Modeling Formal Languages in GF

On the Grammar of Proof

Warrick Macmillan

7^{th} August 2021

One Remark

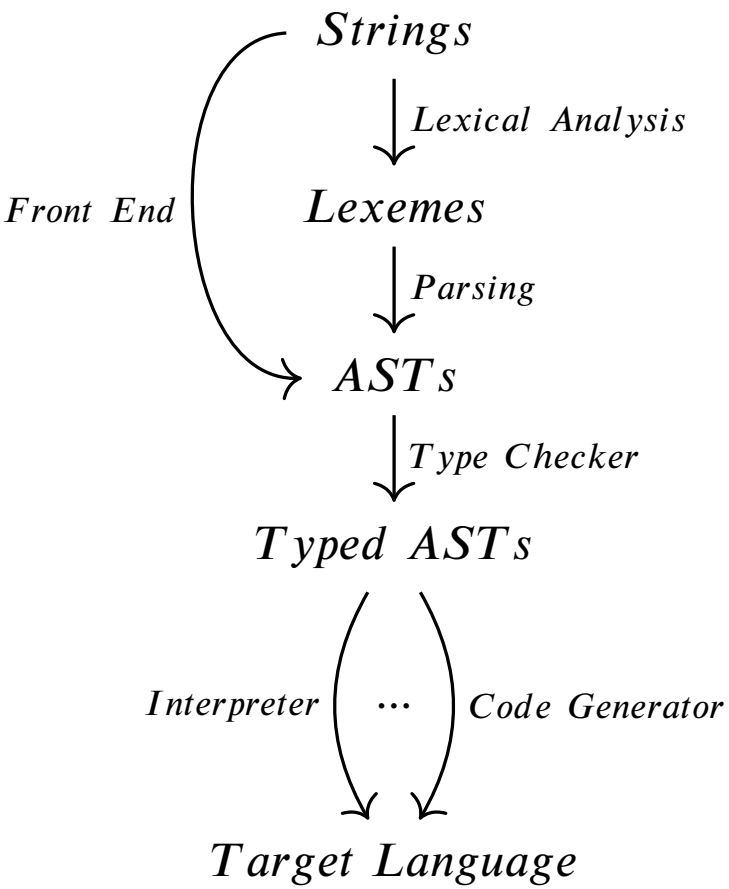
I want feedback on this talk, please

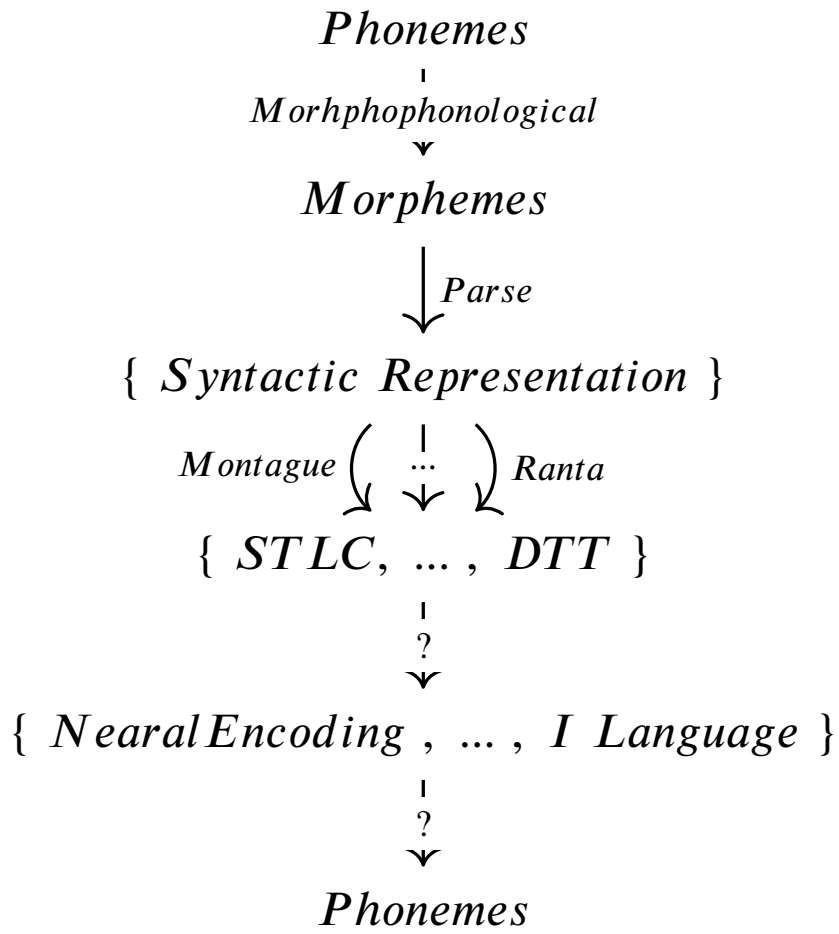
Table of Contents

Abbreviations

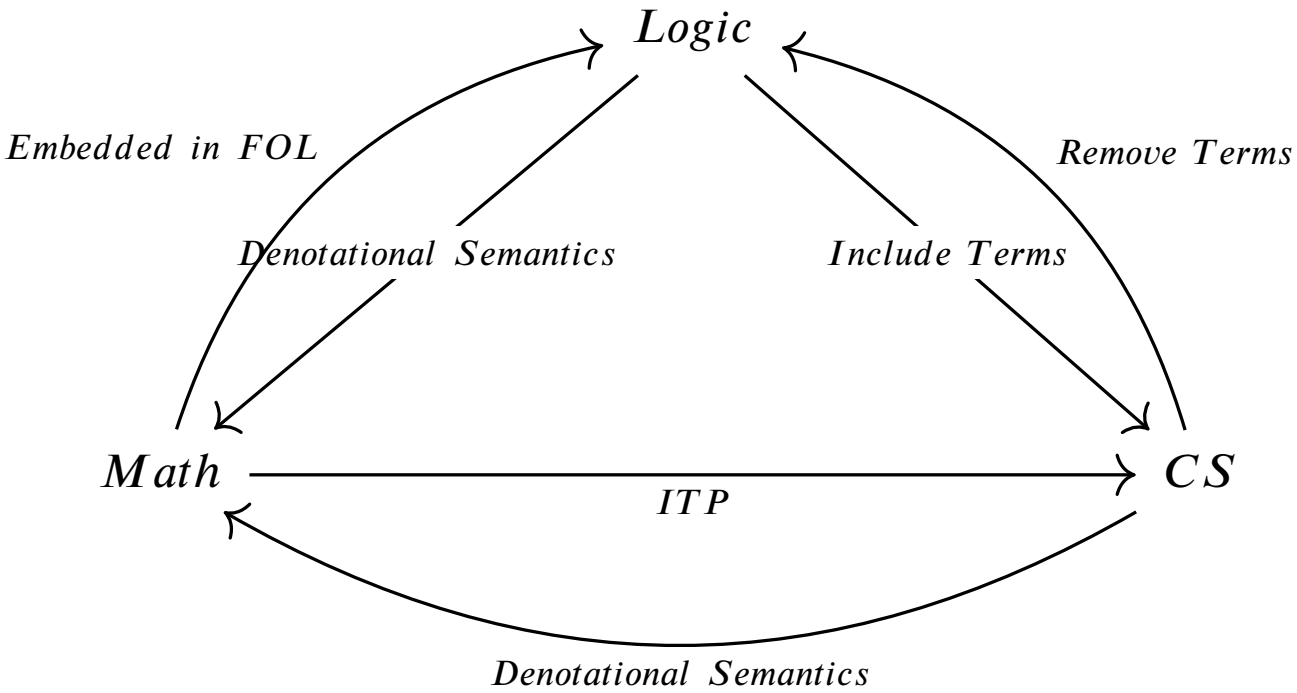
- **TT** : Type Theory
- **MLTT** : Martin-Löf Type Theory
- **MLTT** : Homotopy Type Theory
- **CTT** : Cubical Type Theory
- **NL** : Natural Language
- **PL** : Programming Language
- **GF** : Grammatical Framework
- **PGF** : Portable Grammar Format
- **ITP** : Interactive Theorem Prover
- **FOL** : First Order Logic
- **BNF** : Backus-Naur form
- **CADE** : Conference on Automated Deduction
- **HOL** : Higher Order Logic
- **HIT** : Higher Inductive Type
- **GADT** : Generalized Algebraic Data Type
- **RGL** : Resource Grammar Library

