Stockholm University

Synthetic approaches to cohomology, homology and homotopy

Axel Ljungström

Doctoral Thesis in Computational Mathematics at Stockholm University, Sweden 2025

Synthetic approaches to cohomology, homology and homotopy

Axel Ljungström

Academic dissertation for the Degree of Doctor of Philosophy in Computational Mathematics at Stockholm University to be publicly defended on Wednesday 21 May 2025 at 14.00 in Lärosal 10, Hus 2, Vån 2, Albano, Albanovägen 18 and online via Zoom, public link is available at the department website.

Abstract

This thesis is based on five papers on the development of synthetic homotopy theory in homotopy type theory (HoTT), a relatively recent system of mathematics which extends Martin-Löf type theory with higher inductive types and univalence. The thesis is, in particular, concerned with the development of (co)homology theories and operations, but it also covers other (often related) cornerstone results from homotopy theory. Most results presented here have been *computer formalised*, i.e. digitally verified by a computer, in the verification software (or *proof assistant*) Cubical Agda.

Paper I presents a construction and computer formalisation of cohomology rings in HoTT. To this end, the associativity of cup products is proved – a seemingly easy problem which has turned out to be rather difficult in HoTT due to complicated coherences which arise when attempting a naïve proof. The paper also contains various computations of cohomology groups and rings, the Gysin sequence and a discussion of computational aspects of the computer implementation in Cubical Agda. The paper, in many ways, serves as an introduction to the remainder of the thesis.

Paper II presents a computer formalisation of Brunerie's (2016) computation of the fourth homotopy group of the 3-sphere. In addition to this, we provide a vastly simplified version of Brunerie's proof by introducing a new way of computing (both by hand and by normalisation in Cubical Agda) the so-called *Brunerie number*, i.e. the order of the homotopy group in question.

Paper III is devoted to solving another surprisingly difficult problem in HoTT, namely that of showing that the smash product is (1-coherent) symmetric monoidal. This is done by developing a heuristic for constructing homotopies over large iterated smash products. This heuristic is also expressed as a formal theorem which, in essence, presents iterated smash products as retracts of homotopically simpler spaces.

Paper IV presents work on cellular homology in HoTT. We develop basic theory of CW complexes and prove a constructive analogue of the cellular approximation theorem, as well as a special case of the CW-approximation theorem which states that two different notions of n-connectedness are equivalent. The cellular approximation theorem is then used to prove functoriality of cellular homology, and the special case of the CW-approximation theorem is used to prove the Hurewicz theorem. We also verify that our homology theory satisfies the Eilenberg–Steenrod axioms.

Paper V uses the cohomology theory from Paper I in order to construct the Steenrod squares, a set of cohomology operations for mod 2 cohomology. We use (and generalise) a definition of these operations due to Brunerie (2017), but go much further and prove their characterising properties, e.g. the Cartan formula and the Adem relations. This is done by reducing most problems to a 'master theorem' which is a simple-to-state but difficult-to-prove Fubini-like statement concerning so-called unordered joins. We use the squares to complete an alternative computation of the fourth homotopy group of the 3-sphere suggested in Paper II.

Keywords: *homotopy type theory, constructive mathematics, formalisation.*

Stockholm 2025
http://urn.kb.se/resolve?urn=urn:nbn:se:su:diva-241553

ISBN 978-91-8107-196-2
ISBN 978-91-8107-197-9

Department of Mathematics

Stockholm University, 106 91 Stockholm

SYNTHETIC APPROACHES TO COHOMOLOGY, HOMOLOGY AND HOMOTOPY

Axel Ljungström

Stockholm
University

Synthetic approaches to cohomology, homology and homotopy

Axel Ljungström

ISBN print 978-91-8107-196-2
ISBN PDF 978-91-8107-197-9

Printed in Sweden by Universitetsservice US-AB, Stockholm 2025

Sammanfattning på svenska

Den här avhandlingen består av fem artiklar om syntetisk homotopiteori i homotopi-typteori (HoTT), ett formellt system som utvidgar Martin-Löfs typteori i den bemärkelsen att det även innefattar högre-induktiva typer samt univalens. Avhandlingen handlar i synnerhet om (ko)homologiteorier och kohomologioperationer men behandlar även andra (ofta relaterade) nyckelresultat från homotopiteori. Det är många av resultaten som presenteras här som inte enbart är viktiga från ett matematiskt perspektiv, utan som också är väsentliga eftersom de har *datorformaliserats*, d.v.s. formellt verifierats av ett datorprogram. En majoritet av satserna i den här avhandlingen har formaliserats i bevisassistenten `Cubical Agda`.

Artikel I beskriver en konstruktion av kohomologiringar i HoTT. För att åstadkomma detta bevisas att kopprodukten (eng. *cup product*) är associativ – ett till synes enkelt problem som har visat sig vara icke-trivialt i HoTT då naiva bevismetoder lätt fastnar i komplicerade koherensproblem. Artikeln innehåller även beräkningar av kohomologigrupper och kohomologiringar, Gysinföljden samt en diskussion rörande beräkningsmässiga aspekter av vår datorimplementering i `Cubical Agda`. Artikeln kan ses som en introduktion till resten av avhandlingen.

Artikel II presenterar en datorformalisering av Bruneries (2016) beräkning av den fjärde homotopigruppen av 3-sfären. Utöver detta beskrivs även en avsevärd förenkling av Bruneries bevis. Det här åstadkoms genom att introducera ett nytt sätt att beräkna det så kallade ”Brunerie-talet”, d.v.s. ordningen av homotopigruppen i fråga.

Artikel III är helt och hållet tillägnad beviset av det faktum att smashprodukter är symmetrisk-monoidala. Det här är en sats som vid första anblick ser enkel ut men som har visat sig vara förvånansvärt svårbevisad i HoTT. Här utvecklas en heuristik som förenklar konstruktionen av homotopier över godtyckligt itererade smashprodukter vilken senare används för att bevisa satsen i fråga. Heuristiken uttrycks också som en formell sats.

