
DEPARTMENT OF PHILOSOPHY,
LINGUISTICS AND THEORY OF SCIENCE

# THE GRAMMAR OF PROOF

**Warrick Macmillan**


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

## Abstract

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

# Preface

Acknowledgements, etc.

# Contents

# Introduction

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

To get a feel for this syntactic paradigm, let us look at a basic mathematical example: that of a group homomorphism, as expressed in a variety of sources.

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

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

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

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

**Definition 3** *Let* $G$*,* $H$*, be groups. A map* $\phi : G \to H$ *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 4** *Classically, a group is a monoid in which every element has an inverse (necessarily unique).*

We inquire the reader to pay attention to nuance and difference in presentation that is normally ignored or taken for granted by the fluent mathematician.

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

Details aside, these all mean the same thing–don't they? This thesis seeks to provide an abstract framework to determine whether two lingusitically nuanced presenations mean the same thing via their syntactic transformations.

These syntactic transformations come in two flavors : parsing and linearization, and are natively handled by a Logical Framework (LF) for specifying grammars : Grammatical Framework (GF).

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

[TODO: replace monoidhom with grouphom]

theres supposeed to be agda here

```
monoidHom : {ℓ : Level}
            → ((monoid' a _ _ _ _ _ _) (monoid' a' _ _ _ _ _ _) : Monoid' {ℓ} )
            → (a → a') → Type ℓ
monoidHom
  (monoid' A ε _•_ left-unit right-unit assoc carrier-set)
  (monoid' A₁ ε₁ _•₁_ left-unit₁ right-unit₁ assoc₁ carrier-set₁)
  f
  = (m1 m2 : A) → f (m1 • m2) ≡ (f m1) •₁ (f m2)
```

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

While this example may not exemplify the power of Agda's type checker, it is of considerable interest to many. The typechecker has merely assured us that monoidHom, is a well-formed type. The type-checker is much more useful than is

immediately evident: it delegates the work of verifying that a proof is correct, that is, the work of judging whether a term has a type, to the computer. While it's of practical concern is immediate to any exploited grad student grading papers late on a Sunday night, its theoretical concern has led to many recent developments in modern mathematics. Thomas Hales solution to the Kepler Conjecture was seen as unverifiable by those reviewing it. This led to Hales outsourcing the verification to Interactive Theorem provers HOL Light and Isabelle, during which 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 invalid 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 [4].

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

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

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

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

Formal Abstracts Project.

## Perspectives

Computational linguistics, particularly those in the tradition of type theoretical semantics (Ranta 94), gives one a way of comparing natural and programming languages. Although it is concerned with the semantics of natural language in the logical tradition of Montague, it ended up inspiring the GF system, a side effect of which was to realize that translation was possible with this of abstracted view of natural language semantics. One can then carry this analogy to other areas of computer science, and compare NL and PL phenomena. Indeed, one such description of GF is that it is a compiler tool applied to domain specific machine translation.

* In PL theory, we also have a parallel series of abstraction levels, of which a compiler writing course covers the basics. Lexemes -> Abstract Syntax Trees -> Well Formed ASTs (i.e. trees which type checking) -> (possibly) intermediate language Programs -> Target Language * In what I'll call formal linguistics, we have some kind of hierarchy in terms of the following Phonemes -> Morhemes -> Syntactic Units -> Sentences -> Dialogue

This also has some (external) mapping at the neurological level, where one somehow can say the internal language, mechanism of externalization (generally speech), but we won't be concerned with that here. The point is to recognize their are stark differences, and that classifying both programming languages and natural languages as languages is best read as an incomplete (and even sometimes contradictory) metaphor, due to perceived similarities (of which their are ample).

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

The specific linguistic domain we focus on, that of mathematics, is a particular sweet spot in the intersection of many intersecting interests. It should be noted that this problem, that of translating between formal (in a PL or logic) and informal (in linguistic sense) mathematics as stated, is both vague and difficult. It is difficult in both the computer science sense, that it may be either of infeasible complexity or even perhaps undecidable. But it is also difficult in the philosophical sense, that it a question which one may come up with a priori arguements against its either effectiveness or meaningfulness. Because, like all collective human endeavors, mathematics is a historical construction - that is, its conventions, notations, understanding, methodologies, and means of publication and distribution have all been in a constant flux. While in some sense the mathematics today can be seen today as a much refined version of whatever the Greeks or Egyptians were doing, there is no consensus on what mathematics is, how it is to be done,

and most relevant for this treatise, how it is to be expressed.

