

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 regarded 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 $h : G \rightarrow H$ 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 $\phi : G \rightarrow G'$ 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 $\phi : G \rightarrow H$ 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 in 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 linguistic usage. The inverse and identity are discarded, even though they are necessary data when

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

```

monoidHom
  (monoid' A  $\varepsilon$   $\bullet$  left-unit right-unit assoc carrier-set)
  (monoid' A1  $\varepsilon_1$   $\bullet_1$  left-unit1 right-unit1 assoc1 carrier-set1)
  f
= (m1 m2 : A) → f (m1 • m2) ≡ (f m1) •1 (f m2)

```

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 situations. 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 corollary of proof by contradiction, are much more amenable to formalization in systems with decidable 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

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 interdisciplinary, conceived and constructed by mathematicians, logicians, and computer scientists. It therefore is meant to interface with all these disciplines, 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

Let's start out by examining the inductive definition of the identity type.

```
module Id where

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

infix 20 _≡_

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

--data _≡'_ {A : Set} : A → A → Set where
```

```
-- r : (a : A) → a ≡ a
```

We present this statement 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*

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

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 .

```
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 $D : \prod_{(x,y:A)} (x = y) \rightarrow \mathcal{U}$ be the type family defined by $D(x, y, p) := (y = x)$. 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

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

Thus, the induction principle for identity types gives us an element $\text{ind}_{=_A}(D, d, x, y, p) : (y = x)$ for each $p : (x = y)$. We can now define the desired function $(-)^{-1}$ to be $\lambda p. \text{ind}_{=_A}(D, d, x, y, p)$, i.e. we set $p^{-1} := \text{ind}_{=_A}(D, d, x, y, p)$. The conversion rule [missing reference] gives $\text{refl}_x^{-1} \equiv \text{refl}_x$, 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 refl_x^{-1} to be simply refl_x . The general case then follows by the induction principle, and the conversion rule $\text{refl}_x^{-1} \equiv \text{refl}_x$ 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)
li : {A : Set} {x y : A} (p : x ≡ y) → p ≡ r • p
li {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

-- similarlymeans 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

$$(x = y) \rightarrow (y = x)$$

denoted $p \mapsto p^{-1}$, such that $\text{refl}_x^{-1} \equiv \text{refl}_x$ 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.