Artikel IV handlar om cellulär homologi i HoTT. Vi utvecklar grundläggande teori om CW-komplex och bevisar en konstruktiv version av satsen om cellulär uppskattning (eng. *cellular approximation theorem*) samt ett specialfall av satsen om CW-uppskattning (eng. *CW-approximation theorem*) vilken säger

att två olika definitioner av n-sammanhängighet (eng. *n-connectedness*) sammanfaller. Vi använder den förstnämnda satsen för att bevisa funktorialitet för cellulär homologi och den sistnämnda satsen för att bevisa Hurewicz sats som relaterar vissa homologigrupper och homotopigrupper. Vi verifierar även att vår homologiteori uppfyller Eilenberg-Steenrod-axiomen.

I Artikel V används kohmologiteorin från Artikel I för att konstruera de så kallade Steenrodkvadraterna, ett system av kohomologioperationer för mod 2-kohomologi. Den definition som används kommer från en kort text av Brunerie (2017) men den generaliseras och används för att bevisa de egenskaper som karakteriserar Steenrodkvadraterna: i synnerhet Cartanformeln och Ademrelationerna. Det här görs genom att reducera till en huvudsats, vilken är lätt att formulera men svår att bevisa, som uttrycker ett slags Fubini-egenskap hos så kallade oordnade sammansättningar (eng. *joins*). Vi använder slutligen Steenrodkvadraterna för att fylla i en lucka i en alternativ beräkning av den fjärde homotopigruppen av 3-sfären från Artikel II.

List of Papers

This thesis is based on five papers, numbered I–V, which are included in the following order.

I. Axel Ljungström and Anders Mörtberg. Computational Synthetic Cohomology Theory in Homotopy Type Theory. To appear in *Mathematical Structures in Computer Science*. 2024. arXiv: 2401.16336

II. Axel Ljungström and Anders Mörtberg. Formalising and Computing the Fourth Homotopy Group of the 3-Sphere in Cubical Agda. Preprint. 2024. arXiv: 2302.00151

III. Axel Ljungström. Symmetric monoidal smash products in homotopy type theory. *Mathematical Structures in Computer Science* (2024), pp. 1–23. DOI: 10.1017/S0960129524000318

IV. Axel Ljungström and Loïc Pujet. Cellular Methods in Homotopy Type Theory. Preprint. 2025. URL: https://aljungstrom.github.io/files/cellular2025.pdf

V. Axel Ljungström and David Wärn. The Steenrod squares via unordered joins. To appear at the *40th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 2025. URL: https://aljungstrom.github.io/files/steenrod2025.pdf

Related but excluded papers The following papers have been produced during my PhD but are not included in this thesis.

EI. Guillaume Brunerie, Axel Ljungström and Anders Mörtberg. "Synthetic Integral Cohomology in Cubical Agda". *30th EACSL Annual Conference on Computer Science Logic (CSL 2022)*. Ed. by Florin Manea and Alex Simpson. Vol. 216. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 11:1–11:19. ISBN: 978-3-95977-218-1. DOI: 10.4230/LIPIcs.CSL.2022.11

EII. Thomas Lamiaux, Axel Ljungström and Anders Mörtberg. "Computing Cohomology Rings in Cubical Agda". *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2023. Boston, MA, USA: Association for Computing Machinery, 2023, pp. 239–252. ISBN: 9798400700262. DOI: 10.1145/3573105.3575677

EIII. Axel Ljungström and Anders Mörtberg. "Formalizing $\pi_4(\mathbb{S}^3) \cong \mathbb{Z}/2\mathbb{Z}$ and Computing a Brunerie Number in Cubical Agda ". *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. Los Alamitos, CA, USA: IEEE Computer Society, 2023, pp. 1–13. DOI: 10.1109/LICS56636.2023.10175833

EIV. Stefania Damato, Thorsten Altenkirch and Axel Ljungström. Formalising inductive and coinductive containers. Submitted. 2024. arX-iv: 2409.02603

Papers EI and EII are excluded due to their overlap with Paper I. The relation between Paper EI and Paper I is that the latter is a generalisation of the former (see the coming paragraphs for details). Like Paper I, Paper EII also concerns cohomology rings but has a strong emphasis on the computer formalisation of these. My contribution to this paper is the actual mathematics behind the computations of the cup product structure on the cohomology rings presented there, but these proofs had to be omitted due to space constraints and scope. Paper I contains these details instead, so from the point of view of my personal contribution, it would be superfluous to include both papers. Paper II is an extended (journal) version of Paper EIII, which is why the latter is excluded. Paper EIV is excluded because it does not relate very strongly to any of the other papers included in this thesis. I am also not a main author of this paper.

Relation to licentiate thesis The excluded papers EI and EII were included in my licentiate thesis and thus, since these papers are, respectively, closely related to Papers I and II, there is some overlap between my licentiate thesis and this thesis. In both cases, however, the version in this thesis constitutes a significant extension to that in my licentiate thesis. Paper I generalises Paper EI from integral cohomology to cohomology for arbitrary groups and contains several new cohomology computations. Paper II extends Paper EIII primarily by means of §6 which spells out the details of and generalises the methods behind one of the main novelties of Paper EIII, namely the simplified computation of the Brunerie number.

Author's contribution Paper I is primarily written by me. Most importantly, I am responsible for all mathematical results and the computer formalisation of the paper. Mörtberg supervised the project and was mainly involved in the benchmarking of our computer implementation as well as writing parts of the introduction and the section on related and future work.

For Paper II, the situation is similar to that for Paper I. I am the main author of the paper with Mörtberg acting as a supervisor of the project. As in Paper I, I am responsible for the mathematical results and the computer formalisation while Mörtberg has assisted in benchmarking as well as writing the introduction and conclusion.

Paper III is written solely by me.

Paper IV was written together with Pujet and it is difficult to take or give sole credit for any particular result. Almost the entire paper was conceived during discussions with Pujet and we would work on all mathematical results and the computer formalisation together in person (as we shared an office at the time, this is no exaggeration). There are two main theorems in the paper: the cellular approximation theorem and the Hurewicz approximation theorem. Pujet devised the first proof sketch of the former while I devised the first one of the latter. Nevertheless, in the end, these proofs were reworked and improved on together and have changed since their first renditions.

