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

Abstract

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

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

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

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

Till världens kvinnor

Preface

Having anticipated writing this for so long, it's no surprise that I have no idea what to say. I want to say everything, but this is probably not the place to do it. I write this in my final hours of quarantine in New York, having just contracted the most significant disease the world has faced in a generation, while the remnants of a disastrous hurricane pour outside my window and a crazy wildfire rips through my community back home. Three years ago, burnt out and more confused than ever, I decided not to go to Burning Man for the first time in my adult life. Three years ago *today* I stepped foot in Sweden for the first time, and will be back home in the desert tomorrow for the first time in two years. Life's a fucking trip.

I certainly wouldn't be here writing this right now if it weren't for my mother, Beth, whose endless love helps steer me through life, nor would I be here without the my father, Eddie, whose life has inspired me, whose love has guided me, and whose death thirteen years ago has given me the insight to treat every day as if it's my last. I love you both: for having me and Graham, for loving each other, and for teaching me how to love. I love you too, Graham.

I love and thank all the people who helped get me to a point in life where I can write this, and I hope this love is reflected in the gratitude I feel for y'all, for us. Thank you Daniel for sitting on the phone with me day after day, teaching me algebra and geometry. To Mr. Harris for teaching me to love learning, Mr. McCart for teaching me to think critically, and to Mr. Meinert for teaching me what science is, thank you. To Peter, Danny, Adrian, Bill and the rest of the library crew, thank you. And to the rest of you Reno folk, namaste motherfuckers. At this point I gotta start just listing names : Donna, Andrew, Cliff, James K, Ky, James R, Adrienne, Rachel, Jeremy, Ari, Christophe, David, Kiki, Michelle, Chuck, Erik, Kieran, Jake, Gloria, Mirabai, Ravi, and too many others to mention, I hope I'm an ample reflection of the life and wisdom you've instilled in me, and know that I love you.

To Alessandro, Jose, and Shanshuang, thank you for teaching me to love mathematics and science during my undergraduate years. Thank you Aarne for your time, patience, and conversations. Thank you Inari for your endless support and insight. To the many friends and colleagues I've had the pleasure to know and learn from in Göteborg : Ayberk, Fabian, Nachi, Irena, Sandro, Carlos, Robert and Theresa, and the rest of the ITC crew, tack. And to various mentors whose lectures, writings, and conversations have expanded my love and knowledge of type theory, linguistics, and mathematics : Andreas, Thierry, Peter, Jesper, and Krasimir from the CS department and Robin, Stergios, Jean-Phillipe, Frederik, and Martin from FLOV, tack så mycket.

Efter tre år i Sverige kan jag nu prata och skriva lite svenska, och som den amerikan jag är, vill jag visa upp lite av vad jag kan. Jag har så mycket kärlek för Sverige, trots att de här åren på vissa sätt har varit den svåraste och mest prövande tiden i mitt liv. Men, jag har också vuxit som mest, eftersom jag har fått så mycket kärlek från människor som älskar livet, naturen, och hela planeten. Detta har givit mig ett nytt perspektiv på världen.

Jag vill tacka Per Martin-Löf, vars texter har lärt mig om hur man kan ha så mycket originella idéer, men också vara så ödmjuk. Och ödmjukhet - något jag har upplevt ofta i Sverige. Det var mycket generöst och inspirerande av dig att prata med mig. Tack, Per.

Och till mina nya svenska vänner som jag älskar så mycket, jag har fått mer glädje i mitt liv för att jag har er i mitt liv. Till David, Pär, Maja och Linnea - tack så mycket. Tack också till Charlotte och Uma.

Jag har gråtit mycket när jag har skrivit den här masteruppsatsen, men jag har också skrattat och lett. Jag känner att jag har uttryckt något originellt, även om det inte är mycket. Tack för att jag har fått säga det jag har sagt här.

Contents

1 Introduction

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

1.1 Beyond Computational Trinitarianism

> The doctrine of computational trinitarianism holds that computation manifests itself in three forms: proofs of propositions, programs of a type, and mappings between structures. These three aspects give rise to three sects of worship: Logic, which gives primacy to proofs and propositions; Languages, which gives primacy to programs and types; Categories, which gives primacy to mappings and structures.*Robert Harper* [43]

We begin this discussion of the three relationships between three respective fields, mathematics, computer science, and logic. The aptly named trinity, shown in Figure 1, 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 completely *formal*.

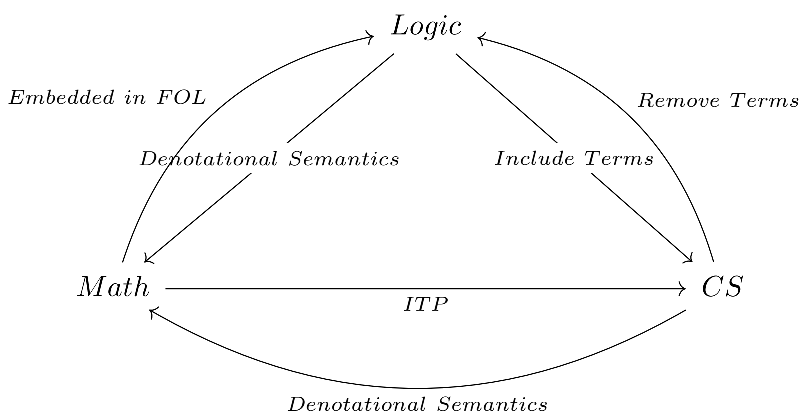

Figure 1: The Holy Trinity

We hope this thesis will help clarify another possible dimension in this diagram, that of Linguistics, and call it the “holy tetrahedron”. The different intellectual communities represented by the three subjects of the trinity also resemble religions in their own right, 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 heretical by including a much less well understood perspective which provides additional challenges and insights into the trinity.

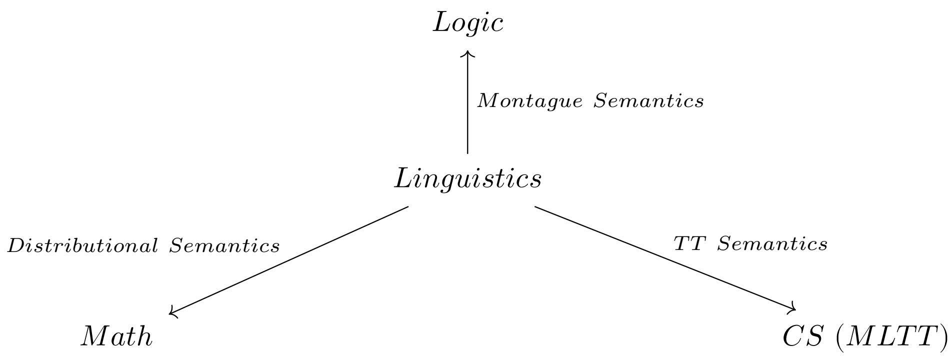

Figure 2: Formal Semantics

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

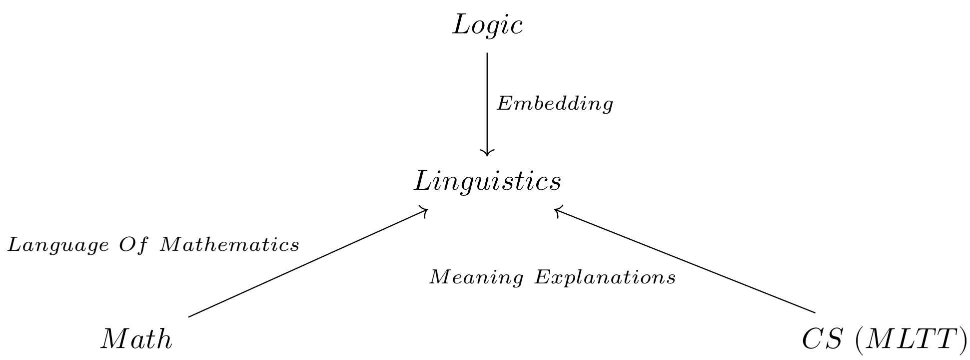

Figure 3: Interpretations of Natural Language

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 it is important to ask how they can be used in applications for mathematicians, logicians, and computer scientists. Therefore we hope this attempt at giving the language of mathematics, in particular how propositions and proofs are expressed and thought about in that language, a stronger foundation.

1.2 The Goal before Us

> Treating propositions as types is definitely not in the way of thinking of ordinary mathematician, yet it is very close to what he actually does. *N. G. de Bruijn* [31].

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).

The type-checker, a language's mechanism of ensuring that programs satisfy the specification of their types, is incredibly useful: 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 its 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 [91].

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 Agda, software updates may cause proofs to break, and the inevitable imperfections of software are 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, $\phi \vee \neg\phi$. She merely observes them, and admits its handiness in certain situations. Excluded middle is indeed a helpful tool - many important results rely on it. The type theorist's contention is that it should be avoided whenever possible - proofs which don't rely on it, or it's corollaries like proof by contradction, are much more amenable to formalization in systems with decidable type checking. Zermelo-Fraenkel Set Theory with the axiom of choice, ZFC, while serving the mathematicians of the early 20th century, is lacking when it comes to the higher dimensional structure of n-categories.

What ITPs 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. Foundational details aside, this thesis is meant to provide insight into one piece, namely in the space of syntax, of a blueprint for what many hope to be a reformation regarding how mathematics is practiced.

We don't insist a mathematician relinquish the beautiful language she has come to love in expressing her ideas. Rather, we asks her to make a hypothetical compromise for the time being, and use a Controlled Natural Language (CNL) to encode key developments in her work. In exchange she may get some of the confidence that Agda provides. Not only that, if adopted at scale, she will 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 [41], and it should practically motivate part of our 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. 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, and how their translation can really take place. This distant goal should be prefaced with a semantic study of mathematics, which itself needs a syntactic basis. 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.

1.3 Philosophical Perspectives

> ...when it comes to understanding the power of mathematical language to guide our thought and help us reason well, formal mathematical languages like the ones used by interactive proof assistants provide informative models of informal mathematical language. The formal languages underlying foundational frameworks such as set theory and type theory were designed to provide an account of the correct rules of mathematical reasoning, and, as Gödel observed, they do a remarkably good job. But correctness isn't everything: we want our mathematical languages to enable us to reason efficiently and effectively as well. To that end, we need not just accounts as to what makes a mathematical argument correct, but also accounts of the structural features of our theorizing that help us manage mathematical complexity. *Jeremy Avigad* [9]

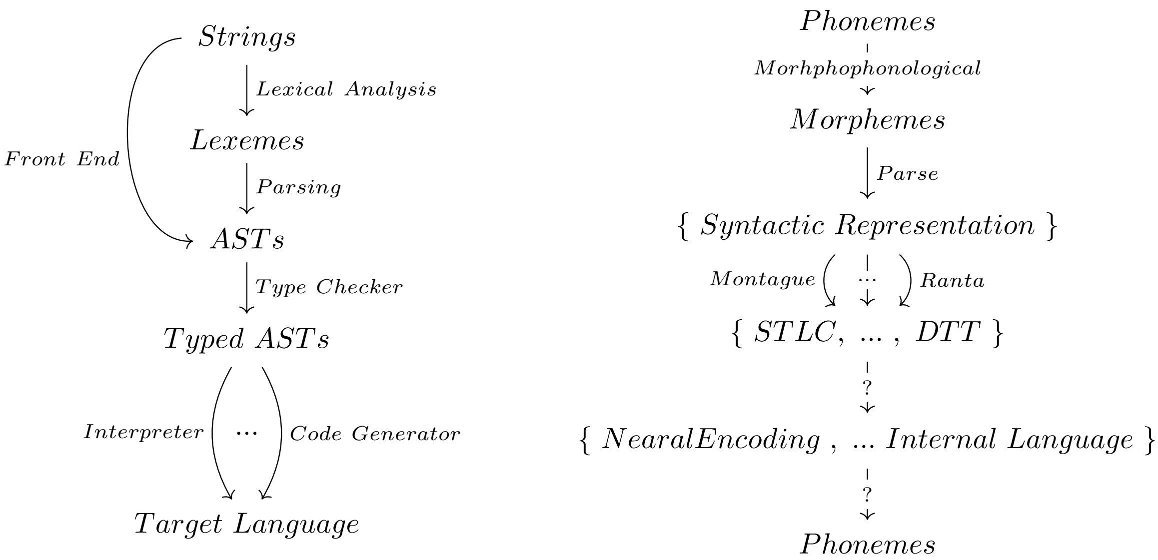

Figure 4: PL (left) and NL (right) Abstraction Ladders

1.3.1 Linguistic and Programming Language Abstractions

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

Computational linguistics, particularly in the tradition of type theoretical semantics[74], gives one a way of comparing natural and programming languages. Type theoretical semantics is concerned with the semantics of natural language in the logical tradition of Montague, who synthesized work in the shadows of Chomsky [23] and Frege [37]. This work ended up inspiring the GF. Indeed, one such description of GF is that it is a compiler tool applied to domain specific machine translation. Comparing the “compiler view” of PLs and the “linguistic view” of NLs, we may interpolate this comparison to other general phenomenon in the respective domains.

We make this comparison via two abstraction ladders, visible in Figure 4. Observe that the PL dimension on the left represents synthetic processes, those which we design, make decisions about, and describe formally. The NL abstractions on the right represent analytic observations, and are subject to empirical observations .

This diagram only serves as an atlas for the different abstractions - it is certainly subject to modifications depending on the mathematician, linguist, or philosopher investigating such matters. The PL abstractions as serve as an actual high altitude blueprint for the design of programming languages. While this view may ignore interesting details and research, it is unlikely to create angst in the computer science communities. The linguistic abstractions, on the other hand, are at the intersection of many fascinating debates. There is consensus among linguists which abstractions and models of those abstractions are more practically useful, theoretically compelling, or empirically testable.

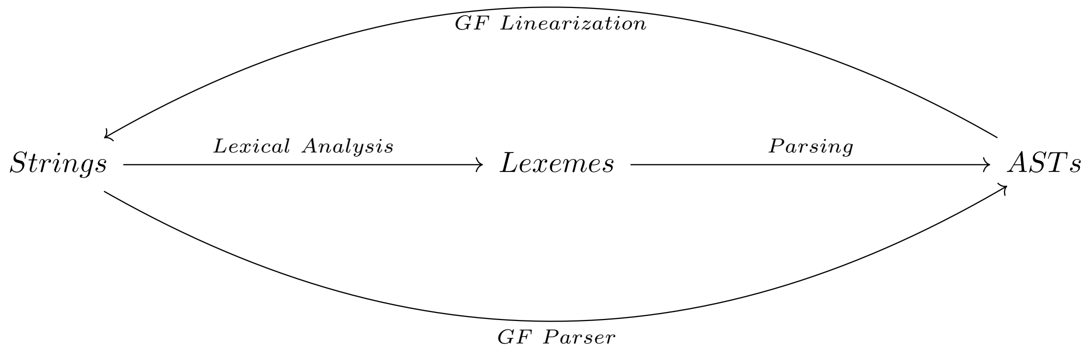

Figure 5: GF in a nutshell

There are also many relevant concerns not addressed in either abstraction chain. We may consider intrinsic and extrensic abstractions that diverge from the idealized picture. For PLs we can inquire about systems with multiple programming languages, or what the intensional behaviors of programs are during evaluation. 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 AST" [2]. 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 one must decide which stage one should connect an Agda AST with GF's representation.

