Roadmap

Warrick Macmillan

February 14, 2021

1 Introduction

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

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

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

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

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

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

Definition 3 *Let G, H, be groups. A map $\phi : G \to H$ is called a* group homomorphism *if*

$$\phi(xy) = \phi(x)\phi(y) \text{ for all } x, y \in G$$

(Note that xy on the left is formed using the group operation in G, whilst the product $\phi(x)\phi(y)$ is formed using the group operation H.)

Definition 4 *Classically, a group is a monoid in which every element has an inverse (necessarily unique).*

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

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

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

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

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

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

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

While this example may not exemplify the power of Agda's type checker, it is of considerable interest to many. The typechecker has merely assured us that monoidHom, is a well-formed type. The type-checker is much more useful than is immediately evident: it delegates the work of verifying that a proof is correct, that is, the work of judging whether a term has a type, to the computer. While it's of practical concern is immediate to any exploited grad student grading papers late on a Sunday night, its theoretical concern has led to many recent developments in modern mathematics. Thomas Hales solution to the Kepler Conjecture was seen as unverifiable by those reviewing it. This led to Hales outsourcing the verification to Interactive Theorem provers HOL Light and Isabelle, during which led to many minor corrections in the original proof which were never spotted due to human oversight.

Fields Medalist Vladimir Voevodsky, had the experience of being told one day his proof of the Milnor conjecture was fatally flawed. Although the leak in the proof was patched, this experience of temporarily believing much of his life's work invalid led him to investigate proof assintants as a tool for future thought. Indeed, this proof verification error was a key event that led to the Univalent Foundations Project [1].

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

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

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

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

It is therefore natural for this thesis, which seeks

2 HoTT Proofs

2.1 Why HoTT for natural language?

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

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

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

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

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

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

Definition 5 *The formation rule says that given a type $A : \mathcal{U}$ and two elements $a, b : A$, we can form the type $(a =_A b) : \mathcal{U}$ in the same universe. The basic way to construct an element of $a = b$ is to know that a and b are the same. Thus, the introduction rule is a dependent function*

$$\mathsf{refl} : \prod_{a:A}(a =_A a)$$

*called **reflexivity**, which says that every element of A is equal to itself (in a specified way). We regard refl_a as being the constant path at the point a.*

We recapitulate this definition in Agda, and treat :

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

2.2 An introduction to equality

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

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

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

Padova lectures were later published in the ”Bibliopolis Book”, and went on to inspire the European (and Gothenburg in particular) approach to implementing proof assitants, whereas the extensionalists were primarily eminating from Robert Constable’s group at Cornell.

This tension has now been at least partially resolved, or at the very least clarified, by an insight Voevodsky was apparently most proud of : the introduction of h-levels. We’ll delegate these details for a later section, it is mentioned here to indicate that extensional type theory was really “set theory” in disguise, and the work over the past 10 years has elucidated the intensional and extensional positions.

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

infix 20 _≡_
```

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

infixr 50 _⁻¹
```

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

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

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

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

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

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

infixl 40 _•_

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

-- similairlymeans uniformly substitute the commuted expression throughout the proof.
iᵣ : {A : Set} {x y : A} (p : x ≡ y) → p ≡ p • r
iᵣ {A} {x} {y} p = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = p ≡ p • r
    d : (a : A) → D a a r
    d a = r
```

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

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

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

3 Goals and Challenges

4 Approach

References

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