Paper V was written with Wärn. Most of the mathematics was developed at in-person meetings which would take place whenever Wärn was in Stockholm. It is especially difficult to estimate the contributions as the project was so extensive, taking roughly 2 years to finish; during this time there have been many different approaches taken that never made it into the final paper. Very often, Wärn would suggest a certain idea (e.g. the usefulness of unordered joins rather than smash products) and I would explore whether this idea is feasible by trying to devise the relevant computer formalised proofs. Nevertheless, this is a rather simplified picture: most theory was simply developed together on the blackboard.

Acknowledgements

First and foremost, I would like to thank Anders Mörtberg. I am very lucky to have had such a present supervisor throughout my doctoral studies. The time and energy he has devoted to me over these five years is, I think, more than any PhD student anywhere could expect.

I also wish to thank all of my colleagues for making our department such a nice workplace. I especially want to thank the logic group for many nice seminars and dinners.

I must also thank my friends and family who have allowed me to maintain some level of normality during my years as a PhD student. Because she told me not to, I have to include a special thanks to Claire Caron, my wife, for putting up with me and my never-ending deadlines.

Finally, I would like to thank everyone who has taken an interest in and supported my academic endeavours, be it collaborators or old teachers. In particular, I want to acknowledge Roussanka Loukanova, who, early on in my education, encouraged me to pursue a career in mathematics and computer science.

Contents

A general introduction to (homotopy) type theory

It is never easy to be wrong. For mathematicians, however, whose whole raison d'être is *being certain*, this prospect is perhaps especially terrifying. Although the papers produced by mathematicians undergo proof-readings and peer-reviews just like in any other field, it is impossible to guarantee that they are completely void of mistakes. In a field as fragmented as mathematics, where two leading experts in their respective subfields can be so far apart academically that they have little else than the weather to chat about in the lunchroom, mistakes have an even better chance of surviving, simply due to the (often) small number of mathematicians who possess the appropriate background to appreciate the material of a given research paper. This can cause errors in influential papers to go unnoticed for years.

A particularly well-known case of an influential paper containing a non-trivial mistake is Kapranov and Voevodsky's '∞-groupoids and homotopy types' [Mik91]. The paper presented a proof of a version of the famous *homotopy hypothesis*. It was not until seven years later that Simpson [Sim98] found a mistake in their proof (and not until yet another 15 years that Voevodsky had, to his own satisfaction, understood the mistake in the original paper [Voe14]). Unhappy with this situation, Voevodsky started to put a serious amount of thought into the prospect of using digital software to check the correctness of mathematical proofs. Such computer programs tend to rely on *type theories*; these are formal systems, some of which have been suggested as an alternative to the *set theory* that is currently the most widely accepted foundation of mathematics. The idea is simple: if we are able to express our mathematical proofs using type theory rather than set theory, we can much more readily ask our computers to verify their correctness. The version of type theory to be used would be the intensional type theory of Martin-Löf [Mar84] endowed with new principles (univalence and higher inductive types) making it suitable for reasoning about the field of *homotopy theory*. The resulting type theory, *homotopy type theory* (HoTT), was designed and studied by Voevodsky and, one must stress, *several* collaborators during a special year at the Institute for Advanced Study, resulting in *the HoTT book* [UF13], the first textbook on

the subject.

This thesis is my contribution to the ongoing work of translating old results into – and developing new results in – the setting of HoTT. It is primarily concerned with results concerning concepts like homotopy, cohomology and homology. While many of the results are well known from the point of view of classical mathematics, their proofs in HoTT often differ significantly. The results are also automatically generalised when expressed in HoTT, due to its rich family of (higher-categorical) models [Shu19]. One of the key motivations behind this work is, like it was for Voevodsky, to be able to digitally verify – to *computer formalise* – these results. Indeed, many of the results presented in this thesis have been computer formalised in the *proof assistant* software `Cubical Agda` [VMA21].

Structure of this chapter This chapter has four main sections and they are all intended for a relatively broad audience of mathematicians. The first three introduce (Martin-Löf) type theory, homotopy type theory and the `Cubical Agda` proof assistant. The expert can likely jump straight to §4. I wish to emphasise here that much of the material presented in the coming sections is also included in some of the papers (albeit in a more compact form).

We start off, in §1, with an introduction to type theory. This section is aimed towards a very general mathematical audience and can be skimmed or even skipped by the expert. For the expert, we summarise this section by saying that it introduces Martin-Löf type theory with universes à la Russell: in essence, a type theory closely resembling that of `Agda`.

In §2, we introduce homotopy type theory. Like the previous section, this section is also intended to be accessible to anyone with a reasonable level of mathematical maturity. It does, however, cater more to algebraic-topological intuitions than §1. The reader who is not helped by this can skip to §2.2. The section also touches on *cubical type theory*, a version of homotopy type theory which has been used when computer formalising the results in this thesis.

In §3, we briefly discuss the `Agda` proof assistant (i.e. verification software) and its extension `Cubical Agda`. The latter proof assistant implements cubical type theory and has been used to check many of the proofs in this thesis. It should be mentioned that Paper II also includes a brief introduction to `Cubical Agda` for those who wish to skip this section.

In §4, we briefly and very generally discuss the contributions of this thesis and the scientific context it was written in. We mention related work and what kind of future projects the material in this thesis could lead to.

1 Type theory

In order to explain what homotopy type theory (HoTT) is, we will need to say a few words about type theory in general. In this thesis, 'type theory' will always refer to a specific type theory, namely the intensional type theory of Martin-Löf (MLTT)[Mar84]. This is a *dependent* and *intuitionistic* type theory which means, respectively, that it allows for dependent types (roughly, indexed sets) and that it is constructive. Let us start off this brief introduction by explaining some of the basic ideas behind type theory and why anyone would ever want to use it for doing mathematics.

