Completeness of Π

Siva Somayyajula

July 2017

Abstract

Π is a reversible programming language by Sabry et al. inspired by type-theoretic isomorphisms. We give a model for Π: a univalent universe of finite types in homotopy type theory. Using properties of *univalent fibrations*, the underlying concept of this model, we give formal proofs in Agda that programs in Π are complete with respect to this model. Additionally, we discuss this model and extensions to Π through the lens of synthetic homotopy theory.

Contents

1 Introduction

1.1 Reversibility

Reversibility is a paradigm in which computations and their effects may be reversed. This is prevalent in computing applications, giving rise to ad hoc implementations in both hardware and software alike. In particular, transactional databases operate on the basic concept that operations on data may be committed to memory or rolled back [11], and version control systems like `darcs` are based on *patch theory*, an algebra for file changes[1]. At the software level, this has motivated the development of general-purpose reversible programming languages.

Instead of relying on an operational model, the Π language by Sabry et al. begins with different foundations. To elaborate, a natural type-theoretic notion of reversibility is given by type isomorphisms i.e. lossless transformations over structured data. Thus, Π is a calculus for such isomorphisms, giving rise to a feature-complete reversible functional programming language [11]. To understand Π and its model, we give a brief introduction to the type theories we use to formalize them.

1.2 Type Theory

A type theory is a formal system for *types*, *terms*, and their computational interactions. A helpful analogy to understand type theory at first is to conceptualize types as sets and terms as their elements. Like set theory, type theories have rules governing *type formation* as there are axioms about set construction e.g. the axiom of pairing, but there are important distinctions. Whereas set theory makes set membership a proposition provable within the system, terms do not exist without an a priori notion of what type they belong to—one writes $a : A$ (pronounced "a inhabits A") to introduce a term a of type A [5]. As a result, terms are also called *inhabitants*, and we will use those terms (pun intended) interchangeably throughout the rest of the paper.

Perhaps the distinguishing feature of type theories are their explicit treatment of computation: computation rules dictate how terms reduce to values. To programming language theorists, type theories formally describe programming languages and computation rules are precisely the structured operational semantics. On the other hand, set theories have no such equivalent concept.

This emphasis on computation has several applications to computer science. First, the type systems of such programming languages as Haskell are based on certain type theories (specifically, System F). Aside from their utility in programming language design, sufficiently sophisticated type theories are suitable as alternative foundations of mathematics to set theory. In fact, Martin-Löf type theory (MLTT) is the basis of many programs aiming to formalize constructive mathematics. To understand how this is possible, recall that set theories consist of rules governing the behavior of sets as well as an underlying logic to express propositions and their truth. Thus, it remains to show that type theories, under the availability of certain type formers, are languages that can express the construction of arbitrary mathematical objects as well as encode propositions as types and act as deductive systems in their own right [6].

Thus, we will first give a brief introduction to MLTT in Agda, a programming language and proof assistant based on MLTT.

1.3 Martin-Löf Type Theory

Continuing the analogy that types are sets, the following table describes the set-theoretic analogue of each type former in MLTT. The syntax of the terms inhabiting these types are in almost one-to-one correspondence with classical mathematics, with caveats explained below [12].

type	set
U or `Type`	universal set
$\mathbb{0}$	$\varnothing$
$\mathbb{1}$	singleton
$\mathbb{N}$	Peano numbers
$A + B$	coproduct $A \sqcup B$
$A \times B$	$A \times B$
$A \to B$	function space B^A

The function type is perhaps the most novel type to mathematicians who are used to set theory. First, functions are no longer specialized sets amenable to implicit descriptions, so we require an explicit syntax to construct them. Inspired by Alonzo Church's lambda calculus, functions of type $A \to B$ are written $\lambda x \to e$ (called a *lambda abstraction*) where x is the argument of type A and e is an expression of type B that may freely use x. In Agda, one may either use lambda abstractions or traditional mathematical notation to write functions—we will use both throughout this paper. Then, to apply a function f to argument x, one can write either $f\ x$ or $f(x)$—we will use the former in writing Agda and the latter elsewhere. As an example, consider the following definition of `add` for the natural numbers. First, the type of the term is declared and then the definition is given.

```
add : ℕ → ℕ → ℕ
add 0  n = n
add (succ m) n = succ (add m n)
```

This definition makes use of *currying*—as opposed to writing this multiargument function as being of type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$, we have written a function that consumes an argument of type $\mathbb{N}$–the first argument–and then returns a function of type $\mathbb{N} \to \mathbb{N}$ that consumes the second argument and produces the sum. While the syntactic shortcuts of Agda abstract this distinction away; one could have written $\lambda m \to \lambda n \to \ldots$. Thus, in classical mathematics, add would be applied as $add(1)(2)$. This technique is common in type theory and will be preferred to traditional notation in this paper. Now, to demonstrate the promises of computational benefits by MLTT, we can request Agda to evaluate the following expression:

$$\texttt{add}\ 1\ 2 \to 3$$

For all types A and B, we can also write a function that swaps the components of a tuple in $A \times B$ and run it on a pair of natural numbers.

```
swap : A × B → B × A
swap (a , b) = (b , a)
```

$$\texttt{swap}\ (1, 2) \to (2, 1)$$

Furthermore, we can define types of our own, like $\mathbb{2}$: the Boolean type consisting of two *canonical inhabitants* representing truth values.

```
𝟚 : Type₀
𝟚 = 𝟙 + 𝟙

pattern true  = i₁ 0₁
pattern false = i₂ 0₁
```

We use the term *canonical* to distinguish *values* inhabiting types, as opposed to the infinite possible expressions that evaluate to said values. Here, $\texttt{i}_1$ and $\texttt{i}_2$ are the canonical injections of A and B into $A + B$, respectively. Furthermore, 0_1

is the canonical inhabitant of $\mathbb{1}$. Agda's pattern syntax allows us to associate the names `true` and `false` with the given values.

This is also our first exposure to MLTT's universe. To avoid Russell's paradox, the universe of types does not contain itself. Instead, Agda has a hierarchy of universes where U_0 is the universe of small types inhabited by $\mathbb{0}$, $\mathbb{1}$, $\mathbb{N}$, etc. Further universes are given by $U_i : U_{i+1}$ and the various type formers like the coproduct inhabit different universes based on its component types. For brevity, we will switch between employing *typical ambiguity*, eliding which universe we are working in by simply writing U, and specifying the level explicitly in code. Now, we may write a function $P : \mathbb{2} \to U$.

```
P : 𝟚 → Type₀
P true  = 𝟙
P false = 𝟘
```

Note that functions like this whose codomains are universes are called *type families*, as they return types instead of ordinary terms.

