ÉCOLE NORMALE SUPÉRIEURE DE LYON

AN INTERNSHIP REPORT
ON

Analyzing Proof Terms in Homotopical Type Theory: Toward Better Proofs

Submitted in partial fullfilment of the requirements for the award of the degree of

MASTER 1
IN
CONCEPTS AND APPLICATIONS OF COMPUTER SCIENCE

by

Johann Rosain

Under the Guidance of

Thierry Coquand

Hosted in

CHALMERS UNIVERSITY OF TECHNOLOGY
LOGIC AND TYPES UNIT

Contents

1 Introduction

Homotopy type theory (HoTT) arose from Vladimir Vœvodsky's desire to write mathematics without "worry[ing] about making a mistake in [his] work"[1]. The achievement of this goal has necessitated to dispense with others mathematicians, as "who would ensure that [he] did not forget something and did not make a mistake, if even the mistakes in much more simple arguments take years to uncover?"[1], and to replace them with a more trustworthy partner — a computer.

Indeed, machine-checked proof systems have been in development since the early sixties, taking full advantage of the constructivity of proofs in Per Martin-Löf type theories [ML82] (MLTT) to yield computable proof terms and provide a witness to the truth of a logical formula. However, taking root in predicate logic, the theoretic foundations of these tools were too limited to express high-level mathematics: many objects could not be defined, and arguments often used by mathematicians were unable to be formalized.

Consequently, HoTT has been constructed to be naturally suited as a foundation of mathematics (i.e., as a theory in which mathematics can be stated formally) that can be implemented in a proof assistant. It is composed of three building blocks, presented in details in §2, MLTT, the *univalence axiom*, and *higher inductive types*, and has seen successful development in popular proof assistants such as Coq and its UniMath [VAG+] library or Agda with agda-unimath [RSPC+].

Moreover, the computability of those proofs is not compromised, as univalence has an actual meaning in cubical sets [BCH13], that is, univalence is actually provable in proof assistants implementing this theory such as Cubical Agda [VMA19]. However, such tools suffer from a severe lack of efficiency [Kov23] when evaluating proof terms. For instance, computing the number of groups of order n up to isomorphism is nigh impossible in Cubical Agda's implementation, for $n \geqslant 2$[2]. Recently, two solutions have been envisaged to counteract the performances issues, the first one being the optimization of the evaluation function, as proposed in [Kov23], and the second one leaning more towards the optimization of the proof itself.

This work takes place in the latter context, with the aim to analyze the computation of Egbert Rijke's proof of finiteness of groups of finite order up to isomorphism [Rij22a], and find out which operation(s) make(s) the computation so difficult. In order to meet this goal, a new formalization of this proof have been implemented in

[1] c.f. his blog post about the origin of univalent foundations: https://www.ias.edu/ideas/2014/voevodsky-origins.
[2] See: https://agda.github.io/cubical/Cubical.Experiments.CountingFiniteStructure.html

a tool called postt[3], that implements head-linear reduction — a step-by-step unfolding of the head symbol — of λ-terms and consequently allows an analysis of the computation.

As the first large-scale project of postt, this work has led to three main contributions: (i) the start of a HoTT standard library containing results of [Uni13, Rij22b] for postt's proof language, (ii) an implementation of E. Rijke's proof in this language and (iii) an analysis of the computation of this proof for fixed values of n.

This document presents these contributions, and is organized as follows. §2 exposes preliminary knowledge, from MLTT to Vœvodsky's univalent mathematics. Then, §3 details the proof of [Rij22a] about finiteness of finite structures up to isomorphism, and §4 analyzes the complexity of the algorithm underlying this proof.

Throughout the document, clickable pictograms λ are present. Each one leads to the formal definition or proof associated to the previous or following statement.

2 Preliminary Knowledge

This section introduces a chosen subset of homotopy type theory that is sufficient to make this document self-contained. However, although §2.1 exposes important concepts and notations, these will be skimmed over as they are standard in the literature. Hence, a reader unfamiliar with type theory may directly refer to [Rij22b, §1, §6] or [Uni13, §1] for a more comprehensive presentation of these essential notions. Then, §2.2 presents Per Martin-Löf's dependent type theory [ML98] by relating types with (the hopefully more familiar) logic formulae (using the so-called Curry-Howard isomorphism). Afterwards, §2.3 tackles the main peculiarity of HoTT — identity types — and §2.4 states standard lemmas and theorems that follow from this interpretation.

2.1 Basic Notions and Notations

Type theory is a formalism that aims to be a foundation for constructive mathematics. Conversely to set theory, type theory is intrinsically equipped with a formal system that is used to construct any object manipulated. These objects are of two kinds: elements (or terms, like $\lambda x.x$, 3, …) and types (like $X \to X$, $\mathbb{N}$, …). The former automatically come together with a type, that is unique. The latter also come with a type, but the definition is more subtle and needs to drop unicity to avoid the type-theoretic version of Russel's paradox λ [Coq92].

Formal Type System. In type theory, introducing a new kind of type is done by giving rules: (i) the formation rules, (ii) the introduction rules, (iii) the elimination rules and (iv) the computation rules. The first explain how to form types of this kind, the second how to construct elements of that type by introducing *constructor(s)* and the third how to use elements of that type by introducing *eliminator(s)*. The fourth specifies how an eliminator acts on a constructor. In this document, we follow Rijke's [Rij22b] presentation and use *inference rules* to represent type theory's formal system. An inference rule is written

$$\frac{\mathcal{H}_1 \qquad \mathcal{H}_2 \qquad \cdots \qquad \mathcal{H}_n}{\mathcal{C}}$$

where $\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_n$ are *judgments* for the premises that allow to conclude $\mathcal{C}$, a judgment for the conclusion. A judgment *for terms* has two forms. The first is $\Gamma \vdash a : A$, where a is an element of type A in the *context* Γ. A context is a *finite list* of variable declarations $x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n$. The shortcut $a : A$ will be used in place of $\emptyset \vdash a : A$ as the terms considered in this document are always *closed*, that is, no context is needed to type them. The second judgment for terms follows the notations of [Uni13] and is of the form $\Gamma \vdash a \equiv b : A$, where a and b are definitionally (or judgmentally) equal elements of type A, ensuring that a can be rewritten to b. For instance, defining the square function will be denoted $(-)^2 :\equiv \lambda x.x \cdot x$, and the judgment $3^2 \equiv 9 : \mathbb{N}$ can be *derived*, i.e., there exists a finite tree rooted at $3^2 \equiv 9 : \mathbb{N}$ such that all its leaves are *axioms* — inference rules without hypotheses. We assume the standard rules [Rij22b, §1.3, §1.4] about the formation of contexts, types, their elements and judgmental equality (which state that definitional equality is an equivalence relation), together with capture-free substitution and weakening rules.

[3] Available at: https://github.com/JonasHoefer/poset-type-theory/

Universes. To give a type to types while avoiding the type-theoretic version of Russel's paradox, a hierarchy of universes $\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2, \ldots$ is considered, where every universe $\mathcal{U}_i$ is an element of the next universe $\mathcal{U}_{i+1}$. *Cumulative* universes are assumed, i.e., universes are such that all elements of the i^{th} universe are also elements of the $(i+1)^{\text{th}}$ universe. It is convenient in most ways, but has the consequence that *types* do not have a unique type. Indeed, A is a type if it inhabits some universe $\mathcal{U}_i$, but as universes are cumulative, if $A : \mathcal{U}_i$ then $A : \mathcal{U}_{i+1}$. However, this property has the pleasant advantage to, given $\mathcal{U}_i$ and $\mathcal{U}_j$ two universes, offer maps $\mathcal{U}_i \to \mathcal{U}_i \sqcup \mathcal{U}_j$ and $\mathcal{U}_j \to \mathcal{U}_i \sqcup \mathcal{U}_j$, where $\mathcal{U}_i \sqcup \mathcal{U}_j$ is the universe that contains all elements of $\mathcal{U}_i$ and of $\mathcal{U}_j$. This map is straightforward as $\mathcal{U}_i \sqcup \mathcal{U}_j$ is $\mathcal{U}_{\max(i,j)}$. As such, it is common practice to (purposefully) forget the level of a universe in pen-and-paper presentations of type theory, writing $A : \mathcal{U}_i$ as $A : \mathcal{U}$. As such, to complete the formal system, judgments for types be defined as follows: $\Gamma \vdash A : \mathcal{U}$ if A is a type, and $\Gamma \vdash A \equiv B : \mathcal{U}$ if A and B are two definitionally equal types.

2.2 Martin-Löf's Dependent Types

Π-types. The core idea of dependent types is to define *functions* such that their output type is parameterized by their input type(s). For example, assume that for all $n : \mathbb{N}$, $\mathsf{array}(n)$ is a type. Then, in Martin-Löf's type theory,

$$\mathsf{append}(n, m) : \mathsf{array}(n) \to \mathsf{array}(m) \to \mathsf{array}(n + m)$$

is a *dependent function*, i.e., for all $n, m : \mathbb{N}$, there is a function $\mathsf{append}(n, m)$ that concatenates arrays of respective sizes n and m. The type of append is then denoted $\prod_{(n,m:\mathbb{N})} \mathsf{array}(n) \to \mathsf{array}(m) \to \mathsf{array}(n + m)$. Here, $\prod$ can be thought of as the type-theoretical equivalent to the logical $\forall$. As such, the grammar of the expressions of dependent type theory cannot be split into a term and a type grammar, and is thus given by

$$e, e' \quad ::= \quad x \mid (e)\ e' \mid \lambda x.e \mid \prod_{x:e} e' \mid \mathcal{U}.$$

Then, as mentioned in §2.1, four types of rules [Mim20, §8.1] are needed to understand and manipulate Π-types.

(formation rule) This rule defines the *type family* B over A.