Let's zoom in a little and observe the so-called front-end part of the compiler. Displayed in Figure 5 is the "kernel" of GF. 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 parser and linearizer . The decision to overlay the abstraction ladders at the syntactic and semantic level led to GF's development.

For natural language, some intrinsic properties of interest could be viewed at the neurological level, where one may contrast the internal language (i-language) with the mechanism of externalization (generally speech) as observed by Chomsky [22]. Extrinsic to the linguistic abstractions depicted, pragmatics is absent. There are stark differences between NLs and PLs where our comparative abstractions break down. Classifying PLs as languages is best read as an incomplete metaphor due to perceived similarities.

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 embedded theory, say arithmetic over $\mathbb{N}$, we ask how do we not just find a natural language representation which interprets our expressions, but also does so in a way that a competent speaker understands?

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 practically difficult in the sense that it may be either infeasibly complexity or undecidable. But it also poses questions of what it means to be a correct specification of a theorem, which is not something we can capture in a formal system.

Like all collective human endeavors, mathematics is a historical construction - that is, its conventions, notations, understanding, methodologies, and means of publication and distribution have 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 on paper. Over time mathematics has been filtered of natural language artifacts, culminating in some sense with a formal proof, despite mathematicians not being intimately familiar this formality as it is treated in type theory. This work emphasizes the need for a new foundational mentality, whereby we try to bring natural language back into "formal mathematics" in a controlled way.

Mathematical constructions like numbers and shapes arose out of ad-hoc needs as humans cultures grew and evolved over the millennia. Unfortunately, most of this evolution remains undocumented. While mathematical intuitions precede mathematical constructions (the observation of a spherical planet precedes the human use of a ruler compass construction to generate a circle), we should assume that mathematics arises naturally out of our linguistic capacity. This may very well not be the case, but it is impossible to imagine humans developing mathematical constructions elaborating anything particularly interesting without linguistic faculties. Regardless of the 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.

1.3.2 Formalization and Informalization

Formalization is the process of taking a 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 - Hales formalization of the Kepler Conjecture took an estimated 20 human years of labor. 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 an error, or other bureaucratic obstacles
- it may not type-check, and only be presumed hypothetically to be a correct formalization with some ad-hoc evidence

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

Informalization, on the other hand is a process of taking a piece formal syntax, and turning it into a natural language utterance, along with commentary motivating and or relating it to other mathematics. It is a clarification of the meaning of a piece of code, suppressing certain details and sometimes redundantly reiterating other details. 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. Just as any machine translating a Virginia Woolf novel from English to Mandarin is incomparable to a human translator's expertise, machines formalizing or informalizing mathematics are destined to be futile relative to a mathematician making such a translation. Despite this futility, it shouldn't deter one so inspired to try.

1.3.3 Syntactic Completeness and Semantic Adequacy

There are two fundamental criteria that must be assessed for one to judge the success of an approach over both formalization and informalization. The first, *syntactic completeness*, means that a construction contains all the syntactic information necessary to verify its correctness. It says that a term type-checks in the PL case, or some natural language form can be deterministically transformed to a term that does type-check. We may therefore ask 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?

This problem of creating a syntactically complete mathematical landscape is certainly infeasible generally - a mathematician might not be able to reconstruct the unstated syntactic details of a proof in an discipline outside her expertise. Additionally, certain necessary syntactic details may also detract from a natural language reading comprehension of a proof. Perhaps most importantly, one does not know a priori that the generated expression in the type theory has its *intended meaning*. The saying "grammars leak" can be transposed to say "natural language proofs leak" insofar as they are certain to contain syntactically necessary omissions.

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 general notion of semantic adequacy is delicate, and certainly not formally definable. Mathematicians themselves may dispute the proof of a given proposition or the correct definition of some notion.

There are few working mathematicians who would not understand some standard theorem in an arbitrary introductory analysis text, even if they may dispute it's presentation, clarity, pedagogy, or take other issues. Whether one asks that semantic adequacy means some kind of sociological consensus among those with relevant expertise, or a more relaxed criterion that some expert herself understands the argument, a dubious perspective in scientific circles, semantic adequacy should appease at least one and potentially more mathematicians.

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

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

Figure 7: Formal and Informal Mathematics

We introduce these definitions, 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 is likely impossible. Even introducing objective criteria to really judge these definitions in special will require a lot of work.

This perspective represents an observation, it is not intended to judge whether the syntactic or semantic perspective in mathematics is better - there is a perpetual dialogue unfolding, where dialectical adjustments are perpetually taking place.

The Syntactic Nature of Agda The act of writing and reading an Agda proof are significantly different endeavors, as the term shadows the application of typing rules which enable its construction. When the Agda user builds her proof, she is outsourcing the bookkeeping to the type-checker. This isn't purely a mechanical process, she often does have to think how her definitions will interact with downstream programs. Like all mathematicians, she must ask if assertions sensible to begin with, and she must carefully craft a strategy of attack before hacking away.

For someone reading Agda code, if proofs were semantically coherent, where only a few occasional comments about various intentions and conclusions would be needed to understand the material. In reality, the human witness of a large term may easily be confused why it fulfills the typing judgment. The reader may have to reexamine parts of the proof by trying to rebuild it interactively with Agda's help.

Yet, papers are often written exclusively in Latex, where Agda proofs have been reverse engineered to appease the audience and only relevant semantic details have been preserved. 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 legible. The description and commentary is still largely necessary to convey the *important material*, regardless if the Agda code is self-contained. And while literate Agda is a bridge, the commentary still unfolds the code.

Coq other ITPs In Coq programming language, proof terms are built using Ltac, a scripting language for writing imperative syntactic meta-programs 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). Ltac often descends into the sorrows of so-called untyped languages, although there are recent attempts to change this [47] [69]. From one perspective, the use of tactics is an additional syntactic obfuscation of what a proof should look like from the mathematicians perspective - and remedies are a research topic. 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 [73]. This has the advantage of making intermediate types visible, something mathematicians do when writing their proofs (even though everything is ultimately inferable by the type-checker).

From another perspective tactics sometimes enhance high level proof understanding, as tactics like *ring* or *omega* often save the reader overhead of parsing pedantic and uninformative details (what mathematicians will often leave to the reader). 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.

Other interactive proof assistants, like Lean, Isabelle, the HOL family, merit attention we don't have the space to give them here. There have been surveys comparing them [95]. It would be incredibly beneficial to explicitly compare these ITPs, in terms of features they offer, how their syntax and semantics differ, and how their syntax and semantics affects the ways proofs are constructed and shown. Additionally, a larger corpus of proofs would showcase these differences and give additional direction to the work we do here.

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.

1.3.4 What is a proof?

> A proof is what makes a judgment evident [61].

The proofs of Agda and any 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 better be considered proofs that may be communicable to aliens. However, formal proofs certainly aren’t the objects mathematicians deal with daily.

Mathematics, and the act of proving theorems, according to Brouwer is a social process. Suppose we have two humans, h_1 and h_2. If h_1 claims to have a proof p_1, and elaborates it to p_2 who claims she can either verify p_1 or reproduce and re-articulate it via p_1', such that h_1 and h_2 agree that p_1 and p_1' are equivalent, then they have discovered some mathematics. In this guise mathematics may be viewed as a science, because there is a notion of reproducibility of evidence.

The Architect and the Engineer 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*.

Informal proofs are specifications and formal proofs are implementations. Two different authors may informalize the same code differently - they may suppress different details and choose to emphasize different details, leading to two unique, 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. It is the architect who has the vision, and the engineers who end up implementing the architects art.

We also pose a different analogy, comparing the mathematician and the physicist. The physicist will often say under her breath, “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 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. In this case, the "pen and paper" mathematician fills the roll of the physicist, and the Agda user the mathematician. Formality when using mathematics is a spectrum.

There isn't a natural notion of equivalence between informal and formal proofs, but perhaps the categorical idea adjunction is more relevant. The fact that the "acceptable" natural language utterances aren't inductively defined precludes us from actually constructing a canonical mathematical model of formal/informal relationship. However, if GF perspective of translation is used, there can at least be an approximation of what a model may look like. The linguist interested in the language of mathematics should perhaps be seen as a scientist, whose purpose is to contribute basic ideas and insights about the natural world from which the architects and engineers can use to inform their designs.

Mathematicians naturally 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). The implementation of a result in Agda versus Coq may lead to intentionally different objects which represent the same thing extensionally. It's also noted a proof doesn't obey the same universality that it does when it's on paper or verbalized - that reliance on Agda 2.6.2, and its current standard library, when updated in the future, may "break proofs". We believe the GF approach offers at least a step in the direction of foundational agnosticism which may appease some of these issues.

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 believe our work provides a nontrivial perspective at many important issues in this canon of great thinkers. We emphasize the following questions:

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

While various branches of linguistics have seen rapid evolution due to, in large part, their adoption of mathematical tools, the dual application of linguistic tools to mathematics is quite sparse and open terrain.

The view of what mathematics is in a philosophical and mathematical sense, particularly with respect to foundations, 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.

A Reconsideration of Proof

> 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. *Marcus Giaquinto* [40]

We here touch upon additional non-syntactic phenomena - "proofs without words" [66] and other diagrammatic and visual reasoning tools in mathematics and programming. The role of visualization in programming, logic, and mathematics offers an abundance of contrast to syntactically oriented alphanumeric alphabets, e.g. strings of symbols. Visualization are ubiquitous in contemporary mathematics: plotting diagrams, 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 [35].

Additionally, graphical programming languages which facilitating diagrammatic programming are one instance of a nonlinear syntax. 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 [10]. Alecytron supports basic data structure visualization, like red-black trees which carry semantic content less easy in a string based-setting [73]. These languages prove tricky but possible to implement grammars for in GF, because one would presumably have to map to an internal AST.

There are also, famously, blind mathematicians who work in topology, geometry, and analysis [46]. 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.

This brief discussion of visualizations should blur the syntactic understanding of mathematics which we emphasize in this work. We highlight the difference between *a proof* and the *the understanding of a proof*. The understanding of a proof, 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.

2 Technical Preliminaries

2.1 Propositions, Sets, and Types

Complete overviews of Martin-Löf type theory been well-articulated elsewhere [32], and we have given a brief introduction in the appendix 8.1. We compare the syntax of mathematical constructions in FOL, a possible natural language use from [79], and MLTT. From this vantage, these look like simple symbolic translations, and in some sense, one doesn't need 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
$\forall\ x\ \in \tau P(x)$	$\Pi x : \tau.\ P(x)$	$for\ all\ x\ in\ \tau,\ p$	$the\ product\ over\ x\ in\ p$
$\exists\ x\ \in \tau P(x)$	$\Sigma x : \tau.\ P(x)$	$there\ exists\ an\ x\ in\ \tau\ such\ that\ p$	$sigma\ x\ in\ \tau\ such\ that\ p$
$p\ \supset\ q$	$p\ \to\ q$	$if\ p\ then\ q$	$p\ to\ q$
$p\ \wedge\ q$	$p\ \times\ q$	$p\ and\ q$	$the\ product\ of\ p\ and\ q$
$p\ \vee\ q$	$p\ +\ q$	$p\ or\ q$	$the\ coproduct\ of\ p\ and\ q$
$\neg\ p$	$\neg\ p$	$it\ is\ not\ the\ case\ that\ p$	$not\ p$
$\top$	$\top$	$true$	top
$\bot$	$\bot$	$false$	$bottom$
$p\ =\ q$	$p\ =_A\ q$	$p\ equals\ q$	$p\ propositionally\ equals\ q\ at\ A$

Figure 8: FOL vs MLTT

Set Theory	MLTT	NL Set Theory	NL MLTT
S	τ	$the\ set\ S$	$the\ type\ \tau$
$\mathbb{N}$	Nat	$the\ set\ of\ natural\ numbers$	$the\ type\ nat$
$S \times T$	$S \times T$	$the\ product\ of\ S\ and\ T$	$the\ product\ of\ S\ and\ T$
$S \to T$	$S \to T$	$the\ function\ \ S\ to\ T$	$p\ to\ q$
$\{x \vert P(x)\}$	$\Sigma x : \tau.\ P(x)$	$the\ set\ of\ x\ such\ that\ P$	$there\ exists\ an\ x\ in\ \tau such\ that\ p$
$\emptyset$	$\bot$	$the\ empty\ set$	$bottom$
$\{\emptyset\}$	$\top$	$the\ singleton\ set$	top
$S \cup T$	$?$	$the\ union\ of\ S\ and\ T$	$?$
$S \subset T$	$S <: T$	$S\ is\ a\ subset\ of\ T$	$S\ is\ a\ subtype\ of\ T$
$\aleph_1$	U_1	$the\ first\ uncountable\ cardinal$	$the\ first\ large\ universe$

Figure 9: Sets 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" in intuitionistic type theory. 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.

We also note that, type theorists often interchange the logical, set theoretic, and type theoretic vocabularies when describing types. Because 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 $Nat\ implies\ Nat$ when expressing $Nat \ \rightarrow \ Nat$.

Terms become even messier, and this can be seen in just a small sample shown in Figure 10. In simple type theory, one distinguishes between types and terms at the syntactic level - this disappears when one allows dependency. As will be seen later, the mixing of terms and types gives MLTT an incredible expressive power, but undoubtedly introduces difficulties. 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. 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 the proof rules themselves contain the necessary information to encode the proofs or constructions. The terms in type theory compress and encode the proof tree derivations - where nodes are displayed during the interactive type-checking phase in ITPs.

Set Theory	MLTT	NL Set Theory	NL MLTT	Logic
$f(x) := p$	$\lambda x.p$	$f\ of\ x\ is\ p$	$lambda\ x,\ p$	$\supset -intro$
$f(p)$	$f\ p$	$f\ of\ p$	$the\ application\ of\ f\ to\ p$	$modus\ ponens$
(x, y)	(x, y)	$the\ pair\ of\ x\ and\ y$	$the\ pair\ of\ x\ and\ y$	$\wedge - i$
$\pi_1\ x$	$\pi_1\ x$	$the\ first\ projection\ of\ x$	$the\ first\ projection\ of\ x$	$\wedge - e_1$

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

We don't include 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 not common relative to coproducts in more general categories.

- λ is a constructor for both the dependent and non-dependent function, so its use in either case will be type-checked by Agda,
- 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.

2.2 Agda

2.2.1 Overview

Agda is an attempt to faithfully formalize Martin-Löf’s intensional type theory [60] into a functional programming language . One can think of Martin-Löf’s original work as a specification of a foundational system, and Agda as one possible implementation.

Through an interactive environment, Agda 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 Agda1. In addition to the core MLTT, Agda incorporates dependent records, inductive definitions with all types of bells and whistles, pattern matching, a versatile module system, and a myriad of other features which are of interest generally but not relevant to this work.

