‘Int‘, ‘Float‘, and ‘String‘. It is my contention and that adding user defined builtin categories would greatly ease the burden of work for people interested in using GF to model programming languages, because ‘String‘ makes the grammars notoriously ambiguous.

In computer science terms, to judge ‘Foo‘ to be a given category ‘cat Foo;‘ corresponds to the definition of a given Algebraic Datatypes (ADTs) in Haskell, or inductive definitions in Agda, whereas the function judgments ‘fun‘ correspond to the various constructors. These connections become explicit in the PGF embedding of GF into Haskell, but examining the Haskell code below makes one suspect their is some equivalence lurking in the corner:

```
data Exp = Add Exp Exp | Mul Exp Exp | EInt Int
```

In linguistics we can interpret the judgments via alternatively simple and standard examples:

1. ‘cat‘ : these could be syntactic categories like Common Nouns ‘CN‘, Noun Phrases ‘NP‘ , and determiners ‘Det‘
2. ‘fun‘ : give us ways of combining words or phrases into more complex syntactic units

For instance, if

```
fun Car_CN : CN ; The_Det : Det ; DetCN : Det -> CN -> NP ;
```

Then one can form a tree ‘DetCN The$_{D}etCar_{C}N`whichshouldlinearizeto`"thecar"`inEnglish,`bilen`inS$

While there was an equivalence suggested Haskell ADTs should be careful not to treat these as the same as the GF judgments. Indeed, the linguistic interpretation breaks this analogy, because linguistic categories aren’t stable mathematical objects in the sense that they evolved and changed during the evolution of language, and will continue to do so. Since GF is primarily concerned with parsing and linearization of languages, the full power of inductive definitions in Agda, for instance, doesn’t seem like a particularly natural space to study and model natural language phenomena.

Arith.gf Below we recapitulate, for completeness, the whole 'Arith.gf' file with all the pieces from above glued together, which, the reader should start to play with.

```
 abstract Arith = {

flags startcat = Exp ;

-- a judgement which says "Exp is a category" cat Exp ;

fun Add : Exp -> Exp -> Exp ; -- "+" Mul : Exp -> Exp -> Exp ; --
"*" EInt : Int
-> Exp ; -- "33"

}
```

The astute reader will recognize some code which has not yet been described. The comments, delegated with '–', can have their own lines or be given at the end of a piece of code. It is good practice to give example linearizations as comments in the abstract syntax file, so that it can be read in a stand-alone way.

The 'flags startcat = Exp ;' line is not a judgment, but piece of metadata for the compiler so that it knows, when generating random ASTs, to include a function at the root of the AST with codomain 'Exp'. If I hadn't included 'flags startcat = *some cat*' , and ran 'gr' in the gf shell, we would get the following error, which can be incredibly confusing but simple bug to fix if you know what to look for!

```
 Category S is not in scope CallStack (from HasCallStack):
error, called at src/compiler/GF/Command/Commands.hs:881:38 in
gf-3.10.4-BNI84g7Cbh1LvYlghrRUOG:GF.Command.Commands
```

Concrete Judgments We now append our abstract syntax GF file 'Arith.gf' with our first concrete GF syntax, some pigdin English way of saying our same expression above, namely 'the product of the sum of 3 and 4 and 5'. Note that 'Mul' and 'Add' both being binary operators preclude this reading : 'product of (the sum of 3 and 4 and 5)' in GF, despite the fact that it seems the more natural English interpretation and it doesn't admit a proper semantic reading.

Reflecting the tree around the 'Mul' root, 'Mul (EInt 5) (Add (EInt 3) (EInt 4))', we get a reading where the 'natural interpretation' matches the actual syntax :'the product of 5 and the sum of 3 and 4'. Let's look at the concrete syntax which allow us to simply specify the linearization rules corresponding to the above 'fun' function judgments.

Our concrete syntax header says that 'ArithEng1' is constrained by the fact that the concrete syntaxes must share the same prefix with the abstract syntax, and extend it with one or more characters, i.e. ‘Arith+.gf‘.

```
concrete ArithEng1 of Arith = { ... }
```

We now introduce the two concrete syntax judgments which compliment those above, namely :

* ‘cat‘ is dual to ‘lincat‘ * ‘fun‘ is dual to ‘lin‘

Here is the first pass at an English linearization :

```
lincat Exp = Str ;
```

```
lin Add e1 e2 = "the sum of" ++ e1 ++ "and" ++ e2 ; Mul e1 e2 = "the product of"
++ e1 ++ "and" ++ e2 ; EInt i = i.s ;
```

The ‘lincat‘ judgement says that ‘Exp‘ category is given a linearization type ‘Str‘, which means that any expression is just evaluated to a string. There are more expressive linearization types, records and tables, or products and coproducts in the mathematician’s lingo. For instance, ‘EInt i = i.s‘ that we project the s field from the integer i (records are indexed by numbers but rather by names in PLs). We defer a more extended discussion of linearization types for later examples where they are not just useful but necessary, producing grammars more expressive than CFGs called Parallel Multiple Context Free Grammars (PMCFGs).

The linearization of the ‘Add‘ function takes two arguments, ‘e1‘ and ‘e2‘ which must necessarily evaluate to strings, and produces a string. Strings in GF are denoted with double quotes ‘”my string”‘ and concatenation with ‘++‘. This resulting string, ‘”the sum of” ++ e1 ++ ”and” ++ e2‘ is the concatenation of ‘”the sum of”‘, the evaluated string ‘e1‘, ‘”and”‘, and the string of a linearized ‘e2‘. The linearization of ‘EInt‘ is almost an identity function, except that the primitive Integer’s are strings embedded in a record for scalability purposes.