1.1 What is type theory?

Type theory is a formal system which, among other things, provides an alternative foundation of mathematics. Here is a one-liner describing it (which I forbid you to quote): *type theory is obtained by taking set theory and swapping the notation '$x \in X$' for '$x : X$'.* While this, from a formal perspective, is a vast oversimplification, it should provide the traditionally schooled mathematician with enough information about the role played by types in type theory required to parse most of the material presented in this thesis. We read the statement (or *judgement*) '$x : X$' as 'x is an element (or *term*) of *type* X'. This, of course, begs the question of what the difference between types and sets actually is. There are many (not necessarily mutually compatible) answers to this, and I will only attempt to provide one point of view here. Types are, unlike sets, not to be seen as a collection of already defined objects. Rather, I claim that a type is, in essence, a description of how to manipulate and interact with objects. In a way, a type is just a label containing user instructions which we slap onto our mathematical constructions. A standard example of this is the type of natural numbers. This is an *inductive type* which can be defined by the following two clauses (inspired by Peano arithmetic [Pea89]).

- $0 : \mathbb{N}$
- $\mathsf{suc} : \mathbb{N} \to \mathbb{N}$

Above, the first clause simply postulates the existence of an element of type $\mathbb{N}$ which we call 0. The second clause postulates the existence of a function suc which takes a natural number $x : \mathbb{N}$ and returns another natural number $\mathsf{suc}(x)$. Note that this presentation only describes how to generate new natural numbers – it never presupposes that we can gather them in one big collection.

A particular consequence of this description is that if you are presented with a natural number n, you can always be certain that it is either of the form 0 or $\mathsf{suc}(n')$ for some 'smaller' natural number n'. This is not only a theorem but a meta-mathematical fact which, in the version of type theory we have in mind here, holds *by definition*. More generally, we will always assume that

inductive types, such as $\mathbb{N}$, come equipped with the appropriate induction and computation rules. In this case, this means (somewhat simplified) that our type theory primitively supports induction on the natural numbers and that any function $f : \mathbb{N} \to A$ defined by induction actually *computes* according to its inductive description.

Never ask a term its type One important feature of type theory is that all mathematical objects are typed upon construction. This means that the question of whether a certain mathematical object belongs to a certain type becomes either tautological or nonsense – *to be a mathematical object* means, in type theory, *to be an element of a type*. If you present me with some $q : \mathbb{Q}$, you cannot then ask me whether $q : \mathbb{Z}$. Indeed, unless $\mathbb{Q}$ and $\mathbb{Z}$ were two names for the *exact* the same type, q cannot be an element of both![1] From a formal perspective, this is, of course, the exact same situation as in set theory where the statement '$q \in \mathbb{Q}$ and $q \in \mathbb{Z}$' is mere syntactic sugar for 'there is some $a \in \mathbb{Z}$ which the canonical embedding $\mathbb{Z} \hookrightarrow \mathbb{Q}$ takes to $q \in \mathbb{Q}$'. Type theory, by design, often forces us to make such syntactic sugar explicit. This is perhaps not always so important when communicating our proofs to humans, but when communicating our proofs to computers, who do not possess quite the same ability to read between the lines, this is crucial.

Often, by being strict about typing, type theory can help us make formal certain informal ideas we are used to from working in a set-theoretic foundation such as ZF (i.e. Zermelo–Fraenkel set theory – see e.g. [BM75]). Most mathematicians happily accept the difference in nature between the natural number 0 and the empty set, despite the fact that, in ZF, these objects are *exactly* the same. In a similar vein, we rarely see an algebraic topologist speak of a manifold of dimension n with $4 \in n$. We know that, although these constructions are entirely grammatical in the language of set theory, it is tacitly agreed among mathematicians that we are not really supposed to make them. The idea is, of course, that objects such as 0 and $\emptyset$ are incomparable in nature – one might even say that they are of different *type*. With any reasonable type-theoretic definitions of 0 and $\emptyset$, we could never ask whether $0 = \emptyset$ – because $0 : \mathbb{N}$ and $\emptyset : \mathsf{Set}$ are of different type, we can never compare them in this way.

Type theory and constructive mathematics The idea we have presented so far is that type theory is a system very similar to set theory but where (i) 'finitary' constructions of infinite objects are preferred to 'infinitary' (recall the type-theoretic definition of $\mathbb{N}$) and (ii) each mathematical object has a type, often forcing the mathematician to explicitly clarify informal ideas. There is, however, (at least) one more aspect of type theory worth mentioning: it is, by default, constructive. Let me mention the, to me, most interesting implications of this fact.

First and foremost, the constructive logic captured by type theory has a very

[1] OK, type theorist reading this: at least in MLTT it cannot.

rich class of models: one says that constructive logic is the *internal language of toposes* [MM94]. Now, if you do not know what a topos is, you can think of it as a kind of landscape in which we can do mathematics. All mathematicians know one topos very well, namely the topos of sets (the standard model of ZF). As it is a topos, any result proved constructively can be interpreted in the category of sets. However, unlike non-constructive results, we can also interpret them in *other* toposes – e.g. sheaves on a site, or any other permutation of serious-sounding words which may or may not excite the reader. This means that anything we prove using type theory will say something both about sets and something about, say, sheaves simultaneously. Many theorems proved in traditional mathematics are, to a significant extent, already constructive. A problem with using classical set-theoretic foundations is that they make it difficult to separate the non-constructive from the constructive parts of a proof. If we instead use a flexible constructive system like type theory, constructive reasoning is the default. When we prove a theorem, we *are* allowed to rely on non-constructive principles, but we have to make sure to explicitly state whenever we do so. This means that we can extract additional (computational) information even from non-constructive proofs by isolating those steps which do not rely on non-constructive principles.