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. And while there are still many reasons one may wish to use other programming languages, there is a sense of purity one gets when writing Agda code. There are many good resources for learning Agda [16] [89] [17] [93] so we’ll only give a cursory overview of what is relevant for this thesis, with a particular emphasis on the 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

2.2.2 Agda Programming

Listed is the syntax Agda uses for judgements: `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. Let's compare these Agda judgements to those keywords ubiquitous in mathematics:

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, $\mathbb{B}$, we present these rules both in the proof theoretic and Agda syntax. We note that the context Γ is not present in Agda.

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

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

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

The elimination forms are deriveable from the introduction rules, and the computation rules can then be extracted by via the harmonious relationship between the introduction and elmination forms [70]. Agda's pattern matching is equivalent to the deriveable dependently typed elimination forms [27], 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. The `if then else` statement shown below is really just the boolean elimination form. It is not standard to include the premises of the eqaulity rules.

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

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

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

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

When using Agda one is interactively building a proof via holes. There is an Agda Emacs mode which enables this. Glossing over many details, we show sample code in the proof development state prior to pattern matching on `b`. We have a hole, `{ b }0`, and the proof state is displayed to the right. It shows both the current context with `A, b, a1, a2`, the goal which is something of type `A`, and what we have, `B`, represents the type of the variable in the hole.

```
if_then_else_ :
  {A : Set} → B → A → A → A
if b then a1 else a2 = { b }0
```

```
Goal: A
Have: B
———————————————
a2 : A
a1 : A
b  : B
A  : Set   (not in scope)
```

The interactivity is performed via emacs commands, and every time one updates the hole with a new term, we can immediately view the next goal with an updated context. The underscore in `if_then_else_` denotes the placement of the arguements, as Agda allows mixfix operations. Agda allows for more nuanced syntacic features like unicode. This is interesting from the *concrete syntax* perspective as the arguement placement and symbolic expressiveness makes Agda's syntax feel 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 boolean negation, `~`, and more interestingly, `functionalNegation`, where one can use functions as arguements. `functionalNegation` is a functional, or higher order functions, which take functions as arguements and return functions. We also notice in `functionalNegation` that one can work directly with a built-in λ to ensure the correct return type.

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

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

This simple example leads us to one of the domains our subsequent grammars will describe, like arithmetic (see 6.1.1). We show how to inductively define natural numbers in Agda, with the formation and introduction rules included beside for contrast.

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

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

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

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

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

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

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

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

Since we are in a dependently typed setting, however, we prove theorems as well as write programs. Therefore, we can see this recursion principle as a special case of the induction principle `natind`, which represents the by induction for the 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.

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

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

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

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

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

We will defer the details of using induction and recursion principles for later when we actually give examples of pidgin proofs some of our grammars can handle.

2.2.3 Formalizing The Twin Prime Conjecture

Inspired by Escardos's formalization of the twin primes conjecture [33], 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, along with the definition of a twin prime, is incredibly compact. We include Escardo's definition below the natural.

Lemma 1 *There are infinitely many twin primes.*

Definition 1 *A* twin prime *is a prime number that is either 2 less or 2 more than another prime number*

```
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 and Agda presentation. First, the Agda 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 embedding a GF grammar in Haskell allows one to add a semantic layer where the symmetry can be explicitly included 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 natural numbers. In this case our notion of infinity we rely on the order of $\mathbb{N}$. 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.

Definition 2 *A* twin prime *is a prime that has a prime gap of two. 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 define something *everyone* alternatively knows. Agda, however, can't infer this.

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. 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 6.2. 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 8.3.

As a personal anecdote, I tried to prove that 2 is prime in Agda, which turned out to be nontrivial. When I told this to a mathematician 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. Agda's standard libary proof is done with tactics and reflection - a way of quoting a term into in abstract syntax tree and then performing some kind of metacomputation. While elegant, this obviously requires a lot of machinery, none of which would be easy to communicate to a mathematician who doesn't know much about coding. Seemingly trivial things, when treated by the type theorist or linguist, can become wonderful areas of exploration.

3 Previous Work

According to legend, Göran Sundholm and Per Martin-Löf were sitting at a dinner table, discussing various questions of interest, and Sundholm presented Martin-Löf with the problem of *donkey sentences* in natural language semantics, statements 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 dependent types, 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 [74], bled into the development of GF, and has now led to this current work.

The prior exploration of formal languages for the interleaving Trinitarian subjects, is vast, and we can only sample the literature[48]. Our approach, using GF ASTs as a has many roots and interconnections with this literature. The success of finding a suitable language for mathematics will obviously require a comparative analysis of the strengths and weaknesses in such a vast bibliography. How the GF approach compares with this long list merits careful consideration and future work. We focus on a few resources.

3.1 Ranta

The initial considerations of Ranta were both oriented towards the language of mathematics [75], as well as purely linguistic concerns [74]. In his treatise, Ranta explores not just the many avenues to describe NL semantic phenomena with dependent types, but, after concentrating on a linguistic analysis, he also proposes 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 [54]. Therefore, if we interpret the set of men and the set of donkeys as types, e.g. we judge $\vdash man : \text{type}$ and $\vdash donkey : \text{type}$ where type really denotes a universe, and ditransitive verbs "owns" and "beats" as predicates, or dependent types over the CN types, i.e. $\vdash owns : man \to donkey \to \text{type}$ 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 [64], 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 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 the projections π_1 and π_2, or Σ eliminators.

In the final chapter of [74], *Sugaring and Parsing*, 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 GF's 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 [53] developed in Göteborg allowed for sugaring that can largely accommodate morphological features of the target natural language [36], the translation between the functional phrase structure (ASTs) and strings [76].

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 [80], it has remained relatively untouched on the GF programmers tool-belt. 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 [55]. Although GF isn't explicitly designed with type-checking in mind , it would be very interesting to apply GF dependent types in the more advanced programming languages to filter 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, the derivative development of GF and their confluence. This led to the development of a natural language layer to Alfa [42], which can be seen as a direct predecessor to this work. In some sense, we seek to recapitulate what was already done in 1998 - but this was prior to both GF's completion, and Alfa's hard fork to Agda.

3.2 Mohan Ganesalingam

> There is a considerable gap between what mathematicians claim is true and what they believe, and this mismatch causes a number of serious linguistic problems. *Mohan Ganesalingam* [39]

The most substantial analysis of the linguistic perspective on written mathematics comes from Ganesalingam [39]. Not only does he pick up and reexamine much of Ranta's early work, but he develops a whole theory for how to understand 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 (DRT) [49], to capture anaphoric use of mathematical 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. Our project should be thought of as more of a way to implement the linguistic features of mathematics rather than Ganesalingam's work focusing on analysis.

Ganesalingam draws insightful, nuanced conclusions from compelling examples. Nonetheless, this subject is somewhat restricted to a specific linguistic tradition and modern, textual mathematics. Therefore, we hope to contrast our GF point of view while offering some perspectives on his work.

He remarks that mathematicians believe "insufficiently precise" mathematical sentences are those which would be result from a failure to translate them into logic. This is much more true from the Agda developers perspective than the mathematicians. Mathematicians generally assume small mistakes may go unnoticed by the reviewers.

Ganesalingam 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.

He also discusses the important distinction between formal (which he focuses on) and informal modes in mathematics, with the informal representing the "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, although it is uncertain if the commentary should just be delegated to comments if translated to Agda.

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 some person's various stages of coming to terms with concepts in mathematics and their generalization in that person's 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, but require proof otherwise. These details are unaccounted for by the mathematician, but can often hamper formalization efforts because the substitution of propositionally equal and definitionally equal terms are treated differently outside of select type theories.

Mathematical variables, it is also noticed, can be treated anaphorically. From the PL perspective 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. These examples should showcase that Ganesalingam's analysis, while certainly helpful when building grammars, may require additional analysis to actually make things work.

3.2.1 Pragmatics in mathematics

Ganesalingam makes one observation which is particularly pertinent to our analysis and understanding of mathematical language, which is that of pragmatics content. The point warranted both a rebuttal [85] and an additional response by Ranta [83]. 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."

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 at 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 , 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 carry pragmatic content in that the author has to decide - how to judge the content by its "importance" and its relation to the *theorems*. Inferring how to judge a keyword seems impossible for a machine, especially since critical results are 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 covered by San Mauro et al. Ranta contends 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 study the *speaker's meaning* of a mathematical utterance by trying to translate them into syntactically complete Agda judgments. *Efficient communication* is the goal of producing a semantically adequate grammar. The prospect of creating a grammar which satisfies both is the most difficult task. We therefore hope that modeling of natural language mathematics will give insights into how understanding of all five pragmatic phenomena are necessary for worthwhile translations between CNLs and formal languages, even if our grammars only really work with syntax. For the CNLs to really be "natural", one must be able to infer and incorporate pragmatic phenomena.

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, and additionally learn about the soil in which we are digging.

3.3 Other authors

> QED is the very tentative title of a project to build a computer system that effectively represents all important mathematical knowledge and techniques. [1]

The ambition of the QED Manifesto, with formalization and informalization of mathematics being a subset of its grandiose vision, is probably impossible. We examine a few of the myriad attempts at languages providing a bridge between the formal and informal mathematics.

N.G. de Brujn's Automath [18], a language for expressing mathematics developed at Eindhoven in the late 60s was a pioneer. It was the first CNL for mathematics well. It gave the first notion of a proof object. It put the notion of substitution of variables at its center, leading to the development of the de Brujn presentation of the lambda calculus. Most interestingly for us is the fact that it was "not a programming language", and didn't have a type-checker capable of guiding proof development, but a notation for encoding constructions. We emphasize this because this notation, which we may now associate with concrete syntax, was actually one of the guiding ideas which made Automath so powerful and caused it to have such a big influence in the development of ITPs generally.

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 [29]. A central goal of Naproche is to develop a controlled natural language, 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 for 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, in addition a database of known results [84]. It is based off Tarski-Grothendeick set theory, and allows for correctness checking of articles. It was originally developed in Poland concurrent to Martin-Löf's work in 1973, and so much of the interest in types instead of sets couldn't be anticipated. Mizar's focus 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 the largest library of for CNL results. Additionally, it has inspired iterations for other vernacular proof assistants, like Isabelle's Intelligible semi-automated reasoning (Isar) extension [67].

Subsequently, in [94], 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. While showing promising initial results, nothing as of yet can be foreseen to manifest in general use. There are opinions that large-scale autoformalization is feasible [90], but our work certainly casts doubts. 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 [30] . 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.

The Boxer system, a CCG parser [15] which allows English text translation into FOL. However, it is not always correct, and dealing with the language of mathematics will present obstacles.

In [28] 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) [77]. 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.

It should be noted that GF's role in this space is primitive. Tools like the Grammatical Logical Inference Framework (GLIF), which uses GF as a front-end for the Meta-Meta-Theory framework [86], offer more evidence of the role GF may play in this space.

4 Grammatical Framework

4.1 Introducing GF

A grammar specification in GF is an abstract syntax, where one specifies trees, and a concrete syntax, where one says how the trees compositionally evaluate to strings. Multiple concrete syntaxes may be attached to a given abstract syntax, and these different concrete syntaxes represent different languages. An AST may then be linearized to a string for each concrete syntax. conversely, given a string admitted by the language being defined, GF's parser will generate all the ASTs which linearize to that tree.

When defining a GF pipeline, one has to merely to construct an abstract syntax file and a concrete syntax file such that they are coherent. In the abstract, one specifies the *semantics* of the domain one wants to translate over, which is ironic, because we normally associate abstract syntax with *just syntax*. However, because GF was intended for implementing the natural language phenomena, the types of semantic categories (or sorts) can grow much bigger than is desirable in a programming language, where minimalism is generally favored. The *foods grammar* is the *hello world* of GF, and should be referred to for those interested in example of how the abstract syntax serves as a semantic space in non-formal NL applications [78].

Let us revisit the "tetrahedral doctrine", now restricting our attention to the subset of linguistics which GF occupies. We first examine how GF fits into the trinity, as seen in Figure 12. A GF abstract syntax with dependent types can just be seen as an implementation of MLTT with the added bonus of a parser once one specifies the linearizations. Additionally, GF is a relatively minimal type theory, and therefore it would be easy to construct a model in a general purpose programming language, like Agda. Embeddings of GF already exist in Coq [12], Haskell [7], and MMT [50]. These applications allow one to use GF's parser so that a GF AST may be transformed into some notion of inductively defined tree these languages all support. From the logical side, we note that GF's parser specification was done using inference rules [5]. Given the coupling of Context-Free Grammars (CFGs) and operads (also known as multicategories) [51] [44] one could use much more advanced mathematical machinery to articulate a categorical semantics of GF.

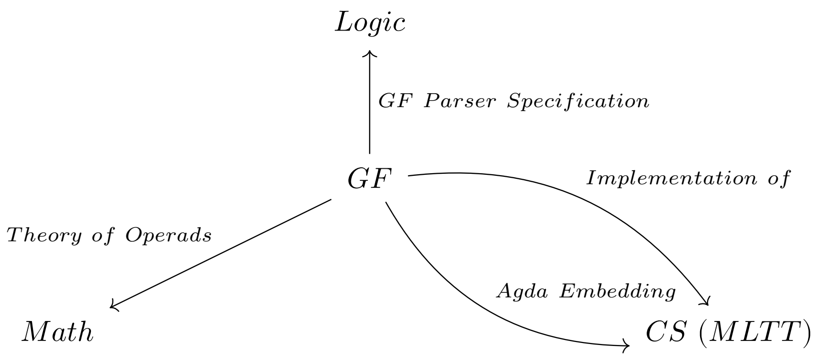

Figure 12: Models of GF

One can additionally model these domains in GF. In Figure 13, we see that there are 3 grammars which allow one to translate in the Trinitarian domains. Ranta's grammar from CADE 2011 [79] built a propositional framework with a core grammar extended with other categories to capture syntactic nuance. Ranta's grammar from the Stockholm University mathematics seminar in 2014 [81] took verbatim text from a publication of Peter Aczel and sought to show that all the syntactic nuance by constructing a grammar capable of NL translation. Finally, our work takes a Backus-Naur Form Converter (BNFC) [3] grammar for the cubicalTT programming language [65], GFifies it, producing an unambiguous grammar [57].

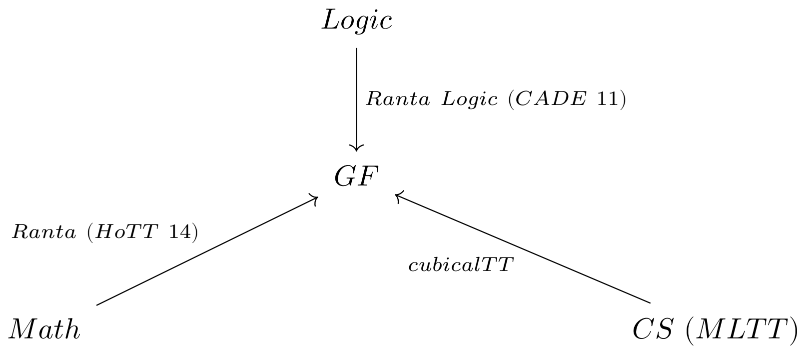

Figure 13: Trinitarian Grammars

While these three grammars offer the most poignant points of comparison between the computational, logical, and mathematical phenomena they attempt to capture, we also note that there were many other smaller grammars developed during the course of this work to supplement and experiment with various ideas presented. Importantly, the "Trinitarian Grammars" do not only model these different domains, but they each do so in a unique way, making compromises and capturing various linguistic and formal language phenomena. The phenomena should be seen on a spectrum of semantic adequacy and syntactic completeness, as in Figure 14.

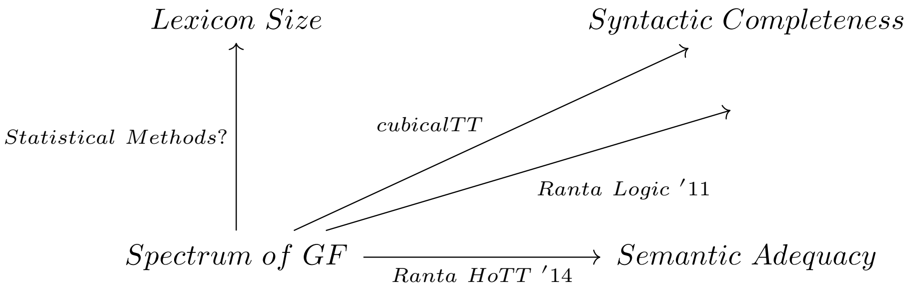

Figure 14: The Grammatical Dimension

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

GF has been show to exist in the PMCFG class of languages [87], between CFGs and context sensitive grammars on the Chomsky Hierarchy [21] Thus, the 'abstract' and 'concrete' coupling is relatively tight, the evaluation is quite simple, and the programs may suggest ways of "writing themselves" once the correct linearization 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 available palette. These constraints allow for fast parsing, but greatly limit the sorts of programs one often thinks of writing.

4.2 GF's Technicalities

GF is a very powerful, yet simple system. GF requires the programmer to work with, in some sense, an incredibly stiff set of constraints compared to general purpose languages, and therefore its lack of expressiveness requires a different way of thinking about programming.

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

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

Both the $\{Strings\}$ and $\{\{ASTs\}\}$ are really parameterized by a grammar G. This property seems somewhat natural from the programmers perspective. The limitation on ASTs to linearize uniquely is actually a benefit, because it saves the user having to make a choice about a translation (although, again, a statistical mechanism could alleviate this constraint). We also want our translations to be well-behaved mathematically, i.e. composing $Linearize$ and $Parse$ ad infinitum should presumably not diverge. Parsing a GF grammar is done in polynomial time, whereby the degree of the polynomial depends on the grammar [5] . It comes equipped with 6 basic judgments:

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

There are two judgments in an abstract file, for categories and named functions defined over those categories, namely `cat` and `fun`. The categories are just (succinct) names. GF dependent types arise as categories which are parameterized over other categories and thereby allow for more fine-grained semantic distinctions. We emphasize 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).

4.2.1 Gödel's T in GF

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

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

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

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

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

So far we have specified how to form expressions : types built out of possibly higher order functions between natural numbers, and expressions built out of variables, λ, application, 0, the successor function, and recursion principle. The variables are kept as a separate syntactic category, and integers, `Int`, are predefined. They allow one to parse numeric expressions. One may then define a functional which takes a function over the natural numbers and returns that function applied to 1 - the AST for this expression is :

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

Dual to the abstract syntax there are parallel judgments when defining a concrete syntax in GF, `lincat` and `lin` corresponding to `cat` and `fun`, respectively. If an 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 (products which support sub-typing and named fields), or tables (coproducts) which can make choices when computing with arbitrarily named parameters. Parameters are naturally isomorphic to the sets of some finite cardinality. The tables are actually derivable from the records and their projections, which is how PGF is defined internally, but they are so fundamental to GF programming and expressiveness that they merit syntactic distincion. The `lin` is a term which matches the type signature of the `fun` with which it shares a name.

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 function are giving correct linearizations, one can now parse and linearize to the abstract syntax tree above the to string "function taking f in the natural numbers to the natural numbers to apply f to 1". This is clearly unnatural for a variety of reasons, but it's an approximation of what a computer scientist might say. Suppose instead, we choose to linearize this same expression to a pidgin expression modeled off Haskell's syntax, "\\ (f : nat -> nat) -> f 1". We should notice the absence of parentheses for application suggest something more subtle is happening with the linearization process, for normally programming languages use fixity declarations to avoid lispy looking code. Here are the linearization functions for our Haskell-like λ-terms:

```
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 GF's standard library, the RGL [77]. We show a few of them below, thereby introducing the final, main GF judgments `param` and `oper` for parameters and operations.

```
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 are data types with nullary constructors - or something isomorphic to them. Operations, on the other hand, encode the logic of GF linearization rules. They are syntactic sugar - they allow one to abstract the function bodies of `lin`s and `lincat`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 use of `variants` is one of the ways to encode multiple linearizations forms for a given tree, so here, for example, we're breaking the key nice property from above.

This more or less resembles a typical programming language, with very little deviation from what when would expect specifying something in Twelf [71]. Nonetheless, because this is both meant to somehow capture the logical form in addition to the surface appearance of a language, the separation of concerns leaves the user with an important decision to make regarding how one couples the linear and abstract syntaxes. There are in some sense two extremes one can take to get a well performing GF grammar.

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

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

We claim syntactically complete grammars are much more naturally dealt with using a simple abstract syntax. However, to take allow a syntactically complete grammar to capture semantic nuance and requires immensely more work on the concrete side. Semantically adequate grammars on the other hand, require significantly more attention on the abstract side, because semantically meaningful expressions often don't generalize - each part of an expressions exhibits unique behaviors which can't be abstracted to apply to other parts of the expression. Semantically complete grammars are vulnerable to over-fitting natural language, making generating formal languages difficult. Producing a syntactically complete expressions which doesn't overgenerate parses also requires a lot work from the grammar writer in this case.

The subsequent examples should illuminate this tension. The problem of merging a syntactically oriented domain like type theory with and a semantically oriented one like natural language mathematics with the same abstract syntax poses very serious problems, but also highlights the power and need of other features of GF, like the RGL and Haskell embedding made available through the PGF API [4].

The GF RGL is a library for parsing grammatically coherent language. It exists for many different natural languages, with various levels of coverage and grammaticality, with a core abstract syntax shared by all of them. The API allows one to easily construct sentence level phrases once the lexicon has been defined. The API also provides helper functions for lexical constructions.

The Haskell embedding of a GF abstract syntax is given via Angelov's PGF library, where the categories are given "shadow types", so that one can transform an abstract syntax into (a possibly massive) Generalized Algebraic Data Type (GADT). The syntax of the embedding is a GADT, `Tree`, with kind `* -> *` where all the functions serve as constructors. If function `h` returns category `c`, the Haskell constructor `Gh` returns `Tree c`. We note that this uses the `--haskell=gadt` flag, of which other options are available but weren't used in this thesis.

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

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

5 Propositions in GF

5.1 CADE 2011

In [79], Ranta designed a grammar which allowed for predicate logic with a domain specific lexicon supporting mathematical theories like geometry or arithmetic. The syntax was both meant to be relatively complete so that typical logical utterances of interest could be accommodated, as well as support relatively non-trivial linguistic nuance. The nuances included lists of terms, predicates, and propositions and in-situ and bounded quantification. The more interesting syntactic details captured in this work was by means of an extended grammar on top of the core. The bidirectional transformation between the core and extended grammars via a Haskell transformation also show the viability and necessity of using more expressive programming languages as intermediaries when doing thorough translations.

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

We see that our criteria of semantic adequacy and syntactic completeness can both occur in the same grammar, with Haskell level transformation allowing one to go between the them. Problematically, this syntactically complete phrase produces four ASTs, with the “or” and “forall” competing for precedence. There is therefore no assurance give the user of the grammar confidence that her phrase was correctly interpreted without deciding which translation is best.

In the opposite direction, the desugaring of a logically “informal” statement into something less linguistically idiomatic is also accomplished. Ranta claims “Finding extended syntax equivalents for core syntax trees is trickier than the opposite direction”. While this may be true for this particular grammar, we argue that this may not hold generally.

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

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

We could therefore transform the extended language phrase “The sum of x , y , and z is equal to itself” in to core language phrase “the sum of the sum of x and y and z is equal to the sum of x and the sum of y and z”. Parsing this core string gives 32 unique trees, and dealing with ambiguities must be solved first and foremost to satisfy the PL designer who only accepts unambiguous parses.

Ranta outlines the mapping, $[\![-]\!] : Core \rightarrow Extended$, which should hypothetically return a set of extended sentences for a more comprehensive grammar.

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

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

While this grammar serves as a precedent for this work generally, we note that the core logic only supports propositions without proofs - it is not a type theory with terms. Additionally, the domain of arithmetic is an important case study, but scaling this grammar (or any other, for that matter) to allow for *semantic adequacy* of real mathematics is still far away, or as Ranta concedes, "it seems that text generation involves undecidable optimization problems that have no ultimate automatic solution."

5.1.1 A Question Answering Example

We wrote a smaller version of the above grammar [58] just focused on propositional logic. It included an added component not just translating between ASTs, but also allowing intermediary computation and of logical propositions and numerical expressions.

After a Haskell evaluation of propositional expressions to their Boolean values, which could possibly in the future be extended to predicate logic, the system allows a question answering system which gave different kinds of answers - the binary valued answer, the most verbose possible answer, and the answer which was deemed the most semantically adequate, `Simple`, `Verbose`, and `Compressed`, respectively. All answers are technically syntactically complete. An example question with the answers follows:

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

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

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

The extended grammar in this case only had lists of propositions and predicates, and so it was much simpler than Ranta's Cade grammar. GF list categories are then transformed into Haskell lists via the embedding, using GF list syntax explicitly is necessary as it is tied to its external behavior as well. The functions of interest for our discussion are:

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

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

The extension adds more complex cases to cover when evaluating propositions, because a normal "propositional evaluator" doesn't have to deal with lists. For the most part, this evaluation is able to just apply Boolean semantics to the *canonical* propositional constructors, like `GNot`. However, one had to dig deeper inside `GIsNumProp` in order to resolve bugs in lists.

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

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

Testing Anecdotes There were many obstructions in engineering this relatively simple example, particularly when it came to writing test cases. The naive way to test with GF is to translate, and the linearization and parsing functions don't give the programmer many degrees of freedom. ASTs are not objects amenable to human intuition, which makes it problematic because understanding their transformations constantly requires parsing and linearizing to see their "behavior". While some work has been done to allow testing of GF grammars for NL applications [52], the domain of formal languages in GF requires a more refined notion of testing because most utterances should be testable relative to some model with well behaved mathematical properties. Debugging something in the pipeline $String \to GADT \to GADT \to String$ for a large scale grammar without a testing methodology for each intermediate state is surely to be avoided.

Unfortunately, there is no published work on using Quickcheck [24] with the PGF library. The bugs in this grammar were discovered via the input and output *appearance* of strings. Often, no string would be returned after a small change, and discovering if the source was in the abstract, concrete, or Haskell embedding was excruciating. In one case, a bug was discovered that was presumed to be from the PGF evaluator, but was then back-traced to Ranta's grammar from which the code had been refactored. The sentence which broke our pipeline from core to extended, "4 is prime , 5 is even and if 6 is odd then 7 is even", would be easily generated (or at least its AST) by Quickcheck.

Theorems and Linguistic Utterances An important observation that was made during this development : that theorems should be the source of inspirations for deciding which PGF transformations should take place. For instance, one could define $odd : \mathbb{N} \to Set$, $prime : \mathbb{N} \to Set$ and prove that $\forall n \in \mathbb{N}.\ n > 2 \times prime\ n \implies odd\ n$. We can use this theorem as a source of translation, and in fact encode a PGF rule that transforms anything of the form "n is prime and n is odd" to "n is prime", subject to the condition that $n \neq 2$. One could then take a whole set of theorems from predicate calculus and encode them as Haskell functions which minify the expressions to something with equivalent meaning. The verbose "if a then b and if a then c", can be more canonically read as "if a then b and c". The application of these theorems as evaluation functions in Haskell could help give our QA example more informative and direct answers.

Reflections on Grammar Refactoring One of the difficulties encountered in this work was reverse engineering Ranta's GF and Haskell code- the large size and declarative nature of a grammar makes it incredibly difficult to isolate individual features one may wish to understand. Significant efforts went into filtering the grammars to understand behaviors of individual components. Careful usage of the GF module system may sometimes allow one to look at "subgrammars", but there is not proper methodology to extract a sub-grammar and therefore it was found that writing a grammar from scratch was often the easiest way to do this. While grammars can be written compositionally, decomposing them is often not a compositional process.

6 Proofs in GF

We now explore the proofs in GF. As a proposed foundational alternative to mathematics, dependent type theories allow types to depend on terms and therefore allow propositions which include terms to be encoded as types.

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

Delaying a more in depth discussion of equality 6.2, we assert that one proves equality in Agda by finding something that is *irrefutably equal* to itself, where the notion of irrefutably gave birth to subject matter of homotopy type theory and cubical type theories, which can both be classified as higher dimensional type theories.

6.1 Natural Numbers Proofs

The most idiomatic kind of proof one would expect are those over the inductively defined natural numbers. In the simple type theory example 4.2.1 we included *types* and *expressions* as distinct syntactic categories, whereby the linearization of a type can't possibly call the linearization of a term. We now experiment with dependently typed programming languages. The big difference in the dependently typed setting is the fact that the recursion principle becomes an induction principle. The types of a sub-expression being evaluated with a recursive call may depend on the values being computing. This means extra work is required in implementing type-checkers for dependent language - they have to deal with a much more sensitive and computationally expensive notion of type. Additionally, a mixture of types and terms creates difficulties in capturing natural language phenomena when trying to distinguish between proposition and proof.

6.1.1 The Associativity of Natural Numbers

We define addition in Agda by recursion on the first argument. Agda has the capacity to always compute the sum of two given natural numbers, via the defining equations, and indeed $2 + 2 = 4$ is irrefutably true.

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

```
2+2=4 : 2 + 2 ≡ 4
2+2=4 = refl
```

We now present the type which encodes the proposition which says 0 plus some number is propositionally equal to that number. Agda is able to compute evidence for this proposition via the definition of addition, and therefore reflexivly know that number is equal to itself. Yet, the novice Agda programmer will run into a `roadblock`: the proposition that any number added to 0 is not definitionally equal to n, i.e. that the defining equations don't give an automatic way of universally validating this fact about the second arguement. This is despite the fact that given any number, like $3 + 0$, Agda can normalize it to 3.

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

3+0=n : 3 + 0 ≡ 3
3+0=n = refl

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

To overcome the `roadblock`, one must use induction, which we show here by pattern matching. We use an auxiliary lemma `ap` which says that all functions are well defined with respect to propositional equality. Then we can simply use `ap` applied the successor function and the induction hypothesis which manifests 5s a simple recursive call. This proof is actually, verabatim, the same as the `associativity-plus` proof - which gives us one perspective that suggests, at least sometimes, types can be even more expressive than programs in Agda.

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

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

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

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