MLTT then introduces *dependent types*, which generalize the function and Cartesian product types.

Definition 1.1 (Dependent types [6]). Let A be a type and $P : A \to U$ be a type family. The *dependent function* type $\prod_{a:A} P(a)$ is inhabited by functions f where if $a : A$, then $f(a) : P(a)$ i.e. functions whose codomain type varies with their input.

Similarly, the *dependent pair* type $\sum_{a:A} P(a)$ is inhabited by (a, b) where $a : A$ and $b : P(a)$ i.e. pairs where the type of the second component varies with the first component.

The utility of these two type formers is elucidated in the following explanation: while we now have a calculus to express arbitrary mathematical objects, we still lack a deductive system to perform mathematical reasoning. In order to develop this, we must first introduce the *Brouwer-Heyting-Kolmogorov (BHK) interpretation*, which not only captures the intuition for proofs in informal mathematics but also expresses them as computable objects.

Definition 1.2 (BHK interpretation [10]). We define a proof by induction on the structure of a logical formula.

- There is no proof of $\bot$

Now, let a be a proof of A and b be a proof of B. A proof of…

- …$A \wedge B$ is (a, b) i.e. a proof of A *and* a proof of B
- …$A \vee B$ is either $(0, a)$ or $(1, b)$ i.e. a proof of A *or* a proof of B
- …$A \implies B$ is a computable function that converts a proof of A to a proof of B
- …$\neg A$ is a proof of $A \implies \bot$

Then, fix a domain of discourse D. A proof of…

- …$\forall_{x \in D} P(x)$ is a computable function that converts $a \in D$ to a proof of $P(a)$
- …$\exists_{x \in D} P(x)$ is a pair (a, b) where $a \in D$ and b is a proof of $P(a)$

The proofs described by this interpretation are in exact one-to-one correspondence with the terms inhabiting the various type formers we have just introduced, as shown below [12].

proposition	type
$\bot$	$\mathbb{0}$
$\top$	$\mathbb{1}$
$A \vee B$	$A + B$
$A \wedge B$	$A \times B$
$A \implies B$	$A \to B$
$\neg A$	$A \to \mathbb{0}$
predicate	type family
$\forall_{a \in A} P(a)$	$\prod_{a:A} P(a)$
$\exists_{a \in A} P(a)$	$\sum_{a:A} P(a)$

We can make concrete the correspondence between propositions and types (and consequently proofs and terms) below.

Definition 1.3 (Propositions-as-types)**.** Let A be a type representing a proposition P. If $a : A$, then a is a proof of P in the sense of the BHK interpretation.

With this principle in mind, we can prove some basic propositions in constructive logic, like DeMorgan's law: $\neg A \wedge \neg B \iff \neg(A \vee B)$.

```
DeMorgans₁ : ¬ A × ¬ B → ¬ (A + B)
DeMorgans₁ (¬a , _) (i₁ a) = ¬a a
```

```
DeMorgans₁ (_ , ¬b) (i₂ b) = ¬b b

DeMorgans₂ : ¬ (A + B) → ¬ A × ¬ B
DeMorgans₂ ¬a+b = ((λ a → ¬a+b (i₁ a)) , (λ b → ¬a+b (i₂ b)))
```

Computationally, DeMorgan's law is simply the universal property of the coproduct. Given morphisms $A \to \mathbb{0}$ and $B \to \mathbb{0}$, one can construct a morphism $A + B \to \mathbb{0}$ and vice versa. As a result, the propositions-as-types principle reduces theorem proving to a purely computational endeavor. Now, we can examine the dependent function and pair types. Let us first define the $\leq$ relation on the natural numbers—in MLTT, it is a type family indexed by two natural numbers.

```
_≤_ : ℕ → ℕ → Type₀
0 ≤ n   = 𝟙
(succ m) ≤ (succ n) = m ≤ n
m ≤ n  = 𝟘
```

This definition is quite straightforward: for any number n, $0 \leq n$, and $S(m) \leq S(n)$ if and only if $m \leq n$. Otherwise, the relation does not hold i.e. is defined as absurdity. This allows us to construct computable evidence that a certain number is less than or equal to another one. We can now prove a basic result like $\forall_{n \in \mathbb{N}} \neg(S(n) \leq n)$ by writing a dependent function. Note that in Agda, the dependent function type $\prod_{a:A} P(a)$ is written $(a : A) \to P(a)$.

```
- The codomain type varies on n
succ−n≰n : (n : ℕ) → ¬ (succ n ≤ n)
- By induction on n
succ−n≰n 0        = id
succ−n≰n (succ n) = succ−n≰n n
```

For the base case, the goal $\neg(1 \leq 0)$ evaluates to $\mathbb{0} \to \mathbb{0}$. Thus, a term of this type is the identity function. For the inductive step, realize that the goal $\neg(S(S(n)) \leq S(n))$ evaluates to $\neg(S(n) \leq n)$. By induction, `succ − n ≤ n` n : $\neg(S(n) \leq n)$, so the proof is complete.

As stated before, existential quantification is encoded as the dependent pair type—in Agda, $\sum_{a:A} P(a)$ is written `Σ A P`. Now, we can prove the analogous proposition that for any set A and predicate P on A, $\neg\exists_{a \in A} P(a) \implies \forall_{a \in A} \neg P(a)$.

```
¬Σ−is−Π¬ : ¬ (Σ A P) → (a : A) → ¬ (P a)
¬Σ−is−Π¬ ¬Σ a Pa = ¬Σ (a , Pa)
```

As a result, we could have proven the penultimate result using existential quantification.

```
succ−n≰n′ : (n : ℕ) → ¬ (succ n ≤ n)
succ−n≰n′ = ¬Σ−is−Π¬ lemma where
  – By induction on n
  lemma : ¬ (Σ ℕ (λ n → succ n ≤ n))
  lemma (0 , 1≰0)  = 1≰0
  lemma (succ n , succ−n≰n) = lemma (n , succ−n≰n)
```

The identification of types and propositions mean that proofs are themselves mathematical objects that may be reasoned about—that is, we are doing *proof-relevant mathematics*. Furthermore, the computational content of MLTT is directly accessible. Although these examples are quite tame, more complex proofs are of great utility in software engineering. For example, Euclid's proof of the existence of a greatest common factor (GCF) formalized in a language like Agda is an executable algorithm which computes the GCF correctly. The implications of proof relevance, amongst other things, have motivated the development of *homotopy type theory*, the type theory underlying the results of this paper.