Here is the relationship between ‘fun‘ and ‘lin‘ from a slightly higher vantage point. Given a ‘fun‘ judgement

$f{:}C_0 \rightarrow C_1 \rightarrow ... \rightarrow C_n$

in the ‘abstract‘ file, the GF user provides a corresponding ‘lin‘ judgement of the form

$f\ c_0\ c_1\ ...\ c_n = t_0$ ++ t_1 ++ ... ++ t_m

in the ‘concrete‘ file. Each c_i must have the linearization type given in the ‘lincat‘ of C_i , e.g. if ‘lincat C$_i = T;‘ then ‘c_i : T‘$.

We step through the example above to see how the linearization recursively evaluates, noting that this may not be the actual reduction order GF internally performs. The relation '->*' informally used but not defined here expresses the step function after zero or more steps of evaluating an expression. This is the reflexive transitive closure of the single step relation '->' familiar in operational semantics.

```
 linearize (Mul (Add (EInt 3) (EInt 4)) (EInt 5)) ->* "the
product of" ++ linearize (Add (EInt 3) (EInt 4)) ++ "and" ++ linearize (EInt 5)
->* "the product of" ++ ("the sum of" ++ (EInt 3) ++ (EInt 4)) ++ "and" ++ ({ s
= "5"} . s) ->* "the product of" ++ ("the sum of" ++ ({ s = "3"} . s) ++ ({ s =
"4"} . s)) ++ "and" ++ "5" ->* "the product of" ++ ("the sum of" ++ "3" ++ "and"
++ "4") ++ "and" ++ "5" ->* "the product of" ++ ("the sum of" ++ "3" ++ "and" ++
"4") ++ "and" ++ "5" ->* "the product of the sum of 3 and 4 and 5"
```

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. We touch upon this in a [previous section](some-preliminary-observations-and-considerations).

Now that the basics of GF have been described, we will augment our grammar so that it becomes slightly more interesting, introduce a second 'concrete' syntax, and show how to run these in the GF shell in order to translate between our two languages.

The GF shell

So now that we have a GF 'abstract' and 'concrete' syntax pair, one needs to test the grammars.

Once GF is [installed](https://www.grammaticalframework.org/download/index-3.10.html), one can open both the 'abstract' and 'concrete' with the 'gf' shell command applied to the 'concrete' syntax, assuming the 'abstract' syntax is in the same directory :

```
 $ gf ArithEng1.gf
```

I'll forego describing many important details and commands, please refer to the [official shell reference](https://www.grammaticalframework.org/doc/gf-shell-reference.html) and Inari Listenmaa's post on [tips and gotchas](https://inariksit.github.io/gf/2018/08/28/gf-gotchas.html) for a more nuanced take than I give here.

The 'ls' of the gf repl is 'gr'. What does 'gr' do? Lets try it, as well as ask gf what it does:

```
Arith> gr Add (EInt 999) (Mul (Add (EInt 999) (EInt 999)) (EInt
999))

0 msec Arith> help gr gr, generate_random generate random trees in the current
abstract syntax
```

We see that the tree generated isn't random - '999' is used exclusively, and obviously if the trees were actually random, the probability of such a small tree might be exceedingly low. The depth of the trees is cut-off, which can be modified with the '-depth=n' flag for some number n, and the predefined categories, 'Int' in this case, are annoyingly restricted. Nonetheless, 'gr' is a quick first pass at testing your grammar.

Listed in the repl, but not shown here, are (some of the) additional flags that 'gr' command can take. The 'help' command reveals other possible commands, included is the linearization, the 'l' command. We use the pipe '|' to compose gf functions, and the '-tr' to trace the output of 'gr' prior to being piped :

```
Arith> gr -tr | l Add (Mul (Mul (EInt 999) (EInt 999)) (Add
(EInt 999) (EInt 999))) (Add (Add (EInt 999) (EInt 999)) (Add (EInt 999) (EInt
999)))

the sum of the product of the product of 999 and 999 and the sum of 999 and 999
and the sum of the sum of 999 and 999 and the sum of 999 and 999
```

Clearly this expression is to too complex to say out loud and retain a semblance of meaning. Indeed, most grammatical sentences aren't meaningful. Dealing with semantics in GF is advanced, and we won't touch that here. Nonetheless, our grammar seems to be up and running.

Let's try the sanity check referenced at the beginning of this post, namely, observe that $linearize \circ parse$ preserves an AST, and vice versa, $parse \circ linearize$ preserves a string. Parse is denoted 'p'.

```
Arith> gr -tr | l -tr | p -tr | l Add (EInt 999) (Mul (Add
(EInt 999) (EInt 999)) (Add (EInt 999) (EInt 999)))

the sum of 999 and the product of the sum of 999 and 999 and the sum of 999 and
999

Add (EInt 999) (Mul (Add (EInt 999) (EInt 999)) (Add (EInt 999) (EInt 999)))

the sum of 999 and the product of the sum of 999 and 999 and the sum of 999 and
999
```

Phew, that's a relief. Note that this is an unambiguous grammar, so when I said

’preserves’, I only meant it up to some notion of relatedness. This relation is indeed equality for unambiguous grammars. Unambiguous grammars are degenerate cases, however, so I expect this is the last time you’ll see such tame behavior when the GF parser is involved. Now that all the main ingredients have been introduced, the reader

Exercises

Exercise 1.1 : Extend the ‘Arith‘ grammar with variables. Specifically, modify both ‘Arith.gf‘ and ‘ArithEng1.gf‘ arithmetic with two additional unique variables, ‘x‘ and ‘y‘, such that the string ‘product of x and y‘ parses uniquely : .notice–danger

