DEPARTMENT OF PHILOSOPHY,
LINGUISTICS AND THEORY OF SCIENCE

ON THE GRAMMAR OF PROOF

Warrick Macmillan

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

Abstract

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

Preface

Acknowledgements, etc.

Contents

1849 Add section numbers to sections

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

> With Kant, something important happened, namely, that the term judgement, Ger. Urteil, came to be used instead of proposition [5].

A central contribution of Per Martin-Löf in the development of type theory was the recognition of the centrality of judgments in logic. Many mathematicians aren't familiar with the spectrum of judgments available, and merely believe they are concerned with *the* notion of truth, namely *the truth* of a mathematical proposition or theorem. There are many judgments one can make which most mathematicians aren't aware of or at least never mention. Examples of both familiar and unfamiliar judgments include,

- A is true
- A is a proposition
- A is possible
- A is necessarily true
- A is true at time t

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

For the logician these judgments may well be interesting because their may be logics in which the discussion of possibility or necessity is even more interesting than the discussion of truth. And for the type theorist interested in designing and building programming languages over many various logics, these judgments become a prime focus. The role of the type-checker in a programming language is to present evidence for, or decide the validity of the judgments. The four main judgments of type theory are given in natural language on the left and symbolically on the right.

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

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

Frege's turnstile, $\vdash$, denotes a judgment.

These judgments become much more interesting when we add the ability for them to be interpreted in a some context with judgment hypotheses. Given a series of judgments $J_1, ..., J_n$, denoted Γ, where J_i can depend on previously listed $J's$, we can make judgment J under the hypotheses, e.g. $J_1, ..., J_n \vdash J$. Often these hypotheses J_i, alternatively called *antecedents*, denote variables which may occur freely in the *consequent* judgment J. For instance, the antecedent, $x : \mathbb{R}$ occurs freely in the syntactic expression $\sin x$, a which is given meaning in the judgment $\vdash \sin x{:}\mathbb{R}$. We write our hypothetical judgement as follows :

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

Rules

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

The fundamental notion of the lambda calculus, the function, is abstracted over a variable and returns a term of some type when applied to an argument which is subsequently reduced via the computational rules. Dependent Type Theory (DTT) generalizes this to allow the return type be parameterized by the variable being abstracted over. The dependent function forms the basis of the LF which underlies Agda and GF.

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 :

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

$$f{:}\mathbb{R} \to \mathbb{R}$$
$$x \mapsto \sin(x)$$

Propositions, Sets, and Types

While the rules of type theory have been well-articulated elsewhere, we provide briefly compare the syntax of mathematical constructions in FOL, one possible natural language use [7], and MLTT. From this vantage, these look like simple symbolic manipulations, and in some sense, one doesn't need a the expressive power of system like GF to parse these to the same form.

Additionally, it is worth comparing the type theoretic and natural language syntax with set theory, as is done in Figure 1 and Figure 2. Now we bear witness to some deeper cracks than were visible above. We note that the type theoretic syntax is *the same* in both tables, whereas the set theoretic and logical syntax shares no overlap. This is because set theory and first order logic are distinct domains classically, whereas in MLTT, there is no distinguishing mathematical types from logical types - everything is a type.

FOL	MLTT	NL FOL	NL MLTT
$\forall\ x\ P(x)$	$\Pi x : \tau.\ P(x)$	$for\ all\ x,\ p$	$the\ product\ over\ x\ in\ p$
$\exists\ x\ P(x)$	$\Sigma x : \tau.\ P(x)$	$there\ exists\ an\ x\ such\ that\ p$	$there\ exists\ an\ x\ in\ \tau such\ that\ p$
$p\ \supset\ q$	$p\ \to\ q$	$if\ p\ then\ q$	$p\ to\ q$
$p\ \wedge\ q$	$p\ \times\ q$	$p\ and\ q$	$the\ product\ of\ p\ and\ q$
$p\ \vee\ q$	$p\ +\ q$	$p\ or\ q$	$the\ coproduct\ of\ p\ and\ q$
$\neg\ p$	$\neg\ p$	$it\ is\ not\ the\ case\ that\ p$	$not\ p$
$\top$	$\top$	$true$	top
$\bot$	$\bot$	$false$	$bottom$
$p\ =\ q$	$p\ \equiv\ q$	$p\ equals\ q$	$definitionally\ equal$

Figure 1: FOL vs MLTT

We show the Type and set comparisons in Figure 2. The basic types are sometimes simpler to work with because they are expressive enough to capture logical and set theoretic notions, but this also comes at a cost. The union of two sets simply gives a predicate over the members of the sets, whereas union and intersection types are often not considered "core" to type theory, with multiple possible ways of interpreting how to treat this set-theoretic concept. The behavior of subtypes and subsets, while related in some ways, also represents a semantic departure from sets and types. For example, while there can be a greatest type in some sub-typing schema, there is no notion of a top set. This is why we use the type theoretic NL syntax when there are question marks in the set theory column.

