Two-level type theory in Lean and application to semi-simplicial types in HoTT

Adrien Mathieu[*]

August 30, 2024

Special thanks to Arthur Adjedj for being the Lean documentation.

Contents

[*]Under the supervision of Christian Sattler.

Abstract

Despite the great interest in homotopy type theory in the study of foundational systems, its profound links with homotopy theory, and the fact that simplicial types are basic yet important tools of homotopy theory, it has proved impossible so far to define simplicial types in homotopy type theory. Over a decade of failed attempts in doing so has led to conjecturing that it is actually impossible. However, stating a conjecture that speaks about the impossibility to define an object in the same language in which it is (conjectured to be) impossible to define such object is challenging on its own. We use tools from the theory of Reedy categories, and of a conservative strengthening of HoTT (two-level type theory) to formalize a precise statement for this conjecture.

1 Introduction

Homotopy type theory has been introduced as a foundational system for mathematics that could be convenient to work with, and whose proofs would be practical to formalize, in an attempt to reduce drastically the number of unnoticed mistakes in very technical arguments [Voe14]. Yet, despite the great expressiveness of homotopical type theory, certain foundational notions of homotopy theory, such as (semi-)simplicial sets, haven't yet been successfully defined in this framework, despite over a decade of efforts in trying to do so. Voevodsky attempted to overcome this apparent deficiency of HoTT by strengthening the theory with a second, strict identity type [Voe13], as did others [Her15]. However, this type system is not conservative over HoTT, meaning that a term in HTS which happens to only use the HoTT parts of HTS cannot be translated into an equivalent term in HoTT. This has led to the development of the two-level type theory [Ann+23], a conservative strengthening of HoTT. 2LTT, as its name suggests, has two type theories: the so-called object theory, which is an embedding of HoTT, and a meta-theory, which is a strengthening of MLTT with UIP and function extensionality. These two theories are glued together by a lifting operation, that maps an object-level type into a meta-level type. Several formalizations of this theory have been carried out [Usk23] [Ann+23], but they are incomplete.

Besides the original motivation for 2LTT, Kovács has highlighted a link between two-level type theory and staged compilation [Kov22]. Staged compilation is a pattern for designing metaprogramming aspects of a language which allows an extremely expressive metalanguage (for instance, a dependently typed meta-language over a simply typed object language) which can be type checked before expansion, giving well-formedness guarantees of the code output.

This interpretation of Kovács of two-level type theory as a staged version of HoTT reveals another possible usage for it in the mathematical view. Since it is conservative over HoTT, it is possible to use it to generate tedious HoTT terms: every term in the meta-theory whose type is a lifted type can be translated into an equivalent HoTT term, even if the original term used meta constructions, just as a program using metaprogramming facilities can be expanded into a "regular" program, by executing the meta-code. This is particularly useful as, for instance, truncated semi-simplicial types, even though theoretically expressible in HoTT, are in practice very painful to write by hand. Writing uniformly the family (in the meta-language) of n-truncated semi-simplicial types (that live in the object language), then choosing a concrete n, and generating the corresponding HoTT type is much more straightforward.

1.1 Contributions

- We provide a full formalization of a two-level type theory in Lean, except for HITs in the object language. In Section 3, we discuss the design of this formalization.
- In this framework, we provide a formalization of a part (about 30% of Part I) of the HoTT book [Uni]. In addition to the proofs and definitions, we also implement several tactics and

other language niceties to make object-level programming and proof making as similar and straightforward as in the meta-level as possible.

- In Section 4, we develop the theory of Reedy presheaves to provide a definition of semi-simplicial types in the meta theory, where every truncation up to a certain dimension of this definition lives in the object language. Except for two intermediate results (which are admitted in our formalization), this definition has been formalized in our framework.

1.2 Overview

This report has two main sections, Section 3 and Section 4, exposing, respectively, the implementation of 2LTT in Lean as a library, and an example of the formalization of semi-simplicial types in said theory, using our library. Before that, we briefly expose the technical requirement to understand the main sections, in Section 2.

The formalization can be found at `https://github.com/jthulhu/2ltt`.

2 Preliminary notions

We expose here the preliminary notions needed to understand what follows. We suggest reading [Uni] and [Ann+23] for more details. We assume familiarity with Martin-Löf type theory, that is, a dependent type theory with universe hierarchy, sigma and pi type formers, inductive types and identity types. Furthermore, we recall the uniqueness of identity proof (UIP) axiom, because it will be an important part of the theory we will use: for every type A, $x, y : A$ and $p, q : x =_A y$, we have

$$p =_{x =_A y} q$$

Using a pure Martin-Löf type theory as a foundational system, though, is too restrictive. Proofs become very tedious, and some useful propositions (such as function extensionality) are undecidable. Hence, proof assistants are usually founded on stronger theories. One that is of particular interest is the homotopy type theory, because it makes reasoning “up to equivalence” (something that is usually very tedious in proof assistants) completely straightforward.

2.1 Homotopy type theory

Homotopy type theory is a Martin-Löf type theory, which additionally postulates the axiom of univalence, and has higher inductive types. The axiom of univalence states that, given two types A and B, the type of equivalences between A and B is equivalent to the type of equalities between A and B, ie.

$$(A \simeq B) \simeq (A = B)$$

This suggests a natural interpretation for types A in HoTT as topological spaces, for elements $x, y : A$ as points in the space, and for equalities $p, q : x = y$ as *paths* between those two points. Higher equalities $r : p = q$ can be seen as homotopies between paths. This interpretation motivates the name *homotopy type theory*.

An other perspective on homotopy type theory is to see types as groupoids, whose (higher) morphisms are identity types.

Homotopy type theory also comes with higher inductive types (ie. inductive type that can generate higher paths over a type, besides points in the type), but since what follows does not use this feature of HoTT, we will not explain them in detail.

2.2 Two-level type theory

No one has achieved defining semi-simplicial types in homotopy type theory (yet?), but it is possible to define them in the models, using the external equality. This has given the idea of internalizing the external world in the theory, which is exactly what two-level type theory is. Concretely, this means that, in two-level type theory, two type theories coexist:

- the *outer* theory, also called the *meta theory*, the *exo theory* or the *strict theory*, which is a MLTT with uniqueness of identity proofs and function extensionality;
- and the *inner* theory, also called the *object theory* or the *fibrant theory*, which is a homotopy type theory.

These two theories are "glued" together by the fact that inner objects can be seen as outer objects. What this means specifically depends on the point of view:

- From the "outer" point of view, there are only "outer" objects; some of them happen to form a sub-theory which we call the "fibrant" fragment of the theory. This works by having a "fibrant" property (which is not a mere proposition!) that allows us to detect the fibrant fragment of the theory.
- From the "middle ground" point of view, there are both inner and outer objects; contexts can be formed with types from both theories, but such mixed contexts can only form judgments about inner terms. There are also lifting maps that map inner types to outer types, and values of inner types to values of the lifted type.
- From the "inner" point of view, we manipulate HoTT terms as usual, except that there exist some exo-types which behave a bit differently, in that they can only appear in contexts. Exo-equality is though of as the (inner) definitional equality (although that's not postulated in 2LTT), and can be used to manipulate object-level objects as if we were manipulating a model of the object theory.

We will adopt the middle ground point of view for what follows. Identifiers referring to points of the object theory will have a subscript $\mathtt{o}$[1], ie. Σ_{o}, and the lifting operation for types will be denoted as such

$$^{\uparrow}- : \mathrm{Type}_{\mathrm{o}} \to \mathrm{Type}$$[2]

Because the proposition and definitional equalities are an aspect of type theory that differs from usual mathematics, which can be confusing, and because the distinction between the object-level equality and the meta equality is a central point of 2LTT, we will recall the different equalities that we are manipulating, as well as their notation in what follows.