1.4 Homotopy Type Theory

In the previous section, we gave an informal exposition of MLTT by appealing to set theory—in other words, we interpreted the various type formers as set constructors, terms as elements, and discussed their computational and logical interactions. However, we are missing a type that expresses *propositional equality* i.e. propositions that two objects a and b are equal.

Definition 1.4 (Identity type [12])**.** For all types A and $a, b : A$, the *identity type* $a = b$ is inhabited by proofs that a and b are equal, called *identifications*.

By definition, the canonical method of introducing an inhabitant of this type is by reflexivity: $refl = \prod_{a:A} a = a$.

Structural induction upon terms of this type is not as straightforward as with the other type formers. One would expect to be able to simply reduce every encounter of the identity type to reflexivity during theorem proving, but that defies the homotopy-theoretic interpretation of type theory due to *homotopy type theory* (HoTT). When types are interpreted as spaces and terms as points, we get the following correspondence [12].

type theory	homotopy theory
type	space
term	point
type family	fibration
$a = b$	path space

The last point is crucial—the identity type on points a and b is interpreted as the space of paths from a to b. As a result, being able to reduce any term inhabiting the identity type to reflexivity is tantamount to contracting any path to a constant loop, which is nonsensical in homotopy theory! In fact, only when at least one endpoint—either a or b—is free to vary, can one contract a path to a constant loop by moving the free point to the other. This intuition allows us to first define the $PathFrom$ type family, which maps a fixed point x to the space of paths emanating from it i.e. an entire subspace of free points.

Definition 1.5 (PathFrom [8])**.**

$$PathFrom(x) \triangleq \sum_{y:A} x = y$$

The following principle then allows us to reduce certain paths to constant loops under the exact conditions described.

Definition 1.6 (Paulin-Mohring's J [8])**.** Given a type family $P : PathFrom(x) \to U$, $J : P(x, refl(x)) \to \prod_{p:PathFrom(x)} P(p)$ with the following computation rule:

$$J\ r\ (x, refl(x)) \to r$$

Thus, it is impossible to prove that *all* inhabitants of the identity type are identical to reflexivity [7]. Likewise, not every path is contractible to a constant loop. In fact, one can only prove that inhabitants of $PathFrom(x)$ are propositionally equal to $(x, refl(x))$ since the second endpoint is left free.

```
PathFrom-unique : (yp : PathFrom x) → yp == (x , refl x)
PathFrom-unique = J (λ yp → yp == (x , refl x)) (refl (x , refl x))
```

As a result, this allows us to add so-called nontrivial (non-reflexivity) inhabitants to the identity type via separate inference rules without rendering the system inconsistent. Motivated by the simplicial set model of type theory, HoTT adds such inhabitants expressing the extensional equality of various objects. For example, given functions $f, g : A \to B$, if one has evidence $\alpha : \prod_{x:A} f(x) = g(x)$, the axiom of function extensionality gives `funext`(α) : $f = g$. However, the crux of HoTT lies in Voevodsky's univalence axiom, which is an extensionality axiom for *types*. Before we introduce it, we must first define what it means for two types to be *equivalent*, or extensionally equal.

Definition 1.7 (Quasi-inverse [12])**.** A *quasi-inverse* of a function $f : A \to B$ is the following dependent triple:

- $g : B \to A$
- $\alpha : \prod_{x:A} g(f(x)) = x$
- $\beta : \prod_{x:B} f(g(x)) = x$

For the purposes of this paper, we will refer to functions that have quasi-inverses as equivalences, although there are other equivalent notions in type theory. In Agda, we must explicitly specify which type of equivalence we are providing i.e. `qinv−is−equiv` for quasi-inverses. We can now give our notion of extensionality for types.

Definition 1.8 (Type equivalence [12])**.** Given types X and Y, $X \simeq Y$ if there exists a function $f : X \to Y$ that is an equivalence.

Perhaps the most trivial equivalence is given below.

Theorem 1.1 (Identity equivalence)**.** For any type A, $A \simeq A$ by the identity function—the dependent pair of id and evidence that it has a quasi-inverse is called `ide` A in Agda.

An immediate result is that an equality between types can be converted to an equivalence.

Theorem 1.2 (`idtoeqv`)**.** For all types A and B, $A = B \to A \simeq B$.

Proof. Using J reduces the proof goal to giving a term of type $A \simeq A$ i.e. the identity equivalence.

```
idtoeqv : A == B → A ≃ B
idtoeqv p = J (λ{(B , _) → A ≃ B}) (ide A) (B , p)
```

□

Axiom 1.1 (Univalence [12])**.** `idtoeqv` is an equivalence.

By declaring that `idtoeqv` has a quasi-inverse, this axiom gives us the following data:

- $ua : A \simeq B \to A = B$, a function that converts equivalences to paths
- $\prod_{f : A \simeq B} idtoeqv(ua(f)) = f$
- $\prod_{p : A = B} ua(idtoeqv(p)) = p$

The last two data are called *propositional computation rules*, as they dictate how *ua* reduces propositionally, outside of the computation rules built into type theory. However, this raises the question: how do terms evaluate to a value in the presence of univalence? This is actually still an open question—for now, homotopy type theory lacks *canonicity*, the guarantee that every term has a canonical form.

Univalence is justified when we broaden our interpretation of types to not just spaces but to *homotopy types*—spaces regarded up to homotopy equivalence. In that sense, *ua* is simply the trivial assertion that spaces that are homotopy equivalent are equal (up to homotopy equivalence).

Before moving onto Π and its model, we must establish one last concept and rethink our previous conception of propositions-as-types. Recall that we are doing proof-relevant mathematics. However, classical mathematics is decidedly proof-irrelevant since propositions are simply assigned a truth value without additional information. In terms of type theory, this would mean the terms of every type would be indistinguishable up to propositional equality. As a result, the only information we would have about a tautology encoded as a type is that it is inhabited by *some* value, and an absurdity would simply be uninhabited. We formalize this intuition below.

Definition 1.9 (Mere proposition [12])**.** A type is a *mere proposition* if all of its inhabitants are propositionally equal. That is, the following type is inhabited:

$$isProp(A) \triangleq \prod_{x,y : A} x = y$$

This allows us to formalize analogies between classical mathematics (we avoid the phrase “classical logic,” which is related to mere propositions but not expounded here) and type theory.

Theorem 1.3 (Logical equivalence [12])**.** For all mere propositions A and B, if $A \to B$ and $B \to A$, then $A \simeq B$. That is, to show that two mere propositions are equivalent, it is sufficient to show that they are logically equivalent.