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

Figure 2: Sets vs MLTT

We also note that pragmatically, type theorists often interchange the logical, set theoretic, and type theoretic lexicons when describing types. Because the types were developed to overcome shortcomings of set theory and classical logic, the lexicons of all three ended up being blended, and in some sense, the type theorist can substitute certain words that a classical mathematician wouldn't. Whereas p *implies* q and *function from* X *to* Y are not to be mixed, the type theorist may in some sense default to either. Nonetheless, pragmatically speaking, one would never catch a type theorist saying $NatimpliesNat$ when expressing $Nat \to Nat$.

Terms become even messier, and this can be seen in just a small sample shown in Figure 3. In simple type theory, one distinguishes between types and terms at the syntactic level - this disappears in DTT. As will be seen later, the mixing of terms and types gives MLTT an incredible expressive power, but undoubtedly makes certain things very difficult as well. In set theory, everything is a set, so there is no distinguishing between elements of sets and sets even though practically they function very differently. Mathematicians only use sets because of their flexibility in so many ways, not because the axioms of set theory make a compelling case for sets being this kind of atomic form that makes up the mathematical universe. Category theorists have discovered vast generalizations of sets (where elements are arrows) which allow one to have the flexibility in a more structured and nuanced way, and the comparison with categories and types is much tighter than with sets. Regardless, mathematicians day to day work may not need all this general infrastructure.

In FOL, terms don't exist at all, and the proof rules themselves contain the necessary information to encode the proofs or constructions. The type theoretic terms somehow compress and encode the proof trees, of which, and in the case of ITPs nodes are displayed during the interactive type-checking phase.

Set Theory	MLTT	NL Set Theory	NL MLTT	Logic
$f(x) := p$	$\lambda x.p$	$f\ of\ x\ is\ p$	$lambda\ x,\ p$	$\supset -elim$
$f(p)$	fp	$fofp$	$the\ application\ of\ f\ to\ p$	$modus\ ponens$
(x, y)	(x, y)	$the\ pair\ of\ x\ and\ y$	$the\ pair\ of\ x\ and\ y$	$\wedge - i$
$\pi_{1,2}\ x$	$\pi_{1,2}\ x$	$the\ first\ projection\ of\ x$	$the\ first\ projection\ of\ x$	$\wedge - e$

Figure 3: Term syntax in Sets, Logic, and MLTT

We don't do all the constructors for type theory here for space, but note some interesting features:

- The disjoint union in set theory is actually defined using pairs - and therefore it doesn't have elimination forms other than those for the product. The disjoint union is also not nearly as ubiquitous, though.
- λ is a constructor for both the dependent and non-dependent function, so its use in either case will be type-checked by Agda, whereas it's natural language counterpart in real mathematics will have syntactic distinction.
- The projections for a Σ type behaves differently from the elimination principle for $\exists$, and this leads to incongruities in the natural language presentation.

Finally, we should note that there are many linguistic presentations mathematicians use for logical reasoning, i.e. the use of introduction and elimination rules. They certainly seem to use linguistic forms more when dealing with proofs, and symbolic notation for Sets, so the investigation of how these translate into type theory is a source of future work. Whereas propositions make explicit all the relevant detail, and can be read by non-experts, proofs are incredibly diverse and will be incomprehensible to those without expertise.

A detailed analysis of this should be done if and when a proper translation corpus is built to account for some of the ways mathematicians articulate these rules, as well as when and how mathematicians discuss sets, symbolically and otherwise. To create translation with "real" natural language is likely not to be very effective or interesting without a lot of evidence about how mathematicians speak and write.

Agda

Agda is an attempt to faithfully formalize Martin-Löf's intensional type theory [4]. Referencing our previous distinction, one can think of Martin-Löf's original work as a specification, and Agda as one possible implementation.

Agda is a functionaly programming language which, through an interactive environment, allows one to iteratively apply rules and develop constructive mathematics. It's current incarnation, Agda2 (but just called Agda), was preceded by ALF, Cayenne, and Alfa, and the Agda1. On top of the basic MLTT, Agda incorporates dependent records, inductive definitions, pattern matching, a versatile module system, and a myriad of other bells and whistles which are of interest generally and in various states of development but not relevant to this work.

For our purposes, we will only look at what can in some sense be seen as the kernel of Agda. Developing a full-blown GF grammar to incorporate more advanced Agda features would require efforts beyond the scope of this work.

Agda's purpose is to manifest the propositions-as-types paradigm in a practical and useable programming language. And while there are still many reasons one may wish to use other programming languages, or just pen and paper to do her work, there is a sense of purity one gets when writing Agda code. There are many good resources for learning Agda [1] [8] [2] [9] so we'll only give a cursory overview of what is relevant for this thesis, with a particular emphasis on the syntax.