Exercise 1.2 : Extend *Exercise 1.1* with unary predicates, so that ‘3 is prime‘ and ‘x is odd‘ parse. Then include binary predicates, so that ‘3 equals 3‘ parses. : .notice–danger

Exercise 2 : Write concrete syntax in your favorite language, ‘ArithFaveLang.gf‘ : .notice–danger

Exercise 3 : Write second English concrete syntax, ‘ArithEng2.gf‘, that mimics how children learn arithmetic, i.e. ”3 plus 4” and ”5 times 5”. Observe the ambiguous parses in the gf shell. Substitute ‘plus‘ with ‘+‘, ‘times‘ with ‘*‘, and remedy the ambiguity with parentheses : .notice–danger

Thought Experiment : Observe that parentheses are ugly and unnecessary: sophisticated folks use fixity conventions. How would one go about remedying the ugly parentheses, at either the abstract or concrete level? Try to do it! : .notice–warning

Solutions

Exercise 1.1

This warm-up exercise is to get use to GF syntax. Add a new category ‘Var‘ for variables, two lexical variable names ‘VarX‘ and ‘VarY‘, and a coercion function (which won’t show up on the linearization) from variables to expressions. We then augment the ‘concrete‘ syntax which is quite trivial.

“‘haskell – Add the following between ‘‘ in ‘Arith.gf‘ cat Var ; fun VExp : Var -> Exp ; VarX : Var ; VarY : Var ; “‘ “‘haskell – Add the following between ‘‘ in ‘ArithEng1.gf‘ lincat Var = Str ; lin VExp v = v ; VarX = ”x” ; VarY = ”y” ; “‘

Exercise 1.2

This is a similar augmentation as was performed above.

“‘haskell – Add the following between ‘‘ in ‘Arith.gf‘ flags startcat = Prop ;

cat Prop ;

fun Odd : Exp -> Prop ; Prime : Exp -> Prop ; Equal : Exp -> Exp -> Prop ; "' "'haskell – Add the following between '' in 'ArithEng1.gf' lincat Prop = Str ; lin Odd e = e ++ "is odd"; Prime e = e ++ "is prime"; Equal e1 e2 = e1 ++ "equals" ++ e2 ; "' The main point is to recognize that we also need to modify the 'startcat' flag to 'Prop' so that 'gr' generates numeric predicates rather than just expressions. One may also use the category flag to generate trees of any expression 'gr -cat=Exp'.

Exercise 2

Exercise 3

We simply change the strings that get linearized to what follows :

"'haskell lin Add e1 e2 = e1 ++ "plus" ++ e2 ; Mul e1 e2 = e1 ++ "times" ++ e2 ; "'

With these minor modifications in place, we make the follow observation in the GF shell, noting that for a given number of binary operators in an expression, we get the [Catalan number](https://en.wikipedia.org/wiki/Catalan$_{n}umber)ofparses$!

"'haskell Arith> gr -tr | l -tr | p -tr | l Add (Mul (VExp VarX) (Mul (EInt 999) (VExp VarX))) (EInt 999)

x times 999 times x plus 999

Add (Mul (VExp VarX) (Mul (EInt 999) (VExp VarX))) (EInt 999) Add (Mul (Mul (VExp VarX) (EInt 999)) (VExp VarX)) (EInt 999) Mul (VExp VarX) (Add (Mul (EInt 999) (VExp VarX)) (EInt 999)) Mul (VExp VarX) (Mul (EInt 999) (Add (VExp VarX) (EInt 999))) Mul (Mul (VExp VarX) (EInt 999)) (Add (VExp VarX) (EInt 999))

x times 999 times x plus 999 x times 999 times x plus 999 x times 999 times x plus 999 x times 999 times x plus 999 x times 999 times x plus 999 "'

"'haskell Arith> gr -tr | l -tr | p -tr Add (EInt 999) (Mul (VExp VarY) (Mul (EInt 999) (EInt 999)))

(999 + (y * (999 * 999)))

Add (EInt 999) (Mul (VExp VarY) (Mul (EInt 999) (EInt 999))) "'

...Blog post...

Natural Number Proofs

Here we open with the perhaps the most natural kind of proof one would expect, that of laws over the inductively defined natural numbers.

A Spectrum of GF Grammars for types

Prior GF Formalizations

Prior to the grammars explored thin this thesis, Ranta produced two main results [44] [46]. These are incredibly important precedents in this approach to proof translation, and serve as important comparative work for which this work responds. In [44], Ranta designed a grammar which allowed for predicate logic with domain specific lexicon supporting mathematical theories on top of the logic like geometry or arithmetic. The the syntax was both meant to be relatively complete, so that typical logical utterances of interest could be accommodated, as well as relatively non-trivial linguistic nuance including lists of terms, predicates, and propositions, in-situ and bounded quantification, and multiple forms of constructing more syntactically nuanced predicates. The syntactic nuance captured in this work was by means of an extended grammar, via a Portable Grammar Format (PGF), on top of the minimal, core logical formalism.

One could translate from the core and extended via a denotational semantics approach. The tree representing the *syntactically complete* phrase “for all natural numbers x, x is even or x is odd” would be evaluated to a tree which linearizes to the *semantically adequate* phrase “every natural number is even or odd”. In the opposite direction, the desugaring of a logically informal statement into something linguistically idiomatic is also accomplished. 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. This means that we are being slightly abusive to our terms, as the formal/informal translation is taking place is at the PGF level. The GF translation between concrete syntaxes supports multiple NLs, but the syntactic completeness has no mechanism of verification via Agda’s type checker. 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.