A second implication of the use of a constructive system like type theory as a foundation is that proofs now can be interpreted as concrete, executable computer programs. The fundamental observation is that type theory, as a constructive language, really is just a programming language [Mar82; NPS90]. The guiding principle of constructive mathematics is that any time we make an existential statement, e.g. the timeless classic 'there are irrational numbers a and b such that a^b is rational', we have to provide a clear *witness* (or, in this case, *witnesses*) of this statement. A classical mathematician may provide a proof by contradiction: 'assume such a pair of irrational numbers does not exist... contradiction'. Such a proof never provides us with any concrete information about the actual values of a and b. In constructive mathematics, we cannot really prove this in any other way than by providing concrete examples (witnesses): 'consider $a = \sqrt{2}$ and $b = \log_2 9$'. Thus, a constructive proof provides more information than a non-constructive proof: it contains information on how to compute witnesses of the statement in question. Now, suppose you devised a constructive proof of something somewhat more eye-opening than the example covered here. It could be that there must be some integer solution a to whatever interesting equation you are trying to solve or that your favourite group is isomorphic to $\mathbb{Z}/n\mathbb{Z}$ for some $n : \mathbb{N}$. Constructive proofs of these statements should, at least in principle, provide enough information for us to deduce the exact values of a and n without having to do any further theorem proving. Indeed, a constructive proof of a theorem can be thought of as an *algorithm* for computing witnesses of the theorem. This idea of proofs as algorithms is especially central to type theory where the so-called *Curry–Howard correspondence* [How80] identifies, in a meaningful way, propositions and proofs with, respectively, types and elements of types. At the same time, types and elements of types have a natural interpretation in terms of, respectively, program

specifications and computer programs. We will return to these ideas in §3. For now, the main takeaway should be that a proof in type theory contains computational content which can be used to simplify or even completely remove the need for future theorem proving.

1.2 A crash course in type theory

So far, there has been a lot of talk about the general idea behind type theory but we have not actually engaged in it formally. In the following couple of pages, I will try to walk you through the very basics of the system. Before we start, I again wish to emphasise the versatile role played by types: we have already seen how a type can be used to capture one of our favourite sets, the natural numbers; in what follows, we will also see how types also can express logical propositions such as '$5 + 7 = 12$' and '2 is an even number'. As these propositions will be captured by types, proofs of these propositions will be precisely elements of these types. Thus, if we write $p : 5 + 7 = 12$, this means that p is a proof of the statement that $5 + 7 = 12$. In exactly the same way, we may write $n : \mathbb{N}$ to say that n is a natural number. Hence, in type theory, a proof is not a meta-mathematical entity but a mathematical object in the same right as, say, the natural number zero. The justification of this apparent conflation of concepts comes from the constructivist's idea of proofs as witnesses. In essence, even an element $n : \mathbb{N}$ is a proof – it is one out of ω many witnesses of the existence of natural numbers. This may take some time getting used to but should hopefully become clearer soon when we consider some concrete constructions in type theory.

Universes The first thing we will need is a universe: a special type $\mathcal{U}$ whose elements are types. Whenever we declare a new type, we write this as $A : \mathcal{U}$. Now, certainly type-theorists know better than to assume the existence of a 'type of all types'. After all, the original version of type theory was invented precisely in order to circumvent Russell's paradox [Rus03]. This can be solved by allowing for a whole hierarchy of universes, $\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \dots$, each an element of the other. This approach – often called *universes à la Russell* – is the one we adhere to in this thesis, although there are alternatives (e.g. *universes à la Tarski*) [Mar84, p. 48]. In practice, however, we will almost always be informal (as is common practice) and simply write $\mathcal{U}$ to denote an arbitrary universe. It is often sufficient to let $\mathcal{U} := \mathcal{U}_0$. We remark that the notation Type may be used instead of $\mathcal{U}$ in some of the papers featured in this thesis.

Functions Given two types $A, B : \mathcal{U}$, we will simply write $A \to B$ for the type of functions from A to B. Functions are introduced in the obvious way: we define a function $f : A \to B$ by defining, for every $a : A$, an element $f(a) : B$. When it comes to functions, there is one piece of notation you need

to know if you wish to be let into any secret gathering of type theorists: lambda notation. Lambdas are a way of introducing functions without naming them. This idea already exists in traditional mathematics where we write e.g. $x \mapsto x^2$ for the squaring function on, say, the natural numbers. Many type theorists would write this function as $\lambda\, x\,.\, x^2$ or, less traditionally, $\lambda\, x \to x^2$ or even $\lambda\,(x : \mathbb{N}) \to x^2$ to clarify the input type. This notation comes from lambda calculus [Chu32], a formal system closely related to type theory which is useful for capturing certain notions of computation.

Under the interpretation of types as logical propositions [How80], function types correspond to implication. In this case, a function $f : P \to Q$ between two types P and Q is thought of as a proof of the implication 'if P then Q'. We will return to this idea later when we speak of dependent functions.

Inductive types One important class of types is that of *inductive types*. These are types whose elements are inductively defined from a set of so-called *constructors*. We have already seen what is probably the most canonical example of an inductive type: the natural numbers $\mathbb{N}$. Recall, we defined these to be the type generated by two constructors, namely $0 : \mathbb{N}$ and $\mathsf{suc} : \mathbb{N} \to \mathbb{N}$. As touched upon earlier, we will always assume recursion/induction principles to be part of the definition of any inductive type. For $\mathbb{N}$, the recursion principle can be stated semi-formally by saying that any function $f : \mathbb{N} \to A$ is can be described (unambiguously) by providing the following data.

- $f(0) : A$
- for any $n : \mathbb{N}$, a term $f(n+1) : A$ whose definition may recursively refer to $f(n)$.

The natural numbers do not constitute the only example of an inductive type. We can, for instance, define the booleans, $\mathbb{B}$, and the unit type, $\mathbb{1}$, respectively to be the inductive types with constructors

$$\mathtt{true}, \mathtt{false} : \mathbb{B} \qquad\qquad \star_1 : \mathbb{1}.$$

The recursion principles for these types say, respectively, that a function $f : \mathbb{B} \to A$ is defined entirely by $f(\mathtt{true}) : A$ and $f(\mathtt{false}) : A$ and a function $g : \mathbb{1} \to A$ is defined entirely by $g(\star_1) : A$. We can also define the empty type $\bot$ to be the inductive type with no constructors. Its recursion principle says that there always exists a (unique) function $\epsilon : \bot \to A$, i.e. the empty function.