```
fun
  Pi : [Tele] -> Exp -> Exp ;     -- types
  Fun : Exp -> Exp -> Exp ;
  Lam : [Tele] -> Exp -> Exp ;    -- terms
  App : Exp -> Exp -> Exp ;
  TeleC : [Var] -> Exp -> Tele ; --telescope
```

This grammar allows us to prove the above right-identity and associativity laws. Before we look at the natural language proof generated by this code, we first look at an idealized version, which is reproduced from the Software Foundations text [Pierce et al.].

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

While overly pedantic relative to a mathematicians preferred conciseness, this illustrates a proof which is both syntactically complete and semantically adequate. Let’s compare this proof with our Agda reconstruction using the an induction principle.

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

This proof, aligned with with the text so-as to allow for idealized translation, is actually overly complicated and unnessary for the Agda programmer. For the proof state is maintained interactively, the definitional equalities are normalized via the typechecker, and therefore the base case and inductive case can be simplified considerably once the *motive*, the type we are eliminating into, is known [63]. Fortunately, Agda's pattern matching is powerful enough to infer the motive, so that one can generally pay attention to "high level details" generally. We see a "more readable" rewriting below, with the motive given in curly braces :

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

Associativity in GF Finally, taking a "desguared" version of the Agda proof term, as presented in our grammar, we can can reconstruct the lambda term which would, in an ideal world, match the Software Foundations proof.

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

```
of x.
```

This is horrendous. There are a few points which make this proof non-trivial to translate. There is little support for punctuation and proof structure - the indentations were added by hand. The syntactic distinctions have been discharged into a single `Exp` category, and therefore, terms like 0, a noun, and the whole proof term above (multiple sentences) are both compressed into the semantic box. This poses a huge issue for the GF developer wishing to utilize the RGL, whereby these grammatical categories (and therefore linearization types) are distinct, but our abstract syntax offers an incredibly course view of the PL syntax.

This may be overcome by creating many fields in the record `lincat` for `Exp`, one for each syntactic category, i.e. `lincat Exp = { nounField : CN ; sentenceField : S ; ...}`. Then one may match on parameters with functions have `Exp` arguements to ensure that different arguements are compatible with the assigned resource gramamr types. This approach has been used at Digital Grammars for a client looking to produce natural language for a code base, but unfortunately the grammar is not publically avaialable [82]. The more expressions one has the more difficult it becomes to define a suitable linearization scheme and generalizing it to full scale mathematics texts with the myriad syntactic uses of different types of mathematical terms seems intractible. Ganesalignam did invent a different theoretical notion "type" to cover grammartical artificats in textual mathematics, and this may be relevant to invsetigate here.

The application function, which is so common it gets the syntactic distinction of whitespace in programming languages, does not have the same luxury in the natural language setting. This is because the typechecker is responsible for determining if the function is applied to the right number of arguements, and we have chosen a *shallow embedding* in our programming language, whereby `Plus` is a variable name and not a binary function. This can also be reconciled at the concrete level, as was demonstrated with a relatively simple example [56]. Nonetheless, to add this layer of complextiy to the linearization seems unneccessarily difficult, as it would be simpler to resolve this by somehow matching the arguement structure of the agda function to some deeply embedded addition function, `Plus : Exp -> Exp -> Exp`. Ranta, for example, does this in the HoTT grammar 6.3.

Finally, we should point out an *error* : "apply ap to the successor of x" is incorrect. The `successor` is actually an arguement of `ap`, and isn't applied to x directly on the final line. This is because `Suc : Exp -> Exp` was deeply embedded into GF, and the η-expanded version should be substituted to correct for the error (which will make it even more unreadable). Alternatively, one could include all permutation forms of an expression's type signature up to η-equivalence (depending on the number of both implicit and explicit arguements). This could make the grammar both overgenerate and also make it significantly more complex to implement. These are relatively simple obstacles for the PL developer where the desugaring process sends η-equivalent expressions to some normal form, so that the programmer can be somewhat flexible. The lack of the same freedom in natural language, however, creates numerous obstacles for the GF developer. These are often non-trivial to identify and reconcile, especially when one layers the complexity of multiple natural language features covered by the same grammar.

6.2 What is Equality?

> ... the univalence axiom validates the common, but formally unjustified, practice of identifying isomorphic objects. *HoTT Book* [91]

Mathematicians have an intuition for equality, that of an identification between two pieces of information which naturally are be indiscernible. The philosophically inclined might ponder identification generally. We showcase different notions of identifying things in mathematics, logic, and type theory :

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

While there are notions of equality, sameness, or identification outside of these formal domains, we don't dare take a philosophical stab at these notions here. We have discussed judgemental and propositional equality. Judgmental equality is the means of computing, for instance, that $2+2=4$, for there is no way of proving this other than appealing to the definition of addition. Propositional equality, on the other hand, is actually a type. It is defined as follows in Agda, with an accompanying natural language definition from [91] :

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

Definition 3 *The formation rule says that given a type $A : \mathcal{U}$ and two elements $a, b : A$, we can form the type $(a =_A b) : \mathcal{U}$ in the same universe. The basic way to construct an element of $a = b$ is to know that a and b are the same. Thus, the introduction rule is a dependent function*

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

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

The astute reader might ask, what does it mean to "construct an element of $a = b$"? For the mathematician use to thinking in terms of sets $\{a = b \mid a, b \in \mathbb{N}\}$ isn't a well-defined notion. Due to its use of the axiom of extensionality, the set theoretic notion of equality is, no suprise, extensional. This means that sets are identified when they have the same elements, and equality is therefore external to the notion of set. To inhabit a type means to provide evidence for that inhabitation. The reflexivity constructor is therefore a means of providing evidence of an equality.

This evidentiary approach is disctinctly constructive, and is a significant reason why classical and constructive mathematics, especially when treated in an intuitionistic type theory suitable for a programming language implementation, are such different beasts.

While propositional equality is inductively defined as a type, 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 properties : extensional type theory sacrificed decideable type checking, while intensional type theories required extra beauracracy when dealing with equality in proofs. This approach in intensional type theories interpreted types as setoids, leading to “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 Martin-Löf’s decision in 1986 to become a born again intensional type theorist. His intensional dispoition was adopted by those in Göteborg who were implementing Agda’s ancestors, whereas the extensionalists were primarily emanating from Robert Constable’s group at Cornell developing NuPrl [26].

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

6.3 Ranta's HoTT Grammar

In 2014, Ranta gave an unpublished talk at the Stockholm Mathematics Seminar [81]. 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 grammar.

Using a page of text from Peter Aczel's writings goes over a few standard HoTT definitions and theorems, the grammar allows the translation of the latex document in English to the same document in French, and to a pidgin logical language. The central motivation of this grammar was to capture entirely "real" natural language mathematics. Therefore, it isn't reminiscent of the slender abstract syntax the type theorist adores, and sacrificed "syntactic completeness" for "semantic adequacy". The abstract syntax is much larger and very expressive, but is no longer easy to reason about. Additionally, certain design choices feel ad-hoc. Another defect is that this grammar overgenerates when parsing from the pidgin logical language. Producing a unique parse from the PL side would require a significant amount of refactoring. While it is presumably possible to carve a subset of the GF HoTT abstract file to accommodate an Agda program, but one encounters rocks as soon as one begins to dig.

In Figure 15 one can see different syntactic presentations of a notion of *contractability*, that a space is deformable into a single point, or that a Type is actually inhabited by a single unique term. Some rendered latex is compared with the translated pidgin logic code (after refactoring of Ranta's linearization scheme) and an Agda program. We see that it was fairly easy to get the notation for our cubicalTT grammar 6.4. When parsing the logical form, unfortunately, the grammar is incredibly ambiguous.

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

```
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 15: Contractibility

In Figure 16, we show the different syntax presentations of the notion of *equivalence*, which is merely a bijection when restricted to sets. This is of such fundamental importance in mathematics that it merits its own chapter in the HoTT book, but we only showcase one of its many equivalent definitions. We see that the pidgin syntax is stuck with the anaphoric artifact, `fiber` has the type `it : Set` instead of something like `(y : B) : Set`, and the `y` variable is unbound in the `fiber` expression. This may possibly be fixed with a few hours more of tinkering, but creates even more angst if we anticipate trying to translate proofs to Agda.

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 16: Contractibility

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 writing a dual grammar for Agda's concrete syntax. This demonstrates that to design a grammar prioritizing *semantic adequacy* and subsequently trying to incorporate *syntactic completeness* is difficult, and probably not the best grammar design choice.

The next grammar we present, taking an actual programming language parser in BNFC, GFifying it, and trying to use the abstract syntax to model natural language, gives in some sense a dual challenge, where the abstract syntax remains simple as in our dependently typed grammar 6.1.1, but its linearizations may become increasingly complex, especially when generating natural language.

6.4 cubicalTT Grammar

Cubical type theories arose out of the desire to give a complete computational interpretation to HoTT, whereby univalence would become a theorem rather than an axiom [25]. The utility of this is that canonicity, the property of an expression having a irreducible normal form, is satisfied for all expressions - all natural number expressions must evaluate to numerals. Univalence in classical HoTT, by introducing a type without computational behavior, means that the constructivist using Agda will be able to define terms which don't normalize.

Cubical Type Theories originated when looking beyond simplicial models of type theory to cubical categories instead [13], and gave a blueprint for a totally new type theory which natively supports proving functional extensionality, which is a especially important for mathematicians. The ideas from cubical type theories cubical generated a series of proof assistants: Cubical [45], cubicalTT [65], and Cubical Agda [92], as well as other in originating from Robert Constables disciples in the NuPrl tradition [8] [88] [34]. cubicalTT, had an unambiguous BNFC grammar which more or less represents a kernel of Agda with cubical primitives. This final grammar, which we simply denote as cubicalTT, took the actual cubicalTT grammar and GFified the subset which is in the intersection with vanilla Agda. Extending our GF version to include cubical primitives would facilitate the extension of the work to Cubical Agda, and we hope future endeavors will go in this direction. Cubical Agda supports Higher Inductive Types [11] natively and is capable of all types of new constructions not mentioned in the HoTT book. It is also incredibly experimental, with large changes to the standard library constantly underway as can be seen in 8.2.

6.4.1 GFification

Our grammar for vanilla dependent Π-types 6.1.1 was actually a subset of the current cubicalTT abstract syntax. We give a brief sketch of the algorithm to go between a BNFC grammar and a GF grammar. BNFC essentially combines the abstract and concrete syntax, enabling a hierarchy of numbered expressions `ExpN` to minimize use of parentheses. So, given m names and choosing $Name_i$, we take the accompanying BNFC rule :

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

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

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

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

```
--BNFC
Lam. Exp  ::= "\\" [PTele] "->" Exp ;
Fun. Exp1 ::= Exp2 "->" Exp1 ;
-- GF
cat Exp ; PTele ;
fun
  Lam : [PTele] -> Exp -> Exp ;
```

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

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

Most interesting is to observe what GF has but is absent in BNFC, namely, the ability to add records and paremeters into the linearization types generally. One could add unique categories in GF $Exp_1, ..., Exp_n$, but this would clutter the abstract syntax with information which isn't *semantically* relevant. And while the Haskell code generated by BNFC for cubicalTT is sent through a resolver to the *actual* abstract syntax used by the type-checker and evaluator, the fact that it parses the concrete syntax into an appropriate intermediary form is enough for our purposes. The grammar is available at [57].

6.4.2 Difficulties

While cubicalTT gives unique parses a PL, linearizing to a CNL for mathematics was not implemented due to time constraints, and the difficulties already encountered for an even simpler programming language 6.1, namely that types and terms in dependent type theory can be of just about any grammatical category. We list a few examples :

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

In [82], the authors, generating human readable natural language from specifications, used a word type with many different fields for different grammatical categories (with the same grammatical categories sometimes accounting for multiple fields), in addition to symbolic fields. While deemed successful by the client, it would be interesting to apply this methodology to cubicalTT the grammar, and see how it scales once one begins to add more of Agda's capabilities. Their system also involved other components like Haskell transformations, and it is uncertain how these specific approaches would also allow for the generation of more *semantically adequate* mathematical language.

Other issues encountered in this grammar were Agda's pattern matching, whereby arguments are arranged in a matrix, as opposed to explicit cases, or *splits*. We define `equalNat` in cubicalTT as:

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

The problem is that when linearizing a split, one cannot know how many further splits will take place, and so going from this form to the more "readable" Agda code below is outside of GF's linearization capabilities - although a proof of this fact would require advanced mathematical capabilities. One could instead just a new form of declarations in the abstract syntax, but this would require more Haskell overhead to allow for the correct AST transformations.

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

The way lists are dealt with in natural language versus programming languages also present obstacles, because the RGL's support for lists require 2 numbers of categories in the end node whereas our Agda grammar may instead have `cat[1]` or `cat[0]` for the same category. Resolving this overloading of categories for the two linearization spaces will also require Haskell transformations.

6.4.3 More advanced Agda features

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

From the semantic side, however, Agda offers many features which extend just the kernel of the Π, Σ, and recursive data type definitions which form the basis of any dependent type theory. These include universes, sized types, modules, overloading for more ad-hoc polymorphism, proof by reflection, a sort system, higher inductive types (only in Cubical Agda), and many more things visible in the Agda documentation [68]. Additionally, it has more traditional PL features, like the ability to perform side effects or call Haskell functions. Adding any one of these not only adds overhead to the parser, but would require lots of thought in terms of how to these features manifest in natural language for mathematicians (and programmers). Additionally, these features make the metatheory of Agda much more expensive to understand, in addition to the practical implications of introducing bugs in its implementation.

Mathematics on the other hand, doesn't often introduce more advanced "semantic machinery" like those Agda features just listed. Perhaps idioms and conventions change, as well as general machinery like category theory offer ways of presenting ideas more succinctly, but these are merely reflected in the presentation, not in the underlying logical formalism. The linguistic evolution of mathematics additionally reflects some kind of meta-changes, but not in a coherent way that is documented or even understood. For many mathematicians are largely interested in proving theorems and solving problems specific to some domain. The resolution of these meta-ideas from both the type theoretic and mathematical perspectives is what makes this problem of translation so philosophically intriguing, as well as even more intractable.

6.5 Comparing the Grammars

We compare the Ranta's HoTT grammar and our cubicalTT grammar, with a focus on comparing syntactic completeness and semantic adequacy.