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma, x : A \vdash B(x) : \mathcal{U}}{\Gamma \vdash \prod_{(x:A)} B(x) : \mathcal{U}}\ \Pi_F$$

(introduction rule) A type family B over A is the type of a dependent function, hence the associated term of this kind of type is a λ-abstraction.

$$\frac{\Gamma, x : A \vdash t : B(x)}{\Gamma \vdash \lambda x.t : \prod_{(x:A)} B(x)}\ \Pi_I$$

(elimination rule) Using a function is done by applying a value of the correct type to it.

$$\frac{\Gamma \vdash t : \prod_{(x:A)} B(x) \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B(u)}\ \Pi_E$$

(computation rules) Dependent functions behave as expected.

$$\frac{\Gamma, x : A \vdash t : B(x) \qquad \Gamma \vdash u : A}{\Gamma \vdash (\lambda x.t)\ u \equiv t[u/x] : B(u)}\ \Pi_C \qquad \frac{\Gamma \vdash f : \prod_{(x:A)} B(x)}{\Gamma \vdash \lambda x.f\ x \equiv f : \prod_{(x:A)} B(x)}\ \Pi_U$$

The first rule is exactly the defining rule of the β-reduction (i.e., the β-reduction is the least binary relation containing Π_C), while the second is known as the η-rule and states the uniqueness of functions.

Remark: *There are some points to take note of in the above rules. First, if the λ-terms are erased, then the introduction (resp. elimination) rule is the same as $\forall$'s introduction (resp. elimination) rule in higher-order natural deduction. As such, a term of a type can be constructed if its associated logical formula is true. Such a type is said* inhabited. *Moreover, Π-types define functions, thus the type $A \to B$ is simply the* constant type family *B over A. Hence, the type of functions from A to B is defined as:*

$$A \to B :\equiv \prod_{\_ :A} B.$$

Finally, two types A and B are logically equivalent *whenever there are back-and-forth maps between those types.*

Inductive Types. The others type-theoretic equivalents of logic connectors are called *inductive types*, as (i) the type corresponding to their logical induction principle is inhabited and therefore offers a way to construct terms of an (inductive) type from its constructors and (ii) they are an instance of a more general inductive framework [Uni13, §5], [Rij22b, §20] (not covered in this document). Hence, they could also be defined using inference rules, but from this paragraph on, a more informal style is adopted.

A straightforward example of inductive types is the *unit type* λ, denoted $\mathbf{1}$ and equipped with a unique term $\star : \mathbf{1}$. It satisfies the induction principle $\mathsf{ind}_{\mathbf{1}}$, that is, for any family of types P indexed by $\mathbf{1}$, there is a function

$$\mathsf{ind}_{\mathbf{1}} : P(\star) \to \prod_{x:\mathbf{1}} P(x).$$

This induction principle can also be thought of as the *eliminator* for $\mathbf{1}$. As such, it induces the computation rule $\mathsf{ind}_{\mathbf{1}}(p, \star) \equiv p$. Moreover, as $\mathbf{1}$ has a unique element, the η-rule behaves as expected — that is, if $\Gamma \vdash t : \mathbf{1}$, then $\Gamma \vdash t \equiv \star : \mathbf{1}$ λ.

As there is a type with a unique element, it is natural to wonder whether there is also a type with no elements. In fact, there is one such type — the *empty type* $\mathbf{0}$ λ — that is a *degenerate* inductive type, in the sense that it is a type with no constructors. It, however, comes with an induction principle $\mathsf{ind}_{\mathbf{0}} : \prod_{(x:\mathbf{0})} P(x)$ that should be familiar: a special case of this induction principle is the *ex falso quodlibet*. That is, for any type A, there is a function $\text{ex-falso} :\equiv \mathsf{ind}_{\mathbf{0}} : \mathbf{0} \to A$. As should now be clear, $\mathbf{1}$ and $\mathbf{0}$ are the type-theoretic equivalents of $\top$ and $\bot$. As such, for any type A, its negation is defined by $\neg A :\equiv A \to \mathbf{0}$ λ. Sometimes, a type A will be qualified as *empty* if there is an element of type $\neg A$, i.e., $\text{is-empty}(A) :\equiv \neg A$.

The disjunction is another uncomplicated inductive type, but one that features multiple constructors and (in general) more than one element. For A and B two types, the *coproduct* $A + B$ λ is the type with two constructors $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$ such that, for any family of types P indexed by $A + B$, there is a term

$$\mathsf{ind}_{+} : \left(\prod_{x:A} P(\mathsf{inl}(x))\right) \to \left(\prod_{y:B} P(\mathsf{inr}(y))\right) \to \prod_{z:A+B} P(z)$$

for which the computation rules $\mathsf{ind}_{+}(f, g, \mathsf{inl}(x)) \equiv f(x)$ and $\mathsf{ind}_{+}(f, g, \mathsf{inr}(y)) \equiv g(y)$ hold.

The last type corresponding to a logical connector is also the second dependent type of Martin-Löf's type theory, the Σ-type. Σ-types correspond to existential quantifiers. In constructive mathematics, the subtle point of such quantifiers is that, conversely to classical logic, a proof of the statement "there exists x such that $P(x)$" requires an explicit exhibition of x. Thus, Σ-types are *pairs* (x, p) where $p : P(x)$. Σ-types are defined by the following rules.

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma, x : A \vdash B(x) : \mathcal{U}}{\Gamma \vdash \sum_{(x:A)} B(x) : \mathcal{U}}\ \Sigma_F \qquad \frac{\Gamma, x : A \vdash B(x) : \mathcal{U} \qquad \Gamma \vdash t : A \qquad \Gamma, t : A \vdash u : B(t)}{\Gamma \vdash (t, u) : \sum_{(x:A)} B(x)}\ \Sigma_I$$

$$\frac{\Gamma, z : \sum_{(x:A)} B(x) \vdash P(z) : \mathcal{U} \qquad \Gamma \vdash t : \sum_{(x:A)} B(x) \qquad \Gamma, x : A, y : B(x) \vdash g : P(x, y)}{\Gamma \vdash \mathsf{ind}_{\Sigma}(g, t) : P(t)}\ \Sigma_E$$

$$\frac{\Gamma \vdash x : A \qquad \Gamma, x : A \vdash u : B(x) \qquad \Gamma, z : \sum_{(x:A)} B(x) \vdash P(z) : \mathcal{U} \qquad \Gamma \vdash g : \prod_{(x:A)} \prod_{(y:B(x))} P(x, y)}{\Gamma \vdash \mathsf{ind}_{\Sigma}(g, (x, y)) \equiv g(x)(y) : P(x, y)}\ \Sigma_C$$

For instance, the first and second projections are defined by Σ-induction:

$$\mathsf{fst} : \left(\sum_{x:A} B(x)\right) \to A \qquad \mathsf{snd} : \prod_{t:\sum_{(x:A)} B(x)} B(\mathsf{fst}(t))$$
$$\mathsf{fst}(x,y) :\equiv \mathsf{ind}_\Sigma(\lambda x\, y.x, (x,y)) \qquad \mathsf{snd}(x,y) :\equiv \mathsf{ind}_\Sigma(\lambda x\, y.y, (x,y))$$

Consequently, if B is a constant family of types over A, then (x,y) where $x : A$ and $y : B$ is a witness of $A \wedge B$. The corresponding type is, of course, the cartesian product and defined as:

$$A \times B :\equiv \sum_{\_:A} B.$$

Actually, this follows the intuition from natural numbers, where $m \times n \equiv \sum_{(i=0)}^{(m-1)} n$. The dependant sum corresponds to summing a finite family $(n_i)_{0 \leqslant i < m}$ of natural numbers, while the cartesian product is the special case where $n_i = n$ for all i.

The final (for our purposes) useful inductive type outside logical connectors are natural numbers $\mathbb{N}$. In type theory, they are defined λ as a *unary* encoding using two constructors, $0_\mathbb{N} : \mathbb{N}$ and $\mathsf{succ}_\mathbb{N} : \mathbb{N} \to \mathbb{N}$, equipped with the usual induction principle λ on $\mathbb{N}$ for any type family P over $\mathbb{N}$:

$$\mathsf{ind}_\mathbb{N} : P(0_\mathbb{N}) \to \left(\prod_{(n:\mathbb{N})} P(n) \to P(\mathsf{succ}_\mathbb{N}(n))\right) \to \prod_{(n:\mathbb{N})} P(n).$$

The computation rules of $\mathbb{N}$ are the usual ones of primitive recursion, i.e.,

$$\mathsf{ind}_\mathbb{N}(p_0, p_S, 0_\mathbb{N}) \equiv p_0$$
$$\mathsf{ind}_\mathbb{N}(p_0, p_S, \mathsf{succ}_\mathbb{N}(n)) \equiv p_S(n, \mathsf{ind}_\mathbb{N}(p_0, p_S, n))$$

Note that, in the informal paragraphs, multiple shortcuts have been used. In particular, right associativity of the arrow is assumed, and functions are parameterized by pairs instead of functional application. The latter is justified by Σ-induction (i.e., curryfication and uncurryfication of expressions). Finally, note that the judgmental equality is actually the reflexive, transitive and symmetric closure of the $\beta\eta$-reduction. As such, judgmental equality can be decided as if a term can be (dependently) typed, then it is strongly normalizing [ML98].

2.3 A Homotopic Interpretation of Identity Types

In Martin Löf's type theory, the equality is also a type, called *propositional equality* to distinguish it from the judgmental one. Indeed, it is often desirable to *prove* that two things, not necessarily judgmentally equal, are propositionally equal. For instance, consider the following functions:

$$\mathsf{add}_\mathbb{N}(m, 0_\mathbb{N}) :\equiv m \qquad \mathsf{add}'_\mathbb{N}(0_\mathbb{N}, n) :\equiv n$$
$$\mathsf{add}_\mathbb{N}(m, \mathsf{succ}_\mathbb{N}(n)) :\equiv \mathsf{succ}_\mathbb{N}(\mathsf{add}_\mathbb{N}(m,n)) \qquad \mathsf{add}'_\mathbb{N}(\mathsf{succ}_\mathbb{N}(m), n) :\equiv \mathsf{succ}_\mathbb{N}(\mathsf{add}'_\mathbb{N}(m,n)).$$

They are *not* judgmentally equal — as can be seen from the defining λ-terms for these functions:

$$\mathsf{add}_\mathbb{N} \equiv \lambda m n.\, \mathsf{ind}_\mathbb{N}(m, \lambda n' r.\, \mathsf{succ}_\mathbb{N}(r), n) \qquad \mathsf{add}'_\mathbb{N} \equiv \lambda m n.\, \mathsf{ind}_\mathbb{N}(n, \lambda m' r.\, \mathsf{succ}_\mathbb{N}(r), m).$$

Nevertheless, for any $m, n : \mathbb{N}$, $\mathsf{add}_\mathbb{N}(m,n) \equiv \mathsf{add}'_\mathbb{N}(m,n)$. As such, one may want to state that $\mathsf{add}_\mathbb{N}$ is *equal* to $\mathsf{add}'_\mathbb{N}$, and that is exactly the purpose of propositional equality, written $\mathsf{add}_\mathbb{N} =_{\mathbb{N}\to\mathbb{N}\to\mathbb{N}} \mathsf{add}'_\mathbb{N}$ or $\mathsf{add}_\mathbb{N} = \mathsf{add}'_\mathbb{N}$ as there is no ambiguity on the type.