Proof. To show $f : A \to B$ and $g : B \to A$ are inverses, we identify $g(f(x))$, x, $f(g(y))$ and y for $x : A$, and $y : B$, respectively, by the fact that A and B are mere propositions.

```
logical-equiv :
  is-prop A → is-prop B → (A → B) → (B → A) → A ≃ B
logical-equiv pA pB f g =
  f , qinv-is-equiv (g , (λ x → pA (g (f x)) x) , (λ y → pB (f (g y)) y))
```

□

For types that are not mere propositions, we may construct an analogue that is.

Definition 1.10 (Propositional truncation [12])**.** For a type A, its propositional truncation $\| A \|$ is described by the following

- If $a : A$, then $| a | : \| A \|$
- $identify : \Pi_{x,y : \|A\|} x = y$

By $identify$, the propositional truncation of any type is a proposition, hence the name.

Structural induction upon inhabitants of a propositional truncation is subtle—a function can only recover the original term underneath the truncation bars if its codomain itself is a mere proposition. We will see this principle show up as `recTrunc` later on.

In short, mere propositions allow us to encode proof-irrelevance into type theory. This is key in defining the *univalent universe of finite types*, the model of Π, which we will do in the next section.

2 Univalent Universe of Finite Types

The underlying characterization of this subuniverse relies on a concept called *univalent fibrations*.

2.1 Univalent Fibrations

An elementary result in homotopy theory is that a path between points x and y in the base space of a fibration induces an equivalence between the fibers over x and y. By univalence, this equivalence is a path as well. We formalize this result below.

Theorem 2.1 (`transporteqv`)**.** For any type A and $x, y : A$, $\prod_{P:A\to U} x = y \to P(x) \simeq P(y)$.

Proof. By J, we may reduce the proof goal to giving a term of type $P(x) \simeq P(x)$ i.e. the identity equivalence.

```
transporteqv : (P : A → Type ℓ) → x == y → P x ≃ P y
transporteqv P p = J (λ{(y , _) → P x ≃ P y}) (ide (P x)) (y , p)
```

□

However, converse is not always true—type families that satisfy this property are called univalent fibrations.

Definition 2.1 (Univalent Fibration [4])**.** For all types A, a type family $P : A \to U$ is a *univalent fibration* if `transporteqv`(P) is an equivalence.

That is, univalent fibrations come with a quasi-inverse of `transporteqv` that converts fiberwise equivalences to paths in the base space. Even though it is rarely the case that any given type family is a univalent fibration, the following theorem characterizes a class of families that are.

Theorem 2.2 (Rose, 2017)**.** Let $P : U \to U$ be a type family. If for all $X : U$, $P(X)$ is a mere proposition, then the first projection $p_1 : \sum_{X:U} P(X) \to U$ is a univalent fibration.

2.2 The `is-finite` Family

We will now examine the `is-finite` type family which forms the basis of the model for Π. First, we require a canonical notion of a finite type.

Definition 2.2 (`El`)**.** The `El` family sends a natural number n to a finite type with n canonical inhabitants.

```
El : ℕ → Type₀
El 0        = 𝟘
El (succ n) = 𝟙 + El n
```

To see that this definition is sufficient, we can enumerate all n canonical inhabitants of `El` n.

$$\begin{array}{r|l} 1 & i_1(0_1) \\ 2 & i_2(i_1(0_1)) \\ 3 & i_2(i_2(i_1(0_1))) \\ & \\ \text{n} & \underbrace{i_2(i_2(\ldots}_{n}(i_1(0_1))\ldots)) \end{array}$$

Notice that we never reach $i_2(i_2(\ldots(i_2(...))\ldots))$ because that would require giving an inhabitant of $\mathbb{0}$, which is impossible. Thus, we are guaranteed n canonical inhabitants. Now, we are ready to define the `is − finite` family.

```
is−finite : Type₀ → Type₁
is−finite A = Σ ℕ (λ n → ∥ A == El n ∥)
```

Viewed as a predicate, this says "a type is finite if it is equivalent to a canonical finite type." Computationally, we require a proof-irrelevant identification of A and `El` n for some n. Then, we define the univalent universe of finite types to be the subuniverse of types satisfying this predicate.

```
M : Type₁
M = Σ Type₀ is−finite
```

Terms of this type are triples consisting of (1) a type A, (2) the "size" of A, and (3) a path witnessing the given size is correct by identifying A with a canonical finite type of the same size. The reason we truncate the above instance of the identity type is to yield the following result.

Theorem 2.3 (Rose, 2017)**.** The first projection p_1 of triples in M is a univalent fibration.

Proof. For any A, $isFinite(A)$ is a mere proposition due to the truncation of its second component, amongst other things. Thus, from theorem 2.2, p_1 is a univalent fibration. □

This is the workhorse of our completeness result—intuitively, to induce a path between two triples, one simply needs to give an equivalence between their first components, which minimizes our proof obligations.

3 Pi

Now that we are acquainted with HoTT and finite types, we can examine the Π programming language by Sabry et al. Π starts with the notion that type equivalences are a natural expression of reversibility—one can write and execute a program and invert its effects via its quasi-inverse. Π then restricts its type calculus to the semiring $(\{\mathbb{0}, \mathbb{1}\}, +, \times)$ up-to type equivalence. As a result, a complete characterization of equivalences over these types is precisely the semiring axioms in figure 1. Note that Π uses $\leftrightarrow$ for $\simeq$.

$$\begin{array}{rrcll}
id\!\leftrightarrow : & \tau & \leftrightarrow & \tau & : id\!\leftrightarrow \\
\\
unite_{+}l : & \mathbb{0} + \tau & \leftrightarrow & \tau & : uniti_{+}l \\
swap_{+} : & \tau_1 + \tau_2 & \leftrightarrow & \tau_2 + \tau_1 & : swap_{+} \\
assocl_{+} : & \tau_1 + (\tau_2 + \tau_3) & \leftrightarrow & (\tau_1 + \tau_2) + \tau_3 & : assocr_{+} \\
\\
unite_{*}l : & \mathbb{1} \times \tau & \leftrightarrow & \tau & : uniti_{*}l \\
swap_{*} : & \tau_1 \times \tau_2 & \leftrightarrow & \tau_2 \times \tau_1 & : swap_{*} \\
assocl_{*} : & \tau_1 \times (\tau_2 \times \tau_3) & \leftrightarrow & (\tau_1 \times \tau_2) \times \tau_3 & : assocr_{*} \\
\\
absorbr : & \mathbb{0} \times \tau & \leftrightarrow & \mathbb{0} & : factorzl \\
dist : & (\tau_1 + \tau_2) \times \tau_3 & \leftrightarrow & (\tau_1 \times \tau_3) + (\tau_2 \times \tau_3) & : factor
\end{array}$$

$$\frac{\vdash c : \tau_1 \leftrightarrow \tau_2}{\vdash !\, c : \tau_2 \leftrightarrow \tau_1} \qquad \frac{\vdash c_1 : \tau_1 \leftrightarrow \tau_2 \quad \vdash c_2 : \tau_2 \leftrightarrow \tau_3}{\vdash c_1 \odot c_2 : \tau_1 \leftrightarrow \tau_3}$$

$$\frac{\vdash c_1 : \tau_1 \leftrightarrow \tau_2 \quad \vdash c_2 : \tau_3 \leftrightarrow \tau_4}{\vdash c_1 \oplus c_2 : \tau_1 + \tau_3 \leftrightarrow \tau_2 + \tau_4} \qquad \frac{\vdash c_1 : \tau_1 \leftrightarrow \tau_2 \quad \vdash c_2 : \tau_3 \leftrightarrow \tau_4}{\vdash c_1 \otimes c_2 : \tau_1 \times \tau_3 \leftrightarrow \tau_2 \times \tau_4}$$