In 2014, Ranta gave an unpublished talk at the Stockholm Mathematics seminar [46]. 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, and allows the translation of the latex 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 expres-

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 12: Rendered Latex

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

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

Figure 13: Contractibility

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 14: Contractibility

sive, 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 will become tricky. Nonetheless, this means that it's 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. For example, in Figure 12 is some rendered latex taken verbatim from Ranta's test code.

With some of hours of tinkering on the pidgin logic concrete syntax and some reverse engineering with help from the GF shell, one is able to get these definitions in Figure 2, which are intended to share the same syntactic space as cubicalTT. We note the first definition of "contractability" actually runs in cubicalTT up to renaming a lexical items, and it is clear that the translation from that to Agda should be a benign task. However, the *equivalence* syntax is stuck with the artifact from the bloated abstract syntax for the of the anaphoric use of "it", which may presumably be fixed with a few hours more of tinkering, but becomes even more complicated when not just defining new types, but actually writing real mathematical proofs, or relatively large terms. 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.

Additionally, we give the Agda code in Figure 14, so-as to see what the end result of such a program would be. The astute reader will also notice a semantic in the pidgin rendering error relative to the Agda implementation. `fiber` has the type `it : Set` instead of something like `(y : B) : Set`, and the y variable is unbound in the `fiber` expression. 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 emphasis on this axis is most assuredly a choice one should consider up front.

While both these grammars have their strengths and weaknesses, one shall see shortly that the approach in this thesis, 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, but its linearizations become must increase in complexity.

HoTT Proofs

Why HoTT for natural language?

We note that all natural language definitions, theorems, and proofs are copied here verbatim from the HoTT book. This decision is admittedly arbitrary, but does have some benefits. We list some here :

- As the HoTT book was a collaborative effort, it mixes the language of many individuals and editors, and can be seen as more "linguistically neutral"
- By its very nature HoTT is interdiscplinary, conceived and constructed by mathematicians, logicians, and computer scientists. It therefore is meant to interface with all these discplines, and much of the book was indeed formalized before it was written
- It has become canonical reference in the field, and therefore benefits from wide familiarity
- It is open source, with publically available Latex files free for modification and distribution

The genisis of higher type theory is a somewhat elementary observation : that the identity type, parameterized by an arbitrary type A and indexed by elements of A, can actually be built iteratively from previous identities. That is, A may actually already be an identity defined over another type A', namely $A :\equiv x =_{A'} y$ where $x, y : A'$. The key idea is that this iterating identities admits a homotpical interpretation :

- Types are topological spaces
- Terms are points in these space
- Equality types $x =_A y$ are paths in A with endpoints x and y in A
- Iterated equality types are paths between paths, or continuous path deformations in some higher path space. This is, intuitively, what mathematicians call a homotopy.

To be explicit, given a type A, we can form the homotopy $p =_{x =_A y} q$ with endpoints p and q inhabiting the path space $x =_A y$.

Let's start out by examining the inductive definition of the identity type. We present this definition as it appears in section 1.12 of the HoTT book.

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

$$\mathsf{refl} : \prod_{a:A} ([A]aa)$$

called ***reflexivity****, which says that every element of A is equal to itself (in a speci-*

fied way). We regard refl_a *as being the constant path at the point* a*.*

We recapitulate this definition in Agda, and treat :

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

An introduction to equality

There is already some tension brewing : most mathematicains have an intuition for equality, that of an identitfication between two pieces of information which intuitively must be the same thing, i.e. $2 + 2 = 4$. They might ask, what does it mean to "construct an element of ab"? For the mathematician use to thinking in terms of sets $\{ab \mid a, b \in \mathbb{N}\}$ 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 for a later section, 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.

All about Identity

We start with a slight reformulation of the identity type, where the element determining the equality is treated as a parameter rather than an index. This is a matter of convenience more than taste, as it delegates work for Agda's typechecker that the programmer may find a distraction. The reflexivity terms can generally have their endpoints inferred, and therefore cuts down on the beauracry which often obscures code.