	Definitional	Propositional
Meta	$\equiv$	$=$
Object	$\equiv_{\mathrm{o}}$	$=_{\mathrm{o}}$

Note that we want to identify the object definitional equality with the meta propositional equality: the whole point of 2LTT is to internalize the external equality in the theory. While the equivalence between both is not a postulated meta-axiom of the theory (because it cannot be expressed in the theory itself), this is the case in the considered models.

Indeed, we can show that the meta propositional equality implies the object propositional equality.

[1]Other authors have used superscript $\mathtt{o}$ (object) [Ann+23] or $\mathtt{i}$ (inner) [Ann+23] to denote object-level components, or superscript $\mathtt{s}$ (strict) [ACK16] or superscript $\mathtt{e}$ (exo) [Ahr+22] to denote meta-level components. The reader familiar with the literature would probably be confused if we didn't, in turn, make up our own notation.

[2]To be precise, this should be $^{\uparrow}\mathrm{Type}_{\mathrm{o}} \to \mathrm{Type}$, but this makes $^{\uparrow}$ appear in its own type. In Section 3.1, we will address this issue.

3 Formalization of 2LTT in Lean

Our goal is to formalize 2LTT in a way that makes it convenient to write proofs in 2LTT. The goal is not to be able to reason *about* 2LTT, but rather, reason *in* 2LTT.

To do so, we aim at a shallow embedding in a proof assistant, Lean. Since 2LTT has two languages (the meta language, and the object language), we can have at most one being embedded directly into the language of the proof assistant itself. We choose it to be the meta language, that is, "plain" Lean will play the role of the outer language, and we will embed an other language in Lean to play the role of the inner language. Lean's theory is a strengthening of MLTT with UIP and function extensionality, so using it for the outer language is sound.

In the object language, we furthermore postulate the univalence axiom. UIP happens to be true in the meta language (because, in Lean, the identity type lives in a proof irrelevant universe).

The distinction between the outer and the inner language is not syntactical. Instead, we encode this distinction in the types of the terms: we have two universe hierarchies, one for outer types (the usual Lean type hierarchy `Type u`); and one for inner types `Typeₒ u`.

3.1 The object language

3.1.1 Object universe hierarchy

The object universe hierarchy should be a type hierarchy `Typeₒ u` indexed by the same universe-level integers as Lean types (which we will mostly leave implicit from now on), equipped with a lifting operation

$$^{\uparrow}- : {}^{\uparrow}\mathrm{Type}_{\mathrm{o}} \to \mathrm{Type}$$

Yet, this signature is problematic because $^{\uparrow}$ appears in its own type. To solve this issue, we first create a "lifted universe" lifted-U alongside a lifting operation

$$^{\uparrow}- : \text{lifted-U} \to \mathrm{Type}$$

such that lifted-U $\equiv {}^{\uparrow}\mathrm{Type}_{\mathrm{o}}$.

```lean
structure MetaU.{u} : Type (u+1) := { intoType : Type u }
def Ty : MetaU := { intoType := MetaU }
structure El (α : MetaU) := { intoU : α.into Type }
def liftedU.{u} : Type (u+1) := El Ty
def Typeₒ : liftedU := { intoU := Ty }
def lift.{u} (α : liftedU.{u}) : Type u := El α.intoU
```

We can check that we have the promised property

```lean
example : lift Typeₒ = liftedU := rfl
```

and therefore, our lifting operator has the right signature

```lean
universe u in
example : lift (Typeₒ u) → Type u := lift
```

We can forget about `MetaU`, `Ty`, `El` and `liftedU`. These are gadgets that, in our implementation, are hidden from the end user: this setup being a correct implementation of 2LTT requires these details not being accessible by the end user. One important thing to note about these gadgets is that they both hide the actual content of an object-level value to the end user, while still exposing enough information to the kernel such that and rules compute as they would in the outer language.

If we step back for a moment, and look back at the actual `lift` signature at its definition, it looks like a universe *à la* Tarski (meaning that points of the universe `liftedU` are not types, but can be mapped to a type using `lift`), whereas the universes in Lean are *à la* Russel (that is, a point of `Type` is, well, a type). To make our object language more idiomatic, and more convenient to use, we hook into the type inference system of Lean to make it infer whenever it has to lift something. That is, for instance, when we expect a type, and provide the term $\mathrm{Type}_{\mathrm{o}}$, it will implicitely be lifted to $^{\uparrow}\mathrm{Type}_{\mathrm{o}}$. Hence, we could also write

```
universe u in
example : Typeₒ u → Type u := lift
```

In fact, once this is setup, we never write `lift` again: the inference is enough in every case!

3.1.2 Type formers

After the universe hierarchy, we need to provide the basic type formers for the object language, that is, Π and Σ types. They're both straightforward to define in our setup.

```
def Piₒ (α : Typeₒ) (β : α → Typeₒ) : Typeₒ :=
  Typeₒ.fromType ((a : α) → β a)

namespace Piₒ
  variable {α : Typeₒ} {β : α → Typeₒ}

  def lam (f : (a : α) → β a) : Piₒ α β := El.mk f
  def app (f : Piₒ α β) : (a : α) → β a := f.intoU
end

def Sigmaₒ (α : Typeₒ) (β : α → Typeₒ) : Typeₒ :=
  Typeₒ.fromType (Σ a : α, β a)

namespace Sigmaₒ
  variable {α : Typeₒ} {β : α → Typeₒ}

  def mk (x : α) (y : β x) : Sigmaₒ α β := El.mk (Sigma.mk x y)
  def pr₁ (x : Sigmaₒ α β) : α := x.intoU.fst
  def pr₂ (x : Sigmaₒ α β) : β (pr₁ x) := x.intoU.snd
end
```

They both inherit η rules from Lean's η rules, that is, object pi types (resp. sigma types) have extensionality up to definitional meta equality! We also define similarly a primitive object-level identity type, that is required for object higher inductive types (which we have not implemented, but for which everything is ready).

3.1.3 Object inductive types

For our object language to be fully featured, we only need to be able to define arbitrary inductive types. Our encoding will be similar to that used for Σ types[3], ie. defining a corresponding inductive type in Lean, then wrapping it in our black box to make it live in Type_o. Then, we expose its constructors and its induction principle, but that can only eliminate to object-level types. If the inductive type is structure-like, it additionally has η computation rules.

For instance, the following holds

```
inductiveₒ Unitₒ : Typeₒ where
  | point : Unitₒ
theorem unit_eta (x : Unitₒ) : x = Unitₒ.point := rfl
```

This lets us define the object-level identity type, very straightforwardly:

```
inductiveₒ Idₒ {α : Typeₒ} : α → α → Typeₒ where
  | refl (x : α) : Id x x
```

For the sake of debugging, private, inner types that are wrapped to produce object-level types live in the private namespace of a module "owned" by our library, rather than the module they were defined in (which is the usual behavior for private items in Lean). This means that it is impossible for anyone *but* us to access those items (which is good, as being able to access them would break the conservativity over HoTT). Of course, this holds up to meta programming: this

[3]In fact, we could directly define Σ types using inductive types in our formalization.

constraint is only enforced in the elaborator, not in the kernel (which has no notion of private items).

Generating the induction principle means wrapping the induction principle that the kernel automatically provides for the wrapped inductive type. This means that our implementation must adhere to the signature generated by the kernel (which is, to the best of our knowledge, not documented anywhere). This is complicated by the fact that the elaborator will perform additional modification of the inductive declarations, such as the parameter promotion: when Lean recognizes that an index of an inductive type family could be, instead, a parameter, it will *sometimes* "promote" it as such. For instance, the identity type as we just defined it might be promoted to the following form

```
inductiveₒ Idₒ {α : Typeₒ} (x : α) : α → Typeₒ where
  | refl : Id x x
```

Those two types are equivalent, but the induction principle of the latter is known as the *based path induction*, which is more convenient to use, because we require only one of the endpoints of the path we are inducting on to be a universally quantified over variable, while the other "end" of the path could be any expression. Of course, it can be derived from the induction principle of our former definition.

