UNIVERSITY OF GOTHENBURG

DEPARTMENT OF PHILOSOPHY,
LINGUISTICS AND THEORY OF SCIENCE

ON THE GRAMMAR OF PROOF

Warrick Macmillan

Abstract

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

Preface

Acknowledgements, etc.

Contents

1304

TODO Add section numbers to sections Fix figure titles

Grammatical Framework

Thinking about GF

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

When defining a GF pipeline, one 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 [1].

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

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

$$\begin{tikzcd}
 & Logic & \\
 & GF \arrow[u, "GF\ Parser\ Specification"'] \arrow[dl, "Theory\ of\ Operads"'] \arrow[dr, bend left, "Implementation\ of"] \arrow[dr, bend right, "Agda\ Embedding"'] & \\
Math & & CS\ (MLTT)
\end{tikzcd}$$

Figure 1: Models of GF

One can additionally model these domains in GF, which is obviously the main focus of this work. In Figure 2, we see that there are 3 grammars which give allow one to translate in these domains. Ranta's grammar from CADE 2011, built a propositional framework with a core grammar extended with other categories to capture syntactic nuance. Ranta's grammar from the Stockholm University mathematics seminar in 2014 took verbatim text from a publication of Peter Aczel and sought to show that all the syntactic nuance by constructing a grammar capable of NL translation. Finally, our work takes a BNFC grammar for a real programming language cubicaltt [cite], GFifies it, producing an unambiguous grammar.

$$\begin{tikzcd}
 & Logic \arrow[d, "Ranta\ Logic\ (CADE\ 11)"] & \\
 & GF & \\
Math \arrow[ur, "Ranta\ (HoTT\ 14)"] & & CS\ (MLTT) \arrow[ul, "cubicalTT"]
\end{tikzcd}$$

Figure 2: Trinitarian Grammars

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

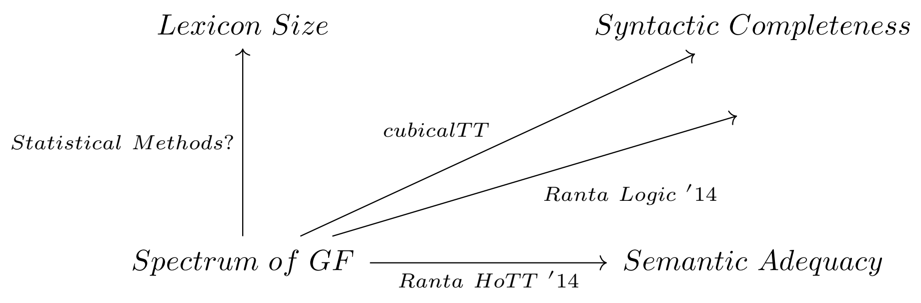

Figure 3: The Grammatical Dimension

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

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

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

A Brief Introduction to GF

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

The two functions displayed in **??**, $Parse : \{Strings\} \to \{\{ASTs\}\}$ and $Linearize : \{ASTs\} \to \{Strings\}$, obey the important property that :

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

PGF, is an embedding of a GF abstract syntax into Haskell, where the categories are given "shadow types", so that one can build turn an abstract syntax into (a possibly massive) Generalized Algebraic Data Type (GADT) `Tree` with kind `* -> *` where all the functions serve as constructors. If function `h` returns category `c`, the Haskell constructor `Gh` returns `Tree c`.

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

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

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

Mathematical Model of GF

Note on the construction of free monoids

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

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

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

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

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

Example : consider the arithmetic grammar of exponentiation, multiplication, and addition defined over a single category of natural number expressions, whereby the function symbol is to be interpreted as a string and the tensor product, $\otimes$ as the concatenation during evaluation.

$$\_\,\hat{}\,\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
$$\_ * \_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
$$\_ + \_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

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

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

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

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

$x + +y\ x \quad y\ ``x \quad y"$

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

$$\begin{aligned}
&((\lambda x, y.\, x \otimes \hat{} \otimes y)\\
&\qquad (\lambda x, y.\, x \otimes + \otimes y)\\
&\qquad (\lambda x, y.\, x \otimes * \otimes y))\\
&\qquad 1\ 2; 3; 4;
\end{aligned}$$

(1 + 2) ^ (3 * 4)

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

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

(1 + 2) ^ (3 * 4)

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

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

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

where

□y : Pow y 1 : Nat -> Nat

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

(plus 2 3) (plus 4 5)

can either be

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

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

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

AST$_L String_L 0 denotethesetsGFASTsandStringsinthelanguagesgeneratedbytherulesofL's abstractsyntaxar$

$$Parse : String- > AST$$

$$Linearize : AST- > String$$

with the important property that given a string 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.

Prior GF Formalizations

Prior to the grammars explored thesis, Ranta produced two main results [2] [3]. These are incredibly important precedents in this approach to proof translation, and serve as important comparative work for which this work responds.