Abstraction Ladders





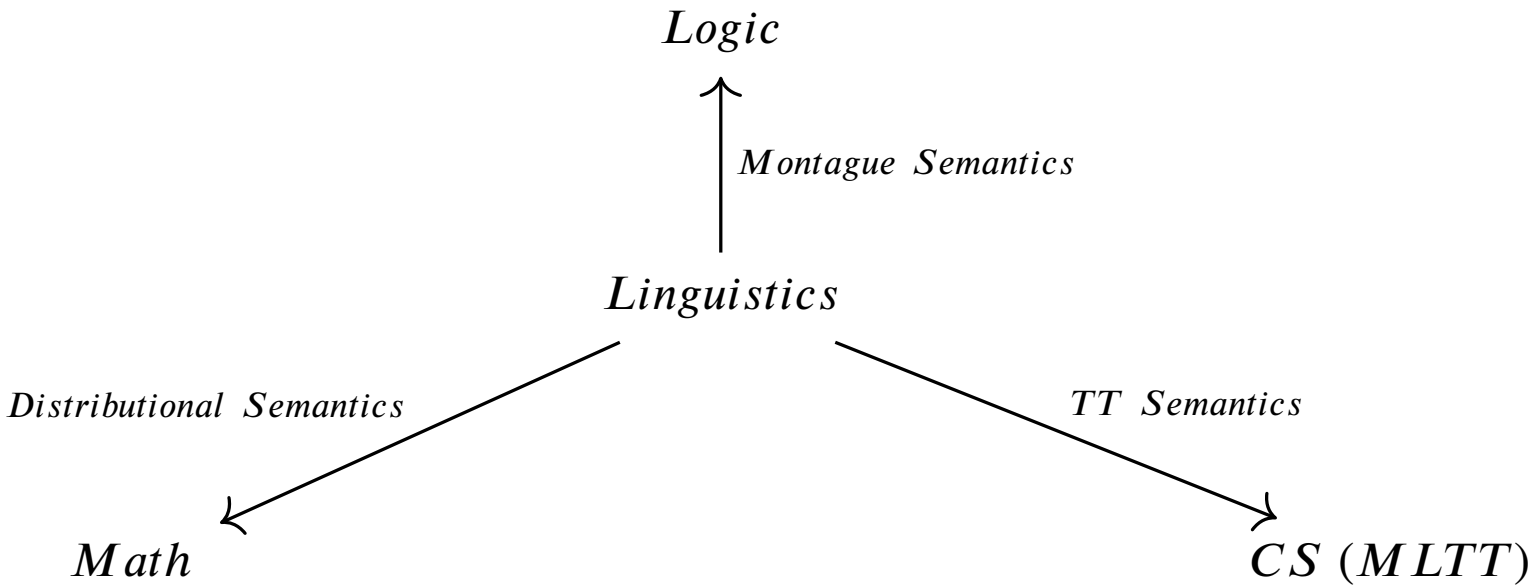
Computational Trinitarianism



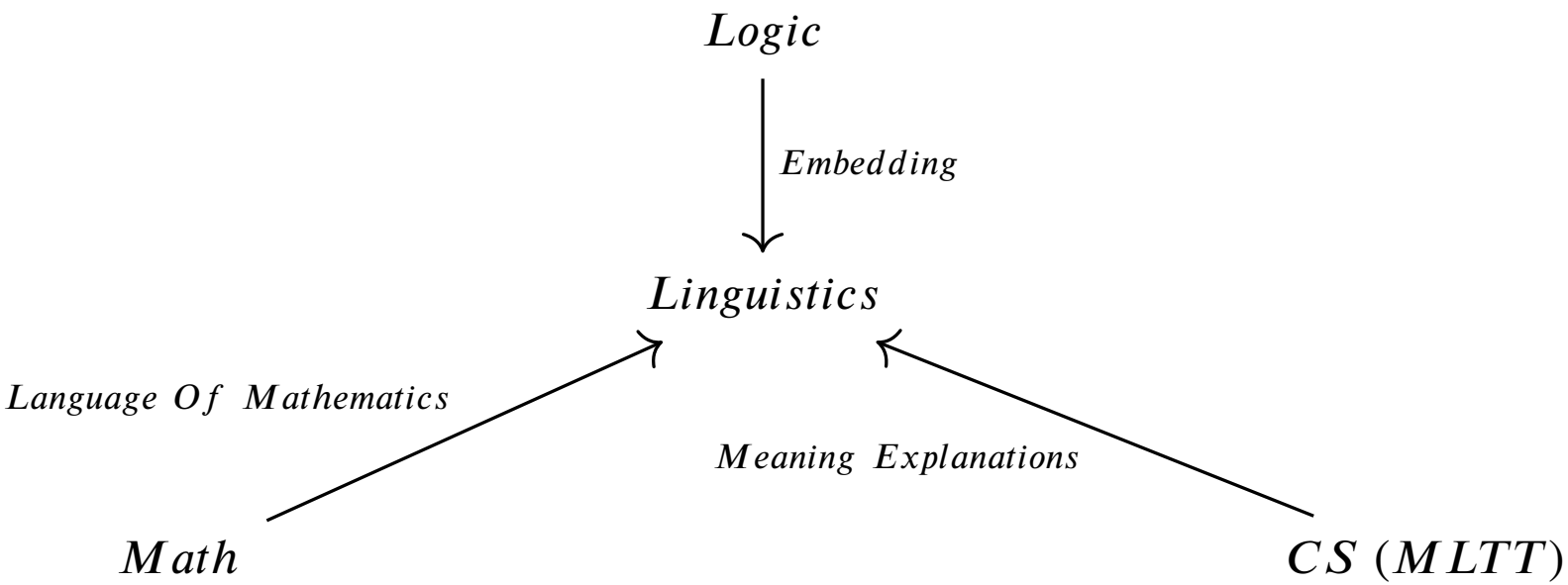
Interpretation of Language

Observation