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

* syntactic Completeness and semantic adequacy

The GF pipeline, that of bidirectional translation through an intermediary abstract syntax representation, has, two fundamental criteria that must be assessed for one to judge the success of the approach for the informal/formal translation. The first, which we'll call *syntactic completeness*, asks the following : given an utterance or natural language expression that a mathematician might understand, does the GF grammar emit a well-formed syntactic expression in the target logic or programming language? [Example?]

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

Conversely, given a well formed syntactic expression in, for instance, Agda, one can ask if the resulting English expression generated by GF is *semantically adequate*. This notion of semantic adequacy is also delicate, as mathematicians themselves may dispute, for instance, the proof of a given proposition or the correct definition of some notion. However, if it is doubtful that there would be many mathematicians who would not understand some standard theorem/proof pair in a 7th edition introductory real analysis text, even if they dispute it's presentation, clarity, pedagogy, or other pedantic details. So, 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 possibly dubious perspective in science), semantic adequacy should appease at least one and potentially more mathematicians.

We introduce these terms, syntactic completeness and semantic adequacy, to highlight a perspective and insight that seems to underlie the biggest differences between informal and formal mathematics. We claim that mathematics, as done on a theorem prover, is a syntax oriented endeavor, whereas mathematics, as practiced by mathematicians, prioritizes semantic understanding.

This perspective represents an observation, and not intended to take a side as to whether the syntactic or semantic perspective on mathematics is better - there is an dialectical phenomena between the two.

Let's highlight some advantages both provide, and try to distinguish more precisely what a syntactic and semantic perspective may be.

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

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

This is particularly pronounced in Coq, where proof terms are built using LTac, which can be seen as some kind of imperative syntactic metaprogramming over the core language, Gallina. The Tactics themselves are not typed, often feel very adhoc, and carry very little semantic value (or even possibly muddy one's understanding when reading proofs with unknown tactics). Indeed, since LTac isn't itself typed, it often descends into the sorrows of so-called untyped languages (which really are really unityped), and there are multiple arguements that this should be changed. But from our perspective, the use of tactics is an additional syntactic obfuscation of what a proof should look like from the mathematicians perspective - and attempt to remedy this is. This isn't always the case, however, as tactics like 'ring' or 'omega' often save the reader overhead of parsing pendantic and unin-

foramtive details. And for certain proofs, especially those involving many cases, the metaprogramming facilities actually give one exclusive advantages not offered to the classical mathematician using pen and paper. Nonetheless, the dependent type theorist's dream that all mathematicians begin using theorem provers in their everyday work is largely just a dream, and with relatively little mainstream adoption by mathematicians, the future is all but clear.

Mathematicians may indeed like some of the facilities theorem provers provide, but ultimately, they may not see that as the "essence" of what they are doing.

## Previous Work

The prior exploration of these interleaving subjects is vast, and we can only sample the available literature here.

### Ranta

Hallgren/Ranta/Alfa

HoTT Grammar

### Mohan Ganesalingam

The Language of Mathematics

### other authors

NaProche, Mizar, Coq (coquand),

# Preliminaries

We give brief but relevant overviews of the background ideas and tools that went into the generation of this thesis.

## Martin-Löf Type Theory

### Judgments

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. These include, for instance,

- $A$ is a proposition
- $A$ is possible
- $A$ is probable

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

For the logician these judgments may well be interesting because their may be logics in which the discussion of possibility or probability 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 :

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

We succinctly present these in a mathematical notation where Frege's turnstile, $\vdash$, denotes a judgment :

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

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 $\Gamma$, 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 :

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

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. This comes out in the following introduction rule :

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

Using this rule, we now see a typical judgment, typical in a field like from real analysis,

$\vdash \lambda x. \sin x :\to$

Equality :

Mathematicians denote this judgement

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

## Grammatical Framework

### Overview

I will introduce Grammatical Framework (GF) through an extended example targeted at a general audience familiar with logic, functional programming, or mathematics. GF is a powerful tool for specifying grammars. A grammar specification in GF is actually an abstract syntax. With an abstract syntax specified, one can then define various linearization rules which compositionally evaluate to strings. An Abstract Syntax Tree (AST) may then be linearized to various strings admitted by different concrete syntaxes. Conversely, given a string admitted by the language being defined, GF's powerful parser will generate all the ASTs which linearize to

that tree.