But as propositional equality is a type, there have to be terms (or proofs) of this type. Actually, in Martin-Löf type theories, propositional equality is defined as an inductive type with one constructor, $\mathsf{refl}_x : x = x$. As such,

an induction principle follows, stating that if $P(x,y,p)$ is a type family indexed over $x,y:A$ and $p:x=_A y$, then there is a function:

$$\mathsf{ind}_= : \left(\prod_{x:A} P(x,x,\mathsf{refl}_x)\right) \to \prod_{x,y:A}\prod_{p:x=y} P(x,y,p)$$

such that $\mathsf{ind}_=(f,x,x,\mathsf{refl}_x) \equiv f(x)$. Often, it is easier to use *based path induction* (λ) that considers a fixed element $a:A$ and a type family $P(x,p)$ over $x:A$ and $p:a=x$. The inductive principle then becomes:

$$\mathsf{ind}'_= : P(a,\mathsf{refl}_a) \to \prod_{x:A}\prod_{p:a=x} P(x,p)$$

with computation rule $\mathsf{ind}'_=(u,a,\mathsf{refl}_a) \equiv u$. As $\mathsf{ind}_=$ and $\mathsf{ind}'_=$ are equivalent [Uni13, §1.12], a (purposeful) confusion between them will be made throughout this document when proving properties.

Note that, as the identity $x=y$ is a type, it may be populated by multiple elements — that is, there may be multiple *proofs* that x and y can be identified. This situation is analogous to homotopy theory, where two elements can be connected by more than one (continuous) path. Hence, homotopy type theory chooses to interpret types as topological spaces, where elements of a type are points in the type's space and identifications are *continuous paths* (also called *continuous deformations*) between points in the space.

Moreover, this interpretation gives a complex algebraic structure to types — types are *higher groupoids*. Indeed, consider (i) the *path concatenation* operation (λ) and (ii) the inversion function (λ):

$$\mathsf{concat} : \prod_{x,y,z:A} (x=y) \to (y=z) \to (x=z) \qquad \mathsf{inv} : \prod_{x,y:A} (x=y) \to (y=x).$$

Given $x,y,z:A$, $p:x=y$ and $q:y=z$, $\mathsf{concat}(p,q)$ is denoted $p \cdot q$ and constructed by path induction over p. Assuming that y is x and $p \equiv \mathsf{refl}_x$, the goal[4] is to give a function of type $(x=z)\to(x=z)$ and the identity suffices. In turn, given $x,y:A$ and $p:x=y$, $\mathsf{inv}(p)$ is denoted p^{-1} and also constructed by path induction over p, with defining equation $\mathsf{refl}_x^{-1} :\equiv \mathsf{refl}_x$. Then:

Theorem 2.1: Types are Higher Groupoids

> Let $A:\mathcal{U}$, $x,y,z,w:A$ and $p:x=y$, $q:y=z$, $r:z=w$. The following types are inhabited:
>
> (i) $p = p\cdot\mathsf{refl}_y$ (λ) and $p = \mathsf{refl}_x\cdot p$ (λ) ;
>
> (ii) $p^{-1}\cdot p = \mathsf{refl}_y$ (λ) and $p\cdot p^{-1} = \mathsf{refl}_x$ (λ) and ;
>
> (iii) $(p\cdot q)\cdot r = p\cdot(q\cdot r)$ (λ).

The three proofs are straightforward by path induction over p. Moreover, note that for $p,q:x=_A y$, $p =_{x=_A y} q$ is also a type, and may also be inhabited. This goes on infinitely, thus giving the structure of a (weak) ∞-groupoid to types. Nevertheless, they keep the usual properties of equality — that is, an equivalence relation equipped with congruence.

Lemma 2.2:

> (i) Let $A,B:\mathcal{U}$ and $f:A\to B$. Then there is an *action on path* operation $\mathsf{ap}_f : \prod_{(x,y:A)}(x=y)\to(f(x)=f(y))$ (λ).
>
> (ii) Let $A:\mathcal{U}$ and $B:A\to\mathcal{U}$. Then there is a *transport* operation $\mathsf{tr}_B : \prod_{(x,y:A)}(x=y)\to B(x)\to B(y)$ (λ).

Proof: Both by path induction: (i) $\mathsf{ap}_f(\mathsf{refl}_x) :\equiv \mathsf{refl}_{f(x)}$ and (ii) $\mathsf{tr}_B(\mathsf{refl}_x) :\equiv \mathsf{id}_{B(x)}$. □

[4] To be completely formal, the goal would be to give an element of $\prod_{(z:A)}(x=z)\to(x=z)$. For instance, $\lambda z.\mathsf{id}$ suffices. One can then easily show that the order of (non inter-dependent) arguments of a function can be swapped.

2.4 The Univalent Foundations of Mathematics

Univalent foundation of mathematics arises from the formalization of mathematics in dependent type theory, using the homotopy interpretation of identity types and Vœvodsky's univalence axiom [Voe15]. The latter characterizes the identity types of universes, stating that equivalent types are equal

$$(A =_{\mathcal{U}} B) \simeq (A \simeq B).$$

This axiom is very useful, as it makes *isomorphic structures* identifiable, turning an often-used informal argument mechanizable. For instance, it is an important element of the proof that there are a finite number of groups of finite order k, that will be developed later in this document. Moreover, it has been found to be consistent when interpreting types as Kan simplicial sets [KL18], hence validating homotopy type theory as a plausible system to formalize mathematics in. Finally, a system that offers an actual computational meaning to univalence has been discovered when interpreting types as cubical sets [BCH13], overcoming the last objection of constructivists toward univalent mathematics. Most of the material covered in this section can be found in greater details in [Uni13, §3–4, §6–7] and [Rij22b, §9–10, §12–18].

Furthermore, the natural hierarchization of types originating from their homotopic interpretation gives a way to formally describe natural constructions such as propositions and sets. This hierarchy starts at level -2 so that the usual set-level mathematics take place at level 0.

Definition 2.3: Contractible Type λ

A type A is *contractible* if it comes equipped with an element of type

$$\mathsf{is\text{-}contr}(A) :\equiv \sum_{a:A} \prod_{x:A} a = x.$$

Given $(a, C) : \mathsf{is\text{-}contr}(A)$, $a : A$ is the *center of contraction* of A and $C : \prod_{(x:A)} a = x$ is the *contraction* of A.

Example: *The unit type is contractible* λ *as* $(\star, \mathsf{ind}_1(\mathsf{refl}_\star)) : \mathsf{is\text{-}contr}(\mathbf{1})$.

Definition 2.4: Vœvodsky's Homotopy Levels λ

Vœvodsky's homotopy levels (also called h-levels) are defined by induction on $\mathbb{Z}_{\geqslant -2}$ by

$$\mathsf{is\text{-}}(-2)\mathsf{\text{-}type}(A) :\equiv \mathsf{is\text{-}contr}(A)$$
$$\mathsf{is\text{-}}(k+1)\mathsf{\text{-}type}(A) :\equiv \prod_{x,y:A} \mathsf{is\text{-}}k\mathsf{\text{-}type}(x = y)$$

Definition 2.5: Proposition λ**, Set** λ

A type A is a *proposition* if it is a (-1)-type and a *set* if it is a 0-type. In particular, given a universe $\mathcal{U}$, the type of $\mathcal{U}$-propositions and $\mathcal{U}$-sets are defined as:

$$\lambda\ \mathsf{Prop}_{\mathcal{U}} :\equiv \sum_{X:\mathcal{U}} \mathsf{is\text{-}prop}(X) \qquad \lambda\ \mathsf{Set}_{\mathcal{U}} :\equiv \sum_{X:\mathcal{U}} \mathsf{is\text{-}set}(X)$$

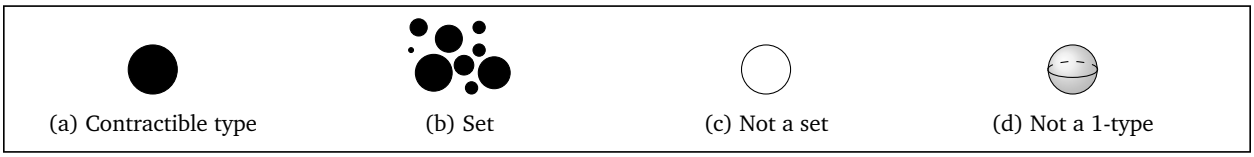
Example: **0** *is a proposition by vacuity and* $\mathbb{N}$ *is a set* λ *(by Thm. 2.15).*

Remark: *For any* $A : \mathcal{U}$, $\mathsf{is\text{-}prop}(A)$ *is logically equivalent* λ *to:*

$$\mathsf{all\text{-}elements\text{-}equal}(A) :\equiv \prod_{x,y:A} x = y.$$

Moreover, λ *a type is a set iff the only path of a point to itself is reflexivity*[5].

[5] This principle is also known under the name of *axiom K*.



(a) Contractible type (b) Set (c) Not a set (d) Not a 1-type

Figure 1: Some examples and counter-examples of n-types as spaces.