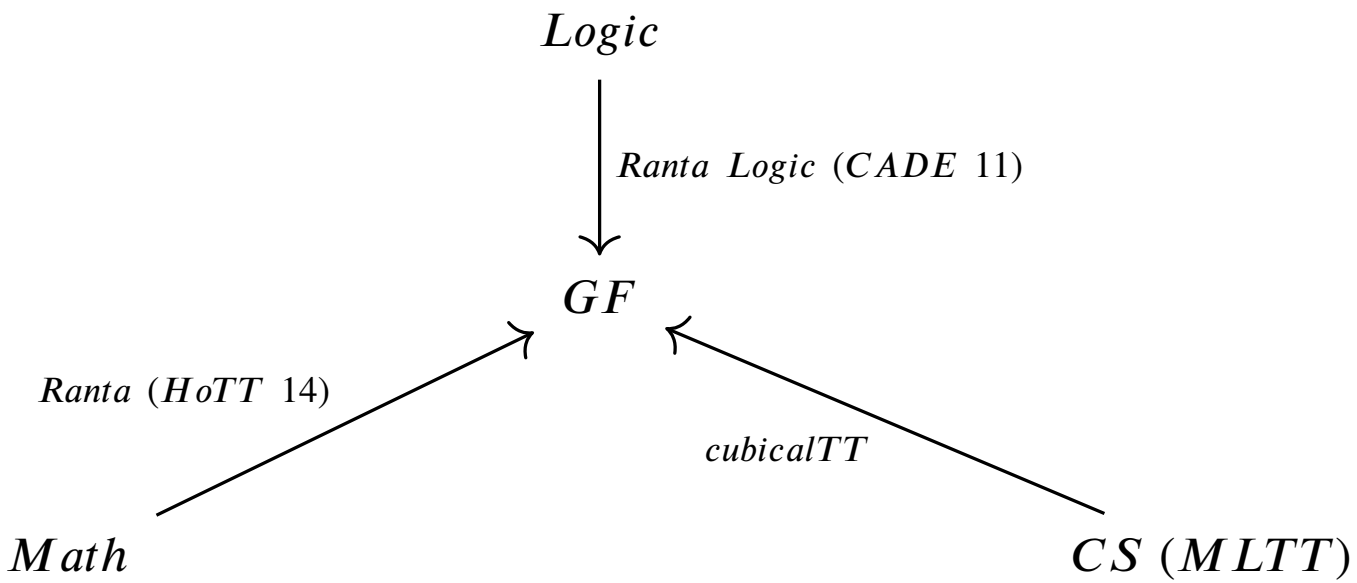
Only semantic interpretations in these domains. One may decide on syntactic, pragmatic, or other paradigms to view this through.