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

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

```
cat
  Paragraph ;         -- definition, theorem, etc
  Definition ;        -- definition of a new concept
  Assumption ;        -- assumption in a proof
  [Assumption]{1} ;   -- list of assumptions in one sentence
  Conclusion ;        -- conclusion in a proof
  Prop ;              -- proposition (sentence or formula)
```

```
  Sort ;             -- set, type, etc corresponding to a common noun
  Ind ;              -- individual element, a singular term
  Fun ;              -- function with individual value
  Pred ;             -- predicate: function with proposition value
  [Ind] ;            -- list of individual expressions
  UnivPhrase ;       -- universal noun phrase
  ConclusionPhrase ; -- conclusion word
  Label ;            -- name/number of definition, theorem, etc
  Title ;            -- title for theorem, definition, etc
```

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

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

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

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

Ranta chose to prioritize semantic adequacy by placing the manifold grammatical categories at the forefront. This was not an error, as Peter Azcel's writing mixed notations from set theory, type theory, first order logic, and homotopy theory. For as much as the type theorist insists on her exclusive use of types, the written language tradition is still tied to the logical and set theoretic tradition of presenting mathematics - this results in a more expressive abstract syntax.

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

Another observation about Ranta's grammar is that the certain notions come with more semantic information that the type-checker would be able to infer, so for instance, `fiberExp` is a binary function, as opposed to the cubicalTT grammar which treats it as a variable. This we may remember, leads to the "application hell" observed earlier 6.1.1.

Despite the complexity of the abstract syntax relative to cubicalTT, it is remarkable that Ranta was able to capture the entire text with a few days of labor. Expertise in GF, however, reveals itself through trial, error, and patience. Despite the success, we hypothesize that extending it to longer lengths of text would very difficult for anyone without deep knowledge of GF and type theory generally. The ease of extending cubicalTT to cover more text, despite its limitations regarding language generation, poses a dual problem of how to extending the concrete syntax each time a new grammatical "feature" is discovered. We have included the text parsed by Ranta's HoTT grammar implemented both an Agda representation which typechecks, as well as the cubicalTT syntax for these terms, in the appendix 8.4.

6.5.1 Ideas for resolution

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

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

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

Let's suppose that our hypothetical "core" should consist of a desugared type theory with Π, Σ, and Equality types, with their respective introduction and elimination forms, inductive definitions and a means of case analysis, and declarations for building types and terms. We could then *extend* this with telescopes for syntactic sugar, "where" and "let" bindings to allow for local definitions, and modules to allow for the basic needs of a suitable programming language - and we'd essentially have the cubicalTT grammar. One thing to be emphasized is that the extension should already map to the core. As was noted when we discussed Haskell transformations for Ranta's logic grammar 5.1, the mapping $[\![-]\!] : Extended \rightarrow Core$ can follow relatively conventional techniques.

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

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

The "in-between" depth is to include `Nat` as an expression, whereby the zero, successor, and induction principle, included as functions, retain their arities from the actual programming language, but don't actually specify what types of expressions work for them - this work is delegated to the type-checker. While this has the benefit of disallowing the "application hell" we saw in 6.1.1, it also requires what we'll passively call "arity inference", and therefore some components of the type-checker would be needed to scale this approach. Additionally, the use of the phrase "successor function" to refer to η-expanded form in contrast to "the successor of" reveals the deep difficulties of how to delegate unique linguistic forms to all the possible arity assignments, something that a programming language can infer automatically based of a term's use.

Once the grammar has been extended *enough*, satisfying the programmer's needs, one would encounter the even more difficult task of pleasing the mathematician. One could have categories for things like sets and propositions, as Ranta does in the HoTT grammar. These extensions of the $\{\Pi, \Sigma, \equiv, ...\}$ core, if given a Haskell embedding, could be made such that any proposition in the extended langauge would be mapped to its type in the core language. In Ranta's code, the existential propositions and set comprehension syntax could be evaluated, via Haskell, to the

Σ type in the core grammar, thereby allowing for a translation from semantically sound utterances to syntactically complete ones.

We offer a conterexample, namely the definition of a left coset in group theory, $gH = \{gh{:}h \in H\}\ \forall g \in G$, because $h \in H$ is a judgment and not a type. Indeed, quotients, subsets, and subgroups in type theory must be treated differently than their set-theoretic counterparts. The sets must be given clever encodings which don't precisely match their set theoretic intuition, especially with respect to syntax [96]. Additionally, in the reverse direction, taking a Σ type and generating a natural language utterance which may be of the type, set, or one the many propositional flavors would require some pragmatic knowledge that our system is not capable of handling.

Our proposal is to build a core, *syntactically complete* grammar similar to cubicalTT and extend it to a *semantically adequate* which follows Ranta's HoTT grammar methodology. It should be based of the condition that they are coherent in a sense that the extended grammar can be compositionally evaluated to the core, via a Haskell function, following Ranta's lead in his Cade grammar.

Typechecker's Role One of the central pieces of a programming language missing from this approach is the role of the type-checker. In dependently typed languages like Agda, where the type-checker evaluates programs in types to a canonical form, this is especially acute - for the typechecker tells you when a proof is *syntactically valid*. For the mathematician, a proof may be valid even if it doesn't type-check, because they can account for many details which ensure that an argument is articulated honestly, or at least as honestly as possible. While there may be small errors, presumptions, or holes in a syntactic proof, this ultimately doesn't detract from the *semantic ideas* being portrayed and perceived.

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

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

7 Conclusion

> Concrete syntax is in some sense where programming language theory meets psychology. *Robert Harper, Oregon Programming Languages Summer School 2017*

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

The first is how to make a dependently typed programming language, capable of formulating proposition and proof, more amenable to mathematicians with the goal of improving semantic adequacy. The second asks how to facilitate the formalization of mathematics, specifically the translation of theorem statements and proofs to controlled natural languages, so-as to slightly disambiguate the many ambiguities which pervade mathematical language and practice.

Progress in either of these directions will only be realized and through significant time, labor, expertise, and most vitally, *original thinking* through collaborative efforts. It is uncertain what the role of parsers, abstract syntax trees, linearization schemes, and other components of the GF ecosystem will have on these efforts, but we have offered a taste of how these tools may be applied..

Our work has perhaps only made a small contribution to these incredibly difficult problems. Compiling various ideas across many different fields has given some philosophical clarifications as to why the problems are so difficult.

Additionally, through the analysis and comparison different GF grammars gives has provided evidence that there's a feasibility of actually applying these ideas to solve real problems. There is doubtless a role to play for statistical methods in tackling these problems as well, and how these data-based methods interface with the rule-based techniques is up to future scholarship.

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

- Introduce notions of *syntactic completeness* and *semantic adequacy*, so-as to allow for understanding a piece of mathematics on a spectrum of formality and clarity
- Offer explicit comparisons, through examples of mathematics in a textual form and a type theoretic presentation
- The developments of new GF grammars for analyzing this problem
- The first comparison of all known GF grammars in this domain with respect to *syntactic completeness* and *semantic adequacy*
- The development of an Agda library which mirrors the HoTT book so that future work can seek a possible "large-scale" translation case study
- Recognition that the GF approach is limited, especially as regards pragmatic concerns, but that it still provides insights here as well

It has been remarked that the bigger grammars gets, the more it begins to resemble a domain specific resource grammar [6]. We advocate to actually produce a "formal language RGL", whereby many of the ideas observed in this work, like document structure, latex (and symbolic support generally), custom lexical classes (like in BNFC), and many more may be accounted for. The future grammar writers' time could be spent either focusing on the scaling of programming language features or the actual linguistic analysis of mathematics text - thereby making a more natural CNL for mathematics.

Despite the promise of various topics discussed here like Cubical Agda, the Formal Abstracts Project , and the use of ITPs in mathematics education [19], we don't foresee a convergence of type theorists and mathematicians, even though devotion to the holy trinity would compel us to believe so. GF as a programming language paradigm applied to this problem gives us a stark contrast of how different these two approaches to mathematical language. For the grammars of proof are insufficient to capture the complexity and nuance about the language of proof, so much of which has yet to captured in an existing linguistic framework. Grammars for propositions and definitions offer a much more limited and seemingly feasible solution, because mathematicians make these utterances with the explicit intention of being comprehensible and unambiguous.

7.1 The Mathematical Library of Babel

The Library of Babel [14] is a profound mirror held up to the human species as regards our comprehension of the world through language. It reflects our inability to grasp and reconcile human finitude. The infinite stack of shelves, containing every book with every permutation of letters from the Hebrew alphabet, leaves the humans who inhabit the closed space in a state of discontent as regards their failures to navigate and interpret the myriad texts.

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

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

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

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

b

References

[1] (1994). The qed manifesto. In *Proceedings of the 12th International Conference on Automated Deduction*, CADE-12 (pp. 238–251). Berlin, Heidelberg: Springer-Verlag.

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

[3] Abel, A. et al. (2002). BNFC. `https://github.com/bnfc/bnfc`.

[4] Angelov, K. (2010). PGF. `https://hackage.haskell.org/package/gf-3.11.0/docs/PGF.html`.

[5] Angelov, K. (2011). *The Mechanics of the Grammatical Framework*. PhD thesis, Chalmers University of Technology.

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

[7] Angelov, K., Bringert, B., & Ranta, A. (2010). PGF: A Portable Run-time Format for Type-theoretical Grammars. *Journal of Logic, Language and Information*, 19(2), 201–228.

[8] Angiuli, C., Cavallo, E., Hou (Favonia), K.-B., Harper, R., & Sterling, J. (2018). The redprl proof assistant (invited paper). *Electronic Proceedings in Theoretical Computer Science*, 274, 1–10.

[9] Avigad, J. (2015). Mathematics and language. In *Mathematics, Substance and Surmise* (pp. 235–255). Springer.

[10] Bar, K., Kissinger, A., & Vicary, J. (2016). Globular: An online proof assistant for higher-dimensional rewriting. In *FSCD*.

[11] Basold, H., Geuvers, H., & van der Weide, N. (2017). Higher Inductive Types in Programming. *J. Univ. Comp. Sci.*, 23(1), 63–88.

[12] Bernardy, J.-P. & Chatzikyriakidis, S. (2017). A type-theoretical system for the FraCaS test suite: Grammatical framework meets coq. In *IWCS 2017 - 12th International Conference on Computational Semantics - Long papers*.

[13] Bezem, M., Coquand, T., & Huber, S. (2019). The Univalence Axiom in Cubical Sets. *J. Autom. Reason.*, 63(2), 159–171.

[14] Borges, J. L. (1998). *The Library of Babel*. Penguin New York.

[15] 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.

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

[17] 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.

[18] Bruijn, de, N. (1970). The mathematical language automath, its usage and some of its extensions. In M. Laudet, D. Lacombe, L. Nolin, & M. Schützenberger (Eds.), *Proceedings Symposium on Automatic Demonstration (Versailles, France, December 1968)*, Lecture Notes in Mathematics (pp. 29–61). Germany: Springer.

[19] Buzzard, K. (2020). When will computers prove theorems? `https://www.ma.imperial.ac.uk/~buzzard/xena/computers.pdf`.

[20] Castellan, S., Clairambault, P., & Dybjer, P. (2021). Categories with families: Unityped, simply typed, and dependently typed. In *Joachim Lambek: The Interplay of Mathematics, Logic, and Linguistics* (pp. 135–180). Springer.

[21] Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, 2(3), 113–124.

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

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

[24] Claessen, K. & Hughes, J. (2000). QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.*, 35(9), 268–279.

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

[26] Constable, R. L., Allen, S. F., Bromley, H. M., Cleaveland, W. R., Cremer, J. F., Harper, R. W., Howe, D. J., Knoblock, T. B., Mendler, N. P., Panangaden, P., Sasaki, J. T., & Smith, S. F. (1986). *Implementing Mathematics with the Nuprl Proof Development System*. USA: Prentice-Hall, Inc.

[27] Coquand, T. (1992). Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs* (pp. 71–83).: Citeseer.

[28] 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.

[29] 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.

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

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

[32] Dybjer, P. & Palmgren, E. (2020). Intuitionistic Type Theory. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2020 edition.

[33] Escardó, M. H. (2020). Introduction to Univalent Foundations of Mathematics with Agda. `https://www.cs.bham.ac.uk/~mhe/HoTT-UF-in-Agda-Lecture-Notes/index.html`.

[34] Favonia et al. (2020). coolTT. `https://github.com/redprl/cooltt`.

[35] Fong, B. (2016). *The Algebra of Open and Interconnected Systems*. PhD thesis, Oxford University.

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

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

[38] Gabbay, M. J. & Pitts, A. M. (2002). A new approach to abstract syntax with variable binding. *Formal aspects of computing*, 13(3), 341–363.

[39] Ganesalingam, M. (2013). *The Language of Mathematics*. Springer Berlin Heidelberg.

[40] 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.

[41] Hales, T. (2019). An Argument for Controlled Natural Languages in Mathematics. `https://jiggerwit.wordpress.com/2019/06/20/an-argument-for-controlled-natural-languages-in-mathematics/`.

[42] 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.

[43] Harper, R. (2011). The Holy Trinity. `https://existentialtype.wordpress.com/2011/03/27/the-holy-trinity/`.

[44] Hermida, C., Makkai, M., & Power, J. (1998). Higher dimensional multigraphs. In *Proceedings. Thirteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.98CB36226)* (pp. 199–206).

[45] Huber, S. et al. (2014). Cubical. `https://github.com/simhu/cubical`.

[46] Jackson, A. (2002). Communications-the world of blind mathematicians. *Notices of the American Mathematical Society*, 49(10), 1246–1251.

[47] 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).

[48] 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.

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

[50] Kohlhase, M. & Schaefer, J. F. (2019). GF + MMT = GLF – From Language to Semantics through LF. *Electronic Proceedings in Theoretical Computer Science*, 307, 24–39.

[51] Lambek, J., of Mathematics, M. U. D., & Statistics (1989). *Multicategories Revisited*. Rapport du Department of Mathematics and Statistics. McGill University, Department of Mathematics and Statistics.

[52] Listenmaa, I. (2019). *Formal Methods for Testing Grammars*. PhD thesis, University of Gothenburg.

[53] Ljunglöf, P. (2004). Expressivity and complexity of the Grammatical Framework.

[54] 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.

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

[56] Macmillan, W. (2021a). Application Precdences Grammar. `https://github.com/wmacmil/GF-Application-Precedences/tree/master`.

[57] Macmillan, W. (2021b). cubicalTT. `https://github.com/wmacmil/AgdaGrammar`.

[58] Macmillan, W. (2021c). QA Grammar. `https://github.com/wmacmil/AgdaGrammar/tree/main/pgfTests`.

[59] Martin-Löf, P. (1982). Constructive mathematics and computer programming. In L. J. Cohen, J. Łoś, H. Pfeiffer, & K.-P. Podewski (Eds.), *Logic, Methodology and Philosophy of Science VI, Proceedings of the Sixth International Congress of Logic, Methodology and Philosophy of Science, Hannover 1979*, volume 104 of *Studies in Logic and the Foundations of Mathematics* (pp. 153–175). North-Holland.

[60] Martin-Löf, P. (1984). *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980.

[61] 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.