Note that these definitions of propositions and sets effectively capture the natural meaning usually given to these objects — the formers are proof irrelevant (i.e., any two proofs of a proposition are the same) and the latters are composed of a bunch of elements not identifiable between themselves. For instance, the space described by Fig. 1b is a set as all identifications happen in a space without holes and thus the space always admits a continuous deformation between two paths. In turn, Fig. 1c is not a set as, for any point, there is a clockwise path p and a counter-clockwise path q such that p cannot be continuously deformed into q due to the hole inside the circle. Likewise, the sphere of Fig. 1d is not a 1-type, as there are no continuous deformations between the continuous deformation of the clockwise and counter-clockwise path between two points.

Another type that is not a set, but more familiar to mathematicians, is the type of all sets of a universe $\mathcal{U}$, $\mathsf{Set}_\mathcal{U}$. A simple way to remark this is through the use of equivalences, that are defined very naturally using n-types as maps $A \to B$ such that all elements of B have a unique preimage in A.

Definition 2.6: Fiber λ

Let $A, B : \mathcal{U}$, $f : A \to B$ and $y : B$. The *fiber* of f at y is the type

$$\mathsf{fib}_f(y) :\equiv \sum_{x:A} f(x) = y.$$

Remark: *The fiber of f at y can be thought of as the type-theoretic version of the preimage of y by f.*

Definition 2.7: Equivalence λ

Let $A, B : \mathcal{U}$ and $f : A \to B$. f is an equivalence whenever all its fibers are contractible; and A is equivalent to B whenever there is a map $f : A \to B$ that is an equivalence:

$$\mathsf{is\text{-}equiv}(f) :\equiv \prod_{y:B} \mathsf{is\text{-}contr}(\mathsf{fib}_f(y)), \qquad A \simeq B :\equiv \sum_{f:A\to B} \mathsf{is\text{-}equiv}(f).$$

Note that $A \simeq A$, as id has contractible fibers. The usual notion of equivalence, inverses, are defined in HoTT using pointwise identification of back-and-forth maps with the identity. Recall that the homotopic interpretion of identity types makes two points identifiable if there is a continuous deformation between those points in the space of the type. Thus, functions are pointwise identifiable if it is possible to "fill the space" between the two functions. This notion is an instance of a *morphism between morphisms*, also called a *homotopy*.

Definition 2.8: Homotopy λ

Let $A : \mathcal{U}$, $B : A \to \mathcal{U}$ and $f, g : \prod_{(x:A)} B(x)$. The type of *homotopies* from f to g is defined as

$$f \sim g :\equiv \prod_{x:A} f(x) = g(x).$$

Of course, being a homotopy is an equivalence relation λ, it is reflexive (with witness $\mathsf{refl\text{-}htpy}$), symmetric and transitive.

Definition 2.9: Inverse λ

> Let $A, B : \mathcal{U}$ and $f : A \to B$. *f has an inverse* whenever there is a function $g : B \to A$ and two homotopies $f \circ g \sim \mathsf{id}_B$ and $g \circ f \sim \mathsf{id}_A$,
>
> $$\mathsf{has\text{-}inverse}(f) :\equiv \sum_{g:B\to A} (f \circ g \sim \mathsf{id}_B \times g \circ f \sim \mathsf{id}_A).$$

This definition has the (practical) advantage to immediatly provide an actual inverse to f, denoted f^{-1}. Unfortunately, it introduces further *structure* on the map [Rij22b, Ex. 22.5], something deeply undesirable as one would like being an equivalence to be a *property* of a map.

Definition 2.10: Subtype λ

> A type family B over A is said to be a *subtype* of A if for each $x : A$, $B(x)$ is a proposition. In this case, $B(x)$ is also said to be a *property* of A.

In fact, the definition of equivalences as a map with contractible fibers has the exact advantage of being a property on the map. Indeed, as a Π-type is a proposition whenever its target type is a (family of) proposition(s) λ, and $\mathsf{is}\text{-}k\text{-}\mathsf{type}(X)$ is a proposition λ forall $k \in \mathbb{Z}_{\geqslant 2}$, it follows directly that is-equiv is a proposition. As such, the univalence axiom only states a property over a map.

Axiom 2.11: Univalence λ

> All universes $\mathcal{U}$ are univalent, that is, for any two types $A, B : \mathcal{U}$, the canonical map
>
> $$\lambda\ \mathsf{path\text{-}to\text{-}equiv} : A =_{\mathcal{U}} B \to A \simeq B$$
>
> given by $\mathsf{path\text{-}to\text{-}equiv}(\mathsf{refl}) :\equiv \mathsf{id}$ is an equivalence.

This axiom implies function extensionality [Uni13, §4.9].

Definition 2.12: Function Extensionality λ

> For any $A : \mathcal{U}$, $B : A \to \mathcal{U}$ $f, g : \prod_{x:A} B(x)$, the family of maps
>
> $$\mathsf{htpy\text{-}eq} : (f = g) \to (f \sim g)$$
>
> defined by $\mathsf{htpy\text{-}eq}(\mathsf{refl}) :\equiv \mathsf{refl\text{-}htpy}_f$ is a family of equivalences.

Moreover, as any equivalence has an inverse [Rij22b, Thm. 10.3.5] λ (given by the center of contraction), the univalence axiom and the function extensionality principle yield the functions

$$\mathsf{equiv\text{-}to\text{-}path} : A \simeq B \to A =_{\mathcal{U}} B \quad \text{and} \quad \mathsf{eq\text{-}htpy} : (f \sim g) \to (f = g).$$

This allows Christian Sattler and David Wärn's proof[6] of the *graduate lemma*, that will be sketched here as it differs greatly from the ones found in [Uni13, §4.2] or [Rij22b, §10.4].

[6] Note that function extensionality is not strictly necessary here, even though it simplifies the proof greatly (computation wise). For instance, [Uni13, Thm. 2.11.1] shows that ap_f has an inverse whenever f has an inverse directly.

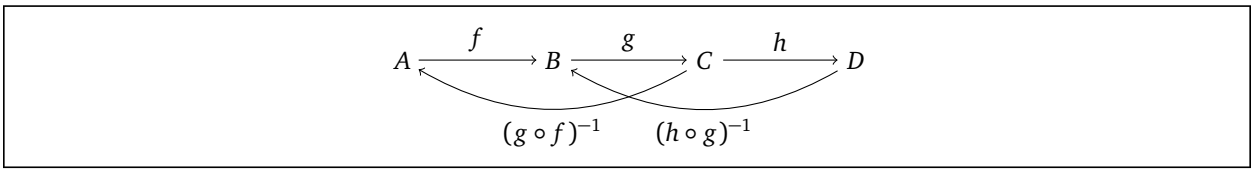


Figure 2: Setting which gives an inverse to g.

Lemma 2.13: Graduate Lemma λ

> Let $A, B : \mathcal{U}$ and $f : A \to B$. Then f is an equivalence iff f has an inverse.

Proof: Only the converse direction is sketched, refer to [Rij22b, Thm. 10.3.5] for the forward direction. Assume that has-inverse(f), that is, $f : A \to B$ comes equipped with

$$g : B \to A, \qquad G : f \circ g \sim \mathsf{id}_B, \qquad H : g \circ f \sim \mathsf{id}_A.$$

By function extensionality, eq-htpy(G) : $f \circ g = \mathsf{id}_B$ and eq-htpy(H) : $g \circ f = \mathsf{id}_A$. As id is the inverse of $\mathsf{ap}_{\mathsf{id}}$ ($\mathsf{ap}_{\mathsf{id}}(p) = p$ by path induction λ), let i : has-inverse($\mathsf{ap}_{\mathsf{id}}$) and $P :\equiv \lambda h.\text{has-inverse}(\mathsf{ap}_h)$. Then, both $\mathsf{ap}_{f\circ g}$ and $\mathsf{ap}_{g\circ f}$ have an inverse

$$\mathsf{tr}_P(\text{eq-htpy}(G)^{-1}, i) : \text{has-inverse}(\mathsf{ap}_{f\circ g}) \quad \text{and} \quad \mathsf{tr}_P(\text{eq-htpy}(H)^{-1}, i) : \text{has-inverse}(\mathsf{ap}_{g\circ f}).$$

Moreover, in a setting of Fig. 2 where $f : A \to B$, $g : B \to C$, $h : C \to D$, if $g \circ f$ and $h \circ g$ have an inverse, then g also has an inverse λ: $f \circ (g \circ f)^{-1}$. Indeed, $g \circ f \circ (g \circ f)^{-1}(x) = x$ and $g \circ f \circ (g \circ f)^{-1} \circ g(x) = g(x)$. Furthermore, g is injective as if $p : g(x) = g(y)$, $\mathsf{ap}_h(p) : h(g(x)) = h(g(y))$ and thus, by path concatenation, $x = y$, so $f \circ (g \circ f)^{-1} \circ g(x) = x$. Returning to the main proof, as $\mathsf{ap}_{f\circ g}$ and $\mathsf{ap}_{g\circ f}$ both have an inverse, the previous setting holds and thus ap_f also has an inverse. As such, forall $a : A$, $\sum_{(x:A)} f(x) = f(a)$ is contractible, with center of contraction $(a, \mathsf{refl}_{f(a)})$. Then, given $(x, p) : \sum_{(x:A)} f(x) = f(a)$, $\mathsf{ap}_f^{-1}(p^{-1}) : a = x$ and so the path between $(a, \mathsf{refl}_{f(a)})$ and (x, p) can be built by path induction over $\mathsf{ap}_f^{-1}(p^{-1})$. Finally, as f has an inverse, $f(a)$ corresponds to a $y : B$ forall $a : A$ and thus $\mathsf{fib}_f(y)$ is contractible; that is, f is an equivalence. □

As such, the relation $\simeq$ is an equivalence relation λ. Indeed, given an equivalence $e : A \simeq B$, there is an inverse equivalence $e^{-1} : B \simeq A$ by Lem. 2.13 as the inverse of an inverse also has an inverse. Moreover, equivalences are transitive as the composition of two functions that have an inverse also has an inverse. Therefore, the notations $e(x)$ and $e^{-1}(x)$ will be used as a shortcut of the underlying maps $\mathsf{fst}(e)(x)$ and $\mathsf{fst}(e^{-1})(x)$.

With those tools introduced, it is now easy to show that $\mathsf{Set}_\mathcal{U}$ is not a set. First, consider $\mathbf{2} :\equiv \mathbf{1} + \mathbf{1}$ the type of booleans, with true :≡ inl(⋆) and false :≡ inr(⋆).