Initially, our implementation tried to follow closely the (undocumented[4]) behavior of the Lean compiler; but that was a bad decision, because this entails a high coupling with the Lean internals, implying more efforts to maintain our library. Instead, we have reimplemented our own higher level elaborator steps, that roughly perform the same job as their counterpart in the Lean compiler.

3.2 Object level tactics

Most of the usual tactics also work seamlessly for object level proofs. There are some exceptions, though: the `rewrite` tactic, for instance, can rewrite everywhere, but only along meta equalities. Our object-level version, `rewriteₒ`, can only rewrite on object types, and uses object paths. Similarly, we have rewritten `intro` and `apply` (which, respectively, are the introduction and elimination principle of Π types). We have also partially rewritten the `induction` tactic: our object-level induction tactic only works with the object equality, and not arbitrary inductive types[5] We have also implemented two other general purpose tactics, which are, in principle, not useful, but were especially handy for debugging our implementation: `check` and `replace`, which respectively checks that the current goal is well-typed[6], and performs a computation step that preserves definitional equality.

Furthermore, we have implemented some elaboration niceties that are available in plain Lean, but do not work out of the box. For instance, rather than writing

```
example : Nat := Nat.succ (Nat.succ (Nat.succ Nat.zero))
```

Lean allows you to write `Nat.zero.succ.succ.succ`. When elaborating this method-like syntax, Lean tries to infer the type of the receiver (whatever is on the left of the dot), and then looks for a function named like the "method" (here, `succ`) in the namespace of that type. Due to our encoding, this doesn't work with object-level values: their type is an implementation detail whose namespace only contains private defintions. We therefore provide a dot-like syntax `Natₒ.zero▫succ▫succ▫succ`, whose behavior is similar to Lean's.

To give a feeling of tactics development in Lean, we will show how (a simplified version of) `exhibitₒ` is implemented. This tactics is used to advance a goal of type $\Sigma_o x : \alpha, \beta x$, by providing

[4]At the time of writing, the paragraph `https://ammkrn.github.io/type_checking_in_lean4/declarations/inductive.html#recursors` simply reads "TBD".

[5]The reason for this restriction is simply that we have implemented the `inductionₒ` tactic before the object inductive types. Our implementation, though, follows closely Lean's implementation of the (general purpose) `induction` tactic, and could therefore easily be extended to handle arbitrary object inductive types.

[6]Which is, in principle, always the case. Things can go sour if there is a bug in a tactic though, so this is useful for debugging tactics.

a value $e : \alpha$, and rewriting the goal to βe, similarly to what `exists` does for existential quantifier goals.

First of all, we need to register a custom syntax for this tactic. Tactics have their own syntax category, `tactic`:

```
syntax (name := exhibit) "exhibit₀ " term : tactic
```

This means that the keyword `exhibit₀` followed by a term is a syntactically valid tactic. Now, we must provide an implementation for this tactic, that is, we must inform the elaborator how to deal with this syntactic node. Contrary to our previous example, this is not just a tree translation, transforming a node of the syntax tree into a node of the intermediate representation of Lean's code. Instead, this elaboration requires both partially building a proof term and interacting with the goal state. Note that, in Lean, these two are separate, and there is nothing that checks they stay in sync. The goal state showing there is nothing more to prove doesn't necessarily mean we have produced a valid proof term, and more generally speaking a goal state being transformed in certain ways by tactics doesn't necessarily mean that a corresponding proof term is being built. It is up to the implementation of the tactic to ensure this is true. Failure to uphold this assumption may result in an elaborator error (if someone, at some point, checks certain assumptions), or, if the proof term is wrongly elaborated without further checks, it produces a kernel error. The kernel always checks the proofs in their intermediate representation form, so mistakes in the elaborator cannot lead to unsound logical conclusions. They can, however, make Lean crash, if, for instance, an unchecked out of bound array indexing operation is performed.

The actual implementation of the tactic is as follows

```
1   @[tactic exhibit]
2   def exhibit_impl : Tactic
3     | `(tactic| exhibit₀ $w) => do
4       let goal ← getMainGoal
5       goal.withContext do
6         let goal_type ← goal.getType
7         let some (u₁, u₂, α, β) ← (← goal_type.objType!).sigma?
8           | throwTacticEx `exhibit₀ goal m!"...error message..."
9         let w ← elabTermEnsuringType witness (α.liftWith u₁)
10        let new_goal ← mkFreshExprSyntheticOpaqueMVar ((.app β w).liftWith u₂)
11        new_goal.mvarId!.withContext do
12          goal.assign (mkAppN (.const ``Sigma₀.mk [u₁, u₂]) #[α, β, w, new_goal])
13          replaceMainGoal [new_goal.mvarId!]
```

Since Lean has an open-ended elaborator, meaning that anyone can hook to elaborate any syntactic node, an elaborator always starts by pattern-matching on all possible syntax nodes. The pattern matching need not to be exhaustive: if an unsupported pattern exception is raised by an elaborator (which is the default behavior for every pattern not explicitely handled), Lean will try the next elaborator, up until one succeeds, one fails with an other error, or none are left.

Here, we are only interested in the node we have just declared, in the `tactic` syntax category. In lines 4-5, we retrieve the active goal, and do the rest of our computations in its context. The context of a goal is an environment for local hypothesis, ie. the bindings for free variables that occur in the goal, and which are not constants defined somewhere else. Then, line 6-8, we retrieve the type of the goal, and check whether it is indeed an object-level sigma type, otherwise, an error is raised. If everything went well, we can actually elaborate the witness provided by the user (which, so far, is still is syntactic node). We pass to the elaborator the information of the type we expected for this term, rather than simply letting the elaborator infer the type, and checking afterwards that the two types match, as type inference is undecidable. By doing it this way, we increase the number of cases where the user is not prompted for additional type information; and this also takes care of reporting and error in case of type mismatch.

Before further explaining this code, an explanation of how Lean goals work is important. Lean goals are simply metavariables; goals are closed by assigning to the metavariable, and are opened by creating a new metavariable, and by registering it as a goal. This is very convenient, as it makes it possible to write proof terms that depend on a goal that is still open. We use it as any other term, and, eventually, it will be assigned to an actual value, or the whole proof has failed. Furthermore, this means that the usual mechanism for filling in automatically metavariables when unifying them with other expressions will solve goals whose value can be implicitly determined, if it suitable that this happens[7].

Hence, line 10-13, we create a new goal, by creating a new metavariable. We close the old goal, by assigning it a partial proof term (because it depends on the newly created goal, which is still open), and then update the goal state by removing the old one, and inserting the new one in place.

3.3 Homotopy type theory

To have a fully functional 2LTT implementation, we still need to postulate the axiom of univalence in the object language. In order for this axiom to be of practical usage, it needs to be correctly formulated, which involves proving a certain number of properties first about equivalences. Skipping these details (which can be found in the formalization), we start by defining the canonical map from the identity type to the equivalence type:

```
def canonical (α β : Typeₒ) : α =ₒ β →ₒ α ≃ₒ β := by
  introₒ p                   -- Let p : α =ₒ β.
  path_inductionₒ p          -- By induction, we can assume α ≡ β.
  exact Equivalence.refl α -- We conclude by observing that the identity function
                             -- is an equivalence.
```

Indeed, when stating univalence, we don't want to just assert that there is an equivalence between the identity type and the equivalence between types; we want to state, more precisely, that this canonical map is an equivalence.

```
axiom univalence {α β : Typeₒ} : (canonical α β)▫is_equiv
```

This, indeed, implies our original formulation

```
def eqv_eqv_eq {α β : Typeₒ} : (α ≃ₒ β) ≃ₒ (α =ₒ β) := by
  apply Equivalence.symm
  exhibitₒ canonical α β    -- As an equivalence, we exhibit the canonical map,
  exact univalence          -- and prove it is indeed an equivalence using univalence.
```

To have a full implementation of HoTT, we would also need to support higher inductive types. We have not implemented them because we didn't need them in our work, but if we had to implement them, we would have done as follows, given a higher inductive declaration:

- extract, from the declaration, the first-order part (that is, the constructors that do not involve higher paths);
- use these constructors to build a hidden inductive type that would be the underlying type, just as we did for regular inductive types;
- postulate every higher constructors;
- implement the induction principle, using the induction principle generated by the kernel for our hidden inductive type;
- implement the computation rules of the induction principle, for the regular constructors;
- postulate the computation rules of the induction principle, for the higher constructors.