Dependent types, functions and sums So far, we have described a plain, non-dependent type theory. The type theory we are concerned with here, i.e. MLTT, is a *dependent* type theory. Dependent types are simply types parameterised by other types. This should not be a foreign concept to the classical mathematician who, perhaps without knowing, relies on such dependency for many crucial constructions. Now, a type being parameterised by another type

may, at least for the more algebraically inclined among us, invoke ideas of e.g. parameterised spaces, fibre products and so on. While these certainly are examples of dependent types, there are much more basic examples. Consider, for example, the n-dimensional reals, $\mathbb{R}^n$. This is a construction we often like to speak of with respect to an arbitrary natural number n. Really, $\mathbb{R}^{(-)}$ is a function which takes as input some natural number n and returns the set $\mathbb{R}^n$. In other words, it is (size-issues aside) a function $\mathbb{N} \to \mathsf{Set}$. In type theory, the idea is the same, the only difference being that we write $\mathcal{U}$ instead of Set; we say that $\mathbb{R}^{(-)} : \mathbb{N} \to \mathcal{U}$ is a *dependent type* (over $\mathbb{N}$). In general, a dependent type indexed by some type A is simply a function $B : A \to \mathcal{U}$.

Note that if we wish to speak of elements of a dependent type, we can only speak of elements of $B(a)$ given some fixed $a : A$. For instance, in the case of $\mathbb{R}^n$, it makes sense to write $(5, 7) : \mathbb{R}^2$ but not $(5, 7) : \mathbb{R}^{(-)}$. Some constructions are, however, universal enough to make sense for any input. For instance, regardless of the value of n, we always have a neutral element $0^n : \mathbb{R}^n$. In other words, $0^{(-)}$ is a function which for every $n : \mathbb{N}$ returns $0^n : \mathbb{R}^n$. This kind of function differs in one fundamental way from the functions discussed previously: the output type ($\mathbb{R}^n$) depends on the input element (n). We call such a function *dependent*. In general, given a dependent type $B : A \to \mathcal{U}$, we shall write $(a : A) \to B(a)$ for its associated dependent function type. Note that the non-dependent function type can be obtained as the special case when B is constant, i.e. when, for any $a : A$, we set $B(a) := B'$ for some (non-dependent) type B'. The reader should know that the traditional notation for the type $(a : A) \to B(a)$ is $\prod_{a:A} B(a)$ (and, in fact, dependent function types are commonly referred to as Π-types or dependent products). While I think the former notation is often clearer, the Π-notation could be useful for the reader who likes to think of types as sets. With this interpretation, a dependent type $B : A \to \mathcal{U}$ corresponds to a family of A-indexed sets, while the dependent function type $\prod_{a:A} B(a)$ is simply the product over this family.

Another crucial construction is that of dependent sums. Given a dependent type $B : A \to \mathcal{U}$, its dependent sum is a type which we denote by $(a : A) \times B(a)$, although traditionally (and sometimes in this thesis) it is also written $\sum_{a:A} B(a)$. An element of this type is a pair (a, b_a) where $a : A$ and $b_a : B(a)$. For instance, both $(2, (5, 7))$ and $(3, (5, 7, 12))$ are elements of $(n : \mathbb{N}) \times \mathbb{R}^n$. We use the notation fst and snd to denote the first and second projections, i.e. $\mathsf{fst}(a, b_a) = a$ and $\mathsf{snd}(a, b_a) = b_a$. Set-theoretically, we can view dependent sums as playing the role of indexed disjoint unions with $(a : A) \times B(a)$ corresponding to the A-indexed union $\bigsqcup_{a:A} B(a)$. Whenever B does not depend on A, we simply write $A \times B$ as this construction, by definition, is a binary product.

There is an elementary but important interplay between (dependent) functions and sums. In type theory, it is not uncommon to ask for a function $f : A \to (B \to C)$. Formally, this is a function taking an argument $a : A$ and returning another function $f(a) : B \to C$. In practice, however, we simply think of this as a function taking two arguments – one in A and one in

B – and returning something in C. Given $a : A$ and $b : B$, we *should* write $f(a)(b) : C$ for the output of f applied to first a and then b. In practice, however, it is often nicer to simply write $f(a, b)$ and think of f as a function of type $A \times B \to C$. The translation between these two representations of binary functions is called, in this direction, *uncurrying* and in the other direction *currying* (or, less commonly but perhaps more justly, had the name not been so long, *Schönfinkelisation* [Sch24]). We will use this trick in many places and without comment. Now, currying is not a concept reserved for non-dependent functions: thanks to the existence of dependent sums, we can also curry dependent functions. For the most general case, consider three types of increasing dependency $A : \mathcal{U}$, $B : A \to \mathcal{U}$ and $C : (a : A) \to B(a) \to \mathcal{U}$. First of all, we can uncurry C, viewing it instead as a family of types defined over the dependent sum over B. That is, $C : (a : A) \times B(a) \to \mathcal{U}$. Now, suppose we are given a dependent function $f : (a : A) \to (b : B(a)) \to C(a, b)$. Just like in the non-dependent case, we will write the application of f as $f(a, b)$ rather than $f(a)(b)$ which is motivated by the fact that, by uncurrying, we may view f as a dependent function $f : ((a, b) : (a : A) \times B(a)) \to C(a, b)$.

Finally, let us make a remark concerning the interplay between dependent types and inductive types. It was previously pointed out that all inductive types are thought of as automatically coming equipped with recursion rules. For instance, the recursion rule for $\mathbb{N}$ describes how functions of type $\mathbb{N} \to A$ are constructed. We never saw, however, how to construct a dependent function $f : (n : \mathbb{N}) \to B(n)$. For natural numbers, this is the obvious dependent analogue of the recursion rule: it suffices to specify $f(0)$ and provide, for any $n : \mathbb{N}$, a function $B(n) \to B(n + 1)$. Every recursion rule has such a dependent analogue which we call an *induction rule*. Of course, recursion rules are mere special cases of these more general induction rules. Just like with recursion, we will assume all inductive types to satisfy their obvious induction rules.