Definition 2.14: Decidable Type λ

> A type A is said *decidable* if $A + \neg A$ is inhabited. Moreover, it has a *decidable equality* whenever $x =_A y$ is decidable for all $x, y : A$.

Theorem 2.15: Hedberg λ

> Any type that has decidable equality is a set.

By Hedberg's theorem, $\mathbf{2}$ is a set. But $\mathsf{neg}_\mathbf{2}$ (that sends true to false and conversely) and $\mathsf{id}_\mathbf{2}$ are both equivalences, as they are their own inverse. As such, by univalence, there are two distinct paths $\mathbf{2} = \mathbf{2}$, and hence $\mathsf{Set}_\mathcal{U}$ is not a set. Thus, equivalences carry further informations than the types themselves.

This is not always desirable — sometimes, the bijection must not be explicit. Examples of this situation arise from finite sets or surjective maps. Stating "a set is finite" and "there is a bijection between this set and a standard finite set" is fundamentally different, as in the latter case, the object manipulated is a totally ordered set. Likewise,

stating that "a map is surjective" and "for every element in the image of a map, its preimage can be computed" makes a vast difference, as the latter defines a left inverse of the map[7]. In constructive mathematics, one thus needs a way to express that a type P is inhabited, without exhibiting any of its inhabitant(s). Doing so yields a *subtype* only containing the objects that satisfy the property, i.e., that satisfy the propositional version of P. In homotopy type theory, this concept is called the *propositional truncation* of P, and often referred to as the "mere" version of P.

Propositional truncation is defined as an *higher inductive type*, the last component of HoTT. Higher inductive types have the peculiarity to introduce constructors to generate *identifications* between its elements. For instance, the propositional truncation of a type A is defined as an higher inductive type with two constructors

$$|\cdot| : A \to \|A\|$$
$$\alpha : \prod_{x,y:\|A\|} x = y.$$

The first states that any type has a propositional truncation, and the second that the propositional truncation of a type is a proposition. Of course, for any family of propositions P over $\|A\|$, it has an induction principle and a recursor (when P is constant)

$$\mathsf{ind}_{\|A\|} : \left(\prod_{x:A} P(|x|)\right) \to \prod_{x:\|A\|} P(x), \qquad \mathsf{rec}_{\|A\|} : (A \to P) \to \|A\| \to P$$

that enable proper reasoning when working with truncated objects. This notion can then be used to define the notions of *finite type* and *surjective map* as properties of a type rather than an additional structure.

Definition 2.16: Standard Finite Types

The family of standard finite types are defined inductively over $\mathbb{N}$ by

$$\mathsf{Fin}_0 :\equiv \mathbf{0}$$
$$\mathsf{Fin}_{k+1} :\equiv \mathsf{Fin}_k + \mathbf{1}.$$

Definition 2.17: Finite Type

A type X is said *finite* whenever it comes equiped with an unspecified equivalence $\mathsf{Fin}_k \simeq X$ for some $k : \mathbb{N}$

$$\mathsf{is\text{-}finite}(X) :\equiv \left\| \sum_{k:\mathbb{N}} \mathsf{Fin}_k \simeq X \right\|.$$

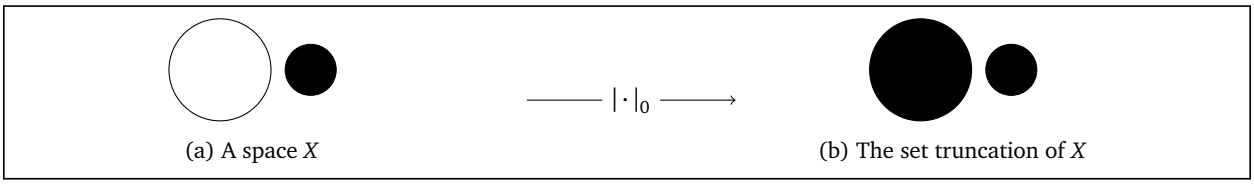
Definition 2.18: Surjective Map

Let $A, B : \mathcal{U}$ and $f : A \to B$. f is said *surjective* whenever there is an *unspecified* x such that $f(x) = b$, for any $b \in B$,

$$\mathsf{is\text{-}surj}(f) :\equiv \prod_{b:B} \left\| \mathsf{fib}_f(b) \right\|.$$

Note that, by definition, both $\mathsf{is\text{-}finite}(X)$ and $\mathsf{is\text{-}surj}(f)$ are properties. However, for the former, one might still want to get the *cardinal* of a finite set.

[7] Note that this is the exact argument used in the $\mathsf{is\text{-}equiv}(f) \to \mathsf{has\text{-}inverse}(f)$ proof.



(a) A space X (b) The set truncation of X

Figure 3: Example of set truncation on a space.

Theorem 2.19: Cardinality λ

For any $X : \mathcal{U}$, the type

$$\mathsf{has\text{-}cardinality}(X) :\equiv \sum_{k:\mathbb{N}} \| \mathsf{Fin}_k \simeq X \|$$

is a proposition and there is a logical equivalence

$$\mathsf{has\text{-}cardinality}(X) \leftrightarrow \mathsf{is\text{-}finite}(X).$$

Proof: Let $(k, e), (k', e') : \mathsf{has\text{-}cardinality}(X)$. As $\| \mathsf{Fin}_k \simeq X \|$ is a proposition, $(k, e) = (k', e')$ whenever $k = k'$ λ. $\mathbb{N}$ is a set, hence its equality type is a proposition and thus $\mathsf{rec}_{\| \mathsf{Fin}_k \simeq X \|}$ can be applied, as well as $\mathsf{rec}_{\| \mathsf{Fin}_{k'} \simeq X \|}$, yielding $\mathsf{Fin}_k \simeq X$ and $\mathsf{Fin}_{k'} \simeq X$. As such, $\mathsf{Fin}_k \simeq \mathsf{Fin}_{k'}$ and thus λ $k = k'$. The back-and-forth maps are given by

$$\mathsf{rec}_{\mathsf{is\text{-}finite}(X)}(\lambda(k, e).(k, |\, e \,|)) : \mathsf{is\text{-}finite}(X) \to \mathsf{has\text{-}cardinality}(X)$$
$$\lambda(k, e).\mathsf{rec}_{\| \mathsf{Fin}_k \simeq X \|}(\lambda e'. \left| (k, e') \right|, e) : \mathsf{has\text{-}cardinality}(X) \to \mathsf{is\text{-}finite}(X).$$

The cardinal of a finite type X is hence denoted $|X|$. □

Types can be truncated above the propositional level. An interesting example of that is the set truncation. Of course, it can be defined as an higher inductive type λ, as the propositional truncation:

$$|\cdot|_0 : A \to \|A\|_0$$
$$\alpha_0 : \prod_{x,y:\|A\|_0} \prod_{p,q:x=y} p = q$$

and is, *de facto*, equipped with an induction principle for any family of sets P over $\|A\|_0$

$$\lambda\ \mathsf{ind}_{\|A\|_0} : \left(\prod_{x:A} P(|\, x \,|_0) \right) \to \prod_{x:\|A\|_0} P(x),$$

such that $\mathsf{ind}_{\|A\|_0}(f, |\, x \,|_0) \equiv f(x)$. But there is another, more insightful, way to define set truncation — as a set quotiented by *mere* equality λ. Indeed, for any X, there is a map $f : \prod_{(x,y:X)} \prod_{(p:\| x=y \|)} |\, x \,|_0 = |\, y \,|_0$ defined as

$$f(x, y) :\equiv \mathsf{rec}_{\| x=y \|}(\lambda p.\, \mathsf{ap}_{|\cdot|_0}(p)).$$

Hence, set truncating simply consists in choosing a representant given by the mere equality, as shown in Fig. 3. An immediate consequence of this is that there is an equivalence $|\, x \,|_0 = |\, y \,|_0 \simeq \| x = y \|$ λ. As such, the set truncation of a type is also called the *connected components* of this type. Thus, a *connected* type is such that its set truncation has a unique element; $\mathsf{is\text{-}conn}(A) :\equiv \mathsf{is\text{-}contr}(\|A\|_0)$. An important property of set truncations is that, for any map f from A to a set X, there is a *unique* extension $g : \|A\|_0 \to X$ of f such that $g \circ |\cdot|_0 \sim f$ λ. Often, this property is represented by saying that the following diagram *commutes*:

$$\begin{array}{ccc} A & & \\ {\scriptstyle |\cdot|_0}\downarrow & \searrow^{f} & \\ \|A\|_0 & \dashrightarrow & X. \end{array}$$

For instance, the diagram introduced in Fig. 2 also commutes.

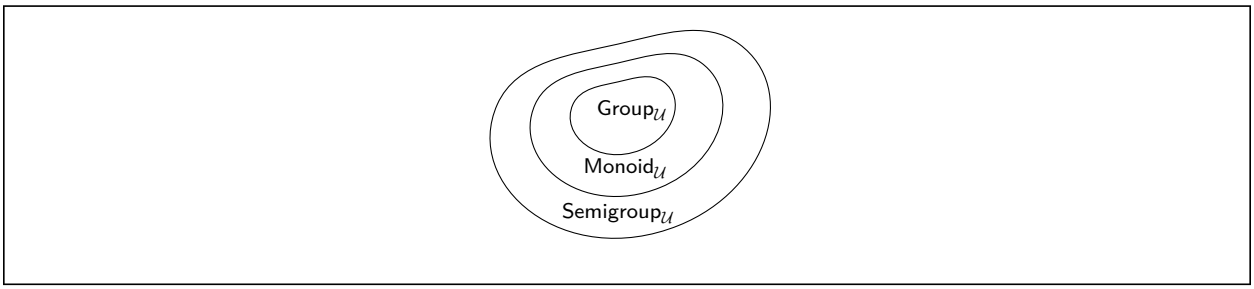


Figure 4: Expected relationship between group structures.