The advantage of implementing anything that can be implemented, rather than postulating it, is that computation rules hold up to definitional equality, and not just propositional equality. Also, this is compatible with our current implementation of inductive types, meaning that we wouldn't have to make a special case out of higher inductive types, simply add the support for higher constructors.

[7]Sometime, goals are not meant to be implicitly solved.

3.4 Examples

To have an idea of how proofs in the two theories compare, consider the following example, which is twice an implementation of natural number, one in each theory, their addition, and the proof that it is associative.

```
inductiveₒ Natₒ where
  | zero : Natₒ
  | succ (n : Natₒ) : Natₒ

namespace Natₒ
def add (n m : Natₒ) : Natₒ :=
  Natₒ.recₒ n (fun k => succ k) m

instance : Add Natₒ where
  add := add

@[simp]
theorem add_zero (n : Natₒ) : n + zero = n := rfl

@[simp]
theorem add_succ (n m : Natₒ) : n + (succ m) = succ (n + m) := rfl

def add_assoc (n m k : Natₒ) : (n + m) + k =ₒ n + (m + k) := by
  refine recₒ (motive := fun k => (n + m) + k =ₒ n + (m + k)) ?zero ?succ k
  case zero =>
    simp
    rflₒ
  case succ =>
    intro k ih
    rwₒ [ih]
    rflₒ
end Natₒ
```

```
inductive Nat' where
  | zero : Nat'
  | succ (n : Nat') : Nat'

namespace Nat'
def add (n m : Nat') : Nat' :=
  match m with
  | zero => n
  | succ m => succ <| add n m

instance : Add Nat' where
  add := add

@[simp]
theorem add_zero (n : Nat') : n + zero = n := rfl

@[simp]
theorem add_succ (n m : Nat') : n + (succ m) = succ (n + m) := rfl

def add_assoc (n m k : Nat') : (n + m) + k = n + (m + k) := by
  induction k
  case zero =>
    simp
  case succ k ih =>
    simp
    rw [ih]
end Nat'
```

3.5 Soundness of the embedding

Our implementation of 2LTT is sound in that we could syntactically translate every term built using the two languages that we expose into 2LTT's syntax. This wouldn't be true if, when writing the terms, the implementation details were exposed: indeed, if that were the case, it would be trivial to build an equivalence between `Nat` and `Natₒ` (which is undecidable in 2LTT). What prevents "normal" users from doing so is that those implementation details live in namespaces that are impossible to write out normally in Lean, akin to how the **`private`** keyword works. We deviate slightly from the behavior of the **`private`** keyword, though, because this restriction doesn't hold anymore in the elaborator. There are no mechanisms in Lean to truly hide implementation details, because the kernel has no notion of visibility of declarations[8]. This means that somehow doing meta programming *could* break soundness. Hence, we have used a "hiding" scheme slightly different from that used by **`private`**, so that no one *accidentally* breaks the abstraction layer.

Furthermore, we guarantee that (if one adheres to the interface we provide, and doesn't try too hard to break through it), our implementation is sound, because we publicly expose exactly what is postulated to exist in 2LTT.

Furthermore, although we haven't proved the soundness of our implementation (which would involve embedding our whole implementation in a proof assistant), we never postulate anything except for the axiom of univalence, contrary to how some other implementations of 2LTT are actually implemented. This means that we can be confident that we haven't accidentally postulated an inconsistent object.

4 Definition of semi-simplicial types in 2LTT

The trick to stating that it is impossible to define semi-simplicial types in HoTT is defining semi-simplicial types in 2LTT, and stating that there is no object in the HoTT fragment of 2LTT that is equal to the object we have just defined. Unfortunately, we cannot directly compare something that lives in HoTT with something that lives outside of HoTT so, instead, we are going to build a family of "properties" of semi-simplicial types, such that each of these "properties" lives in the HoTT fragment. Then, our statement will be that there is no similar family of properties that lives in the HoTT fragment (that is, the whole *family* lives in the object language, not just pointwise). This implies that there is no object in HoTT that has similar properties to that that we have defined in 2LTT.

To be specific, the properties that we will consider are the truncation of semi-simplicial types up to a certain dimension, the type of semi-simplicial types being the colimit of these truncations. We will show that each of these truncations lives in HoTT.

A byproduct of this construction is that we can easily generate terms in HoTT that define arbitrarily high truncation of semi-simplicial types, due to the conservativity property of 2LTT over HoTT. While it was already possible to do this by hand before, it is very tiresome to do so, especially because, to the best of our knowledge, the best size of the definition of semi-simplicial types truncated to a certain dimension that we currently know is exponential in the dimension, and the scheme for generating these definitions by hand is not trivial.

Since we will define semi-simplicial types as presheaves over the semi-simplicial category, we will start with some preliminary definitions, notations and facts about the fragment of category theory that will be useful to our work.

4.1 Preliminary notions and notations

We will use the notation $\begin{bmatrix} a_0 : \tau_0 \\ a_1 : \tau_1(a_0) \\ \vdots \\ a_n : \tau_n(a_0, a_1, \dots, a_{n-1}) \end{bmatrix}$ to refer to an anonymous structure type, as well as a_i for the projections of such type. If we need a canonical choice for such a type, we can

[8]The kernel has a similar feature, called opaque declarations, but those completely hide implementation details including for the kernel, so an opaque declaration is basically useless. Indeed, opaque declarations are mainly used to encode FFI calls in the kernel.

consider such a type to be defined as being

$$\sum_{a_0:\tau_0} \sum_{a_1:\tau_1(a_0)} \cdots \sum_{a_{n-1}:\tau_{n-1}(a_0,\dots,a_{n-2})} \tau_n(a_0, a_1, \dots, a_{n-1})$$

For a category C, and $x, y : C$ two objects in C, we will denote by $C(x, y)$ the type of morphisms in C from x to y.

For $n : \mathbb{N}$, we will note

$$[n] := \{k : \mathbb{N} \mid k < n\}$$

the canonical finite type of cardinal n, which is called `Fin n` in Lean.

4.1.1 Presheaves and their limits

Definition 4.1 (presheaf)**.** A presheaf over a category C is a functor $C^{\mathrm{op}} \to \mathrm{Type}$.

Definition 4.2 (Category of elements)**.** Let C be a category, and F be a presheaf over C. We note by $\int F$ the *category of elements of F*, whose points are the (dependent) couples (x, a) where $x : C$ and $a : F(x)$, and whose morphisms

$$\left(\int F\right)((x, a), (y, b)) = \{f : C(x, y) \mid F(f)(b) = a\}$$

The category of elements of a presheaf is interesting, because it corresponds exactly to the shape over which we have an explicit formula for the limit of the said presheaf.

Lemma 4.3. Let J be a category, and $A, F : \mathrm{Psh}(J)$. Then

$$\lim_{(j,a):\int A} F(j) \cong \mathrm{Psh}(J)(A, F)$$

In particular, this means that we can compute the "regular" (or non-weighted) limit of a presheaf, by choosing accordingly the presheaf A:

Lemma 4.4. Let J be a category, and 1 be the constant unit presheaf over J. Then

$$\int 1 \cong J$$

Proof. The natural functors between the two categories are easily shown to be each other's inverse. □

Let us now prove Lemma 4.3

Proof. Consider the cone defined by, for every $j : J$ and $a : A(j)$,

$$\mathrm{Psh}(J)(A, F) \xrightarrow{\phi_{j,a}} F(j)$$

where $\phi_{j,a}(u) \simeq u(j)(a)$ for $u : \mathrm{Psh}(J)(A, F)$.