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

infix 20 _≡_
```

It is of particular concern in this thesis, because it hightlights a fundamental difference between the lingusitic and the formal approach to proof presentation. While the mathematician can whimsically choose to include the reflexivity arguement or ignore it if she believes it can be inferred, the programmer can't afford such a laxidasical attitude. Once the type has been defined, the arguement strcuture is fixed, all future references to the definition carefully adhere to its specification. The advantage that the programmer does gain however, that of Agda's powerful inferential abilities, allows for the insides to be seen via interaction windown.

Perhaps not of much interest up front, this is incredibly important detail which the mathematician never has to deal with explicity, but can easily make type and term translation infeasible due to the fast and loose nature of the mathematician's writing. Conversely, it may make Natural Language Generation (NLG) incredibly clunky, adhering to strict rules when created sentences out of programs.

[ToDo, give a GF example]

A prime source of beauty in constructive mathematics arises from Gentzen's recognition of a natural duality in the rules for introducing and using logical connectives.

The mutually coherence between introduction and elmination rules form the basis of what has since been labeled harmony in a deductive system. This harmony isn't just an artifact of beauty, it forms the basis for cuts in proof normalization, and correspondingly, evaluation of terms in a programming langauge.

The idea is simple, each new connective, or type former, needs a means of constructing its terms from its constiutuent parts, yielding introduction rules. This however, isn't enough - we need a way of dissecting and using the terms we construct. This yields an elimantion rule which can be uniquely derived from an inductively defined type. These elimination forms yield induction principles, or a general notion of proof by induction, when given an interpration in mathematics. In the non-depedent case, this is known as a recursion principle, and corresponds to recursion known by programmers far and wide. The proof by induction over natural numbers familiar to mathematicians is just one special case of this induction principle at work–the power of induction has been recognized and brought to the fore by computer scientists.

We now elaborate the most important induction principle in HoTT, namely, the induction of an identity type.

Definition 9 (Version 1) *Moreover, one of the amazing things about homotopy type theory is that all of the basic constructions and axioms—all of the higher groupoid structure—arises automatically from the induction principle for identity types. Recall from [section 1.12] that this says that if*

- *for every* $x, y : A$ *and every* $p : [A]xy$ *we have a type* $D(x, y, p)$*, and*
- *for every* $a : A$ *we have an element* $d(a) : D(a, a, \mathsf{refl}_a)$*,*

then

- *there exists an element* $A(D, d, x, y, p) : D(x, y, p)$ *for* every *two elements* $x, y : A$ *and* $p : [A]xy$*, such that* $A(D, d, a, a, \mathsf{refl}_a) \equiv d(a)$*.*

The book then reiterates this definition, with basically no natural language, essentially in the raw logical framework devoid of anything but dependent function types.

Definition 10 (Version 2) *In other words, given dependent functions*

$$D : \prod_{(x,y:A)} (x = y) \ \to \ \mathcal{U}$$
$$d : \prod_{a:A} D(a, a, \mathsf{refl}_a)$$

there is a dependent function

$$A(D, d) : \prod_{(x,y:A)} \prod_{(p:xy)} D(x, y, p)$$

such that

$$A(D, d, a, a, \mathsf{refl}_a) \equiv d(a) \qquad (1)$$

for every $a : A$. Usually, every time we apply this induction rule we will either not care about the specific function being defined, or we will immediately give it a different name.

Again, we define this, in Agda, staying as true to the syntax as possible.

```
J : {A : Set}
  → (D : (x y : A) → (x ≡ y) → Set)
  → ((a : A) → (D a a r )) -- → (d : (a : A) → (D a a r ))
  → (x y : A)
  → (p : x ≡ y)
  ------------------------------------
  → D x y p
J D d x .x r = d x
```

It should be noted that, for instance, we can choose to leave out the d label on the third line. Indeed minimizing the amount of dependent typing and using vanilla function types when dependency is not necessary, is generally considered "best practice" Agda, because it will get desugared by the time it typechecks anyways. For the writer of the text; however, it was convenient to define d once, as there are not the same constraints on a mathematician writing in latex. It will again, serve as a nontrivial exercise to deal with when specifying the grammar, and will be dealt with later [ToDo add section]. It is also of note that we choose to include Martin-Löf's original name J, as this is more common in the computer science literature.

Once the identity type has been defined, it is natural to develop an "equality calculus", so that we can actually use it in proof's, as well as develop the higher groupoid structure of types. The first fact, that propositional equality is an equivalence relation, is well motivated by needs of practical theorem proving in Agda and the more homotopically minded mathematician. First, we show the symmetry of equality–that paths are reversible.

Lemma 2 *For every type A and every $x, y : A$ there is a function*

$$(x = y) \to (y = x)$$

*denoted $p \mapsto p$, such that $\mathsf{refl}_x \equiv \mathsf{refl}_x$ for each $x : A$. We call p the **inverse** of p.*

Proof 1 (First proof) *Assume given $A : \mathcal{U}$, and let $D : \prod_{(x,y:A)}(x = y) \to \mathcal{U}$ be the type family defined by $D(x, y, p) :\equiv (y = x)$. In other words, D is a function assigning* *to any $x, y : A$ and $p : x = y$ a type, namely the type $y = x$. Then we have an element*

$$d :\equiv \lambda x.\ \mathsf{refl}_x : \prod_{x:A} D(x, x, \mathsf{refl}_x).$$

Thus, the induction principle for identity types gives us an element $A(D, d, x, y, p) : (y = x)$ for each $p : (x = y)$. We can now define the desired function $()$ to be $\lambda p.\ A(D, d, x, y, p)$, i.e. we set $p :\equiv A(D, d, x, y, p)$. The conversion rule [missing reference] gives $\mathsf{refl}_x \equiv \mathsf{refl}_x$, as required.

The Agda code is certainly more brief:

```
_⁻¹ : {A : Set} {x y : A} → x ≡ y → y ≡ x
_⁻¹ {A} {x} {y} p = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = y ≡ x
    d : (a : A) → D a a r
    d a = r

infixr 50 _⁻¹
```

While first encountering induction principles can be scary, they are actually more mechanical than one may think. This is due to the the fact that they uniquely compliment the introduction rules of an an inductive type, and are simply a means of showing one can "map out", or derive an arbitrary type dependent on the type which has been inductively defined. The mechanical nature is what allows for Coq's induction tactic, and perhaps even more elegantly, Agda's pattern matching capabilities. It is always easier to use pattern matching for the novice Agda programmer, which almost feels like magic. Nonetheless, for completeness sake, the book uses the induction principle for much of Chapter 2. And pattern matching is unique to programming languages, its elegance isn't matched in the mathematicians' lexicon.

Here is the same proof via "natural language pattern matching" and Agda pattern matching:

Proof 2 (Second proof) *We want to construct, for each $x, y : A$ and $p : x = y$, an element $p : y = x$. By induction, it suffices to do this in the case when y is x and p is refl_x. But in this case, the type $x = y$ of p and the type $y = x$ in which we are trying to construct p are both simply $x = x$. Thus, in the "reflexivity case", we can define refl_x to be simply refl_x. The general case then follows by the induction principle, and the conversion rule $\mathsf{refl}_x \equiv \mathsf{refl}_x$ is precisely the proof in the reflexivity case that we gave.*

```
_⁻¹' : {A : Set} {x y : A} → x ≡ y → y ≡ x
```

```
_⁻¹' {A} {x} {y} r = r
```

Next is trasitivity–concatenation of paths–and we omit the natural language presentation, which is a slightly more sophisticated arguement than for symmetry.