Figure 1: Level 1 Programs (equivalences) in Π [3]

For example, recall that the Boolean data type can be encoded as $\mathbb{1} + \mathbb{1}$. Negation, which sends `true` to `false` and vice versa, is an equivalence. We may define it in many ways—we give two below.

$$\begin{array}{l}
\mathtt{NOT}_1 : \mathbb{2} \longleftrightarrow \mathbb{2} \\
\mathtt{NOT}_1 = \mathtt{swap}_{+}
\end{array}$$

```
NOT₂ : 𝟚 ⟷ 𝟚
NOT₂ = id⟷ ◎ (swap₊ ◎ id⟷)
```

Furthermore, one can ask whether two equivalences are extensionally equal. Π then includes a language which encodes such proofs, called *coherences*, shown in figure 2.

As a result, we can write a proof that NOT₁ and NOT₂ are equivalent by cancelling out the instances of id⟷.

```
NOT₁⇔NOT₂ : NOT₁ ⇔ NOT₂
NOT₁⇔NOT₂ = 2! (idl◎l 2◎ idr◎l)
```

Now that we have a language that describes various finite types and their equivalences as well as a model for them in HoTT, we would like to determine whether the language is complete with respect to the model—that is, for every object in the model, there exists an equivalent one in the language and vice versa.

4 Completeness of Level 0

Now, we can discuss the completeness of level 0, or types in Π with respect to the given model. First, we require translations from the syntax to the model and vice versa. Assume we have the following functions defined.

```
- Converts a type in the syntax
- to the exact same type in MLTT
#⟦_⟧₀ : S → Type₀

- Computes the number of canonical
- inhabitants of a type in the syntax
size : S → ℕ

- Converts an equivalence in the
- syntax to the same one in HoTT
#⟦_⟧₁ : {X Y : S} → X ⟷ Y → #⟦ X ⟧₀ ≃ #⟦ Y ⟧₀
```

In order to write the translation into the model, we need a way of relating any type in the semiring T to $El(n)$ where $n = size(T)$. Note that the image of $El(n)$ is a subtype of S, allowing us to write an analogous function into S.

$\texttt{fromSize} : \mathbb{N} \to \texttt{S}$

We can formalize the relationship between fromSize and El n as follows.

$\texttt{fromSize=El} : \{n : \mathbb{N}\} \to \#[\![\ \texttt{fromSize}\ n\ ]\!]_0 == \texttt{El}\ n$

Then, we define canonical, which converts a type in the semiring to its "canonical" form.

$\texttt{canonical} : \texttt{S} \to \texttt{S}$
$\texttt{canonical} = \texttt{fromSize} \circ \texttt{size}$

Here is an example of the action of canonical:

$$\texttt{canonical}\ ((\mathbb{1} + \mathbb{1}) \times (\mathbb{1} + \mathbb{1})) \to \mathbb{1} + \mathbb{1} + \mathbb{1} + \mathbb{1} + \mathbb{0}$$

Intuitively, a type is equivalent to its canonical form, allowing us to write a function that constructs an equivalence in the syntax between them (due to Sabry et al).

$\texttt{normalize} : (T : \texttt{S}) \to T \longleftrightarrow \texttt{canonical}\ T$

We can finally write the translation by using the above functions. Note that we use univalence to convert the equivalence between a type and its canonical form to a path. Then, we use ■ to concatenate that with the path given by fromSize=El where $n = size(T)$ to generate a path of type $T = El(n)$.

$[\![\_]\!]_0 : \texttt{S} \to \texttt{M}$
$[\![\ T\ ]\!]_0 = (\#[\![\ T\ ]\!]_0\ ,\ \texttt{size}\ T\ ,\ |\ \texttt{ua}\ \#[\![\ \texttt{normalize}\ T\ ]\!]_1 \blacksquare \texttt{fromSize=El}\ |)$

This definition is quite complex, so figure 3 demonstrates its action as an injection into the model.

The translation of the model into the syntax is much simpler—since one cannot perform induction on the opaque type in the first component, we must return the next best thing: a conversion of the size in the second component to a type in the syntax.

$[\![\_]\!]_0^{-1} : \texttt{M} \to \texttt{S}$
$[\![(T\ ,\ n\ ,\ p)]\!]_0^{-1} = \texttt{fromSize}\ n$

We can again view the action of this translation as an injection in figure 4, taking a triple in the model to a canonical form in the syntax.

We now have the sufficient tools to discuss the completeness of level 0. Let us formalize the statements of completeness we made two sections ago.

$$cmpl_1^0 : \prod_{T_1 : S} \sum_{T_2 : M} T_1 \leftrightarrow [\![T_2]\!]_0^{-1} \qquad cmpl_2^0 : \prod_{T_1 : M} \sum_{T_2 : S} \| T_1 = [\![T_2]\!]_0 \|$$

$$T \mapsto ([\![T]\!]_0, lem_1) \qquad T \mapsto ([\![T]\!]_0^{-1}, lem_2)$$

By sending each input to their respective translations, we have proof obligations $lem_1 : \prod_{T:S} T \leftrightarrow [\![[\![T]\!]_0]\!]_0^{-1}$ and $lem_2 : \prod_{T:M} \| T = [\![[\![T]\!]_0^{-1}]\!]_0 \|$. Intuitively, these each say that going back and forth between the syntax and model (and vice versa) produces an equivalent object—let us prove them. To prove the first lemma, consider figure 5, which depicts the round trip of applying both translations.