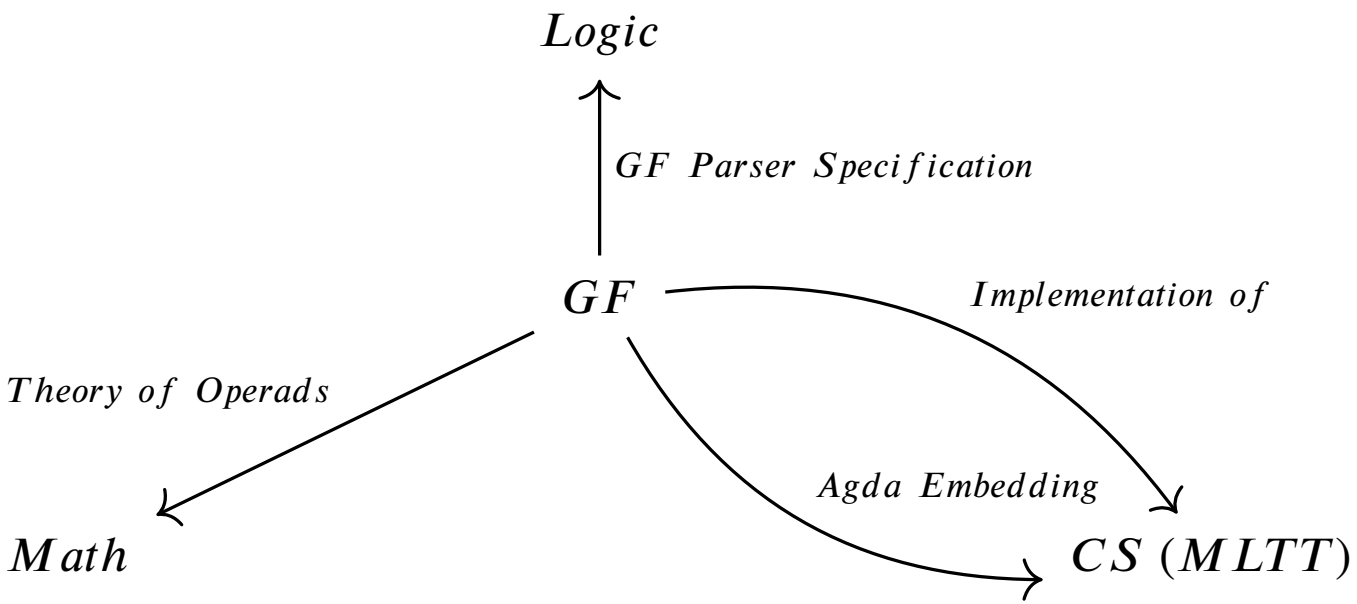
Trinitarian Linguistics



Trinitarian Grammars



Models of GF


Logic
GF Parser Specification
GF
Implementation of
Theory of Operads
Agda Embedding
Math
CS (MLTT)

Remarks

Concrete syntax is, in some sense, where programming meets psychology.

(Robert Harper, OPLSS)

- Trinitarian doctrine is in the “formal” space
- Trinitarianism + linguistics is partially informal and very underexplored
- Introduces many philosophical concerns
- Perhaps a rereading of Wittgenstein should take place in this context

Predecessors

- Frege : Formal proof, predicate logic
- Russel : Invents TT to resolve his paradox
- Brouwer : Constructivism
- Heyting, Kolmogorov, Church, Gödel, Kleene, Bishop, ...

Mathematical logic and the relation between logic and mathematics have been interpreted in at least three different ways:

i. mathematical logic as symbolic logic, or logic using mathematical symbolism;
ii. mathematical logic as foundations (or philosophy) of mathematics;
iii. mathematical logic as logic studied by mathematical methods, as a branch of mathematics.

We shall here mainly be interested in mathematical logic in the second sense. What we shall do is also mathematical logic in the first sense, but certainly not in the third.

(Per Martin-Löf, Padua Italy, June 1980)

Props vs Types

First Order Logic

- $\forall$
- $\exists$
- $\supset$
- $\wedge$
- $\vee$
- $\neg$
- $\top$
- $\bot$
- $=$

Dependent Type Theory

- Π
- Σ
- $\rightarrow$
- $\times$
- $+$
- $\neg$
- $\top$
- $\bot$
- $\equiv$

Sets vs Types

Sets

- $\mathbb{N}$
- $\mathbb{N} \times \mathbb{N}$
- $\mathbb{N} \to \mathbb{N}$
- $\{x | P(x)\}$
- $\varnothing$
- ?
- $\cup$
- ?

Types

- Nat
- $Nat \times Nat$
- $Nat \to Nat$
- $\Sigma x : \_.P(x)$
- $\bot$
- $\top$
- ?
- U_1

Sets

- 1
- $(1, 0)$

Programs

- $suc\ \ zero$
- $(suc\ \ zero, zero)$

Judgments

Type Theoretic Judgments

- 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

Mathematical Judgments

- P is a proposition
- P is true

- Notice that judgmental equality is uniquely type theoretic
- Judgments in type theory are decidable
- Truth (inhabitation) is not decidable
- More exotic judgments are available in TT, i.e. P is possible.

TT vs classical FOL

- The rules of the types make explicit that they are not equivalent to those of classical FOL
- An existential assertion in type theory requires data
- Excluded middle and double negation are not admitted in MLTT
- To be *not unhappy* is clearly of a different meaning than to be *happy*.
- This makes our approach to general translation of non-constructive mathematics *impossible*... at least such that it type-checks
- perhaps this is possible for other TTs, like those based of HOL

Other issues

- One doesn't define logics and type systems in mathematics (e.g. metamathematics)
- Encoding concepts like rational and real numbers in TT are difficult
- Sets are just one way of encoding mathematics
- Already category theorists and set theorists are at odds. Think small and large categories, higher categories, simplicial, cubical, globular, ... enrichment, etc.
- Intensional TT comes with two distinct notions of equality : judgmental (definitional, computational) and propositional

Donkey Anaphora

- Interpret the sentence “every man who owns a donkey beats it” in MLTT via the following judgment :

$$\Pi z : (\Sigma x : man.\ \Sigma y : donkey.\ owns(x, y)).\ beats(\pi_1 z, \pi_1(\pi_2 z))$$

- We judge $\vdash man$:type and $\vdash donkey$:type.
- type really denotes a universe

HoTT

- Homotopy Type Theory is an all out coming to terms with what equality is in type theory
- Equality is perhaps the most confusing detail for mathematicians learning TT
- Propositional equality can be iterated
- given a type A, we can form the homotopy $p =_{x=_A y} q$ with endpoints p and q inhabiting the path space $x =_A y$.
- Univalence axiom : Equivalence is equivalent to equality
- Allows one to admit a topological interpretation of types
- Has led to HITs, where constructors can include equality types.

Interpretations of $t : \tau$

- Set theory : n is an element of $\mathbb{N}$
- Type theory : n is a term of type $\mathbb{N}$
- Homotopy theory : n is a point in the space $\mathbb{N}$
- Category theory : n is an arrow between the object $\mathbb{N}$ and itself
- Logic : n is a proof of the proposition $\mathbb{N}$ –broken

What is Agda?

- Implementation of MLTT
- Logical Framework
- Think a kernel with Π and Σ (think $\forall$ and $\exists$, respectively)
- Interactive proof development environment
- Inductive types, modules, pattern matching, & more
- Large standard library, many other libraries
- Universe Hierarchy (not present classically)
- Tons of experimental stuff
  sized types, coinduction, tactics, etc.