```
_•_ : {A : Set} → {x y : A} → (p : x ≡ y) → {z : A} → (q : y ≡ z) → x ≡ z
_•_ {A} {x} {y} p {z} q = J D d x y p z q
    where
    D : (x₁ y₁ : A) → x₁ ≡ y₁ → Set
    D x y p = (z : A) → (q : y ≡ z) → x ≡ z
    d : (z₁ : A) → D z₁ z₁ r
    d = λ v z q → q

infixl 40 _•_
```

Putting on our spectacles, the reflexivity term serves as evidence of a constant path for any given point of any given type. To the category theorist, this makes up the data of an identity map. Likewise, conctanation of paths starts to look like function composition. This, along with the identity laws and associativity as proven below, gives us the data of a category. And we have not only have a category, but the symmetry allows us to prove all paths are isomorphisms, giving us a groupoid. This isn't a coincedence, it's a very deep and fascinating articulation of power of the machinery we've so far built. The fact the path space over a type naturally must satisfies coherence laws in an even higher path space gives leads to this notion of types as higher groupoids.

As regards the natural language–at this point, the bookkeeping starts to get hairy. Paths between paths, and paths between paths between paths, these ideas start to lose geometric inutiotion. And the mathematician often fails to express, when writing $p = q$, that she is already reasoning in a path space. While clever, our brains aren't wired to do too much book-keeping. Fortunately Agda does this for us, and we can use implicit arguements to avoid our code getting too messy. [ToDo, add example]

We now proceed to show that we have a groupoid, where the objects are points, the morphisms are paths. The isomorphisms arise from the path reversal. Many of the proofs beyond this point are either routinely made via the induction principle, or even more routinely by just pattern matching on equality paths, we omit the details which can be found in the HoTT book, but it is expected that the GF parser will soon cover such examples.

```
iₗ : {A : Set} {x y : A} (p : x ≡ y) → p ≡ r • p
iₗ {A} {x} {y} p = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = p ≡ r • p
    d : (a : A) → D a a r
    d a = r
```

```
iᵣ : {A : Set} {x y : A} (p : x ≡ y) → p ≡ p • r
iᵣ {A} {x} {y} p = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = p ≡ p • r
    d : (a : A) → D a a r
    d a = r

leftInverse : {A : Set} {x y : A} (p : x ≡ y) → p ⁻¹ • p ≡ r
leftInverse {A} {x} {y} p = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = p ⁻¹ • p ≡ r
    d : (x : A) → D x x r
    d x = r

rightInverse : {A : Set} {x y : A} (p : x ≡ y) → p • p ⁻¹ ≡ r
rightInverse {A} {x} {y} p = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = p • p ⁻¹ ≡ r
    d : (a : A) → D a a r
    d a = r

doubleInv : {A : Set} {x y : A} (p : x ≡ y) → p ⁻¹ ⁻¹ ≡ p
doubleInv {A} {x} {y} p = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = p ⁻¹ ⁻¹ ≡ p
    d : (a : A) → D a a r
    d a = r

associativity :{A : Set} {x y z w : A} (p : x ≡ y) (q : y ≡ z) (r' : z ≡ w ) → p • (q • r') ≡ p •
associativity {A} {x} {y} {z} {w} p q r' = J D₁ d₁ x y p z w q r'
  where
    D₁ : (x y : A) → x ≡ y → Set
    D₁ x y p = (z w : A) (q : y ≡ z) (r' : z ≡ w ) → p • (q • r') ≡ p • q • r'
    -- d₁ : (x : A) → D₁ x x r
    -- d₁ x z w q r' = r -- why can it infer this
    D₂ : (x z : A) → x ≡ z → Set
    D₂ x z q = (w : A) (r' : z ≡ w ) → r • (q • r') ≡ r • q • r'
    D₃ : (x w : A) → x ≡ w → Set
    D₃ x w r' = r • (r • r') ≡ r • r • r'
    d₃ : (x : A) → D₃ x x r
    d₃ x = r
    d₂ : (x : A) → D₂ x x r
    d₂ x w r' = J D₃ d₃ x w r'
    d₁ : (x : A) → D₁ x x r
    d₁ x z w q r' = J D₂ d₂ x z q w r'
```

When one starts to look at structure above the groupoid level, i.e., the paths between paths between paths level, some interesting and nonintuitive results emerge. If one defines a path space that is seemingly trivial, namely, taking the same starting and end points, the higherdimensional structure yields non-trivial structure. We now arrive at the first "interesting" result in this book, the Eckmann-Hilton Arguement. It says that composition on the loop space of a loop space, the second loop space, is commutitive.

Definition 11 *Thus, given a type A with a point $a : A$, we define its **loop space** $\Omega(A, a)$ to be the type $[A]aa$. We may sometimes write simply ΩA if the point a is understood from context.*

Definition 12 *It can also be useful to consider the loop space* of *the loop space of A, which is the space of 2-dimensional loops on the identity loop at a. This is written $\Omega^2(A, a)$ and represented in type theory by the type $[([A]aa)]\mathsf{refl}_a\mathsf{refl}_a$.*

Theorem 1 (Eckmann–Hilton) *The composition operation on the second loop space*

$$\Omega^2(A) \times \Omega^2(A) \to \Omega^2(A)$$

is commutative: $\alpha \cdot \beta = \beta \cdot \alpha$, for any $\alpha, \beta : \Omega^2(A)$.

Proof 3 *First, observe that the composition of 1-loops $\Omega A \times \Omega A \to \Omega A$ induces an operation*

$$\star : \Omega^2(A) \times \Omega^2(A) \to \Omega^2(A)$$

as follows: consider elements $a, b, c : A$ and 1- and 2-paths,

$$\begin{aligned} p &: a = b, & r &: b = c \\ q &: a = b, & s &: b = c \\ \alpha &: p = q, & \beta &: r = s \end{aligned}$$

as depicted in the following diagram (with paths drawn as arrows).

[TODO Finish Eckmann Hilton Arguement]

[Todo, clean up code so that its more tightly correspondent to the book proof] The corresponding agda code is below :