To give a brief overview of the syntax Agda uses for judgements, namely $T : Set$ means T is a Type, $t : T$ means a term t has type T, and $t = t'$ means t is defined to be equal to t'. Let's compare it to those keywords ubiquitous in mathematics Figure 4, and show how those are represented in Agda directly below. Warrick

- Axiom
- Definition
- Lemma
- Theorem
- Proof
- Corollary
- Example

Figure 4: FOL vs MLTT

```
data inductiveType : Set where --Formation Rule
  constructr : inductiveType --Introduction Rules
  constructr' : inductiveType

postulate -- Axiom
  axiom : A

definition : stuff → Set --Definition
definition s = definition-body

theorem : T -- Theorem Statement
theorem = proofNeedingLemma lemma -- Proof
  where
    lemma : L -- Lemma Statement
    lemma = proof

corollary : corollaryStuff → C
corollary coro-term = theorem coro-term

example : E -- Example Statement
example = proof
```

Formation rules, are given by the first line of the data declaration, followed by some number of constructors which correspond to the introduction forms of the type being defined.

Therefore, to define a type Booleans, $\mathbb{B}$, we present for the formation rule

$$1[]\vdash \mathbb{B} : \text{type}$$

along with two introduction rules for the bits,

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

Agda's allows us to succintly put these together as

```
data 𝔹 : Set where
  true : 𝔹
  false : 𝔹
```

Now we can define our first type, term judgement pair, and define, for instance, the Boolean or, ∨. We detail the definition which is just a result of the pattern match Agda performs when working interactively via holes in the emacs mode, and that once one plays around with it, one recognizes both the beauty and elegance in how Agda facilitates programming. The colon reresents the judgmenet that ∨ is a type, whereas the equality symbol denotes the fact that ∨ is computationally equal to the subsequent expression over the given inputs. Once one has made this equality judgement, agda can normalize the definitionally equal terms to the same normal form when defining subsequent judgements.

The underscore denotes the placement of the arguement. We see the $_{\vee}constructorallows formorenu$

```
_∨_ : 𝔹 → 𝔹 → 𝔹
true ∨ b     = true
false ∨ true = true
false ∨ false = false
```

As the elimination forms are deriveable from the introduction rules, the computation rules can then be extracted by via the harmonious relationship between the introduction and elmination forms [6]. Agda's pattern matching is equivalent to the deriveable dependently typed elimination forms [3], and one can simply pattern match on a boolean, producing multiple lines for each constructor of the variable's type, to extract the classic recursion principle for Booleans.

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

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

The Agda code reflects this, and we see the first use of parametric polymorphism, namely, that we can extract a member of some arbtitrary type `A` from a boolean value given two possibly equal values members of `A`.

This polymorphism therefore allows one to implement simple programs like the boolean not, , via the eliminator. More interestingly, one can work with functionals, or higher order functions which take functions as arguements and return functions as well. We also notice in `functionalExample` below that one can work directly with lambda's if the typechecker infers a function type for a hole.

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

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

This simple example

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

ℕrec : {X : Set} -> (ℕ -> X -> X) -> X -> ℕ -> X
ℕrec f x zero = x
ℕrec f x (suc n) = f n (ℕrec f x n)

-- data List (A : Type) : Type where

-- data Vector :


-- \begin{code}

-- Type : Set₁
-- Type = Set

-- \end{code}
```

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 $L^{E}nglish, L^{B}antu, L^{S}wedish for L^{E}nglsih_{A}mericanvs L^{E}nglsih_{E}nglishor L$

One could also further elaborate these $L^{E}nglish_0 L^{E}nglish_1 tovaryingdegrees of granularity, like L^{E}nglis$

$L_0 < L_1 -> L_E nglish_0 < L_E nglish_1$

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

$Logic_{C}ore^{E}nglish Logic_{E}xtended^{E}nglish$

whereas in the PL domain

$Logic_{C}ore^{A}gda Logic_{E}xtended^{A}gda maycollapseatcertainpoints, butalsoinsomewaysextendbeyondourLangu$

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

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

Mathematics$^{E}nglishI--Somethingabout$

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 ques-

tion 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$_{D}etCar_{C}N\`which shouldlinearizeto\`"thecar"\`inEnglish,\`bilen\`inS$

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

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

```
 abstract Arith = {

flags startcat = Exp ;

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

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

}
```

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

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

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

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

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

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

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

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

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

Here is the first pass at an English linearization :

```
 lincat Exp = Str ;

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

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

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

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

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

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...

Natural Language and Mathematics

References

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

[2] Bove, A., Dybjer, P., & Norell, U. (2009). A brief overview of agda – a functional language with dependent types. In S. Berghofer, T. Nipkow, C. Urban, & M. Wenzel (Eds.), *Theorem Proving in Higher Order Logics* (pp. 73–78). Berlin, Heidelberg: Springer Berlin Heidelberg.

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

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

[5] Martin-Löf, P. (1996). On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1), 11–60.

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

[7] 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.

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

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

Appendices