3 Structures in Univalent Mathematics — an Application with Groups

One of the main goal of HoTT is to qualify the identity types of groupoids. Indeed, only a type that has its identity types fully characterized can be wholly understood. In this section, a common structure of mathematics, groups, are defined in §3.1. Then, the univalence axiom is used to demonstrate how any two isomorphic groups can be identified. This allows to formalize long-used arguments such as "two isomorphic structures satisfy the same properties" using a simple transport, and to mechanize proofs that hold "up to isomorphism". §3.2 is an example of such a proof, where finiteness of structures up to isomorphism is shown. Finally, §3.3 is an application of this proof to the type of groups of finite order.

3.1 Semigroups, Monoids and Groups

Recall that a *group* is a set, equipped with (i) an associative operation, (ii) a neutral element and (iii) such that each element has an inverse. A set satisfying only condition (i) is called a *semigroup*, and one verifying conditions (i) and (ii) is a *monoid*. Naturally, it is expected that the types behave as Fig. 4; that is, groups being subtypes of monoids, themselves subtypes of semigroups. As such, by Def. 2.10, these structures should be defined carefully using propositions.

Definition 3.1: Semigroup λ

A set G in a universe $\mathcal{U}$ is a *semigroup* if it is equipped with an associative multiplication

$$\mathsf{is\text{-}semigroup}(G) :\equiv \sum_{\mu : G \to G \to G} \prod_{x,y,z:G} \mu(\mu(x,y),z) = \mu(x,\mu(y,z)).$$

The type of all semigroups of $\mathcal{U}$ is $\mathsf{Semigroup}_{\mathcal{U}} :\equiv \sum_{(G:\mathsf{Set}_{\mathcal{U}})} \mathsf{is\text{-}semigroup}(G)$.

Note that $\mathsf{is\text{-}semigroup}(G)$ is *not* a proposition, hence semigroups are *not* a subtype of sets.

Definition 3.2: Unital Semigroup λ

A semigroup G is *unital* whenever there exists a unit $e : G$ which satisfies

$$\mathsf{right\text{-}unit\text{-}law}(G,e) :\equiv \prod_{x:G} \mu(x,e) = x \quad \text{and} \quad \mathsf{left\text{-}unit\text{-}law}(G,e) :\equiv \prod_{y:G} \mu(e,y) = y.$$

Hence, $\mathsf{is\text{-}unital}(G) :\equiv \sum_{(e:G)} (\mathsf{right\text{-}unit\text{-}law}(G,e) \times \mathsf{left\text{-}unit\text{-}law}(G,e))$.

Being unital is a property λ. Indeed, as the right and left unit laws state equalities over elements of a set, they are both propositions. Thus, it suffices to show that the neutral element is unique. Let $e, e' : G$. Then:

$$e = \mu(e,e') = e'.$$

The type of all monoids of $\mathcal{U}$ is $\mathsf{Monoid}_{\mathcal{U}} :\equiv \sum_{(G:\mathsf{Semigroup}_{\mathcal{U}})} \mathsf{is\text{-}unital}(G)$.

Definition 3.3: Group λ

A unital semigroup has *inverses* if it comes equipped with a function $(-)^{-1} : G \to G$ such that

$$\mathsf{right\text{-}inv}(G,e,(-)^{-1}) :\equiv \prod_{x:G} \mu(x,x^{-1}) = e \quad \text{and} \quad \mathsf{left\text{-}inv}(G,e,(-)^{-1}) :\equiv \prod_{y:G} \mu(y^{-1},y) = e.$$

A semigroup G is a group whenever it is unital and has inverses;

$$\mathsf{is\text{-}group}(G) :\equiv \sum_{e:\mathsf{is\text{-}unital}(G)} \sum_{(-)^{-1}:G\to G} (\mathsf{right\text{-}inv}(G,e,(-)^{-1}) \times \mathsf{left\text{-}inv}(G,e,(-)^{-1})).$$

The type of all groups of $\mathcal{U}$ is $\mathsf{Group}_{\mathcal{U}} :\equiv \sum_{(G:\mathsf{Semigroup}_{\mathcal{U}})} \mathsf{is\text{-}group}(G)$.

Being a group is, once again, a property λ. Indeed, as being unital is a property, it suffices to show that the type of triples $((-)^{-1},\alpha,\beta)$ is a proposition. Of course, being a right inverse and a left inverse are propositions as they state equalities over elements of a set. Hence, showing that $(-)^{-1}$ is unique is enough. If $(-)^{-1'}$ is an inverse for G, by function extensionality, it suffices to show that $x^{-1} = x^{-1'}$ for any $x \in G$:

$$x^{-1} = \mu(x^{-1},e) = \mu(x^{-1},\mu(x,x^{-1'})) = \mu(\mu(x^{-1},x),x^{-1'}) = \mu(e,x^{-1'}) = x^{-1'}.$$

Definition 3.4: (Semi)group Homomorphism λ

Let G and H be (semi)groups, with respective multiplications μ_G and μ_H. A map $f : G \to H$ *preserves multiplication* if the type

$$\mathsf{preserves\text{-}mul}(f) :\equiv \prod_{x,y:G} f(\mu_G(x,y)) = \mu_H(f(x),f(y))$$

is inhabited. Homomorphisms are maps f equipped with $\mathsf{preserves\text{-}mul}(f)$. The type of all homomorphisms from G to H is written $\mathsf{hom}(G,H) :\equiv \sum_{(f:G\to H)} \mathsf{preserves\text{-}mul}(f)$.

Note that preserves-mul is a property over a map. Hence, for $\mathrm{h}_f, \mathrm{h}_g : \mathsf{hom}(G,H)$ with respective underlying maps f and g,

$$\lambda\ (\mathrm{h}_f = \mathrm{h}_g) \simeq (f = g) \simeq (f \sim g).$$

Definition 3.5: (Semi)group Isomorphism λ

Let $h : \mathsf{hom}(G,H)$. h is an *isomorphism* if it comes equipped with a triple (h^{-1},p,q) where $h^{-1} : \mathsf{hom}(H,G)$,

$$p : h^{-1} \circ h = \mathsf{id}_G \quad \text{and} \quad q : h \circ h^{-1} = \mathsf{id}_H.$$

Such triples have type $\mathsf{is\text{-}iso}(h)$, and the type of all isomorphisms between G and H is

$$G \cong H :\equiv \sum_{h:\mathsf{hom}(G,H)} \mathsf{is\text{-}iso}(h).$$

is-iso is a property of a map. Indeed, as $h^{-1} \circ h = \mathsf{id}_G \simeq h^{-1}(h(x)) = x$ and $h \circ h^{-1} = \mathsf{id}_H \simeq h(h^{-1}(y)) = y$, these equalities are propositions. Hence, it suffices to show that the inverse is unique. Let $h^{-1}, h^{-1'} : \mathsf{hom}(H,G)$ two inverses of h. It is enough to show that $h^{-1} \sim h^{-1'}$:

$$h^{-1}(y) = h^{-1'}(h(h^{-1}(y))) = h^{-1'}(y).$$

As Σ-types are closed under sets, and any proposition is a set, $G \cong H$ is also a set.

Lemma 3.6:

A (semi)group homomorphism $h : G \to H$ is an isomorphism iff its underlying map is an equivalence. Consequently, there is an equivalence

$$(G \cong H) \simeq \sum_{e:G\simeq H} \mathsf{preserves\text{-}mul}(e)$$

Proof: The forward map is trivial by Lem. 2.13 as h has an inverse. For the converse, let $h^{-1} : H \to G$ be the inverse of h. It is also a group homomorphism:

$$h^{-1}(\mu_H(x,y)) = h^{-1}(\mu_H(h(h^{-1}(x)), h(h^{-1}(y)))) = h^{-1}(h(\mu_G(h^{-1}(x), h^{-1}(y)))) = \mu_G(h^{-1}(x), h^{-1}(y)).$$

As is-iso and is-equiv are propositions, $\mathsf{is\text{-}iso}(f) \simeq \mathsf{is\text{-}equiv}(f)$ and the equivalence is directly implied by $\sum_{(t:\sum_{(x:A)} B(x))} C(\mathsf{fst}(t)) \simeq \sum_{(t:\sum_{(x:A)} C(x))} B(\mathsf{fst}(t))$ whenever $B(x) \simeq C(x)$ forall $x : A$. □

Theorem 3.7: Isomorphic Groups are Equal

Let G, H be (semi)groups of a universe $\mathsf{Set}_\mathcal{U}$. Then

$$(G \cong H) \simeq (G = H).$$

Proof: Remark that if $p : G = H$, then $\mathsf{preserves\text{-}mul}(\mathsf{path\text{-}to\text{-}equiv}(p)) \simeq \mathbf{1}$ as, by path induction, $\mathsf{path\text{-}to\text{-}equiv}(\mathsf{refl}) \equiv \mathsf{id}$ and preserves-mul is a proposition. Hence, by univalence, there is a chain of equivalences

$$(G \cong H) \simeq \sum_{e:G\simeq H} \mathsf{preserves\text{-}mul}(e) \simeq \sum_{p:G=H} \mathsf{preserves\text{-}mul}(\mathsf{path\text{-}to\text{-}equiv}(p)) \simeq (G = H).$$

□

3.2 A Counting of Structures up to Isomorphism

A structure finite *up to isomorphism* is not expected to be finite in general. For instance, take the type of types with two elements. They are all equivalent to Fin_2. Thus, the number of two-element types up to isomorphism is one. But the number of such types is not finite. However, by univalence, types with two elements can be identified, and hence form a unique *connected component* in the space of the universe. As presented in §2.4, the set of connected components of a type can be represented by the set truncation of this type. The notion of *finiteness up to isomorphism* takes full advantage of this contruction.

Definition 3.8: Homotopy Finiteness

$A : \mathcal{U}$ is said π_n-finite if it has finitely many connected components, and all its i^{th} paths have also finitely many connected components, for $i < n$:

$$\mathsf{is\text{-}}\pi_0\mathsf{\text{-}finite}(A) :\equiv \mathsf{is\text{-}finite}\|A\|_0$$
$$\mathsf{is\text{-}}\pi_{n+1}\mathsf{\text{-}finite}(A) :\equiv \mathsf{is\text{-}finite}\|A\|_0 \times \prod_{x,y:A} \mathsf{is\text{-}}\pi_n\mathsf{\text{-}finite}(x = y).$$

The goal of this section is to show that homotopy finiteness is closed by Σ-type former. Indeed, in univalent mathematics, *structures* are built using Σ-types. Moreover, as seen in §3.1, isomorphic structures can be identified. As such, the notion of homotopy finiteness is adapted for counting up to isomorphism. The proof is done by induction on n, and starts by a difficult base case.