To show that we have properly defined a cone, be need to show that, for $i, j : J$, $a_i : A(i)$, $a_j : A(j)$ and $f : i \to j$ such that $A(f)(a_j) = a_i$, the following diagram commutes

$$\begin{tikzcd}
 & F(i) \\
\mathrm{Psh}(J)(A, F) \arrow[ur, "\phi_{i,a_i}"] \arrow[dr, "\phi_{j,a_j}"'] & \\
 & F(j) \arrow[uu, "F(f)"']
\end{tikzcd}$$

By extensionality, it suffices to have the result pointwise. Let $u : \mathrm{Psh}(J)$, we have to check that

$$\begin{aligned}\phi_{i,a_i}(u) &= F(f)(\phi_{j,a_j}(u)) \\ &= F(f)(u(j)(a_j)) \\ &= (F(f) \circ u(j))(a_j)\end{aligned}$$

and, since we also have

$$\begin{aligned}\phi_{i,a_i}(u) &= u(i)(a_i) \\ &= u(i)(A(f)(a_j)) \\ &= (u(i) \circ A(f))(a_j)\end{aligned}$$

which follows from the naturality of u

$$\begin{tikzcd} A(j) \arrow[r, "A(f)"] \arrow[d, "u(j)"] & A(i) \arrow[d, "u(i)"] \\ F(j) \arrow[r, "F(f)"] & F(i) \end{tikzcd}$$

We now need to show that this cone is a limit cone. Let X : Type and, for every $j : J$ and $a_j : A(j)$,

$$X \xrightarrow{\psi_{j,a_j}} F(j)$$

such that for every $i, j : J$, $a_i : A(i)$, $a_j : A(j)$ and every $f : J(i,j)$ such that $A(f)(a_j) = a_i$, the following diagram commutes

$$\begin{tikzcd} & F(i) \\ X \arrow[ur, "\psi_{i,a_i}"] \arrow[dr, "\psi_{j,a_j}"'] & \\ & F(j) \arrow[uu, "F(f)"'] \end{tikzcd}$$

We have to show that there exists a unique $\varphi : X \to \mathrm{Psh}(J)(A, F)$ such that, for every $j : J$ and $a_j : A(j)$, the following diagram commutes

$$\begin{tikzcd} & F(j) \\ X \arrow[ur, "\psi_{j,a_j}"] \arrow[r, "\varphi"'] & \mathrm{Psh}(J)(1, F) \arrow[u, "\phi_{j,a_j}"'] \end{tikzcd}$$

If such a φ exists, it satisfies, for every $x : X$,

$$\begin{aligned}\psi_{j,a_j}(x) &= \phi_{j,a_j}(\varphi(x)) \\ &= \varphi(x)(j)(a_j)\end{aligned}$$

which uniquely determines it. In fact, by letting φ be defined exactly as such, it trivially makes the former diagram commute. Hence $\mathrm{Psh}(J)(A, F)$ is indeed a limit of $F \circ \pi_1$. □

In particular, we have an explicit formula for regular limits.

Corollary 4.4.1. Let J be a category, and F be a presheaf over J. Then

$$\lim F \cong \mathrm{Psh}(J)(1, F)$$

Proof.

$$\begin{aligned}
\mathrm{Psh}(J)(1, F) &\cong \lim_{(j,*):\int 1} F(j) && \text{by Lemma 4.3} \\
&\cong \lim_{j:J} F(j) && \text{by Lemma 4.4}
\end{aligned}$$

□

4.1.2 Semi-simplicial category

Definition 4.5 (Semi-simplicial category)**.** Let Δ_+ be the category whose objects are natural numbers. Given $n, m : \Delta_+$, the morphisms between n and m are strictly monotone functions from $[n]$ to $[m]$.

In a "classical" setting (ie. in set theory, or in a type theory with UIP), we would define semi-simplicial types (or sets) as presheaves over the semi-simplicial category. In this case, if we unfold the definition, a semi-simplicial type is a type family $(X_n)_{n:\mathbb{N}} : \mathbb{N} \to \mathrm{Type}$, alongside border operators for every dimension $n : \mathbb{N}$, and $k : [n]$, $\partial^n_k : X_{n+1} \to X_n$, that satisfies a commutativity condition: for every $n : \mathbb{N}$, and $i < j < n + 1$,

$$\partial^n_i \circ \partial^{n+1}_j = \partial^n_{j-1} \circ \partial^{n+1}_i$$

In HoTT, though, this is not enough. The following cube is automatically filled if we have UIP, but not in HoTT.

$$\begin{tikzcd}
& n+3 \arrow[rr, "\partial^{n+2}_j"] \arrow[dl, "\partial^{n+2}_i"'] \arrow[dd, "\partial^{n+2}_k"] & & n+2 \arrow[dl, "\partial^{n+1}_i"] \arrow[dd, "\partial^{n+1}_{k-1}"] \\
n+2 \arrow[rr, "\partial^{n+1}_{j-1}"] \arrow[dd, "\partial^{n+1}_{k-1}"'] & & n+1 \arrow[dd, "\partial^{n}_{k-2}"] & \\
& n+2 \arrow[rr, "\partial^{n+1}_j"] \arrow[dl, "\partial^{n+1}_i"'] & & n+1 \arrow[dl, "\partial^{n}_i"] \\
n+1 \arrow[rr, "\partial^{n}_{j-1}"] & & n &
\end{tikzcd}$$

We could enforce the equality between these paths as an additional condition for being a semi-simplicial type, but then there would have a similar problem with the composition of four border operations, then five, then so forth, and the diagrams that have to be filled become increasingly complex. We call this unfolding in higher and higher dimension of coherence paths an *infinite tower of higher-dimension equalities.* From a categorical perspective, this is an instance of the difference between a functor and an ∞-functor between ∞-groupoids, the latter having an infinity of isomorphism up to higher isomorphism conditions in place of equalities.

4.1.3 Direct categories

In what follows, we will be interested in diagrams over direct categories.

Definition 4.6 (direct category)**.** A category C is said to be *direct* if there is a *rank functor* $\varphi : C \to \omega$ that reflect identities, that is, for every $x, y : C$, if we have a morphism $f : x \to y$ such that $\varphi(x) = \varphi(y)$, then f is the identity, that is, $(y, f) = (x, \mathrm{id}_x)$ (note that this is an equality between dependent pairs). For $x : C$, we say that $\varphi(x)$ *is the rank of* x.

For $n : \mathbb{N}$, we note $C^{<n}$ for the full subcategory of objects whose rank is less than n, and $C^{=n}$ for the type of objects whose rank is exactly n. The latter is also a full subcategory of C, but since it is also a discrete category, its categorical structure is not very interesting.

Example 4.7. Δ_+ is a direct category.

Proof. The rank functor simply maps $n : \Delta_+$ to n. It is a rank functor because

- for $n, m : \mathbb{N}$, if there is a strictly monotone function from $[n]$ to $[m]$, then $n \leq m$;
- for $n : \mathbb{N}$, the only strictly monotone function from $[n]$ to itself is the identity.

$\square$

For the rest of the article, D will be a direct category.

Definition 4.8. D is said to have *finite layers* if, for every $n : \mathbb{N}$, $D^{=n}$ is finite.

Example 4.9. Δ_+ has finite layers. Indeed, each layer is just a singleton.

Direct categories are interesting because we can "peel them layer by layer", ie. we can do prove theorems on truncated direct categories by induction on the rank up to which we have truncated. In particular, what interests us are the presheaves on direct categories, so we want to be able to characterize the presheaves overs $D^{<n+1}$ using the presheaves over $D^{<n}$. We therefore introduce the type of layered presheaves up to rank $n+1$: it's the presheaves up to rank n, plus the information on the extra layer $D^{=n}$. Because we are talking about functors, this extra information consists of:

- the map on objects of the layer $D^{=n}$;
- the map on morphisms (because a direct category reflects identity, the only information that we need is how the functor behaves on morphisms whose origin is in lower layers);
- some coherence equalities, to ensure that this extra information still behaves like a functor.