Dependent types as predicates Until now, our explanation of dependent types has primarily catered to set-theoretic intuitions. However, dependent types also play a crucial role in expressing logical predicates. Say, for instance, that we wish to encode the predicate $\mathsf{isEven}(n)$ expressing that a given $n : \mathbb{N}$ is even. Clearly, this predicate depends (both syntactically and semantically) on the input variable $n : \mathbb{N}$. Thus, this predicate must be encoded as a dependent type $\mathsf{isEven} : \mathbb{N} \to \mathcal{U}$. Using the recursion rule for $\mathbb{N}$, we can actually define a version of this predicate already:

$$\begin{aligned} \mathsf{isEven}(0) &:= \mathbb{1} \\ \mathsf{isEven}(1) &:= \bot \\ \mathsf{isEven}(2 + n) &:= \mathsf{isEven}(n). \end{aligned}$$

Recall that if we view types as logical propositions, elements of types correspond to proofs of these propositions. For instance, an element $p : \mathsf{isEven}(48)$ is a proof of the fact that 48 is even. Under this interpretation, the type of dependent functions plays the role of proofs of universally quantified statements. In the

case of isEven, a dependent function $(n : \mathbb{N}) \to \mathsf{isEven}(n)$ would, if it existed, constitute a proof of the statement that every natural number is even. We can now also build up more complex statements. For instance, we can encode a proposition like 'for every natural number n, if n is even, then so is $2 + n$' by the type

$$(n : \mathbb{N}) \to \mathsf{isEven}(n) \to \mathsf{isEven}(2 + n).$$

To prove the statement, we need to define $p(n, q) : \mathsf{isEven}(2+n)$ for every $n : \mathbb{N}$ and $q : \mathsf{isEven}(n)$. Actually, we can complete this proof already now. Since we *defined* $\mathsf{isEven}(2 + n)$ to be $\mathsf{isEven}(n)$, we are really trying to construct an element $p(n, q) : \mathsf{isEven}(n)$. Thus, we may simply define $p(n, q) := q$, and we have thereby constructed our first proof in type theory.

Just like dependent function types correspond to universal quantifiers, dependent sums correspond to existential quantifiers. To continue on our example of the isEven predicate, suppose we were interested in providing a proof of the statement that 'there exists an even number'. In type theory, this amounts to providing two pieces of data: a natural number n and a proof $p : \mathsf{isEven}(n)$. Hence, this existential statement can be captured by the dependent sum $(n : \mathbb{N}) \times \mathsf{isEven}(n)$ whose objects are precisely pairs of these two pieces of data. Note that with this interpretation of the existential quantifier, there may be different ways of proving the same statement – for instance, both $(0, \star_1)$ and $(2, \star_1)$ are elements of $(n : \mathbb{N}) \times \mathsf{isEven}(n)$. This is a quirk which becomes entirely obvious when we think of existence proofs in terms of witnesses.

Identity types The type theory we have described so far looks quite a bit like a system of first order logic, and indeed this intuition is not far from the truth. However, if we wish for our type theory to be able to play this role, there is one construction missing: identity types. With the type theory we have so far, we are not able to express even the most basic theorems of arithmetic such as, say, the commutativity of natural number addition. Perhaps surprisingly, the treatment of identity types is one of the key differences between set theory and type theory. We wish to define, for any type A, a dependent type $\_ = \_ : A \to A \to \mathcal{U}$ such that, for any $x, y : A$, an element $p : x = y$ constitutes a proof of the fact that x and y are equal. One naïve definition would look something like

$$(x = y) := \begin{cases} \mathbb{1} & \text{if } x \text{ and } y \text{ are exactly the same object, syntactically} \\ \bot & \text{otherwise} \end{cases}$$

but this does not look like a definition in type theory. However, the right idea is there: we wish to express that any two general elements $x, y : A$ can only be declared equal if x and y are *literally* the same element. In other words, the only equality we should be able to infer automatically is $x = x$. The way we encode this in type theory is by an ingenious use of inductive types. This idea, due to Martin-Löf [Mar84], is really what distinguishes modern intuitionistic

type theory (i.e. MLTT) from other alternatives. For any $x : A$, we define the identity type $x = \_ : A \to \mathcal{U}$ to be the family of inductive types with one constructor:

- $\mathsf{refl}_x : x = x$.

This perhaps strange-looking construction is an *indexed inductive* type. These are essentially dependent analogues of inductive types but we will not need to discuss how to formally make sense of them here. This definition of identity may take some time getting used to. The intuition should be that it is valid to form the type $x = y$ for any $x, y : A$, but the only time you can actually construct an element of it (i.e. prove the equality of x and y) is when x and y are exactly the same object.

Like any inductive type, the identity type comes with an induction rule. It is so important that we will give it a name: the *J-rule*, or simply *J*.

Definition 1 (J)**.** *Fix $x : A$ and let $B : (y : A) \times (x = y) \to \mathcal{U}$ be a dependent type equipped with an element $b_0 : B(x, \mathsf{refl}_x)$. In this case, there is a dependent function $f : (y : A) \times (p : x = y) \to B(y, p)$. Furthermore, f satisfies the equality $f(x, \mathsf{refl}_x) = b_0$ and it holds strictly (i.e. by refl_{b_0}).*

Informally, the J-rule says that 'whenever we have a proof $p : x = y$, we may always replace y by x and p by refl_x'. There are some caveats here: for instance, y has to be a fresh variable, but let us postpone these discussions for later. Let us instead look at some examples of how the J-rule can be used to prove two elementary results about the identity type. We will do this with a decreasing level of formality, with the second proof looking much more like one written by the working type theorist.

Lemma 2 (Symmetry)**.** *For any $x, y : A$, if $x = y$, then $y = x$.*

A rather formal proof. The goal is to construct, for any $x, y : A$, a function $g : x = y \to y = x$. To this end, let us apply Definition 1 with $B(y, p) := (y = x)$. We have an element of $B(x, \mathsf{refl}_x)$, namely refl_x. We thus obtain a dependent function $f : (y : Y) \times (p : x = y) \to y = x$. We set $g(p) := f(y, p)$ and we are done. □

Lemma 3 (Transitivity)**.** *Let $x, y, z : A$. If $x = y$ and $y = z$, then $x = z$.*

A not so formal but close-to-practice proof. Let $p : x = y$ and $q : y = z$. We wish to produce an element $r : x = z$. By applying J to p, we may replace every y by x. After doing so, we have $q : x = z$, and thus we may simply set $r := q$ and we are done. □

As a sanity check, we should probably verify one of the defining features of any reasonable definition of equality: it is preserved by function application. This construction is important, so we will give it a name.