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

[63] McBride, C. (2002). Elimination with a motive. In P. Callaghan, Z. Luo, J. McKinna, R. Pollack, & R. Pollack (Eds.), *Types for Proofs and Programs* (pp. 197–216). Berlin, Heidelberg: Springer Berlin Heidelberg.

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

[65] Mörtberg, A. et al. (2018). *cubicaltt*. `https://github.com/mortberg/cubicaltt`.

[66] Nelsen, R. B. (1993). *Proofs without words: Exercises in visual thinking*. Number 1. MAA.

[67] Nipkow, T., Wenzel, M., & Paulson, L. C. (2002). *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Berlin, Heidelberg: Springer-Verlag.

[68] Norell, U. et al. (2005). Agda Docs. `https://agda.readthedocs.io/en/v2.6.2/team.html`.

[69] Pédrot, P.-M. (2019). Ltac2: tactical warfare. In *The Fifth International Workshop on Coq for Programming Languages, CoqPL* (pp. 13–19).

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

[71] Pfenning, F. & Schürmann, C. (1999). System description: Twelf — a meta-logical framework for deductive systems. In *Automated Deduction — CADE-16* (pp. 202–206). Berlin, Heidelberg: Springer Berlin Heidelberg.

[Pierce et al.] Pierce, B. C., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., & Yorgey, B. Software foundations. `https://github.com/wmacmil/AgdaGrammar/tree/main/pgfTests`.

[73] 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.

[74] Ranta, A. (1994a). *Type Theoretical Grammar*. Oxford University Press.

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

[76] Ranta, A. (2004). Grammatical Framework. *Journal of Functional Programming*, 14(2), 145–189.

[77] Ranta, A. (2009). The GF Resource Grammar Library. *Linguistics in Language Technology*, 2.

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

[79] 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.

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

[81] Ranta, A. (2014). Translating Homotopy Type Theory in Grammatical Framework . https://github.com/GrammaticalFramework/gf-contrib/tree/master/homotopy-typetheory. Stockholm University, Department of Mathematics.

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

[83] 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.

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

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

[86] Schaefer, J. F. & Kohlhase, M. (2020). The GLIF System: A Framework for Inference-Based Natural-Language Understanding.

[87] Seki, H., Matsumura, T., Fujii, M., & Kasami, T. (1991). On Multiple Context-Free Grammars. *Theoretical Computer Science*, 88(2), 191–229.

[88] Sterling, J. et al. (2018). redTT. https://github.com/redprl/redtt.

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

[90] Szegedy, C. (2020). A promising path towards autoformalization and general artificial intelligence. In C. Benzmüller & B. Miller (Eds.), *Intelligent Computer Mathematics* (pp. 3–20). Cham: Springer International Publishing.

[91] Univalent Foundations Program, T. (2013). *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book.

[92] Vezzosi, A., Mörtberg, A., & Abel, A. (2019). Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *Proc. ACM Program. Lang.*, 3(ICFP).

[93] Wadler, P., Kokke, W., & Siek, J. G. (2020). Programming language foundations in Agda. http://plfa.inf.ed.ac.uk/20.07/.

[94] 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.

[95] Wiedijk, F. (2006). *The Seventeen Provers of the World: Foreword by Dana S. Scott (Lecture Notes in Computer Science / Lecture Notes in Artificial Intelligence)*. Berlin, Heidelberg: Springer-Verlag.

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

8 Appendices

8.1 Martin-Löf Type Theory

8.1.1 Judgments

> With Kant, something important happened, namely, that the term judgement, Ger. Urteil, came to be used instead of proposition. *Per Martin-Löf* [61].

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 feel it's tautological to claim 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. They would, 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 :

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

8.1.2 Rules

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

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

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`.

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

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

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

As an addendum to this brief overview, we also mention that substitution and variable binding are an incredibly delicate and important aspect of type theory, especially from an implementor's perspective. This is because internally, de Brujn's indices for variable binding are much easier to reason about and ensure correctness, despite the syntactically more intuitive classical ways of treating variables when writing actual programs. The theory of nominal sets [38] and interpretation of variables in the categorical semantics of type theories [20] are some of the many compelling research areas which have arisen due to interest in variables in type theories.

We recommend reading Martin-Löf's original papers [60] [59] to see all the rules elaborated in full detail, as well as his philosophical papers [61] [62] to understand type theory as it was conceived both practically and philosophically.

8.2 What is a Homomorphism?

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

Definition 4 *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 5 *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 6 *Let G, H, be groups. A map $\phi : G \to H$ is called a* group homomorphism *if*

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

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

Definition 7 *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 wants to distill the meaning of each of these presentations, there is a significant amount of subliminal interpretation happening very much analogous to our innate linguistic usage. The inverse and identity are discarded, even though they are necessary data when defining a group. The order of presenting the information is inconsistent, as well as the choice to use symbolic or natural language information. In Definition 6, the group operation is used implicitly, and its clarification a side remark. Details aside, these all mean the same thing - don't they?

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, distinguising between 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 differences via changes in camel case, new uses of Group vs GroupStr, as well as, most significantly, the identity and inverse preservation data not appearing as corollaries, but part of the definition. Additionally, we had to refactor the commented lines to those shown below to be compatible with our outdated version of cubical. These changes reflect interesting syntactic changes.

```
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 be somewhat compressible to a programmer or mathematician not exposed to Agda, it is certainly comprehensible to a computer : that is, the colors indicate it type-checks on a computer where Cubical Agda is installed. While GF is designed for multilingual syntactic transformations and is targeted for natural language translation, its 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! While the Agda definitions are present another concrete presentation of a group homomorphism, they are distinct in that they have inherent semantic content.

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.

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 7 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.

8.3 Twin Primes Conjecture Revisited

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

```
prime = Σ[ p ∈ ℕ ] isPrime p

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

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

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

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

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

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

8.4 Hott and cubicalTT Grammars

```
id : A → A
id = λ z → z

iscontr : (A : Set) → Set
```

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

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

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

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

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

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

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

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

Compared with the latex code

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

$$(x, z) = y$$

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

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

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

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

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

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

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

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

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

Figure 17: Mathematical Assertions and Agda Judgements

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

What we notice :

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

also, refactor to have the sigma proof here

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

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

```
3 PropParagraph
    * NoConclusionPhrase
      ForAllProp
        * if_thenUnivPhrase
            * BaseVar
                * U_Var
              type_universe_Sort
          ForAllProp
            * plainUnivPhrase
                * ConsVar
                    * X_Var
                      BaseVar
                        * Y_Var
                  ExpSort
                    * VarExp
                        * U_Var
              LabelledExpProp
                * DisplayMathEnv
                  StarLabel
                  mapExp
                    * equalExp
                        * VarExp
                            * X_Var
                          VarExp
                            * Y_Var
                      equivalenceExp
                        * VarExp
                            * X_Var
                          VarExp
                            * Y_Var
4 ConclusionParagraph
    1 NoConclusionPhrase
      ApplyLabelConclusion
        * the_lemma_Label
          BaseInd
          ForAllProp
            * plainUnivPhrase
                * BaseVar
                    * X_Var
                  ExpSort
                    * VarExp
                        * U_Var
              ExpProp
                * DollarMathEnv
                  equivalenceExp
                    * VarExp
                        * X_Var
                      VarExp
                        * X_Var
    2 henceConclusionPhrase
      ApplyLabelConclusion
        * id_induction_Label
          ConsInd
            * FunInd
                * ExpFun
                    * TypedExp
                        * ConsExp
```