Definition 4.10. For $n : \mathbb{N}$, we let

$$\mathrm{LPsh}_n(D) := \left[\begin{array}{r} \mathrm{X} : \mathrm{Psh}(D^n) \\ \mathrm{Y} : D^{=n} \to \mathrm{Type} \\ \mathrm{u} : \prod_{i:D^{=n}} \prod_{j:D^{<n}} (j \to i) \to (Y(i) \to X(j)) \\ \mathrm{coh} : \prod_{i:D^{=n}} \prod_{j,j':D^{<n}} \prod_{f:j\to i} \prod_{g:j'\to j} \mathrm{X}(g) \circ \mathrm{u}_{i,j}(f) = \mathrm{u}_{i,j'}(g \circ f) \end{array}\right.$$

Our definition is built exactly so that we have the following equivalence.

Lemma 4.11. Let $n : \mathbb{N}$. We have

$$\mathrm{Psh}(D^{<n+1}) \overset{e_n}{\simeq} \mathrm{LPsh}_n(D)$$

Proof. Let $s : \mathrm{Psh}(D^{<n+1}) \to \mathrm{LPsh}_n(D)$ be defined, for $F : \mathrm{Psh}(D^{<n+1})$, by

$$s(F) = \left[\begin{array}{ll} \mathrm{X} = F \circ \iota_n & \text{where } \iota_n \text{ is the inclusion functor} \\ \mathrm{Y} = i \mapsto F(i) & \\ \mathrm{u} = (i,j,f) \mapsto F(f) & \end{array}\right.$$

Additionally, the coherence conditions stems directly from the functoriality of F.

Conversely, let's define $u : \mathrm{LPsh}_n(D) \to \mathrm{Psh}(D^{<n+1})$. Let $G : \mathrm{LPsh}_n(D)$. For $i : D^{n+1}$. We define $u(G)(i)$ by

$$u(G)(i) = \begin{cases} G.\mathrm{X}(i) & \text{if } \varphi(i) < n \\ G.\mathrm{Y}(i) & \text{otherwise} \end{cases}$$

For $i, j : D^{<n+1}$ and $f : i \to j$. We define $u(G)(f)$ by case distinction.

- If $\varphi(i) = n$, because

$$\begin{aligned} \varphi(i) &\leq \varphi(j) && \text{by functoriality of } \varphi \\ &\leq n && \text{by definition of } D^{<n+1} \end{aligned}$$

  we have $\varphi(j) = n$. By identity reflection, $i = j$ and $f = \mathrm{id}_j$. So we can just define $u(G)(f) = \mathrm{id}_{u(G)(j)}$.

- Otherwise, if $\varphi(j) = n$, we can define
$$u(G)(f) = G.\mathrm{u}_{i,j}(f)$$

- Otherwise, we can define
$$u(G)(f) = G.\mathrm{X}(f)$$

This is functorial. It is clear that it maps identities to identities. Furthermore, given $i, j, k : D^{<n+1}$, $f : i \to j$ and $g : j \to k$,

- either they all have a rank less than n, in which case
$$u(G)(g \circ f) = G.\mathrm{X}(g \circ f) = G.\mathrm{X}(g) \circ G.\mathrm{X}(f) = u(G)(g) \circ u(G)(f)$$

- or exactly one of them has a rank equal to n, in which case it must be k, and the functoriality is given by $G.\mathrm{coh}_{k,j,i}$;

- or two or more of them have a rank equal to n, in which case g is an identity morphism, and the identity becomes trivial.

It is clear that s and u are inverse of each other, because the coherence type is a mere proposition (because we have uniqueness of identity proof in the meta theory). □

4.2 Fibrant types

In consistency with the outer point of view, we'll call outer types that are lifts of object types *fibrant.* More specifically, we'll distinguish between *strict fibrancy* and *weak fibrancy.*

Definition 4.12 (Fibrant types)**.** Let

$$\text{is-fibrant}_{\text{strict}}(\alpha) = \left[\begin{array}{r}\alpha_{\mathrm{o}} : \mathrm{Type}_{\mathrm{o}} \\ c : \alpha = {}^{\uparrow}\alpha_{\mathrm{o}}\end{array}\right.$$
$$\text{is-fibrant}_{\text{weak}}(\alpha) = \left[\begin{array}{r}\alpha_{\mathrm{o}} : \mathrm{Type}_{\mathrm{o}} \\ c : \alpha \simeq {}^{\uparrow}\alpha_{\mathrm{o}}\end{array}\right.$$

We can extend this definition from types to maps.

Definition 4.13 (Fibrations)**.** A map $f : X \to Y$ is said to be a (strict) *fibration* if each of its fibers are (strictly) fibrant.

Definition 4.14 (Reedy presheaves)**.** Let D be a direct category. A presheaf $X : \mathrm{Psh}(D)$ is said to be *Reedy* if, for every $i : D$, $\mathrm{can}_{i,X}$ is a strict fibration, where $\mathrm{can}_{i,X} : X_i \to \mathrm{M}_i\, X$ is the canonical map.

4.2.1 Layered presheaf lemmas

Lemma 4.15. Let D be a direct category and $n : \mathbb{N}$.

$$\mathrm{Psh}_{\mathrm{rd}}(D^{<n+1}) \simeq \left[\begin{array}{rl}\mathrm{X} : & \mathrm{Psh}(D^{<n}) \\ \mathrm{reedy} : & \prod_{i:D^{<n}} \text{is-fibration}(\mathrm{can}_{i,\mathrm{X}}) \\ \mathrm{Y} : & D^{=n} \to \mathrm{Type} \\ \mathrm{u} : & \prod_{i:D^{=n}} \prod_{j:D^{<n}} (j \to i) \to (\mathrm{Y}(i) \to \mathrm{X}(j)) \\ \mathrm{coh} : & \prod_{i:D^{=n}} \prod_{j,j':D^{<n}} \prod_{f:j \to i} \prod_{g:j' \to j} \mathrm{X}(g) \circ \mathrm{u}_{i,j}(f) = \mathrm{u}_{i,j'} \\ \mathrm{fib} : & \prod_{i:D^{=n}} \text{is-fibration}(\mathrm{can}_{i,e_n^{-1}(\mathrm{X},\mathrm{Y},\mathrm{u},\mathrm{coh})})\end{array}\right.$$

Proof. The direct part of the equivalence is obtained by reusing the previously defined equivalence e_n, seeing that the only things that we have more than in $\mathrm{LPsh}_n(D)$ are the Reedy fibrancy conditions, which are verified because we consider Reedy fibrant presheaves to start with.

The reverse direction is, similarly, obtained by building a presheaf in $\mathrm{Psh}(D^{<n+1})$ first, using e_n^{-1}, and then proving it is Reedy fibrant, which stems directly from the additional information we have stored. □

Lemma 4.15 states that Reedy presheaves below $n+1$ are exactly Reedy presheaves below n plus some glueing information (the same as for regular presheaves), plus some fibrancy conditions on that glueing information. However, it turns out that this fibrancy condition allows us to reformulate the glueing in a much more compact way, that is, we can encode it by storing just the (fibrant) fibers over a *matching object.* The following definition and statement will make this intuition clearer.

Definition 4.16 (Matching object)**.** Let $i : D$, and X be a presheaf over $D^{<\varphi(i)}$. We define the matching object $\mathrm{M}_i\, X$ as being the limit of the composite

$$i/\!\!/D \hookrightarrow D^{<\varphi(i)} \xrightarrow{X} \mathrm{Type}$$

where $i/\!\!/D$ is the "category (strictly) under i", whose elements are couples (j, f) where $j : D$ and $f : j \to i$, where $j \neq i$.