## Overview

We introduce this subject assuming no familiarity with GF, but a general mathematical and programming maturity. While GF is certainly geared to applications in domain specific machine translation, this writing will hopefully make evident that learning about GF, even without the intention of using it for its intended application, is a useful exercise in abstractly understanding syntax and its importance in just about any domain.

A working high-level introduction of Grammatical Framework emphasizing both theoretical and practical elements of GF, as well as their interplay. Specific things covered will be

- Historical developments leading to the creation and discovery of the GF formalism
- The difference between Abstract and Concrete Syntax, and the linearization and parsing functions between these two categories
- The basic judgments :
  - Abstract : 'cat, fun'
  - Concrete : 'lincat, lin'
  - Auxiliary : 'oper, param'
- A library for building natural language applications with GF
  - The Resource Grammar Library (RGL)
  - A way of interfacing GF with Haskell and transforming ASTs externally
  - The Portable Grammar Format (PGF)
- Module System and directory structure
- A brief comparison with other tools like BNFC

These topics will be understood via a simple example of an expression language, 'Arith', which will serve as a case study for understanding the many robust, theoretical topics mentioned above - indeed, mathematics is best understood and learned through examples. The best exercises are the reader's own ideas, questions, and contentions which may arise while reading through this.

## Theoretical Overview

**What is a GF grammar ?** [TODO : update with latex]

Consider a language L, for now given a single linear presentation $C_0^L, where AST_L String_L 0 denote the$

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

with the important property that given a string s,

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

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

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

We can should note that computa

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

is understood as the set of translations of a phrase of one language to possibly grammatical phrases in the other languages connected by a mutual abstract syntax. So we could denote these L$^{E}nglish, L^{B}antu, L^{S}wedishforL^{E}nglsih_{A}mericanvsL^{E}nglsih_{E}nglishorL$

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

L$_0 < L_1 -> L_{E}nglish_0 < L_{E}nglish_1$

But this would be similair to a set of expressions translateable between programming languages, like Logic$^{E}nglish, Logic^{L}atex, Logic^{A}gda, Logic^{C}oq, etc$

where one could extend the

Logic$_{C}ore^{E}nglishLogic_{E}xtended^{E}nglish$

whereas in the PL domain

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

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

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

Mathematics$^{E}nglishiI --Somethingabout$

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

break down to different classifications of

– Something about

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

break down to different classifications of

The indexes here, while seemingly arbitrary,

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

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

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

## Some preliminary observations and considerations

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

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

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

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

[Abstract <-> Concrete] syntax

Have your cake and eat it ?

## A grammar for basic arithmetic

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

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

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

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

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

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

```
cat Exp ;

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

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

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

The trees nodes are just the function names, and the leaves, while denoted above as numbers, are actually function names for the built-in numeric strings which happen to be linearized to the same piece of syntax, i.e. 'linearize 3 == 3', where the left-hand 3 has type 'Int' and the right-hand 3 has type 'Str'. GF has support

for very few, but important categories. These are ‘Int‘, ‘Float‘, and ‘String‘. It is my contention and that adding user defined builtin categories would greatly ease the burden of work for people interested in using GF to model programming languages, because ‘String‘ makes the grammars notoriously ambiguous.

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

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

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

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

For instance, if

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

Then one can form a tree ‘DetCN The$_{D}etCar_{C}N`which should linearize to`"the car"`in English,`bilen`in Su$

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

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

```
abstract Arith = {
```

```
flags startcat = Exp ;

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

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

}
```

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

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

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

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

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

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

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

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

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

Here is the first pass at an English linearization :

```
 lincat Exp = Str ;

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

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

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

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

$f$:$C_0 \to C_1 \to ... \to C_n$

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

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

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

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

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

The PMCFG class of languages is still quite tame when compared with, for instance, Turing complete languages. Thus, the ‘abstract‘ and ‘concrete‘ coupling tight, the evaluation is quite simple, and the programs tend to write themselves once the correct types are chosen. This is not to say GF programming is easier than in other languages, because often there are unforeseen constraints that the programmer must get used to, limiting the palette available when writing code. These constraints allow for fast parsing, but greatly limit the types of programs one often thinks of writing. We touch upon this in a [previous section](some-preliminary-observations-and-considerations).