```
-- whiskering
_•ᵣ_ : {A : Set} → {b c : A} {a : A} {p q : a ≡ b} (α : p ≡ q) (r′ : b ≡ c) → p • r′ ≡ q • r′
_•ᵣ_ {A} {b} {c} {a} {p} {q} α r′ = J D d b c r′ a α
  where
    D : (b c : A) → b ≡ c → Set
    D b c r′ = (a : A) {p q : a ≡ b} (α : p ≡ q) → p • r′ ≡ q • r′
```

```
    d : (a : A) → D a a r
    d a a' {p} {q} α = iᵣ p ⁻¹ • α • iᵣ q

-- iᵣ == ruₚ

_•ₗ_ : {A : Set} → {a b : A} (q : a ≡ b) {c : A} {r' s : b ≡ c} (β : r' ≡ s) → q • r' ≡ q • s
_•ₗ_ {A} {a} {b} q {c} {r'} {s} β = J D d a b q c β
  where
    D : (a b : A) → a ≡ b → Set
    D a b q = (c : A) {r' s : b ≡ c} (β : r' ≡ s) → q • r' ≡ q • s
    d : (a : A) → D a a r
    d a a' {r'} {s} β = iₗ r' ⁻¹ • β • iₗ s

_⋆_ : {A : Set} → {a b c : A} {p q : a ≡ b} {r' s : b ≡ c} (α : p ≡ q) (β : r' ≡ s) → p • r' ≡
_⋆_ {A} {q = q} {r' = r'} α β = (α •ᵣ r') • (q •ₗ β)

_⋆'_ : {A : Set} → {a b c : A} {p q : a ≡ b} {r' s : b ≡ c} (α : p ≡ q) (β : r' ≡ s) → p • r' ≡
_⋆'_ {A} {p = p} {s = s} α β = (p •ₗ β) • (α •ᵣ s)

Ω : {A : Set} (a : A) → Set
Ω {A} a = a ≡ a

Ω² : {A : Set} (a : A) → Set
Ω² {A} a = _≡_ {a ≡ a} r r

lem1 : {A : Set} → (a : A) → (α β : Ω² {A} a) → (α ⋆ β) ≡ (iᵣ r ⁻¹ • α • iᵣ r) • (iₗ r ⁻¹ • β • iₗ r)
lem1 a α β = r

lem1' : {A : Set} → (a : A) → (α β : Ω² {A} a) → (α ⋆' β) ≡ (iₗ r ⁻¹ • β • iₗ r) • (iᵣ r ⁻¹ • α • iᵣ r
lem1' a α β = r

-- ap\_
apf : {A B : Set} → {x y : A} → (f : A → B) → (x ≡ y) → f x ≡ f y
apf {A} {B} {x} {y} f p = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = {f : A → B} → f x ≡ f y
    d : (x : A) → D x x r
    d = λ x → r

ap : {A B : Set} → {x y : A} → (f : A → B) → (x ≡ y) → f x ≡ f y
ap f r = r

lem20 : {A : Set} → {a : A} → (α : Ω² {A} a) → (iᵣ r ⁻¹ • α • iᵣ r) ≡ α
lem20 α = iᵣ (α) ⁻¹

lem21 : {A : Set} → {a : A} → (β : Ω² {A} a) → (iₗ r ⁻¹ • β • iₗ r) ≡ β
lem21 β = iᵣ (β) ⁻¹

lem2 : {A : Set} → (a : A) → (α β : Ω² {A} a) → (iᵣ r ⁻¹ • α • iᵣ r) • (iₗ r ⁻¹ • β • iₗ r) ≡ (α • β)
lem2 {A} a α β = apf (λ - → - • (iₗ r ⁻¹ • β • iₗ r) ) (lem20 α) • apf (λ - → α • -) (lem21 β)

lem2' : {A : Set} → (a : A) → (α β : Ω² {A} a) → (iₗ r ⁻¹ • β • iₗ r) • (iᵣ r ⁻¹ • α • iᵣ r) ≡ (β • α
```

```
lem2' {A} a α β = apf (λ - → - • (iᵣ r ⁻¹ • α • iᵣ r)) (lem21 β) • apf (λ - → β • -) (lem20 α)
-- apf (λ - → - • (iₗ r ⁻¹ • β • iₗ r) ) (lem20 α) • apf (λ - → α • -) (le

⋆≡• : {A : Set} → (a : A) → (α β : Ω² {A} a) → (α ⋆ β) ≡ (α • β)
⋆≡• a α β = lem1 a α β • lem2 a α β

-- proven similairly to above
⋆'≡• : {A : Set} → (a : A) → (α β : Ω² {A} a) → (α ⋆' β) ≡ (β • α)
⋆'≡• a α β = lem1' a α β • lem2' a α β