Theorem 4.17. Let D be a direct category, and $n : \mathbb{N}$.

$$\mathrm{Psh}_{\mathrm{rd}}(D^{<n+1}) \simeq \left[\begin{array}{c} \mathrm{X} : \mathrm{Psh}_{\mathrm{rd}}(D^{<n}) \\ \mathrm{glueing} : \prod_{i:D^{=n}} \mathrm{M}_i\, \mathrm{X} \to \mathrm{Type}_{\mathrm{o}} \end{array}\right.$$

Proof. We will reuse the very last lemma. We want to show that the glueing field is exactly as much information as the fields Y, u, coh and fib. More precisely, given a $X : \mathrm{Psh}_{\mathrm{rd}}(D^{<n})$, we want to show that

$$\prod_{i:D^{=n}} \mathrm{M}_i\, X \to \mathrm{Type}_{\mathrm{o}} \simeq \left[\begin{array}{r} \mathrm{Y} : D^{=n} \to \mathrm{Type} \\ \mathrm{u} : \prod_{i:D^{=n}} \prod_{j:D^{<n}} (j \to i) \to (\mathrm{Y}(i) \to X(j)) \\ \mathrm{coh} : \prod_{i:D^{=n}} \prod_{j:D^{<n}} \prod_{f:j\to i} \prod_{g:j'\to j} X(g) \circ \mathrm{u}_{i,j}(f) = \mathrm{u}_{i,j'} \\ \mathrm{fib} : \prod_{i:D^{=n}} \text{is-fibration}(\mathrm{can}_{i,e_n^{-1}(X,\mathrm{Y},\mathrm{u},\mathrm{coh})}) \end{array}\right.$$

For the forward direction, consider $g : \prod_{i:D^{=n}} \mathrm{M}_i\, X \to \mathrm{Type}_{\mathrm{o}}$ a glueing. We let

$$Y(i) :\equiv \sum_{z:\mathrm{M}_i\, X} g_i(z)$$

This precisely makes the canonical morphism a fibration. Furthermore, we define

$$u_{i,j}(f) :\equiv (z, y) \mapsto \psi_{j,f}(z)$$

where $\psi_{j,f} : \mathrm{M}_i\, X \to X(j)$ is the component of the cone over $\mathrm{M}_i\, X$, at (j, f). The coherence condition coh is true because of the defining property of ψ being a cone.

Conversely, consider $l : \left[\begin{array}{r} \mathrm{Y} : D^{=n} \to \mathrm{Type} \\ \mathrm{u} : \prod_{i:D^{=n}} \prod_{j:D^{<n}} (j \to i) \to (\mathrm{Y}(i) \to X(j)) \\ \mathrm{coh} : \prod_{i:D^{=n}} \prod_{j:D^{<n}} \prod_{f:j\to i} \prod_{g:j'\to j} X(g) \circ \mathrm{u}_{i,j}(f) = \mathrm{u}_{i,j'} \\ \mathrm{fib} : \prod_{i:D^{=n}} \text{is-fibration}(\mathrm{can}_{i,e_n^{-1}(X,\mathrm{Y},\mathrm{u},\mathrm{coh})}) \end{array}\right.$. For $i : D^{=n}$ and $z : \mathrm{M}_i\, X$, we define $g_i(z)$ by

$$g_i(z) = \mathrm{fib}_{\pi_1}(z)$$

where $\pi_1 : l.\mathrm{Y}(i) \to \mathrm{M}_i\, X$ □

4.2.2 Fibrancy of the matching object

Theorem 4.18. Let D be a direct category of finite height, A a presheaf over D, and X a (weakly) Reedy fibrant presheaf over D. $\mathrm{Psh}(D)(A, X)$ is weakly fibrant.

To prove this theorem, we will need to "truncate" functors, in the same spirit as we "truncate" a direct category D up to a rank n: $D^{<n}$.

Definition 4.19 (skeleton functor)**.** Let n be a natural number, and A a presheaf over D. We define the *skeleton functor* $\mathrm{Sk}^{<n}(A)$ as the presheaf over D which only keeps elements of rank less than n in a, ie. for $d : D$

$$\mathrm{Sk}^{<n}(A)(d) = \begin{cases} A(d) & \text{if } \operatorname{rank} d < n \\ 0 & \text{otherwise} \end{cases}$$

Proof. We will show that the function $\mathrm{Psh}(D)(A, X) \to 1$ is a fibration. To do so, we can consider the (unique) morphism $0 \to A$ as follows

$$\begin{tikzcd}
\mathrm{Sk}^{<0}(A) \arrow[r, equal] \arrow[d] & 0 \\
\mathrm{Sk}^{<1}(A) \arrow[d] & \\
\mathrm{Sk}^{<2}(A) \arrow[d] & \\
\vdots \arrow[d] & \\
\mathrm{Sk}^{<h}(A) \arrow[r, equal] & A
\end{tikzcd}$$

where $h : \mathbb{N}$ is the height of D, and at each step the morphism is the subpresheaf morphism.

We can check that, for every $n < h$, the morphism from $\mathrm{Sk}^{<n}(A) \to \mathrm{Sk}^{<n+1}(A)$ is part of a pushout

$$\begin{tikzcd}
\coprod_{i:D^{=n}} \coprod_{a:A(i)} \partial i \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near end] & \mathrm{Sk}^{<i}(A) \arrow[d] \\
\coprod_{i:D^{=n}} \coprod_{a:A(i)} \mathbf{y}i \arrow[r] & \mathrm{Sk}^{<i+1}(A)
\end{tikzcd}$$

Indeed, this can be checked pointwise. Let $d : D$. We want to check that the following diagram is a pushout

$$\begin{tikzcd}
(\coprod_{i:D^{=n}} \coprod_{a:A(i)} \partial i)(d) \arrow[r] \arrow[d] & \mathrm{Sk}^{<n}(A)(d) \arrow[d] \\
(\coprod_{i:D^{=n}} \coprod_{a:A(i)} \mathbf{y}i)(d) \arrow[r] & (\mathrm{Sk}^{<n+1}(A))(d)
\end{tikzcd}$$

If $\operatorname{rank} d < i$, then

$$\begin{aligned}
(\coprod_{i:D^{=n}} \coprod_{a:A(i)} \partial i)(d) &= \sum_{i:D^{=n}} \sum_{a:A(i)} D(d, i) \\
(\coprod_{i:D^{=n}} \coprod_{a:A(i)} \mathbf{y}i)(d) &= \sum_{i:D^{=n}} \sum_{a:A(i)} D(d, i) \\
\mathrm{Sk}^{<n}(A)(d) &= A(d) \\
\mathrm{Sk}^{<n+1}(A)(d) &= A(d)
\end{aligned}$$

so the diagram is clearly a pushout

$$\begin{tikzcd}
\sum_{i:D^{=n}} \sum_{a:A(i)} D(d,i) \arrow[r] \arrow[d, equal] & A(d) \arrow[d, equal] \\
\sum_{i:D^{=n}} \sum_{a:A(i)} D(d,i) \arrow[r] & A(d)
\end{tikzcd}$$

If $\operatorname{rank} d = i$, then

$$\begin{aligned}
(\coprod_{i:D^{=n}} \coprod_{a:A(i)} \partial i)(d) &= \sum_{i:D^{=n}} \sum_{a:A(i)} 0 = 0 \\
(\coprod_{i:D^{=n}} \coprod_{a:A(i)} \mathbf{y} i)(d) &= \sum_{i:D^{=n}} \sum_{a:A(i)} D(d,i) = A(d) \\
\mathrm{Sk}^{<n}(A)(d) &= 0 \\
\mathrm{Sk}^{<n+1}(A)(d) &= A(d)
\end{aligned}$$

so the diagram is a pushout

$$\begin{tikzcd}
0 \arrow[r, equal] \arrow[d] & 0 \arrow[d] \\
A(d) \arrow[r, equal] & A(d)
\end{tikzcd}$$