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

## The GF shell

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

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

```
 $ gf ArithEng1.gf
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Exercises

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

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

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

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

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

Solutions

*Exercise 1.1*

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

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

*Exercise 1.2*

This is a similar augmentation as was performed above.

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

cat Prop ;

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

*Exercise 2*

*Exercise 3*

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

“‘haskell lin Add e1 e2 = e1 ++ ”plus” ++ e2 ; Mul e1 e2 = e1 ++ ”times” ++ e2 ; “‘

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

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

x times 999 times x plus 999

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

x times 999 times x plus 999 x times 999 times x plus 999 x times 999 times x plus 999 x times 999 times x plus 999 x times 999 times x plus 999 “‘

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

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

Add (EInt 999) (Mul (VExp VarY) (Mul (EInt 999) (EInt 999))) “‘

...Blog post...

## Agda

Agda is an attempt to formalize Martin-Löf’s intensional type theory (reference 1984). For our purposes, we will only look at what can in some sense be seen as the kernel of Agda.

## Natural Language and Mathematics

## Natural Number Proofs

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

natproof?

# HoTT Proofs

## Why HoTT for natural language?

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

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

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

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

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

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

**Definition 5** *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* $\mathsf{refl}_a$ *as being the constant path at the point* $a$.

We recapitulate this definition in Agda, and treat :

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

## An introduction to equality

There is already some tension brewing : most mathematicains have an intuition for equality, that of an identitfication between two pieces of information which intuitively must be the same thing, i.e. $2+2=4$. They might ask, what does it mean to "construct an element of $a=b$"? For the mathematician use to thinking in terms of sets $\{a=b \mid a,b \in \mathbb{N}\}$ isn't a well-defined notion. Due to its use of the axiom of extensionality, the set theoretic notion of equality is, no suprise, extensional. This means that sets are identified when they have the same elements, and equality is therefore external to the notion of set. To inhabit a type means to provide evidence for that inhabitation. The reflexivity constructor is therefore a means of providing evidence of an equality. This evidence approach is disctinctly constructive, and a big reason why classical and constructive mathematics, especially when treated in an intuitionistic type theory suitable for a programming language implementation, are such different beasts.

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

Prior to the homotopical interpretation of identity types, debates about extensional and intensional type theories centred around two features or bugs : extensional type theory sacrificed decideable type checking, while intensional type theories required extra beauracracy when dealing with equality in proofs. One approach in intensional type theories treated types as setoids, therefore leading to so-called "Setoid Hell". These debates reflected Martin-Löf's flip-flopping on the issue. His seminal 1979 Constructive Mathematics and Computer Programming, which took an extensional view, was soon betrayed by lectures he gave soon thereafter in Padova in 1980. Martin-Löf was a born again intensional type theorist. These Padova lectures were later published in the "Bibliopolis Book", and went on to

inspire the European (and Gothenburg in particular) approach to implementing proof assitants, whereas the extensionalists were primarily eminating from Robert Constable's group at Cornell.

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

## All about Identity

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

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

infix 20 _≡_
```

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

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

[ToDo, give a GF example]

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

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

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

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

**Definition 6 (Version 1)** *Moreover, one of the amazing things about homotopy type theory is that all of the basic constructions and axioms—all of the higher groupoid structure—arises automatically from the induction principle for identity types. Recall from [section 1.12] that this says that if*

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

*then*

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

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

**Definition 7 (Version 2)** *In other words, given dependent functions*

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

*there is a dependent function*

$$\mathsf{ind}_{=_A}(D, d) : \prod_{(x,y:A)} \prod_{(p:x=y)} D(x, y, p)$$

*such that*

$$\mathsf{ind}_{=_A}(D, d, a, a, \mathsf{refl}_a) \equiv d(a) \tag{1}$$

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

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

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

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

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

**Lemma 1** *For every type $A$ and every $x, y : A$ there is a function*

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

*denoted $p \mapsto p^{-1}$, such that $\mathsf{refl}_x{}^{-1} \equiv \mathsf{refl}_x$ for each $x : A$. We call $p^{-1}$ the **inverse** of $p$.*

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

*to any $x, y : A$ and $p : x = y$ a type, namely the type $y = x$. Then we have an element*

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

*Thus, the induction principle for identity types gives us an element* $\mathsf{ind}_{=_A}(D, d, x, y, p) : (y = x)$ *for each* $p : (x = y)$*. We can now define the desired function* $(-)^{-1}$ *to be* $\lambda p.\, \mathsf{ind}_{=_A}(D, d, x, y, p)$*, i.e. we set* $p^{-1} :\equiv \mathsf{ind}_{=_A}(D, d, x, y, p)$*. The conversion rule [missing reference] gives* $\mathsf{refl}_x{}^{-1} \equiv \mathsf{refl}_x$*, as required.*