Math Keywords in Agda

Mathematical Declarations

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

```
postulate -- Axiom
  axiom : A

definition : stuff → Set
definition s = definition-body

theorem : T -- Theorem Statement
theorem = proof -- Proof

lemma : L -- Lemma Statement
lemma = proof

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

Twin Prime Conjecture

Definition

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

Alternatively, we may state it as follows :

Definition

A *twin prime* is a prime that has a prime gap of two.

Definition

A *prime gap* is the difference between two successive prime numbers.

Theorem

There are infinitely many twin primes.

Twin Prime Conjecture in Agda

```
is-prime : ℕ → Set
is-prime n =
  (n ≥ 2) ×
  ((x y : ℕ) → x * y ≡ n → (x ≡ 1) + (x ≡ n))

twin-prime-conjecture : Set
twin-prime-conjecture = (n : ℕ) → Σ[ p ∈ ℕ ] (p ≥ n)
  × is-prime p
  × is-prime (p + 2)
```

A proof is what makes a judgment evident

(Per Martin-Löf, On the Meanings of the Logical Constants And the Justifications of the Logical Laws, 1983)

...there is a considerable gap between what mathematicians claim is true and what they believe, and this mismatch causes a number of serious linguistic problems.

(Mohan Ganesalingam)

Comparing Formal and Informal Proofs

Category	Formal Proof	Informal Proof
Audience	Agda (and Human)	Human
Translation	Compiler	Human
Objectivity	Objective	Subjective
Historical	20th Century	<= Euclid
Orientation	Syntax	Semantics
Inferability	Complete	Domain Expertise Necessary
Verification	PL Designer	Human
Ambiguity	Unambiguous	Ambiguous

Specification and Implementation

My Takeaway

An informal proof is a specification and a formal proof is an implementation.

- Historically, we think of semantics preceding
  i.e. the abstract notion of a circle preceded its “algebraic” understanding
- Syntax oriented thinking is very popular in the CS tradition
- Sometimes given an expressive enough type, the programs seem to *write themselves*
- Both syntactic and semantic thinking are necessary in the end, especially for big proofs

Formal Abstracts vision

The Formal Abstracts (FAbstracts) project will establish a formal abstract service that will express the results of mathematical publications in a computer-readable form that captures the semantic content of publications

(The Formal Abstracts Project)

A Smaller Problem

Propositions or theorem statements in natural language are *intentionally unambiguous*

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

(Avigad, Mathematics and language, 2015)

Syntactic Completeness

Given an natural language expression that a mathematician understands, does the GF grammar emit a well-formed and well-typed expression in the target logic or programming language?

- The intended meaning manifests differently for different people - there may be no absolute intention
- Many possible syntaxes beyond just alpha conversion
- The details will have to change depending on the tools and machines available
- Begin with math as developed in an ITP

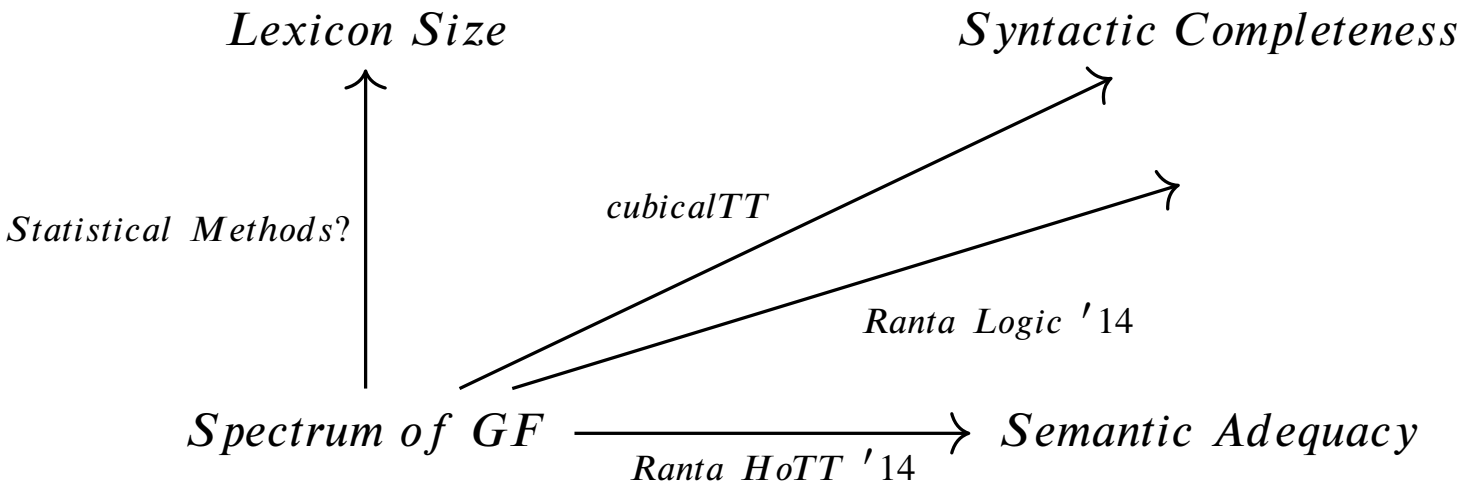
Semantic Adequacy

Semantic Adequacy

Given a well formed syntactic expression in, for instance, Agda, one can ask if the resulting NL expression generated by GF is *semantically adequate* and coherent to a "fluent speaker"

- Think expressive adequacy in logic
- In dispute among mathematicians
- Kind of like a Turing Test for machine proof translation
- Depends on historical developments as our language develops, both internally and more interestingly, as a society
- Ganesalingam calls a related notion *adaptivity*, the phenomenon whereby "the grammar of an individual mathematician changes as definitions are encountered"
- Grammar should just be a starting point for mathematical literacy generally
- Begin with *real mathematics* empirically

The Dimensions of Grammar



Choosing Sides: Concrete vs Abstract

