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

1946

TODO Add section numbers to sections Fix figure titles

Prior GF Formalizations

Prior to the grammars explored thesis, Ranta produced two main results [1] [2]. 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 [1], 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, $[\![-]\!] : Core \rightarrow Extended$, which should hypothetically return a set of extended sentences for a more comprehensive grammar.

- Flattening a list $x\ and\ y\ and\ z\ \mapsto x,\ y\ and\ z$
- Aggregation $x\ is\ even\ or\ x\ is\ odd\ \mapsto x\ is\ even\ or\ odd$
- In-situ quantification
  $\forall\ n\ \in Nat,\ x\ is\ even\ or\ x\ is\ odd \mapsto every\ Nat\ is\ even\ or$
- 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, $[\![-]\!] : \{pgfAST\} \rightarrow \{pgfAST\}$, but also evaluate them in more classical domains $[\![-]\!] : \{pgfAST\} \rightarrow \mathbb{N}$ for the arithmetic objects and $[\![-]\!] : \{pgfAST\} \rightarrow \mathbb{B}$, `evalProp`, for the propositions.

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

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

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

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

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

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

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

Stockholm Math Seminar 2014

In 2014, Ranta gave an unpublished talk at the Stockholm Mathematics seminar [2]. 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 1 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

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 1: 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 2: 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 3: Contractibility

when defining Agda's concrete syntax.

Additionally, we give the Agda code in Figure 3, 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.

– Proof using isPropIsContr. This is slow and the direct proof below is better

isPropIsEquiv' : (f : A → B) → isProp (isEquiv f) equiv-proof (isPropIsEquiv' f u0 u1 i) y = isPropIsContr (u0 .equiv-proof y) (u1 .equiv-proof y) i

– Direct proof that computes quite ok (can be optimized further if – necessary, see:
q– HTTPSqqq://github.com/mortberg/cubicaltt/blob/pi4s3$_{d}imclosures/examples/brunerie2.cttL562$

References

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

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

Appendices