IdIsEquiv Var * X

8.5 HoTT Agda Corpus

```
module Id where

open import Agda.Builtin.Sigma public
open import Data.Product

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

infix 20 _≡_

-- (2.0.1)
J : {A : Set}
     → (D : (x y : A) → (x ≡ y) →   Set)
     -- → (d : (a : A) → (D a a r ))
     → ((a : A) → (D a a r ))
     → (x y : A)
     → (p : x ≡ y)
     ------------------------------------
     → D x y p
J D d x .x r = d x

-- Lemma 2.1.1
_⁻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

-- Lemma 2.1.2
_•_ : {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 _•_

-- Lemma 2.1.4 (i)_1
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

-- Lemma 2.1.4 (i)_2
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

-- Lemma 2.1.4 (ii)_1
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

-- Lemma 2.1.4 (ii)_2
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

-- Lemma 2.1.4 (iii)
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

-- Lemma 2.1.4 (iv)
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'
    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′

module Eckmann-Hilton where
  -- Lemma 2.1.6
  -- 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

  _•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)

  -- Definition 2.1.8
  -- loop space
  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
  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 • i
  lem1′ a α β = r

  -- Lemma 2.2.1
  -- first proof
  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

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

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

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

  _⋆≡⋆'_ : {A : Set} → {a b c : A} {p q : a ≡ b} {r' s : b ≡ c} (α : p ≡ q) (β : r' ≡ s) → (α
  _⋆≡⋆'_ {A} {a} {b} {c} {p} {q} {r'} {s} α β = J D d p q α c r' s β
    where
      D : (p q : a ≡ b) → p ≡ q → Set
      D p q α = (c : A) (r' s : b ≡ c) (β : r' ≡ s) → (α ⋆ β) ≡ (α ⋆' β)
      E : (r' s : b ≡ c) → r' ≡ s → Set
      E r' s β = (_⋆_ {A} {b = b} {c} {r} {r} {r' = r'} {s = s} r β) ≡ (r ⋆' β)
      e : ((s : b ≡ c) → E s s r)
      e r = r
      d : ((p : a ≡ b) → D p p r)
      d r a r r r = r -- book uses J

  -- cheating, not using the same arguement as the book
  eckmannHilton : {A : Set} → (a : A) → (α β : Ω² {A} a) → α • β ≡ β • α
  eckmannHilton a r r = r

open Eckmann-Hilton

-- Lemma 2.2.2 (i)
apfHom : {A B : Set} {x y z : A} (p : x ≡ y) (f : A → B) (q : y ≡ z) → apf f (p • q) ≡ (apf f p
apfHom {A} {B} {x} {y} {z} p f q = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = {f : A → B} {q : y ≡ z} → apf f (p • q) ≡ (apf f p) • (apf f q)
    d : (x : A) → D x x r
    d x = r

-- Lemma 2.2.2 (ii)
apfInv : {A B : Set} {x y : A} (p : x ≡ y) (f : A → B) → apf f (p ⁻¹) ≡ (apf f p) ⁻¹
```

```
apfInv {A} {B} {x} {y} p f = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = {f : A → B} → apf f (p ⁻¹) ≡ (apf f p) ⁻¹
    d : (x : A) → D x x r
    d x = r

-- compostion, not defined in hott book
infixl 40 _∘_
_∘_ : {A B C : Set} → (B → C) → (A → B) → (A → C)
(g ∘ f) x = g (f x)

-- Lemma 2.2.2 (iii)
apfComp : {A B C : Set} {x y : A} (p : x ≡ y) (f : A → B) (g : B → C) → apf g (apf f p) ≡ ap
apfComp {A} {B} {C} {x} {y} p f g = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = {f : A → B} {g : B → C} → apf g (apf f p) ≡ apf (g ∘ f) p
    d : (x : A) → D x x r
    d x = r

-- not defined explicitly, different from Id_A
id : {A : Set} → A → A
id = λ z → z

-- apfId : {A B : Set} {x y : A} (p : x ≡ y) (f : _≡_ {A}) → apf f p ≡ p

-- Lemma 2.2.2 (iv)
apfId : {A : Set} {x y : A} (p : x ≡ y) → apf id p ≡ p
apfId {A} {x} {y} p = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = apf id p ≡ p
    d : (x : A) → D x x r
    d = λ x → r

-- Lemma 2.3.1
transport : ∀ {A : Set} {P : A → Set} {x y : A} (p : x ≡ y) → P x → P y
transport {A} {P} {x} {y} = J D d x y
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = P x → P y
    d : (x : A) → D x x r
    d = λ x → id

p* : {A : Set} {P : A → Set} {x : A} {y : A} {p : x ≡ y} → P x → P y
p* {P = P} {p = p} u = transport p u

_* : {A : Set} {P : A → Set} {x : A} {y : A} (p : x ≡ y) → P x → P y
(p *) u = transport p u
```

```
-- Lemma 2.3.2
lift : {A : Set} {P : A → Set} {x y : A} (u : P x) (p : x ≡ y) → (x , u) ≡ (y , p* {P = P} {p =
lift {P} u r = r --could use J, but we'll skip the effort for now


-- Lemma 2.3.4
    -- the type inference needs p below
apd : {A : Set} {P : A → Set} (f : (x : A) → P x) {x y : A} {p : x ≡ y}
  → p* {P = P} {p = p} (f x) ≡ f y
apd {A} {P} f {x} {y} {p} = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = p* {P = P} {p = p} (f x) ≡ f y
    d : (x : A) → D x x r
    d = λ x → r



-- Lemma 2.3.5
transportconst : {A B : Set} {x y : A} {p : x ≡ y} (b : B) → transport {P = λ _ → B} p b ≡
transportconst {A} {B} {x} {y} {p} b = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = transport {P = λ _ → B} p b ≡ b
    d : (x : A) → D x x r
    d = λ x → r

-- missing 2.3.8

-- Lemma 2.3.9
twothreenine : {A : Set} {P : A → Set} {x y z : A} (p : x ≡ y) (q : y ≡ z ) {u : P x} → ((q *)
twothreenine r r = r

-- Lemma 2.3.10
twothreeten : {A B : Set} {f : A → B} {P : B → Set} {x y : A} (p : x ≡ y) {u : P (f x) } → tr
twothreeten r = r

-- Lemma 2.3.11
twothreeeleven : {A : Set} {P Q : A → Set} {f : (x : A) → P x → Q x} {x y : A} (p : x ≡ y) {
twothreeeleven r = r

-- 2.4

infixl 20 _~_

-- Lemma 2.4.1
_~_ : {A : Set} {P : A → Set} (f g : (x : A) → P x) → Set
f ~ g = (x : _) → f x ≡ g x

-- Lemma 2.4.2 (i)
refl~ : {A : Set} {P : A → Set} → ((f : (x : A) → P x) → f ~ f)
refl~ f x = r
```

```
-- Lemma 2.4.2 (ii)
sym∼ : {A : Set} {P : A → Set} → (f g : (x : A) → P x) → f ∼ g → g ∼ f
sym∼ f g fg = λ x → fg x ⁻¹

-- Lemma 2.4.2 (iii)
trans∼ : {A : Set} {P : A → Set} → (f g h : (x : A) → P x) → f ∼ g → g ∼ h → f ∼ h
trans∼ f g h fg gh = λ x → (fg x) ∙ (gh x)

-- transrightidentity, note not defitionally equal
translemma : {A : Set} {x y : A} (p : x ≡ y) → p ∙ r ≡ p
translemma r = r

-- first use of implicit non-definitional equality (oudside of the eckmann h
-- Lemma 2.4.3
hmtpyNatural : {A B : Set} {f g : A → B} {x y : A} (p : x ≡ y) → ((H : f ∼ g) → H x ∙ apf g
hmtpyNatural {x = x} r H = translemma (H x)

-- syntactic sugar for equational reasoning, borrowed from Wadler's presenta
module ≡-Reasoning {A : Set} where

  infix 1 begin_
  infixr 2 _≡⟨⟩_ _≡⟨_⟩_
  infix 3 _∎

  begin_ : ∀ {x y : A}
    → x ≡ y
    -----
    → x ≡ y
  begin x≡y = x≡y

  _≡⟨⟩_ : ∀ (x : A) {y : A}
    → x ≡ y
    -----
    → x ≡ y
  x ≡⟨⟩ x≡y = x≡y

  _≡⟨_⟩_ : ∀ (x : A) {y z : A}
    → x ≡ y
    → y ≡ z
    -----
    → x ≡ z
  x ≡⟨ x≡y ⟩ y≡z = x≡y ∙ y≡z

  _∎ : ∀ (x : A)
    -----
    → x ≡ x
  x ∎ = r

open ≡-Reasoning

-- Corollary 2.4.4
```

```
coroll : {A B : Set} {f : A → A} {x y : A} (p : x ≡ y) → ((H : f ∼ (id {A})) → H (f x) ≡ apf f (
coroll {A} {f = f} {x = x} p H =
  begin
    H (f x)
  ≡⟨ translemma (H (f x)) ⁻¹ ⟩
    H (f x) • r
  ≡⟨ apf (λ - → H (f x) • -) ll51 ⟩
    H (f x) • (apf (λ z → z) (H x) • H x ⁻¹ )
  ≡⟨ associativity (H (f x)) (apf (λ z → z) (H x)) ((H x ⁻¹)) ⟩
    H (f x) • apf (λ z → z) (H x) • H x ⁻¹
  ≡⟨ whisk ⟩
    apf f (H x) • H (x) • H x ⁻¹
  ≡⟨ associativity (apf f (H x)) (H (x)) (H x ⁻¹) ⁻¹ ⟩
    apf f (H x) • (H (x) • H x ⁻¹)
  ≡⟨ apf (λ - → apf f (H x) • -) locallem ⟩
    apf f (H x) • r
  ≡⟨ translemma (apf f (H x)) ⟩
    apf f (H x) ∎
  where
    thatis : H (f x) • apf (λ z → z) (H x) ≡ apf f (H x) • H (x)
    thatis = hmtpyNatural (H x) H
    whisk : H (f x) • apf (λ z → z) (H x) • H x ⁻¹ ≡ apf f (H x) • H (x) • H x ⁻¹
    whisk = thatis •ᵣ (H x ⁻¹)
    locallem :  H x • H x ⁻¹ ≡ r
    locallem = rightInverse (H x)
    ll51 : r ≡ apf (λ z → z) (H x) • H x ⁻¹
    ll51 = locallem ⁻¹ • (apf (λ - → - • H x ⁻¹) (apfId (H x))) ⁻¹

-- Definition 2.4.6
qinv : {A B : Set} → (f : A → B) → Set
qinv {A} {B} f = Σ (B → A) λ g → (f ∘ g ∼ id {B}) × (g ∘ f ∼ id {A})

-- Example 2.4.7
qinvid : {A : Set} → qinv {A} {A} id
qinvid = id , (λ x → r) , λ x → r

-- syntactic sugar, is redundant
p• : {A : Set} {x y z : A} (p : x ≡ y) → ((y ≡ z) → (x ≡ z))
p• p = λ - → p • -

-- Example 2.4.8
qinvcomp : {A : Set} {x y z : A} (p : x ≡ y) → qinv (p• {A} {x} {y} {z} p)
qinvcomp p = (λ - → p ⁻¹ • -) , sec , retr
  where
    sec : (λ x → p• p (p ⁻¹ • x)) ∼ (λ z → z)
    sec x =
      begin
        p• p (p ⁻¹ • x)
      ≡⟨ associativity p (p ⁻¹) x ⟩
        (p • p ⁻¹) • x
```

```
      ≡⟨ apf (λ - → - • x) (rightInverse p) ⟩
        r • x
      ≡⟨ iₗ x ⁻¹ ⟩
        x ∎
    retr : (λ x → p ⁻¹ • p• p x) ∼ (λ z → z)
    retr x =
      begin
        p ⁻¹ • p• p x
      ≡⟨ associativity (p ⁻¹) p x ⟩
        (p ⁻¹ • p) • x
      ≡⟨ apf (λ - → - • x) (leftInverse p) ⟩
        x ∎

-- Example 2.4.9
qinvtransp : {A : Set} {P : A → Set} {x y : A} (p : x ≡ y) → qinv (transport {P = P} p)
qinvtransp {A} {P} {x} {y} p = transport (p ⁻¹) , sec , retr p
  where
    sec' : {A : Set} {P : A → Set} {x y : A} (p : x ≡ y) → (λ x₁ → transport {P = P} p (tran
    sec' r x = r
    sec : (λ x₁ → transport p (transport (p ⁻¹) x₁)) ∼ (λ z → z)
    sec z = sec' p z
    retr : (p : x ≡ y) → (λ x₁ → transport (p ⁻¹) (transport p x₁)) ∼ (λ z → z)
    retr r z = r

-- Definition 2.4.10
isequiv : {A B : Set} → (f : A → B) → Set
isequiv {A} {B} f = Σ (B → A) λ g → (f ∘ g ∼ id {B}) × Σ (B → A) λ g → (g ∘ f ∼ id {A})

-- (i) prior to 2.4.10
qinv->isequiv : {A B : Set} → (f : A → B) → qinv f → isequiv f
qinv->isequiv f (g , α , β) = g , α , g , β

-- (ii) prior to 2.4.10
-- not the same is as the book
isequiv->qinv : {A B : Set} → (f : A → B) → isequiv f → qinv f
isequiv->qinv f (g , α , g' , β ) = (g' ∘ f ∘ g) , sec , retr
  where
    sec : (λ x → f (g' (f (g x)))) ∼ (λ z → z)
    sec x = apf f (β (g x)) • α x
    retr : (λ x → g' (f (g (f x)))) ∼ (λ z → z)
    retr x = apf g' (α (f x)) • β x

-- book defn, confusing because of the "let this be the composite homotopy"
isequiv->qinv' : {A B : Set} → (f : A → B) →  isequiv f → qinv f
isequiv->qinv' f (g , α , h , β ) = g , α , β'
  where
    γ : (λ x → g x) ∼ λ x → h x
    γ x = β (g x) ⁻¹ • apf h (α x)
    β' : (λ x → g (f x)) ∼ (λ z → z)
    β' x = (γ (f x)) • β x
```

```
-- Definition 2.4.11
_≃_ : (A B : Set) → Set
A ≃ B = Σ (A → B) λ f → isequiv f

-- Lemma 2.4.12 (i)
≃refl : {A : Set} → A ≃ A
≃refl = (id) , (qi qinvid)
  where
    qi : qinv (λ z → z) → isequiv (λ z → z)
    qi = qinv->isequiv (id )
-- type equivalence is an equivalence relation, 2.4.12
-- qinv->isequiv

comm× : {A B : Set} → A × B → B × A
comm× (a , b) = (b , a)

-- Lemma 2.4.12 (ii)
≃sym : {A B : Set} → A ≃ B → B ≃ A
≃sym (f , eqf) = f-1 , ef (f , comm× sndqf)
  where
    qf : isequiv f → qinv f
    qf = isequiv->qinv f
    f-1 : _ → _
    f-1 = fst (qf eqf)
    sndqf : ((λ x → f (fst (isequiv->qinv f eqf) x)) ∼ (λ z → z)) ×
                  ((λ x → fst (isequiv->qinv f eqf) (f x)) ∼ (λ z → z))
    sndqf = snd (qf eqf)
    ef : qinv f-1 → isequiv f-1
    ef = qinv->isequiv f-1

-- Lemma 2.4.12 (iii)
≃trans : {A B C : Set} → A ≃ B → B ≃ C → A ≃ C
≃trans (f , eqf) (g , eqg) = (g ∘ f) ,
  qinv->isequiv (λ z → g (f z)) ((f-1 ∘ g-1) , sec , retr)
  where
    qf : isequiv f → qinv f
    qf = isequiv->qinv f
    f-1 = fst (qf eqf)
    qg : isequiv g → qinv g
    qg = isequiv->qinv g
    g-1 = fst (qg eqg)
    sec : (λ x → g (f (f-1 (g-1 x)))) ∼ (λ z → z)
    sec x =
              begin g (f (f-1 (g-1 x)))
              ≡⟨ apf g (fst (snd (qf eqf)) (g-1 x)) ⟩
              g (g-1 x)
              ≡⟨ fst (snd (qg eqg)) x ⟩
              x ∎
    retr : (λ x → f-1 (g-1 (g (f x)))) ∼ (λ z → z)
    retr x =
```

```
        begin f-1 (g-1 (g (f x)))
        ≡⟨ apf f-1 ((snd (snd (qg eqg)) (f x))) ⟩
        f-1 (f x)
        ≡⟨ snd (snd (qf eqf)) x ⟩
        x ∎

-- No section 2.5

-- Lemma 2.6.1
fprodId : {A B : Set} {x y : A × B} → _≡_ {A × B} x y → ((fst x) ≡ (fst y)) × ((snd x) ≡ (snd
fprodId p = (apf fst p) , (apf snd p)
-- fprodId r = r , r

-- Theorem 2.6.2
equivfprod : {A B : Set} (x y : A × B) → isequiv (fprodId {x = x} {y = y} )
equivfprod (x1 , y1) (x2 , y2) = qinv->isequiv fprodId (sn , h1 , h2)
  where
    sn : (x1 ≡ x2) × (y1 ≡ y2) → (x1 , y1) ≡ (x2 , y2)
    sn (r , r) = r
    h1 : (λ x → fprodId (sn x)) ∼ (λ z → z)
    h1 (r , r) = r
    -- h1 (r , r) = r
    h2 : (λ x → sn (fprodId x)) ∼ (λ z → z)
    h2 r = r

-- helper type for below
×fam : {Z : Set} {A B : Z → Set} → (Z → Set)
×fam {A = A} {B = B} z = A z × B z

-- Theorem 2.6.4
transport× : {Z : Set} {A B : Z → Set} {z w : Z} (p : z ≡ w) (x : ×fam {Z} {A} {B} z) → (
transport× r s = r

fprod : {A B A′ B′ : Set} (g : A → A′) (h : B → B′) → (A × B → A′ × B′)
fprod g h x = g (fst x) , h (snd x)

-- inverse of fprodId
pair= : {A B : Set} {x y : A × B} → (fst x ≡ fst y) × (snd x ≡ snd y) → x ≡ y
pair= (r , r) = r

-- Theorem 2.6.5
functorProdEq : {A B A′ B′ : Set} (g : A → A′) (h : B → B′) (x y : A × B) (p : fst x ≡ fst y) (q
functorProdEq g h (a , b) (.a , .b) r r = r


-- Theorem 2.7.2
-- rename f to g to be consistent with book
equivfDprod : {A : Set} {P : A → Set} (w w′ : Σ A (λ x → P x)) → (w ≡ w′) ≃ Σ (fst w ≡ fst w
equivfDprod (w1 , w2) (w1′ , w2′) = f , qinv->isequiv f (f-1 , ff-1 , f-1f)
  where
    f : (w1 , w2) ≡ (w1′ , w2′) → Σ (w1 ≡ w1′) (λ p → p* {p = p} w2 ≡ w2′)
    f r = r , r
```

```
    f-1 : Σ (w1 ≡ w1') (λ p → p* {p = p} w2 ≡ w2') → (w1 , w2) ≡ (w1' , w2')
    -- f-1 (r , psndw) = apf (λ - → (w1 , -)) psndw
    f-1 (r , r) = r
    ff-1 : (λ x → f (f-1 x)) ~ (λ z → z)
    ff-1 (r , r) = r
    f-1f : (λ x → f-1 (f x)) ~ (λ z → z)
    f-1f r = r

-- Corollary 2.7.3
etaDprod : {A : Set} {P : A → Set} (z : Σ A (λ x → P x)) → z ≡ (fst z , snd z)
etaDprod z = r

-- helper type for 2.7.4
Σfam : {A : Set} {P : A → Set} (Q : Σ A (λ x → P x) → Set) → (A → Set)
Σfam {P = P} Q x = Σ (P x) λ u → Q (x , u)

-- helper function for 2.7.4
dpair= : {A : Set} {P : A → Set} {w1 w1' : A} {w2 : P w1 } {w2' : P w1'} → (p : Σ (w1 ≡
dpair= (r , r) = r

-- Theorem 2.7.4
transportΣ : {A : Set} {P : A → Set} (Q : Σ A (λ x → P x) → Set) (x y : A) (p : x ≡ y) ((u , z)
              → _* {P = λ - → Σfam Q - } p (u , z) ≡ ((p *) u    , _* {P = λ - → Q ((fst -) , (snd
transportΣ Q x .x r (u , z) = r

fDprod : {A A' : Set} {P : A → Set} {Q : A' → Set} (g : A → A') (h : (a : A) → P a → Q (g a))
fDprod g h (a , pa) = g a , h a pa

ap2 : {A B C : Set} {x x' : A} {y y' : B} (f : A → B → C)
      → (x ≡ x') → (y ≡ y') → (f x y ≡ f x' y')
ap2 f r r = r

transportd : {X : Set } (A : X → Set ) (B : (x : X) → A x → Set )
  {x : X} ((a , b) : Σ (A x) λ a → B x a) {y : X} (p : x ≡ y)
  → B x a → B y (transport {P = A} p a)
transportd A B (a , b) r = id

data Unit : Set where
  ⋆ : Unit

-- Theorem 2.8.1
path1 : (x y : Unit) → (x ≡ y) ≃ Unit
path1 x y = (λ p → ⋆) , qinv->isequiv (λ p → ⋆) (f-1 x y , ff-1 , f-1f x y)
  where
    f-1 : (x y : Unit) → Unit → x ≡ y
    f-1 ⋆ ⋆ ⋆ = r
    ff-1 : (λ x₁ → ⋆) ~ (λ z → z)
    ff-1 ⋆ = r
    f-1f : (x y : Unit) → (λ _ → f-1 x y ⋆) ~ (λ z → z)
    f-1f ⋆ .⋆ r = r

-- 2.9
```

```
-- theorem 2.9.2
happly : {A : Set} {B : A → Set} {f g : (x : A) → B x} → f ≡ g → ((x : A) → f x ≡ g x )
happly r x = r

postulate
  funext : {A : Set} {B : A → Set} {f g : (x : A) → B x} → ((x : A) → f x ≡ g x ) → f ≡ g

->fam : {X : Set} (A B : X → Set) → X → Set
->fam A B x = A x → B x

-- Lemma 2.9.4
transportF : {X : Set} {A B : X → Set} {x1 x2 : X} {p : x1 ≡ x2} {f : A x1 → B x1} →
              transport {P = ->fam A B} p f ≡ λ x → transport {P = B} p (f (transport {P =
transportF {X} {A} {B} {x1} {.x1} {r} {f} = funext (λ x → r)
```