Abstract

- Capture more semantic content from a NL perspective and more "freedom of expression"
- Simpler makes easier to work with in Haskell - really large GADTs after PGF embedding

Concrete

- Simpler semantic space, easier for designing PL for it eliminates ambiguity
- Focus on this may be more feasible for real implementation

As will be seen, a smaller abstract syntax leans towards syntactic completeness, but it comes at the cost of larger, more complex *lincats*.

A Grammar for Propositions

- Ranta, *Translating between Language and Logic : What Is Easy and What Is Difficult*
- Application grammar for logical propositions defined over some mathematical domain (integral arithmetic, euclidean geometry)
- Doesn't cover proofs
- Uses a core and extended set of categories
- Translates between them via PGF extension

Ideas

- One can use theorems from FOL to simplify the NL expression (semantics preserving normalizer)
- Serves as a possible basis for other logics

Logic Grammar Properties

Core

- Minimal necessary expressivity
- Syntactically complete
- Ambiguous parses : Catalan explosion with n conjunctions
- Needs to be evaluated to remain legible

Core Example

- $\forall x(Nat(x) \supset Even(x) \vee Odd(x))$
- "for all x, if x is a natural number then x is even or x is odd"

Semantically Inadequate Example

"is it the case that the sum of 3 and the sum of 4 and 10 is prime and 9999 is odd"

Core Syntax

construction	**symbolic**	**verbal**
negation	$\sim P$	*it is not the case that P*
conjunction	$P \,\&\, Q$	*P and Q*
disjunction	$P \vee Q$	*P or Q*
implication	$P \supset Q$	*if P then Q*
universal quantification	$(\forall x)P$	*for all x, P*
existential quantification	$(\exists x)P$	*there exists an x such that P*

Extended Properties

Extended

- Much more expressive
- Semantically adequate
- increases both number of categories and functions
- also need for more complicated linearization categories
- complex PGF backend to keep this syntactically complete
- questions about scalability

Extended Example

- $\forall x(Nat(x) \supset Even(x) \vee Odd(x))$
- "every natural number is even or odd"

Extended Syntax

construction	**symbolic**	**verbal (example)**
atom negation	$\overline{A}$	*x is not even*
conjunction of proposition list	$\&[P_1, \ldots, P_n]$	*P, Q and R*
conjunction of predicate list	$\&[F_1, \ldots, F_n]$	*even and odd*
conjunction of term list	$\&[a_1, \ldots, a_n]$	*x and y*
bounded quantification	$(\forall\, x_1, \ldots, x_n : K)P$	*for all numbers x and y, P*
in-situ quantification	$F(\forall\, K)$	*every number is even*
one-place predication	$F^1(x)$	*x is even*
two-place predication	$F^2(x, y)$	*x is equal to y*
reflexive predication	$\mathit{Refl}(F^2)(x)$	*x is equal to itself*
modified predicate	$\mathit{Mod}(K, F)(x)$	*x is an even number*

Translating to Core

$[\![-]\!] : Extended \to Core$

- Core syntax as a model for extended
- relatively simple in the sense that it is be deterministic
- More or less uses the same logical structure from the "standard view"

Translating to Extended

$[\![-]\!] : Core \to Extended$

- Difficult problem, infeasible at scale
- Obscures formal detail, but is more intuitive

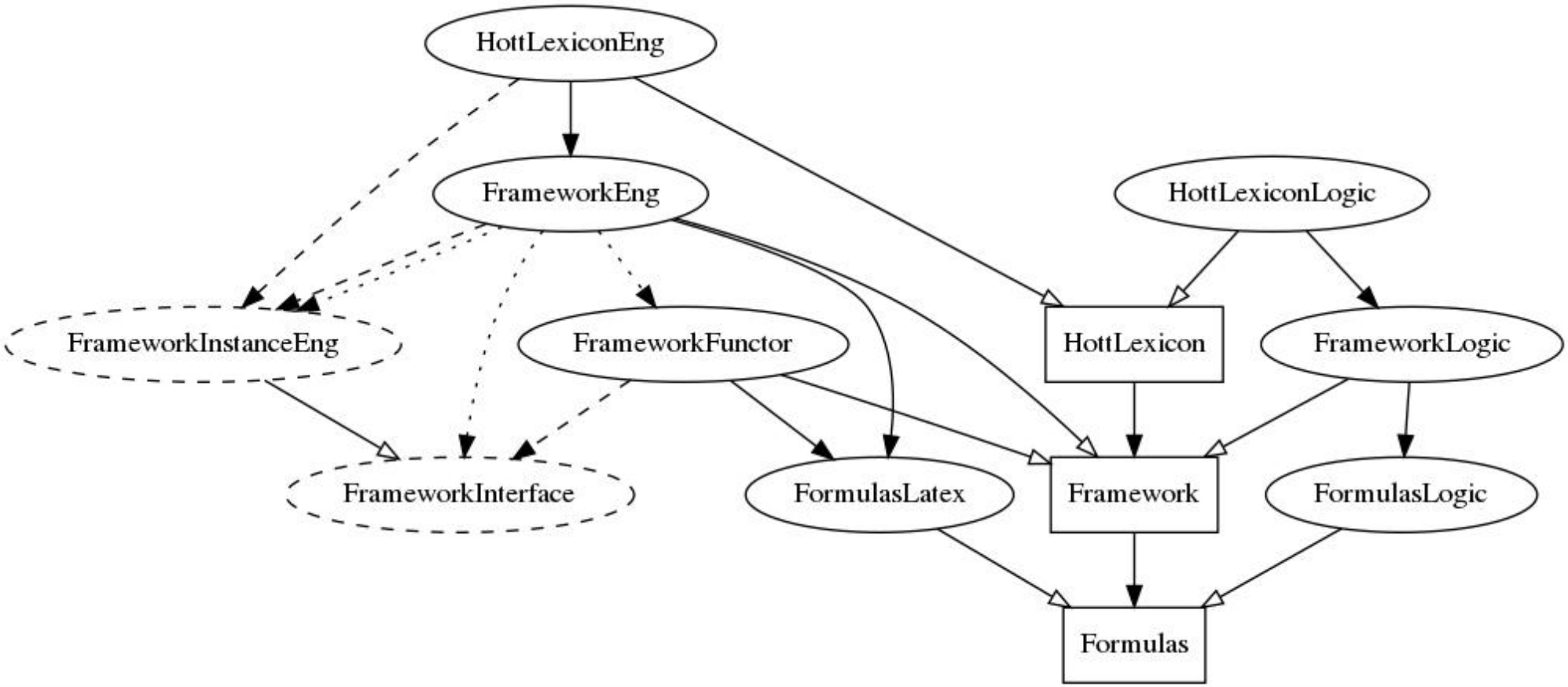
Conversion

- 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
  $x\ is\ a\ number\ and\ x\ is\ even \mapsto x\ is\ an\ even\ number$

An Unpublished Talk at the Stockholm University Math Seminar

- Ranta, *Translating Homotopy Type Theory in Grammatical Framework*, Fall 2014
- Case study for text of *real* mathematics writing
- Purely GF translation
- Complex abstract structure including latex, metadocument structure, etc.
- HoTT specific lexicon
- Includes proofs, but AST not similar to PL
- Includes both expressions and propositions (breaks Curry-Howard)
- *Semantically adequate*
- *Syntactically incomplete*

HoTT Grammar Modules



Framework.gf