--eckmannHilton : {A : Set} → (a : A) → (α β : Ω² {A} a) → α • β ≡ β • α
--eckmannHilton a r r = r
```

[TODO, fix without k errors]

Goals and Challenges

The parser is still quite primitive, and needs to be extended extensively to support natural language ambiguity in mathematics as well as other linguistic nuance that GF captures well, like tense and aspect. This can follow a method expored in Aarne's paper : "Translating between Language and Logic: What Is Easy and What Is Difficult" where one develops a denotational semantics for translating between natural language expressions with the desired AST. The bulk of this work will be writing a Haskell back-end implementing this AST transformation. The extended syntax, designed for linguistic nuance, will be filtered into the core syntax, which is essentially what I have done.

The Resource Grammar Library (RGL) is designed for out-of-the box grammar writing, and therefore much of the linearization nuance can be outsourced to this robust and well-studied library. Nonetheless, each application grammar brings its own unique challenges, and the RGL will only get one so far. My linearization may require extensive tweaking.

Thus far, our parser is only able to parse non-cubical fragments of the cubicalTT standard library. Dealing with Agda pattern matching, it was realized, is outside the theoretical boundaries of GF (at least, if one were to do it in a non ad-hoc way) due to its inability to pass arbitrary strings down the syntax tree nodes during linearization. Pattern matching therefore needs to be dealt with via pre and post processing. Additionally, cubicaltt is weaker at dealing with telescopes than Agda, and so a full generalization to Agda is not yet possible. Universes are another feature future iterations of this Grammar would need to deal with, but as they aren't present in most mathematician's vernacular, it is not seen as relevant for the state of this project.

Records should also be added, but because this grammar supports sigma types, there is no rush. The Identity type is so far deeply embedded in our grammar, so the first code fragment may just be for explanatory purposes. The degree to which the library is extended to cover domain specific information is up to debate, but for now the grammar is meant to be kept as minimal as possible.

One interesting extension, time dependnet, would be to allow for a bidrectional feedback between GF and Agda : thereby allowing ad hoc extensions to GF's ASTs to allow for newly defined Agda functions to be treated with more care, i.e. have an arguement structure rather than just treating everything as variables. This may be too ambitious for the time being.

Category theory in agda paper, differences in formalization

* my agda hott library * escardo's hott library - if successful on mine, with universe support - mix of latex, agda code , and natural language * dummy example for non-hott math (spivak et al, type-in-type) * alternatively, trying digging in the mountain at the other end, and try extedning ad-hoc grammar with various syntactic nuance * Latex Unicode support - * Degenerate cases - find examples which are unable to be supported by this grammar, explain why and offer future possible patches

Talk about all the things that need to be done

Pattern Matching, additional parser vs internal to GF

How to decide an optimal phrase (this seems like itd be some rule based) from agda program

* Support for cs math - e.g. specifications of algorithms and their actual implementations * Alternative syntaxes - graphical languages like grasshopper * user interface - QA - Hoogle for proofs * NL semantics (the semantic content is precisely the formal statements) * Comparison / integration with ML approaches * studies in concerete syntax -Harper psychology programming

Testing, with particular reference to the pgf grammar I developed

It is also worth noting that with respect to our earlier comparative analysis of PLs and NLs [cite earlier section], there has been work comparing things like numeracy, natural language acquisition skills, and programming language skills [39]. This account offers evidence that PL and NL acquisition in humans who have no experience coding and it is claimed that the study “are consistent with previous research reporting higher or unique predictive utility of verbal aptitude tests when compared to mathematical one” with respect to learning Python.

However, as Python is unityped, and similar experiments with a typed programming language would perhaps be more relevant - especially for studying mathematical abilities more consistent with the mathematicians notion of mathematics, rather than just numeracy. Additionally, it would be interesting to explore the role of PL syntax in such studies - and if what kind of variation could be linked to the concrete PL syntax.

Andreas comment about the proof state/terms being desugared in every known PL - ask a question of how one can make the interactivity more amenable to a kind of mathematical oracle, and therefore give semantic, not just syntactic goal states (i.e help allow the programmer to reason semantically)

Concrete syntax is in some sense where programming language theory meets psychology, (Bob Harper Oplss 2017)

While a GF grammar makes defining CNLs more convenient by doing the grammatical bookkeeping, it nonetheless is insufficient in the actual linguistic analysis of mathematics text. Angelov says the bigger grammars get, the more the begin to resemble a domain specific RGL []. A n so perhaps an doing this for at least some of the many varied examples in Ganesalingam’s text, bolstered with a corpus with more examples could allow for the creation of a mathematics RGL.

Code

GF Parser

Additional Agda Hott Code

References

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

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

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

[4] 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.

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

[6] 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.

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

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

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

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

[11] 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.

[12] 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.

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

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

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

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

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

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

[19] 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.

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

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

[22] 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.

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

[24] Howard, W. A. (1986). Per martin-löf. intuitionistic type theory. (notes by giovanni sambin of a series of lectures given in padua, june 1980.) studies in proof theory. bibliopolis, naples1984, ix 91 pp. *Journal of Symbolic Logic*, 51(4), 1075–1076.

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

[26] 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).

[27] 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.

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

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

[30] 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.

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

[32] 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.

[33] 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.

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

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

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

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

[38] 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.

[39] Prat, C. S., Madhyastha, T. M., Mottarella, M. J., & Kuo, C.-H. (2020). Relating natural language aptitude to individual differences in learning programming languages. *Scientific reports*, 10(1), 1–10.

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

[41] Ranta, A. (1994). 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.

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

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

[44] 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.

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

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

[47] 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.

[48] Ranta, A. & Ranta, D. (1994). *Type-theoretical Grammar*. Indices (Claredon). Clarendon Press.

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

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

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

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

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

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

[55] 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.

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

Appendices