The Agda code is certainly more brief:

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

infixr 50 _⁻¹
```

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

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

**Proof 2 (Second proof)** *We want to construct, for each $x, y : A$ and $p : x = y$, an element $p^{-1} : y = x$. By induction, it suffices to do this in the case when $y$ is $x$ and $p$ is $\mathsf{refl}_x$. But in this case, the type $x = y$ of $p$ and the type $y = x$ in which we are trying to construct $p^{-1}$ are both simply $x = x$. Thus, in the "reflexivity case", we can define $\mathsf{refl}_x{}^{-1}$ to be simply $\mathsf{refl}_x$. The general case then follows by the induction principle, and the conversion rule $\mathsf{refl}_x{}^{-1} \equiv \mathsf{refl}_x$ is precisely the proof in the reflexivity case that we gave.*

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

```
_⁻¹' {A} {x} {y} r = r
```

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

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

infixl 40 _•_
```

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

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

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

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

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

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

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

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

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

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

**Definition 8** *Thus, given a type $A$ with a point $a : A$, we define its* ***loop space*** $\Omega(A, a)$ *to be the type $a =_A a$. We may sometimes write simply $\Omega A$ if the point $a$ is understood from context.*

**Definition 9** *It can also be useful to consider the loop space* of *the loop space of $A$, which is the space of 2-dimensional loops on the identity loop at $a$. This is written $\Omega^2(A, a)$ and represented in type theory by the type* $\mathsf{refl}_a =_{(a=_A a)} \mathsf{refl}_a$.

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

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

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

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

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

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

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

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

*[TODO Finish Eckmann Hilton Arguement]*

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

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

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

-- iᵣ == ruₚ

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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


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

[TODO, fix without k errors]

## GF Grammar for types

We now discuss the GF implementation, capable of parsing both natural language and Agda syntax. The parser was appropriated from the cubicaltt BNFC parser, de-cubified and then gf-ified. The languages are tightly coupled, so the translation is actually quite simple. Some main differences are:

- GF treats abstract and concrete syntax seperately. This allows GF to support many concrete syntax implementation of a given grammar
- Fixity is dealt with at the concrete syntax layer in GF. This allows for more refined control of fixity, but also results in difficulties : during linearization their can be the insertion of extra parens.
- GF supports dependent hypes and higher order abstract syntax, which makes it suitable to typecheck at the parsing stage. It would very interesting to see if this is interoperable with the current version of this work in later iterations [Todo - add github link referncing work I've done in this direction]
- GF also is enhanced by a PGF back-end, allowing an embedding of grammars into, among other languages, Haskell.

While GF is targeted towards natural language translation, there's nothing stopping it from being used as a PL tool as well, like, for instance, the front-end of a compiler. The innovation of this thesis is to combine both uses, thereby allowing translation between Controlled Natural Languages and programming languages.

Example expressions the grammar can parse are seen below, which have been verified by hand to be isomorphic to the corresponding cubicaltt BNFC trees:

```
data bool : Set where true | false
data nat : Set where zero | suc ( n : nat )
caseBool ( x : Set ) ( y z : x ) : bool -> Set = split false -
> y || true -> z
indBool ( x : bool -> Set ) ( y : x false ) ( z : x true ) : ( b : bool ) -
> x b = split false -> y || true  -> z
funExt  ( a : Set )   ( b : a -> Set )   ( f g :  ( x : a )  -
> b x )  ( p :  ( x : a )  -> ( b x )  ( f x ) == ( g x ) ) : (  ( y : a ) -
> b y )  f == g = undefined
foo ( b : bool ) : bool = b
```

[Todo] add use cases

## Goals and Challenges

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

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

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

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

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

# Code

GF Parser

Additional Agda Hott Code

## Testing

Hello world

Two citation examples: [3] introduced a well-known method for extracting collocations. Bilingual data can be used to train part-of-speech taggers [2]. Another one: [1]

# References

[1] Cortes, C., Kuznetsov, V., & Mohri, M. (2014). Learning ensembles of structured prediction rules. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 1–12). Baltimore, Maryland: Association for Computational Linguistics.

[2] Das, D. & Petrov, S. (2011). Unsupervised part-of-speech tagging with bilingual graph-based projections. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies* (pp. 600–609). Portland, Oregon, USA: Association for Computational Linguistics.

[3] Dunning, T. (1993). Accurate methods for the statistics of surprise and coincidence. *Computational Linguistics*, 19(1), 61–74.

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

# Appendices