```
cat
  Paragraph ; -- definition, theorem, etc
  Definition ; -- definition of a new concept
  Assumption ; -- assumption in a proof -- let ...
  [Assumption]{1} ;  -- list of assumptions in one sentence
  Conclusion ; -- conclusion in a proof -- thus P
  Prop ; -- proposition,sentence or formula, A is contractible
  Sort ; -- set, type, etc corresponding to a common noun
  Ind ; -- individual element corresponding to a singular term
  Fun ; -- function with individual value
  Pred ; -- predicate: function with proposition value
  [Ind] ; -- list of individual expressions -- 1, 2 and 3
  UnivPhrase ; -- universal noun phrase -- for all x,y : A
  ConclusionPhrase ; -- conclusion word -- hence
  Label ; -- name/number of definition, theorem, etc
  Title ; -- title for theorem, definition, etc
```

Formulas.gf

```
cat
  Exp ;           -- formal expression
                  -- x + y = z
  Var ;           -- variable
                  -- x
  [Var]{1} ;      -- list of variables
                  -- x, y, z
  [Exp]{1} ;      -- list of expressions
                  -- 1_{A}, A
  Format ;        -- line other than content
                  -- \begin{document}
  MathEnv ;       -- math environment
                  -- $ ... $
```

Contractability

Comments

- We work with the definition of contractability, the notion that a type (or space) is actually a point, i.e. up to equality, there is only object
- We show the (i) rendered latex, (ii) pidgin Agda syntax after some concrete modifications, and (iii) the Agda code

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

```
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)
```

Equivalence

- We also show that a notion of a map being an equivalence, that of the every element in the codomain having a contractible image (think bijection)
- notice the error in the Pidgin case

Definition: A map $f : A \to B$ is an equivalence, if for all $y : B$, its fiber, $\{x : A \mid f\,x = y\}$, 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)
```

GF as a programming language parser

- How can a PL parser be bootstrapped to linearize to natural language ?
- *Syntactically adequate* out the box
- All precedence information at the concrete level
- There exists many existing programs to test on, hopefully less overhead
- In dependently typed language, almost everything is an expression
- No distinguishing syntactically between types and terms
- This makes the *abstract syntax* easy to write for the PL (in GF) but difficult for the NL
- Caution : This also makes it difficult to implement an ITP

Basic Categories

```
cat
  Exp ;
  Var ;
  Tele ;
  LTele ;
  [Var]{1} ;
  [Tele]{2} ;
  [LTele]{2} ;
  Decl ;

  --cat [C] {n}
  -- =
  --cat ListC ;
  --fun BaseC : C -> ...-> C -> ListC ; -- n C 's
  --fun ConsC : C -> ListC -> ListC
```

Dependent Lambda Calculus with Propositional Equality

```
fun
  Dtype , Dterm : Var -> Exp -> Decl ;

  Earr : Exp -> Exp -> Exp ;
  Epi : [Tele] -> Exp -> Exp ;
  Eid : Exp -> Exp -> Exp -> Exp ; -- for type
  Eid2 : Exp -> Exp -> Exp ;
  Enat , Euni : Exp ;

  Evar : Var -> Exp ;
  Elam : [LTele] -> Exp -> Exp ;
  Eapp : Exp -> Exp -> Exp ;

  Erefl : Exp ;
  Eidind : Exp -> Exp -> Exp -> Exp -> Exp -> Exp ;

  Ezer : Exp ;
  Esuc : Exp -> Exp ;
  EsucEta : Exp ;
  Enatind : Exp -> Exp -> Exp -> Exp -> Exp ;
```

Natural Numbers in Agda

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

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

two : ℕ'
two = suc' (suc' zero')

double : ℕ' → ℕ'
double zero' = zero'
double (suc' n) = suc' (suc' (double n))

four : ℕ'
four = double two

_+'_ : ℕ' → ℕ' → ℕ'
zero' +' y = y
suc' x +' y = suc' (x +' y)

2+2=4 : (two +' two) ≡ four
2+2=4 = refl
```

Proof Associativity of Natural Numbers from *Software Foundations*

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

Pattern Matching Proofs in Agda

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

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

Natural Number Induction Principle (Elimination Rule) in Agda

```
natind : {C : ℕ -> Set} -> -- predicate
         C zero ->         -- base case
         ((n : ℕ) -> C n -> C (suc n)) -> --IH
         (n : ℕ) -> C n
natind base step zero = base
natind base step (suc n) = step n (natind base step n)
```

Agda Associativity Proof

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

Pidgin cubicalTT Associativity Proof

```
p -lang=LHask "
  \\ ( x y z : nat ) ->
  natind
    (\\ (f : nat) ->
     ((plus f (plus y z)) == (plus (plus f y) z)))
    refl
    (
     \\ ( f : nat ) ->
     \\ (g :
      ((plus f (plus y z)) == (plus (plus f y) z))
        )
      -> ap suc g
    )
    x" | l
0 msec
```

Pidgin NL Associativity Proof

```
L>
  function taking x , y z in the natural numbers
  to
  We proceed by induction over x .
  We therefore wish to prove : function taking f,
    in the natural numbers to apply apply plus to
    f to apply apply plus to y to z is equal
    to apply apply plus to apply apply plus to f
    to y to z .
  In the base case, suppose x equals zero.
    we know this by reflexivity .
  In the inductive case,
    suppose x is the successor.
    Then one has one has function taking f ,
      in the natural numbers to function
      taking g , in apply apply plus to f
      to apply apply plus to y to z is equal to
      apply apply plus to apply apply plus to f
      to y to z to apply ap to the successor
      of g.
```

cubicalTT Grammar

Idea

How can we build a GF PL parser that unambiguously parses an existing PL?

HoTT Book

- Canonical source for learning HoTT
- Concurrently formalized as it was written
- Multiple Agda and Coq libraries mirroring the text
- Open source, latex available
- Many authors suggest it's language should be more neutral

My Library

A custom Agda Library was written additionally for this project to mirror the exact HoTT structure. It was an exercise in formalization by hand, and can hopefully soon serve as a test corpus.

cubicalTT

- Existing grammars for experimental PLs, like cubicalTT
- Treats propositional equality natively
- Univalence becomes a theorem
- *Computational Higher Type Theory*
- Experimental & a hot research topic
- Explicit *split* instead of pattern matching
- Only one universe, inconsistent
- Less pretty syntax than Agda, but front-end parser is in BNFC
- cubicalTT discontinued, now Cubical Agda

BNFC → GF

- Backus-Naur form Converter (BNFC)
- Parser generator for PL's
- A cousin of GF for that combines the concrete and abstract into a single grammar file.
- The BNFC functions are quite similar, but resembles classic BNF
- Precedence is native in BNFC
- We have help from Formal.gf in the RGL.