Lemma 3.9: Finite Codomain λ

> Let $A, B : \mathcal{U}$ such that A is finite and $f : A \to B$ is surjective. Then, B is finite whenever it has decidable equality.

Proof: As is-finite is a property, assume without loss of generality that $(k, e) : \sum_{(n:\mathbb{N})} \mathsf{Fin}_n \simeq A$. Note that, consequently, there is a map $g :\equiv f \circ e : \mathsf{Fin}_k \to B$ that is surjective (as the composition of surjective maps is surjective λ). The proof is by induction on k. If $k = 0$, then $\mathbf{0} \simeq B$ as g is surjective and injective by definition of $\mathbf{0}$. If $k > 0$, then if $\mathsf{inr}(\star)$ is the only preimage of $g(\mathsf{inr}(\star))$:

$$B \simeq \left(\sum_{y:B} y \neq g(\mathsf{inr}(\star)) \right) + \mathbf{1}$$

which is finite as $\mathbf{1}$ is finite, $\sum_{(y:B)} y \neq g(\mathsf{inr}(\star))$ is finite by induction hypothesis (the map is g augmented with a simple information, surjectiveness follows from the condition, and decidable equality follows from B's own decidable equality) and finiteness is closed by coproduct λ. If $\mathsf{inr}(\star)$ is not the only preimage of $g(\mathsf{inr}(\star))$, then g is still surjective when restraining the domain to Fin_k and thus the result follows by induction hypothesis. □

Lemma 3.10: Closure Under Σ-types, Base Case λ

> Let B be a family of π_0-finite types over a connected, π_1-finite type A. Then the type $\sum_{(x:A)} B(x)$ is π_0-finite.

Proof: Note that homotopy finiteness is a proposition, being either a proposition or a product of propositions. As A is connected, it is *merely* inhabited by $|a| : \|A\|$ λ. By the recursion principle of propositional truncation, assume that $a : A$. Then, the function $f :\equiv \lambda b.(a, b) : B(a) \to \sum_{(x:A)} B(x)$ is surjective λ as for $(x, y) : \sum_{(x:A)} B(x)$, $p : |a|_0 = |x|_0$ by connectedness, hence $q : \|a = x\|$ and, as is-surj is a proposition, $r : a = x$. By path induction, $f(y) = (x, y)$ for any $(x, y) : \sum_{(x:A)} B(x)$. Consider $\|f\|_0$, defined so that the following diagram commutes:

$$\begin{array}{ccc} B(a) & \xrightarrow{\quad f \quad} & \sum_{(x:A)} B(x) \\ {\scriptstyle |\cdot|_0}\downarrow & & \downarrow{\scriptstyle |\cdot|_0} \\ \|B(a)\|_0 & \xrightarrow[\|f\|_0]{} & \left\| \sum_{(x:A)} B(x) \right\|_0 \end{array}$$

$\|f\|_0$ is also surjective λ, by surjectiveness of f and the induction principle of set truncation. As $\|B(a)\|_0$ is finite by assumption, and $\|f\|_0$ is surjective, $\left\| \sum_{(x:A)} B(x) \right\|_0$ is finite if it has decidable equality by Lem. 3.9. The following chain of equivalences hold:

$$\left(|(x,y)|_0 = \left|(x',y')\right|_0\right) \simeq \left\|(x,y) = (x',y')\right\| \simeq \left\|(a,y) = (a,y')\right\| \simeq \left\| \sum_{(p:a=a)} \mathsf{tr}_B(p,y) = y' \right\| \text{ λ.}$$

Hence, it suffices to show that $\left\| \sum_{(p:a=a)} \mathsf{tr}_B(p,y) = y' \right\|$, or equivalently λ, $\left\| \sum_{(p:a=a)} \|\mathsf{tr}_B(p,y) = y'\| \right\|$ is decidable. As it is a proposition, and decidability is not closed under Σ-types in general, we show that the underlying type is finite, which suffices λ to show that it is decidable. As $\|B(a)\|_0$ is finite by assumption, it has decidable equality λ, hence $\|\mathsf{tr}_B(p,y) = y'\| \simeq |\mathsf{tr}_B(p,y)|_0 = |y'|_0$ is a decidable proposition, so it is finite. But $a = a$ has no reason of being finite. However, as the universe of propositions is a set λ by univalence, let P be the type family over $\|a = a\|_0$ defined by the induction principle of set truncation as follows:

$$\begin{array}{ccc} a = a & & \\ {\scriptstyle |\cdot|_0}\downarrow & \searrow^{p \mapsto \| \mathsf{tr}_B(p,y) = y' \|} & \\ \| a = a \|_0 & \xrightarrow[P]{} & \mathsf{Prop}_{\mathcal{U}}. \end{array}$$

There is a trivial back-and-forth map between $\left\| \sum_{(p:a=a)} \| \mathsf{tr}_B(p,y) = y \| \right\|$ and $\left\| \sum_{(\omega:\| a=a \|_0)} P(\omega) \right\|$ by the recursion principle of propositional truncation and the induction principle of set truncation as if $\omega \equiv |p|_0$ for $p : a = a$ then $P(\omega) \equiv \| \mathsf{tr}_B(p,y) = y' \|$ by the computation rule of the induction principle. Hence, as those two types are propositions, they are equivalent. By hypothesis, A is π_1-finite and consequently, $\| a = a \|_0$ is finite. Thus $\left\| \sum_{(\omega:\| a=a \|_0)} P(\omega) \right\|$ is also finite as finiteness is closed under Σ-types λ. □

Theorem 3.11: Closure under Σ-types λ

> Let B be a family of π_n-finite types over a π_{n+1}-finite type A. Then the type $\sum_{(x:A)} B(x)$ is π_n-finite.

Proof: By induction over n. For $n = 0$, by induction on the number of connected components of A. If there are no connected components, then A is empty λ and hence, as $\mathbf{0}$ is π_k-finite forall $k \in \mathbb{N}$ λ and $\sum_{(x:\mathbf{0})} B(x) \simeq \mathbf{0}$, the result follows. Otherwise, as $\|A\|_0$ is finite by assumption and a property is being proven, let $e : \mathsf{Fin}_{k+1} \simeq \|A\|_0$. Then, it can be shown λ that there is a map $f : \mathsf{Fin}_{k+1} \to A$ such that $|\cdot|_0 \circ f \sim e$. The *image* λ of $h : A \to B$ is defined to be $\mathsf{im}_h :\equiv \sum_{(y:B)} \| \mathsf{fib}_h(y) \|$. By taking advantage that $(|x|_0 = |y|_0) \simeq \| x = y \|$, it can be shown that λ

$$\sum_{x:A} B(x) \simeq \left(\sum_{x:\mathsf{im}_{f \circ \mathsf{inl}}} B \circ \mathsf{fst} \right) + \left(\sum_{x:\mathsf{im}_{f \circ \mathsf{inr}}} B \circ \mathsf{fst} \right).$$

Moreover, as $\mathsf{Fin}_k \simeq \left\| \mathsf{im}_{f \circ \mathsf{inl}} \right\|_0$ λ, $\sum_{(x:\mathsf{im}_{f \circ \mathsf{inl}})} B \circ \mathsf{fst}$ has a finite number of connected components by (inner) induction hypothesis. Furthermore, as $f \circ \mathsf{inr} : \mathbf{1} \to A$, $\mathsf{im}_{f \circ \mathsf{inr}}$ is connected λ and thus Lem. 3.10 applies. As homotopy finiteness is closed under coproduct, the result follows.

If $n > 0$, it suffices to remark that for $t, u : \sum_{(x:A)} B(x)$, $t = u \simeq \sum_{(p:\mathsf{fst}(t)=\mathsf{fst}(u))} \mathsf{tr}_B(p, \mathsf{snd}(t)) = \mathsf{snd}(u)$, and the result is immediate by (outer) induction hypothesis. □

3.3 The Finiteness of Groups of Finite Order

The goal of this section is to define groups of finite order such that Thm. 3.11 can be applied.

Definition 3.12: (Semi)groups of Finite Order λ

> A (semi)group G is of finite order n if there is a *mere* equivalence between Fin_n and G,
>
> $$\mathsf{Semigroup\text{-}of\text{-}Order}(n) :\equiv \sum_{G:\mathsf{Semigroup}_{\mathcal{U}}} \| \mathsf{Fin}_n \simeq G \| \quad \text{and} \quad \mathsf{Group\text{-}of\text{-}Order}(n) :\equiv \sum_{G:\mathsf{Group}_{\mathcal{U}}} \| \mathsf{Fin}_n \simeq G \|.$$

Recall that being finite implies having decidable equality. Then, by Thm. 2.15, every finite type is a set. Hence, in both $\mathsf{Semigroup\text{-}of\text{-}Order}(n)$ and $\mathsf{Group\text{-}of\text{-}Order}(n)$, there is a redundant information in the fact that G is a set. Hence,

$$\mathsf{Semigroup\text{-}of\text{-}Order}(n) \simeq \sum_{\sum_{G:\mathcal{U}} \| \mathsf{Fin}_n \simeq G \|} \sum_{\mu : G \to G \to G} \prod_{x,y,z:G} \mu(\mu(x,y),z) = \mu(x,\mu(y,z)) \equiv: \mathsf{Semigroup\text{-}of\text{-}Order}'(n)$$

Whenever G is finite, the right-hand side of $\mathsf{Semigroup\text{-}of\text{-}Order}'(n)$ is finite as the equality is a decidable proposition. In particular, a finite type is π_n-finite for $n \in \mathbb{N}$ λ, hence it is π_0-finite. Then:

Lemma 3.13:

> $\sum_{G:\mathcal{U}} \| \mathsf{Fin}_n \simeq G \|$ is connected. Consequently, it is π_n-finite for $n \in \mathbb{N}$.