CADE 2011

In [2], Ranta designed a grammar which allowed for predicate logic with a domain specific lexicon supporting mathematical theories , say geometry or arithmetic, on top of the logic. The syntax was both meant to be relatively complete, so that typical logical utterances of interest could be accommodated, as well as support relatively non-trivial linguistic nuance like lists of terms, predicates, and propositions, in-situ and bounded quantification, like other ways of constructing more syntactically nuanced predicates. The more interesting syntactic details captured in this work was by means of an extended grammar on top of the core. The bidirectional transformation between the core and extended grammars via a PGF also show the viability and necessity of using more expressive programming languages (Haskell) when doing thorough translations.

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

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

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

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

In the opposite direction, the desugaring of a logically “informal” statement into something less linguistically idiomatic is also accomplished. Ranta claims “Finding extended syntax equivalents for core syntax trees is trickier than the opposite direction". While this may be true for this particular grammar, we argue that this may not hold generally. Dealing with these ambiguities must be solved first and foremost to satisfy the PL designer who only accepts unambiguous parses. For instance, the gf shell shows "the sum of the sum of x and y and z is equal to the sum of x and the sum of y and z" giving 32 unique parses. Ranta also outlines the mapping, $[\![-]\!] : Core \to Extended$, which should hypothetically return a set of extended sentences for a more comprehensive grammar.

- Flattening a list $x\ and\ y\ and\ z\ \mapsto x,\ y\ and\ z$
- Aggregation $x\ is\ even\ or\ x\ is\ odd\ \mapsto x\ is\ even\ or\ odd$
- In-situ quantification
  $\forall\ n\ \in Nat,\ x\ is\ even\ or\ x\ is\ odd \mapsto every\ Nat\ is\ even\ or$
- 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 [cite ganesalingam], will pose challenges. Extending this work in general without very sophisticated statistical methods is impossible because mathematicians will speak uniquely, and so choosing how to extend a grammar that covers the multiplicity of ways of saying "the same thing" will require many choices and a significant corpus of examples. Efficient communication, is a pragmatic feature which this only begins to barely address. The most interesting linguistic phenomena covered by this grammar, In-situ quantification, has been at the heart of the Montague tradition.

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

An Additional PGF Grammar

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Stockholm Math Seminar 2014

In 2014, Ranta gave an unpublished talk at the Stockholm Mathematics seminar [3]. Fortunately the code is available, although many of the design choices aren't documented in the grammar. This project aimed to provide a translation like the one desired in our current work, but it took a real piece of mathematics text as the main influence on the design of the Abstract syntax.

This work took a page of text from Peter Aczel's book which more or less goes over standard HoTT definitions and theorems, and allows the translation of the latex to a pidgin logical language. The central motivation of this grammar was to capture, entirely "real" natural language mathematics, i.e. that which was written for the mathematician. Therefore, it isn't reminiscent of the slender abstract syntax the type theorist adores, and sacrificed "syntactic completeness" for "semantic adequacy". This means that the abstract syntax is much larger and very expressive, but it no longer becomes easy to reason about and additionally quite ad-hoc. Another defect is that this grammar overgenerates, so producing a unique parse from the PL side will become tricky. Nonetheless, this means that it's presumably possible to carve a subset of the GF HoTT abstract file to accommodate an Agda program, but one encounters rocks as soon as one begins to dig. For example, in Figure 4 is some rendered latex taken verbatim from Ranta's test code.

With some of hours of tinkering on the pidgin logic concrete syntax and some reverse engineering with help from the GF shell, one is able to get these definitions in **??**, which are intended to share the same syntactic space as cubicalTT. We note the first definition of "contractability" actually runs in cubicalTT up to renaming a lexical items, and it is clear that the translation from that to Agda should be a benign task. However, the *equivalence* syntax is stuck with the artifact from the bloated abstract syntax for the of the anaphoric use of "it", which may presumably be fixed with a few hours more of tinkering, but becomes even more complicated when not just defining new types, but actually writing real mathematical proofs, or relatively large terms. To extend this grammar to accommodate a chapter worth of material, let alone a book, will not just require extending the lexicon, but encountering other syntactic phenomena that will further be difficult to compress when defining Agda's concrete syntax.

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

Figure 4: Rendered Latex

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

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

Figure 5: Contractibility

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

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

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

Figure 6: Contractibility

Additionally, we give the Agda code in Figure 6, so-as to see what the end result of such a program would be. The astute reader will also notice a semantic in the pidgin rendering error relative to the Agda implementation. `fiber` has the type `it : Set` instead of something like `(y : B) : Set`, and the y variable is unbound in the `fiber` expression. This demonstrates that to design a grammar prioritizing *semantic adequacy* and subsequently trying to incorporate *syntactic completeness* becomes a very difficult problem. Depending on the application of the grammar, the emphasis on this axis is most assuredly a choice one should consider up front.

While both these grammars have their strengths and weaknesses, one shall see shortly that the approach in this thesis, taking an actual programming language parser in Backus-Naur Form Converter (BNFC), GFifying it, and trying to use the abstract syntax to model natural language, gives in some sense a dual challenge, where the abstract syntax remains simple, but its linearizations become must increase in complexity.

References

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

[2] 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.

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

Appendices