```
Lam. Exp  ::= "\\" [PTele] "->" Exp ;
Fun. Exp1 ::= Exp2 "->" Exp1 ;
App. Exp2 ::= Exp2 Exp3 ;
```

```
fun
  Lam : [Tele] -> Exp -> Exp ;
  Fun : Exp -> Exp -> Exp ;
  App : Exp -> Exp -> Exp ;
lin
  Lam pt e = mkPrec 0 ("\\" ++ pt ++ "->" ++ usePrec 0 e) ;
  Fun = infixr 1 "->" ;
  App = infixl 2 "" ;
```

Symmetry of Equality in Agda

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

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

Lemma

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

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

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

First proof.

Assume given $A : \mathcal{U}$, and let $D : \prod_{(x,y:A)}(x = y) \to \mathcal{U}$ be the type family defined by $D(x, y, p) \equiv (y = x)$. In other words, D is a function assigning to any $x, y : A$ and $p : x = y$ a type, namely the type $y = x$. Then we have an element

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

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

Difficulties in NL Generation

NL Generation

- Properly deal with all the *applys* in the NL case
- Scale to more interesting facts about, number theory or HoTT
- Symbolic module extension
- Modularize the grammar
- GF latex package in RGL, ideally

Difficulties in PL Direction

PL Pains

- Length of list issues, how to get it to properly linearize to two sytnaxes with unique list idioms
- Pattern matching
- Implement inductive types, records, as many features of Agda as possible
- Port to Coq, Lean, or some other ITP
- Inference needed to determine type signature aritities in Agda, probably via abstract syntax, although possibly exponential number of functions n hidden arguments
- Precedence analysis

Mathematicians Objections

- The syntax of programming languages is less elegant, harder to read, and often indecipherable - even to the original author given enough time
- Computers, programs, programming languages, and software have bugs
- Code deprecates over time, and old code often breaks with software updates
- Learning to code requires expertise in its own right, and mathematicians don’t have time to become experts in that, or so they believe
- Tedious details are necessary in machine implementations of proofs
- Martin-Löf type theory is an entirely different foundation of mathematics, and most mathematicians aren’t concerned with foundations
- Natural language proofs often explicitly keep track of proof states, or goals, whereas interactive proof development shields these when one simply witnesses the program syntax
- There is very little *new* mathematics which has been developed in theorem provers, in the sense of mainstream mathematical journals publishing original research that was developed exclusively in theorem provers
- Typecheckers are stubborn

Computer Scientist's Frustrations

- Vague language is prone to error in interpretation
- Details left to be deciphered by the reader often nontrivial, and possibly impossible to infer for non-experts
- No easy way to fork someone else's work, credit them without explicitly referencing papers and theorem number
- No online repository to look up, for instance, theorems proven by their type signature (think Hoogle)
- Many (mostly older) papers written in foreign languages without translations available
- Errors may not be caught until much later
- Peer review incredibly costly
- Type-checker is bookkeeper, mathematician must do this manually

Questions Going Forward

- How will the foundations of mathematics change, and how will this change reflect new ways of speaking and thinking about mathematics?
- What is GF's role in this problem? Are there theoretical or practical limitations?
- HITs are leading to new proofs of topological phenomenon, arguably more close to topological intuitions. How will these effect the ITP community at large?
- What about graphical proofs and graphical programming languages?

Future Work

Unified Grammar

- Goal : with the best of all three approaches taken above
- Objections : maybe there are just different grammars which need to be developed for more domain specific tasks in this area.
- Mirror both CADE ’11 and HoTT ’14 in cubicalTT with their respective qualities maximized and difficulties minimized

Linguistic Analysis

- Gangaselem’s work is both practical and philosophical, but just the tip of the iceberg
- Differences between NLs and PLs still ripe for exploration

“RGL” for Formal Language

- Krasimir : bigger grammar, begins resembling RGL
- We need to build a formal language RGL for GF
- Other ways of connecting it with external ITPs and PLs
- Possibly reinvigorate use of dependent types in GF

TODO

- Amass a domain corpus for comparison of specific concrete syntax
- Testing suite, framework, and methodology for GF CNLs, especially in the domain of propositions and types
- some work on GF itself, to allow for user defined categories (like Int and String for variables)
- More interface with the Agda (or other PL) communities
- Possible investigation of metaprogramming in GF (this may hit the parser complexity bottleneck)
- Mathematical theory of GF (specifically, a categorical semantics with ASTs being represented by operads). There is a presumed adjunction between GF Trees and Strings will us to understand GF more formally, i.e. prove theorems about the relation between concrete and abstract syntax, especially regarding expressivity, complexity, and scalability

Related Work

- NaProChe (Natural Language Proof Checking) : CNL with back-end solver which guarantees correctness
- Mizar : ITP with nl built in (based on set theory)
- NL interface for Coq
- Boxer system parses NL to FOL
- Ranta and Hallgren's work on NL inteface for Alfa
- Machine Learning : Szegedy, Josef Urban

Metaphors

We've covered many issues, about a very immature problem.

Math as Religion

- How one views the foundations of mathematics resembles religion
- All sides are convinced they are correct
- There are constant forks, reinterpretations, and heretics along the way

Actually, in some sense, this is a very tangible problem, and the forces are already at work to integrate these fields.

- Mathematician : Architect
- Type Theorist : Engineer
- Linguist : Scientist