Proof: Take as center of contraction $\left|(\mathsf{Fin}_n, |\,\mathsf{id}\,|)\right|_0$. The contraction is trivial as, by recursion principle of propositional truncation, $\mathsf{Fin}_n \simeq G$ and, by univalence, $(\mathsf{Fin}_n \simeq G) \simeq (\mathsf{Fin}_n = G)$. The homotopy finiteness is shown by induction on n. For $n = 0$, a contractible type is finite. For $n > 0$, the equality between two elements of this type is equivalent to the equality between the underlying types as this is a Σ-type over a proposition. But $(G = H) \simeq (G \simeq H)$ by univalence, and the number of equivalences between finite types is finite λ. □

Theorem 3.14: A Finite Number of Finite Semigroups

> Semigroup-of-Order(n) is π_k-finite, for $n, k \in \mathbb{N}$.

Proof: As $\sum_{G:\mathcal{U}} \| \mathsf{Fin}_n \simeq G \|$ is π_{k+1}-finite by Lem. 3.13, and $\sum_{(\mu:G\to G\to G)} \prod_{(x,y,z:G)} \mu(\mu(x,y),z) = \mu(x,\mu(y,z))$ is finite and hence π_k-finite, Semigroup-of-Order′(n) is π_k-finite by application of Thm. 3.11. Consequently, Semigroup-of-Order(n) is also π_k-finite. □

Theorem 3.15: A Finite Number of Finite Groups λ

> Group-of-Order(n) has a finite number of connected components.

Proof: First, note that is-group(G) is finite whenever G is finite. Moreover:

$$\mathsf{Group\text{-}of\text{-}Order}(n) \simeq \sum_{G:\mathsf{Semigroup\text{-}of\text{-}Order}(n)} \mathsf{is\text{-}group}(G)$$

and Semigroup-of-Order(n) is π_1-finite by Thm. 3.14. Thm. 3.11 applies, and Group-of-Order(n) is π_0-finite. □

4 Computational Analysis of the Proof

As the proof of Thm. 3.15 is constructive, there is a λ-term

$$\mathsf{has\text{-}finite\text{-}connected\text{-}components}(n) : \mathsf{is\text{-}}\pi_0\text{-}\mathsf{finite}(\mathsf{Group\text{-}of\text{-}Order}(n)).$$

Consequently, by definition of homotopy finiteness, the cardinal of $\| \mathsf{Group\text{-}of\text{-}Order}(n) \|_0$ is computable, i.e., it is possible to count the number of groups of order n up to isomorphism.

However, as far as the author knows, there are no implementations of this proof that compute well. For instance, in Cubical Agda's version[8] as well as in ours λ, computing (semi)groups of order 0 and 1 is instantaneous, but from order 2 onwards, the computation either takes tens of minutes or gets shut down due to a lack of RAM.

This section's goal is to analyze the bottlenecks of the computation, taking full advantage of the head-linear reduction of the tool used by the formalization. The complexity of the algorithm executed by Thm. 3.15 is computed in §4.1 and §4.2 points out which steps make the computation difficult.

4.1 Complexity of the Underlying Algorithm of Thm. 3.15

In a proof term, it is difficult to know whether a subterm needs to be unfolded during the computation or not. For instance, terms t that have a propositional type do not always need to be evaluated, as any inhabited proposition is equivalent to $\mathbf{1}$ λ; that is, t should be able to be replaced by $\star$ without affecting the computation[9].

[8] Available here: https://agda.github.io/cubical/Cubical.Experiments.CountingFiniteStructure.html

[9] This is not always true. For instance, finiteness is a proposition, but an information is still extracted from its proof term.

By analyzing the step-by-step unfolding[10] of Lem. 3.14 for $n = 2$, a broad outline of the computation followed can be summarized. For every element of $\left\| \sum_{(G:\mathcal{U})} \| \mathsf{Fin}_n \simeq G \| \right\|_0$, Lem. 3.10 is called. In turn, the number of elements of Semigroup-of-Order$'(n)$ is counted by looping over the number of elements of associative functions. At each iteration, the decidable equality of $\left\| \mathsf{Semigroup\text{-}of\text{-}Order}'(n) \right\|_0$ is unfolded for every associative function to decide whether $\mathsf{inr}(\star)$ is the sole preimage of $\| f \circ e(\mathsf{inr}(\star)) \|_0$[11], which actually depends on the number of isomorphisms between finite types of order n.

As such, the total complexity of the algorithm is in the range of $\mathcal{O}((n^{\mathcal{O}(n^2)})n!)$. The factor $n^{\mathcal{O}(n^2)}$ comes from the number of associative functions $G \to G \to G$, which is looped on in the loop, hence gaining $\mathcal{O}(n^2)$ as exponent. The other one, $n!$, is the number of isomorphisms between finite semigroups of order n. As the proof term of Thm. 3.15 needs the computation of the number of semigroups up to isomorphism to start running, its complexity is even worse.

4.2 Discussion over the Bottlenecks of the Computation

The asymptotic complexity of the β-reduction of has-finite-connected-components(n) is clearly high. It however does not fully explain the inability to compute the result for small values of n. In fact, an additional factor should be taken into account to understand the slow running time of the algorithm: the growth of the proof term.

A usual metric for a λ-term is the size of its *abstract syntax tree*. Formally, the function $|\cdot| : \Lambda \to \mathbb{N}$ is defined as

$$|x| = 1 \qquad |(e)\ e'| = |e| + |e'| \qquad |\lambda x.e| = 1 + |e|.$$

Note that the size of types are not defined, as their unfolding is not expected during the computation.

In practice, the runtime of a step of β-reduction depends on the size of the term. Moreover, it is well-known that the reduction step

$$(\lambda x.t)\ u \to t[u/x]$$

can lead to an exponential growth. In fact, the λ-terms involved in the proof of Thm. 3.15 are quite complex (and thus, not especially short), and a rapid growth can be constated when unfolding the proof for $n = 2$, that reaches a size of about a hundred thousand by the two-hundredth step.

As such, the theoretical complexity of $\mathcal{O}(n^{\mathcal{O}(n^2)}n!)$ hides a non-small constant behind the Landau notation, which may explain the lack of efficiency when computing has-finite-connected-components for small values of n.

Others bottlenecks come from a more theoretical perspective, as analyzed in §4.1. A naive algorithm would be expected to have a factorial runtime. However, the key factor here is the number of associative functions $G \to G \to G$, which all comes down from needing decidability over a type indexed by the type of associative functions. Hence, any proof that makes use of this argument will suffer from the same computability problems as Thm. 3.15.

5 Conclusion

In summary, we have implemented E. Rijke's proof of the finiteness of structures up to isomorphism in the language of postt, and applied this proof to standard structures of mathematics: semigroups and groups. The head-linear β-reduction option of this tool has allowed us to analyze the computations of the proof term for (semi)groups of different orders, and to single out the most likely culprit behind the lack of computational power of the proof, even considering the *naiveté* of the underlying algorithm.

Moreover, as the first large-scale project of this tool, most standard definitions and results of homotopy type theory found in [Uni13, Rij22b] had to be formalized, hence offering a sprout of standard library for postt. Consequently, the total number of lines of λ-terms written for this project stands slightly above 9000. It has also allowed the developers of the tool to witness the difficulties of analyzing a proof's computation through a step-by-step basis, where a lot of noise slips in, especially when a proof is composed of a number of intermediate

[10] Using the commands: `postt repl`, `:load -s src/Playground.ctt` and `:unfold number-of-Semigroup-of-Order-two`.

[11] Recall that $f : B(a) \to \sum_{(x:A)} B(x)$ with $f(y) \equiv (a, y)$ and $e : \mathsf{Fin}_k \simeq A$.

lemmas. This has led postt to grow throughout the duration of this internship into a better-suited tool for proof analysis.

This work is but a first step in the field of proof analysis, and the future work is challenging. One could imagine at least two directions — either by capitalizing on the analysis of this document to find a better proof of the finiteness of structures up to isomorphism, either by the extraction of a more general principle by the assessment of others difficult proofs (such as the Brunerie number [Bru16], ...) — both very exciting ventures.

References

[BCH13] Marc Bezem, Thierry Coquand, and Simon Huber. A Model of Type Theory in Cubical Sets. In *TYPES*, volume 26 of *LIPIcs*, pages 107–128. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013.

[Bru16] Guillaume Brunerie. On the homotopy groups of spheres in homotopy type theory, 2016.

[Coq92] Thierry Coquand. The Paradox of Trees in Type Theory. *BIT*, 32(1):10–14, 1992.

[KL18] Chris Kapulkin and Peter LeFanu Lumsdaine. The Simplicial Model of Univalent Foundations (after Voevodsky), 2018.

[Kov23] András Kovács. Efficient Evaluation for Cubical Type Theories. HoTT 2023, 2023.

[Mim20] Samuel Mimram. *PROGRAM = PROOF*. Independently published, 2020.

[ML82] Per Martin-Löf. Constructive Mathematics and Computer Programming. In L. Jonathan Cohen, Jerzy o, Helmut Pfeiffer, and Klaus-Peter Podewski, editors, *Logic, Methodology and Philosophy of Science VI*, volume 104 of *Studies in Logic and the Foundations of Mathematics*, pages 153–175. Elsevier, 1982.

[ML98] Per Martin-Löf. An intuitionistic theory of types. In *Twenty Five Years of Constructive Type Theory*. Oxford University Press, 10 1998.

[Rij22a] Egbert Rijke. Daily applications of the univalence axiom. Conference of Logic and Higher Structures in the Centre International de Rencontres Mathématiques, 2022.

[Rij22b] Egbert Rijke. Introduction to Homotopy Type Theory, 2022.

[RSPC+] Egbert Rijke, Elisabeth Stenholm, Jonathan Prieto-Cubides, Fredrik Bakke, and others. The agda-unimath library.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[VAG+] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. Unimath — a computer-checked library of univalent mathematics. available at http://unimath.org.

[VMA19] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical agda: a dependently typed programming language with univalence and higher inductive types. *Proc. ACM Program. Lang.*, 3(ICFP):87:1–87:29, 2019.

[Voe15] Vladimir Voevodsky. An experimental library of formalized mathematics based on the univalent foundations. *Math. Struct. Comput. Sci.*, 25(5):1278–1294, 2015.