It seems that we simply must construct an equivalence between a type in the syntax and its canonical form, in the same way we did for $[\![\cdot]\!]_0$.

```
lem₁ : (T : S) → T ⟷ ⟦ ⟦ T ⟧₀ ⟧₀⁻¹
lem₁ = normalize
```

The other direction is a bit more difficult. First, by theorem 2.3 and `idtoeqv`, we can define a function that converts paths between the first components of a triple in the model to a path between the entire triple.

```
induce : {X Y : M} → p₁ X == p₁ Y → X == Y
```

Now, let us observe the this round trup in figure 6—it yields a similar triple but the first component is in canonical form. Precisely by the original path, we may induce a path across both triples by the fact that the first projection is univalent.

This allows us to prove `lem₂` by induction on the truncated path in the third component of a triple, which by `induce`, gives us the necessary result.

```
lem₂ : (X : M) → ‖ X == ⟦ ⟦ X ⟧₀⁻¹ ⟧₀ ‖
lem₂ (T , n , p) =
  recTrunc _ (λ p′ → | induce (p′ ■ ! fromSize=El) |) identify p
```

With these lemmas, we may formally state these completeness results in Agda.

```
cmpl⁰₁ : (T₁ : S) → Σ M (λ T₂ → T₁ ⟷ ⟦ T₂ ⟧₀⁻¹)
cmpl⁰₁ T₁ = (⟦ T₁ ⟧₀ , lem₁ T₁)

cmpl⁰₂ : (T₁ : M) → Σ S (λ T₂ → ‖ T₁ == ⟦ T₂ ⟧₀ ‖)
cmpl⁰₂ T₁ = (⟦ T₁ ⟧₀⁻¹ , lem₂ T₁)
```

5 Future Work

We are currently working on completeness results on levels 1 and 2: isomorphisms and coherences. Furthermore, we would like to develop the formal theory surrounding reversible programming. In particular, there is a deep interplay between homotopy theory and reversibility. For example, we do not have a clear perception of reversible programming with *higher inductive types*, HoTT's internalization of homotopy types. Furthermore, we have the following conjecture which gives a topological characterization of our model, in terms of Eilenberg-MacLane (EM) spaces.

Conjecture 5.1 (Rose, 2017)**.**

$$M = \bigoplus_{n \in \mathbb{N}} K(S_n, 1)$$

where S_n is a symmetric group.

An EM-space $K(G, n)$ has its n^{th} homotopy group (group of n-paths under concatenation and inversion) isomorphic to G and every other one trivial [9]. Thus, this conjecture captures all the necessary information about paths in the model (and equivalences), and therefore the inherent reversibility.

6 Acknowledgements

We thank Prof. Amr Sabry, Prof. Jacques Carette, Robert Rose, Vikraman Choudhury, and Chao-Hong Chen for the collaborative effort on this project. This work is supported by NSF REU Grant #1461061.

References

[1] Patch theory.

[2] Carette, J., Chen, C.-H., Choudhury, V., and Sabry, A. Fractional types.

[3] Carette, J., and Sabry, A. Computing with semirings and weak rig groupoids. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632* (New York, NY, USA, 2016), Springer-Verlag New York, Inc., pp. 123–148.

[4] Christensen, D. A characterization of univalent fibrations. 2015.

[5] Coquand, T. Type theory, Feb 2006.

[6] Dybjer, P., and Palmgren, E. Intuitionistic type theory, Feb 2016.

[7] Hofmann, M., and Streicher, T. The groupoid interpretation of type theory. In *In Venice Festschrift* (1996), Oxford University Press, pp. 83–111.

[8] Licata, D. R. Just kidding: Understanding identity elimination in homotopy type theory, Nov 2015.

[9] Licata, D. R., and Finster, E. Eilenberg-maclane spaces in homotopy type theory. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)* (New York, NY, USA, 2014), CSL-LICS '14, ACM, pp. 66:1–66:9.

[10] Moschovakis, J. Intuitionistic logic, Sep 1999.

[11] Sabry, A. From reversible programming languages to univalent universes and back. 2017.

[12] Univalent Foundations Program, T. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

$$\frac{c : \tau_1 \leftrightarrow \tau_2}{\mathsf{id} : c \Leftrightarrow c} \qquad \frac{c_1, c_2, c_3 : \tau_1 \leftrightarrow \tau_2 \quad \alpha_1 : c_1 \Leftrightarrow c_2 \quad \alpha_2 : c_2 \Leftrightarrow c_3}{\alpha_1 \bullet \alpha_2 : c_1 \Leftrightarrow c_3}$$

$$\frac{c_1 : \tau_1 \leftrightarrow \tau_2 \quad c_2 : \tau_2 \leftrightarrow \tau_3 \quad c_3 : \tau_3 \leftrightarrow \tau_4}{\mathsf{assoc}_{\odot}\mathsf{l} : c_1 \odot (c_2 \odot c_3) \Leftrightarrow (c_1 \odot c_2) \odot c_3 : \mathsf{assoc}_{\odot}\mathsf{r}}$$

$$\frac{c : \tau_1 \leftrightarrow \tau_2}{\mathsf{idl}_{\odot}\mathsf{l} : \mathsf{id} \odot c \Leftrightarrow c : \mathsf{idl}_{\odot}\mathsf{r}} \qquad \frac{c : \tau_1 \leftrightarrow \tau_2}{\mathsf{idr}_{\odot}\mathsf{l} : c \odot \mathsf{id} \Leftrightarrow c : \mathsf{idr}_{\odot}\mathsf{r}}$$

$$\frac{c : \tau_1 \leftrightarrow \tau_2}{\mathsf{rinv}_{\odot}\mathsf{l} : !c \odot c \Leftrightarrow \mathsf{id} : \mathsf{rinv}_{\odot}\mathsf{r}} \qquad \frac{c : \tau_1 \leftrightarrow \tau_2}{\mathsf{linv}_{\odot}\mathsf{l} : c \odot !c \Leftrightarrow \mathsf{id} : \mathsf{linv}_{\odot}\mathsf{r}}$$

$$\frac{}{\mathsf{sumid} : \mathsf{id} \oplus \mathsf{id} \Leftrightarrow \mathsf{id} : \mathsf{splitid}}$$

$$\frac{c_1 : \tau_5 \leftrightarrow \tau_1 \quad c_2 : \tau_6 \leftrightarrow \tau_2 \quad c_3 : \tau_1 \leftrightarrow \tau_3 \quad c_4 : \tau_2 \leftrightarrow \tau_4}{\mathsf{hom}_{\oplus\odot} : (c_1 \odot c_3) \oplus (c_2 \odot c_4) \Leftrightarrow (c_1 \oplus c_2) \odot (c_3 \oplus c_4) : \mathsf{hom}_{\odot\oplus}}$$