Finally, if $\operatorname{rank} d > n$, then

$$\begin{aligned}
(\coprod_{i:D^{=n}} \coprod_{a:A(i)} \partial i)(d) &= 0 \\
(\coprod_{i:D^{=n}} \coprod_{a:A(i)} \mathbf{y} i)(d) &= 0 \\
\mathrm{Sk}^{<n}(A)(d) &= 0 \\
\mathrm{Sk}^{<n+1}(A)(d) &= 0
\end{aligned}$$

Consider now the composition of morphisms that we obtain by applying $\mathrm{Psh}(D)(-,X)$:

$$\begin{tikzcd}
\mathrm{Psh}(D)(\mathrm{Sk}^{<0}(A),X) \arrow[r, equal] & 1 \\
\mathrm{Psh}(D)(\mathrm{Sk}^{<1}(A),X) \arrow[u] & \\
\mathrm{Psh}(D)(\mathrm{Sk}^{<2}(A),X) \arrow[u] & \\
\vdots \arrow[u] & \\
\mathrm{Psh}(D)(\mathrm{Sk}^{<h}(A),X) \arrow[u] \arrow[r, equal] & \mathrm{Psh}(D)(A,X)
\end{tikzcd}$$

Because $\mathrm{Psh}(D)(-,X)$ is continuous, each morphism at step $n < h$ is part of a pullback:

$$\begin{tikzcd}
\mathrm{Psh}(D)(\mathrm{Sk}^{<n+1}(A),X) \arrow[r] \arrow[d] \arrow[dr, phantom, "\ulcorner"] & \prod_{i:D^{=n}} \prod_{a:A(i)} \mathrm{Psh}(D)(\mathbf{y} i,X) \arrow[d] \\
\mathrm{Psh}(D)(\mathrm{Sk}^{<n}(A),X) \arrow[r] & \prod_{i:D^{=n}} \prod_{a:A(i)} \mathrm{Psh}(D)(\partial i,X)
\end{tikzcd}$$

Because X is Reedy fibrant, the map $\mathrm{Psh}(D)(\mathbf{y}i, X) \to \mathrm{Psh}(D)(\partial i, X)$ is a fibration. □

Corollary 4.19.1. Let D be a direct category of finite height, and X a weakly Reedy fibrant presheaf over D. X has a fibrant limit.

Proof. By Theorem 4.18, $\mathrm{Psh}(D)(1, X)$ is weakly fibrant, where 1 is the constant unit presheaf. □

Corollary 4.19.2. Let D be a direct category, $i : D$, and X a Reedy fibrant presheaf over $D^{<\varphi(i)}$. $\mathrm{M}_i(X)$ is weakly fibrant.

Proof. $\mathrm{M}_i(X)$ is a limit of a presheaf over a direct category whose height is bounded by $\varphi(i)$. Hence, by Corollary 4.19.1, it is (weakly) fibrant. □

4.3 Main result

Theorem 4.20. Let D be a fibrant category such that, for every $n : \mathbb{N}$, $D^{=n}$ is cofibrant. Then, for every $n : \mathbb{N}$, $\mathrm{Psh}_{\mathrm{rd}}(D^{<n})$ is weakly fibrant.

Proof. By induction on n. The case where $n = 0$ is straightforward, because $D^{<0} \simeq 0$, hence $\mathrm{Psh}_{\mathrm{rd}}(D^{<0}) \simeq 1$ which is weakly fibrant.

Suppose this is true for $n : \mathbb{N}$. By Theorem 4.17, we have

$$\mathrm{Psh}_{\mathrm{rd}}(D^{<n+1}) \simeq \left[\begin{array}{c} \mathrm{X} : \mathrm{Psh}_{\mathrm{rd}}(D^{<n}) \\ \text{glueing} : \prod_{i:D^{=n}} \mathrm{M}_i(\mathrm{X}) \to \mathrm{Type}_{\mathrm{o}} \end{array}\right.$$

By Corollary 4.19.2, for any $X : \mathrm{Psh}_{\mathrm{rd}}(D^{<n})$, $\mathrm{M}_i(X)$ is fibrant, so $X \to \mathrm{Type}_{\mathrm{o}}$ is too. By assumption, $D^{=n}$ is cofibrant, so $\prod_{i:D^{=n}} \mathrm{M}_i(X) \to \mathrm{Type}_{\mathrm{o}}$ is fibrant too. By the induction hypothesis, $\mathrm{Psh}_{\mathrm{rd}}(D^{<n})$ is fibrant. Because (weak) fibrancy is preserved by equivalence, $\mathrm{Psh}_{\mathrm{rd}}(D^{<n+1})$ is fibrant. □

Corollary 4.20.1. For $n : \mathbb{N}$, $\mathrm{Psh}_{\mathrm{rd}}(\Delta_+^{<n})$ is fibrant.

Proof. This is a direct consequence of Theorem 4.20, as Δ_+ is a direct category with finite layers, and as finite types are cofibrant [Ann+23, Lemma 3.25]. □

Definition 4.21 (Truncated semi-simplicial types)**.** For any $n : \mathbb{N}$, $\mathrm{Psh}_{\mathrm{rd}}(\Delta_+^{<n})$, so there exists a $\Delta_{\mathrm{o}_n} : \mathrm{Type}_{\mathrm{o}}$ such that $\mathrm{Psh}_{\mathrm{rd}}(\Delta_+^{<n}) \simeq {}^{\uparrow}\Delta_{\mathrm{o}_n}$. We define the *truncated semi-simplicial types* as $\Delta_{\mathrm{o}} : \mathbb{N} \to \mathrm{Type}_{\mathrm{o}}$, defined by, for every $n : \mathbb{N}$,

$$\Delta_{\mathrm{o}}(n) \equiv \Delta_{\mathrm{o}_n}$$

This allows us to state the following conjecture

Conjecture 4.22. There is no object-level truncated semi-simplicial type, ie a type family $(\tau_n)_{n:\mathbb{N}_{\mathrm{o}}}$ such that, for every $n : \mathbb{N}$,

$$\tau_{\langle n\rangle} = \Delta_{\mathrm{o}}(n)$$

where $\langle - \rangle : \mathbb{N} \to \mathbb{N}_{\mathrm{o}}$ is the natural map.

References

[Voe13] Vladimir Voevodsky. "A Simple Type System with Two Identity Types". Unpublished note. Feb. 23, 2013. URL: `https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/HTS.pdf`.

[Voe14] Vladimir Voevodsky. *The Origins and Motivations of Univalent Foundations*. Letter. Oct. 3, 2014. URL: `https://www.ias.edu/ideas/2014/voevodsky-origins`.

[Her15] Hugo Herbelin. "A Dependently-Typed Construction of Semi-Simplicial Types". In: 25.5 (June 2015), pp. 1116–1131. ISSN: 0960-1295, 1469-8072. DOI: `10.1017/S0960129514000528`.

[ACK16] Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. "Extending Homotopy Type Theory with Strict Equality". In: 62 (2016), 21:1–21:17. ISSN: 1868-8969. DOI: `10.4230/LIPICS.CSL.2016.21`.

[Ahr+22] Benedikt Ahrens, Paige Randall North, Michael Shulman, and Dimitris Tsementzis. *The Univalence Principle*. Aug. 29, 2022. DOI: `10.48550/arXiv.2102.06275`. arXiv: `2102.06275 [cs, math]`. Pre-published.

[Kov22] András Kovács. "Staged Compilation with Two-Level Type Theory". Aug. 29, 2022. DOI: `10.1145/3547641`. arXiv: `2209.09729 [cs]`.

[Ann+23] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. "Two-Level Type Theory and Applications". Sept. 2023. DOI: `10.1017/S0960129523000130`. arXiv: `1705.03307 [cs]`.

[Usk23] Elif Uskuplu. *Formalizing Two-Level Type Theory with Cofibrant Exo-Nat*. Sept. 17, 2023. DOI: `10.48550/arXiv.2309.09395`. arXiv: `2309.09395 [cs, math]`. Pre-published.

[Uni] Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundation of Mathematics*. URL: `https://homotopytypetheory.org/book`.