$$\frac{c_1, c_3 : \tau_1 \leftrightarrow \tau_2 \quad c_2, c_4 : \tau_2 \leftrightarrow \tau_3 \quad \alpha_1 : c_1 \Leftrightarrow c_3 \quad \alpha_2 : c_2 \Leftrightarrow c_4}{\alpha_1 \boxdot \alpha_2 : c_1 \odot c_2 \Leftrightarrow c_3 \odot c_4}$$

$$\frac{c_1, c_3 : \tau_1 \leftrightarrow \tau_2 \quad c_2, c_4 : \tau_2 \leftrightarrow \tau_3 \quad \alpha_1 : c_1 \Leftrightarrow c_3 \quad \alpha_2 : c_2 \Leftrightarrow c_4}{\mathsf{resp}_{\oplus\Leftrightarrow}\ \alpha_1\ \alpha_2 : c_1 \oplus c_2 \Leftrightarrow c_3 \oplus c_4}$$

$$\frac{c_1, c_3 : \tau_1 \leftrightarrow \tau_2 \quad c_2, c_4 : \tau_2 \leftrightarrow \tau_3 \quad \alpha_1 : c_1 \Leftrightarrow c_3 \quad \alpha_2 : c_2 \Leftrightarrow c_4}{\mathsf{resp}_{\otimes\Leftrightarrow}\ \alpha_1\ \alpha_2 : c_1 \otimes c_2 \Leftrightarrow c_3 \otimes c_4}$$

Figure 2: Level 2 Programs (coherences) in Π [2]

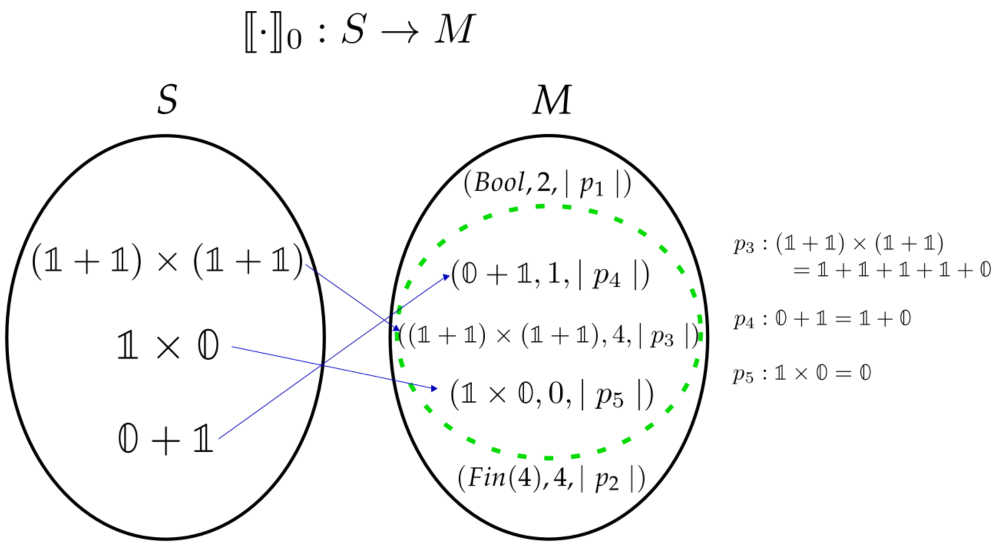
$[\![\cdot]\!]_0 : S \to M$
S
M
$(\mathbb{1}+\mathbb{1}) \times (\mathbb{1}+\mathbb{1})$
$\mathbb{1} \times \mathbb{0}$
$\mathbb{0}+\mathbb{1}$
$(Bool, 2, \mid p_1 \mid)$
$(\mathbb{0}+\mathbb{1}, 1, \mid p_4 \mid)$
$((\mathbb{1}+\mathbb{1}) \times (\mathbb{1}+\mathbb{1}), 4, \mid p_3 \mid)$
$(\mathbb{1} \times \mathbb{0}, 0, \mid p_5 \mid)$
$(Fin(4), 4, \mid p_2 \mid)$
$p_3 : (\mathbb{1}+\mathbb{1}) \times (\mathbb{1}+\mathbb{1}) = \mathbb{1}+\mathbb{1}+\mathbb{1}+\mathbb{1}+\mathbb{0}$
$p_4 : \mathbb{0}+\mathbb{1} = \mathbb{1}+\mathbb{0}$
$p_5 : \mathbb{1} \times \mathbb{0} = \mathbb{0}$

Figure 3: The action of $[\![\cdot]\!]_0$

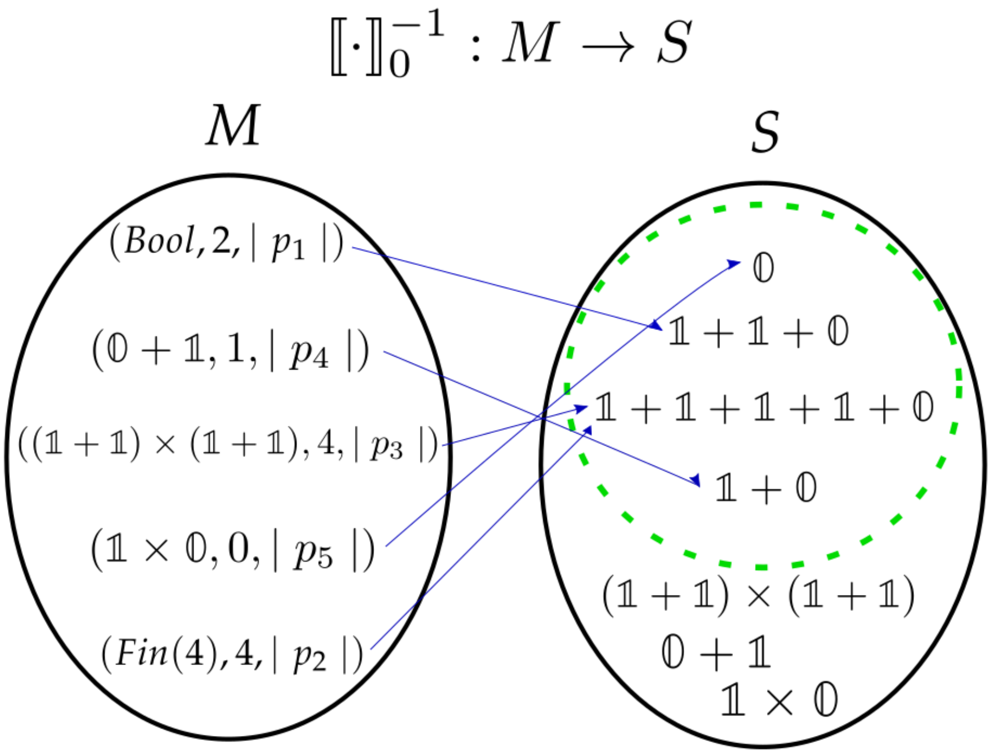
$[\![\cdot]\!]_0^{-1} : M \to S$
M
S
$(Bool, 2, \mid p_1 \mid)$
$(\mathbb{0} + \mathbb{1}, 1, \mid p_4 \mid)$
$((\mathbb{1} + \mathbb{1}) \times (\mathbb{1} + \mathbb{1}), 4, \mid p_3 \mid)$
$(\mathbb{1} \times \mathbb{0}, 0, \mid p_5 \mid)$
$(Fin(4), 4, \mid p_2 \mid)$
$\mathbb{0}$
$\mathbb{1} + \mathbb{1} + \mathbb{0}$
$\mathbb{1} + \mathbb{1} + \mathbb{1} + \mathbb{1} + \mathbb{0}$
$\mathbb{1} + \mathbb{0}$
$(\mathbb{1} + \mathbb{1}) \times (\mathbb{1} + \mathbb{1})$
$\mathbb{0} + \mathbb{1}$
$\mathbb{1} \times \mathbb{0}$

Figure 4: The action of $[\![\cdot]\!]_0^{-1}$

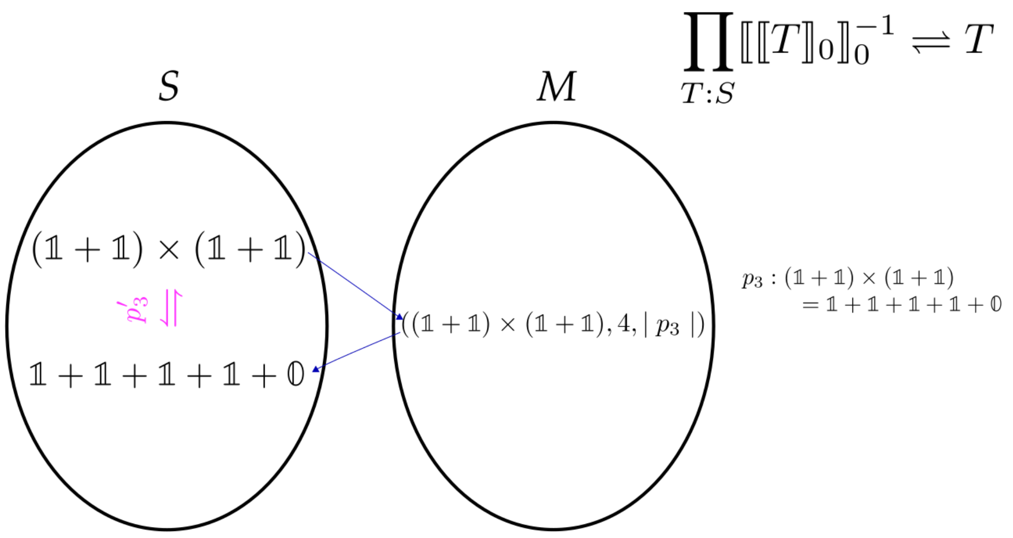
$\prod_{T:S} [\![[\![T]\!]_0]\!]_0^{-1} \rightleftharpoons T$
S
M
$(\mathbb{1}+\mathbb{1})\times(\mathbb{1}+\mathbb{1})$
$p_3' \leftrightharpoons$
$\mathbb{1}+\mathbb{1}+\mathbb{1}+\mathbb{1}+\mathbb{0}$
$((\mathbb{1}+\mathbb{1})\times(\mathbb{1}+\mathbb{1}), 4, |\, p_3 \,|)$
$p_3 : (\mathbb{1}+\mathbb{1})\times(\mathbb{1}+\mathbb{1}) = \mathbb{1}+\mathbb{1}+\mathbb{1}+\mathbb{1}+\mathbb{0}$

Figure 5: The action of the translation then its “inverse”

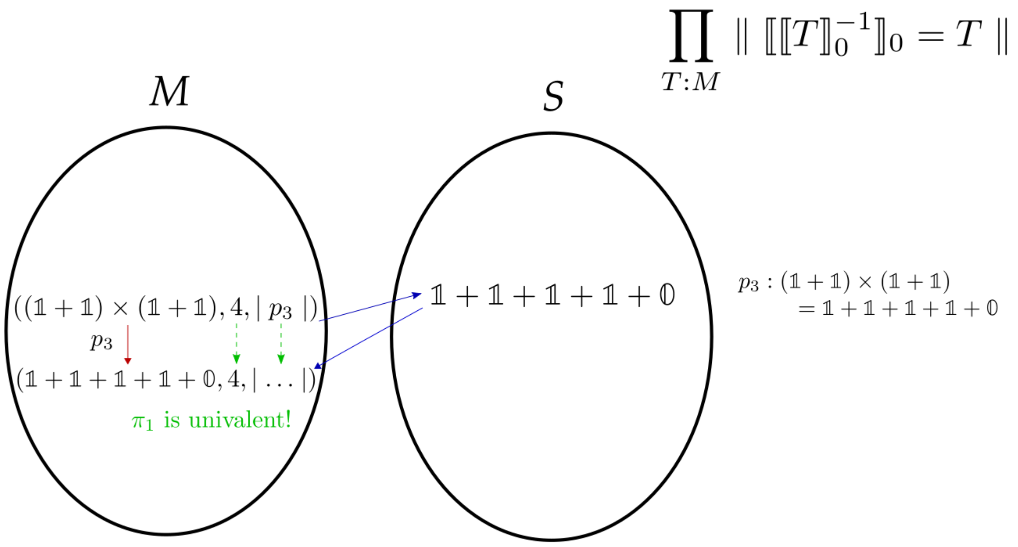
$\prod_{T:M} \| \, [\![[\![T]\!]_0^{-1}]\!]_0 = T \, \|$
M
S
$((\mathbb{1}+\mathbb{1})\times(\mathbb{1}+\mathbb{1}), 4, |\, p_3 \,|)$
p_3
$(\mathbb{1}+\mathbb{1}+\mathbb{1}+\mathbb{1}+\mathbb{0}, 4, |\, \ldots \,|)$
π_1 is univalent!
$\mathbb{1}+\mathbb{1}+\mathbb{1}+\mathbb{1}+\mathbb{0}$
$p_3 : (\mathbb{1}+\mathbb{1})\times(\mathbb{1}+\mathbb{1}) = \mathbb{1}+\mathbb{1}+\mathbb{1}+\mathbb{1}+\mathbb{0}$

Figure 6: The action of the “inverse” then the usual translation