Lemma 4 (Function application)**.** *For any $f : A \to B$ and $x, y : A$, there is a function $\mathsf{ap}_f : x = y \to f(x) = f(y)$.*

Proof/Construction. We define $\mathsf{ap}_f(p)$ for $p : x = y$. By J, it is enough to define it when x is y and p is refl_x. In this case, we need to define $\mathsf{ap}_f(\mathsf{refl}_x) : f(x) = f(x)$ which we do by $\mathsf{ap}_f(\mathsf{refl}_x) := \mathsf{refl}_{f(x)}$ □

It is crucial when applying J to some identity proof $p : x = y$ that the statement we are trying to prove is defined for *any* y. In other words, y must be arbitrary and, in particular, cannot depend on x. We know, thanks to Hofmann and Streicher [HS98], that this would allow us to prove statements which are known to be independent (and even undesirable if we later wish to extend our type theory to obtain homotopy type theory). Consider, for instance, the following non-theorem and its accompanying non-proof.

Non-Theorem 1. *For any $p : x = x$, we have $p = \mathsf{refl}_x$.*

'Proof'. We apply J to p, replacing x by x and p by refl_x. We are left to prove $\mathsf{refl}_x = \mathsf{refl}_x$ which holds by $\mathsf{refl}_{\mathsf{refl}_x}$. △

The issue with the 'proof' above is that J does not apply to the identity type $x = x$ as the endpoint depends on x (indeed, it is literally x). For J to apply, we should be able to replace this endpoint x with an arbitrary element $y : A$. However, if we do this, the statement we are trying to prove is that $p = \mathsf{refl}_x$ for any $p : x = y$. This is not well-typed as the types of p and refl_x now differ (one mentions y and one does not). The significance of our failure to prove this statement will become much clearer soon when we discuss homotopy type theory.

We should briefly mention that there is another notion of identity: *strict* or *definitional* equality. So far, we have often seen statements like $f(x) := x^2$. I have assumed, without comment, that the reader parses this as '$f(x)$ is defined to be x^2'. When we make such a definition, $f(x)$ and x^2 become *definitionally* or *strictly* equal – from the point of view of the type theory, they are syntactically the exact same object. This notation will be used whenever two elements are syntactically the same (even if we did not explicitly define them as such). That is to say, we will see the notation $x := y$ used if $\mathsf{refl}_x : x = y$. Note that this notion of equality is meta-mathematical rather than a construction within the type theory.

Now, back to the actual identity type of our type theory: another important construction related to identity types is the *transport* function.

Definition 5 (Transport)**.** *Let $x, y : A$ and let $B : A \to \mathcal{U}$ be a dependent type. For any $p : x = y$, there is a function*

$$\mathsf{transport}^B(p, -) : B(x) \to B(y)$$

defined by applying J to p: that is, we define $\mathsf{transport}^B(\mathsf{refl}_x, -) : B(x) \to B(x)$ *by* $\mathsf{transport}^B(\mathsf{refl}_x, b) := b$.

On the one hand, thinking of types as propositions, the transport function simply captures Leibniz's law: if x and y are equal, then any property that holds for x also holds for y. On the other hand, the transport function allows us to define a notion of dependent identity.

Definition 6 (Dependent identity)**.** *Let $x, y : A$, $p : x = y$, $B : A \to \mathcal{U}$, $b_x : B(x)$ and $b_y : B(y)$. We define the type of dependent identities between b_x and b_y over p, denoted $b_x =_p^B b_y$, by*

$$(b_x =_p^B b_y) := (\mathsf{transport}^B(p, b_x) = b_y).$$

Whenever p is refl above, the dependent identity type above coincides precisely with the usual one.

The dependent identity type may seem daunting at first, but from a formal point of view, it is crucial. Consider something as simple as the axiom of graded commutativity for a graded ring $R_\bullet$. Informally, we would like to capture this by saying that for any $x : R_i$ and $y : R_j$, we have that $xy = (-1)^{ij}yx$. However, from a strictly formal perspective, the latter expression is not well-typed. Indeed, we have $xy : R_{i+j}$ while $(-1)^{ij}yx : R_{j+i}$ and the types appearing here are only equal up to an identification $c : i + j = j + i$. Thus, if we were to be entirely strict, we should write '$xy =_c^R (-1)^{ij}yx$'. Of course, when doing informal mathematics, we would never be this rigorous and simply use the usual identity type, abusing notation. It should be pointed out, however, that we are not, at this point, quite justified in accepting this convention. Indeed, there could be another identity proof $c' : i + j = j + i$, so which type do we mean when we informally write '$xy = (-1)^{ij}yx$'? Do we mean '$xy =_c^R (-1)^{ij}yx$' or do we mean '$xy =_{c'}^R (-1)^{ij}yx$'? For now, let us just accept it – we will touch upon these kinds of issues when we discuss homotopy type theory in §2.

There are also situations where the notion of dependent identity is more than just a formal necessity. For instance, it plays a crucial role in the description of identity types of dependent sums. To illustrate this, let $B : A \to \mathcal{U}$ be a dependent type and let $(x, b_x), (y, b_y) : (a : A) \times B(a)$ be pairs in its dependent sum. It is natural to ask what it means to prove these two pairs equal. A naive answer would be to say that $(x, b_x) = (y, b_y)$ if $x = y$ and $b_x = b_y$. However, the latter equality is not well-typed. Indeed, $b_x : B(x)$ and $b_y : B(y)$ belong to different types which are only equal up to an identity proof $p : x = y$. What we should say is that $(x, b_x) = (y, b_y)$ if there is an identity proof $p : x = y$ and a dependent identity $b_x =_p^B b_y$.

Since we covered the identity types of (dependent) sums, it only makes sense to say a few words about the identity types of (dependent) functions. This is a big topic which, you guessed it, will be better explained when we discuss homotopy type theory (at this point, the reader will hopefully have started anticipating

Type theory	Set theory	Logic
$A : \mathcal{U}$	A is a set	A is a proposition
$x : A$	$x \in A$	x is a proof of A
$A \to B$	$A \to B$	A implies B
$B : A \to \mathcal{U}$	B_a is a family of sets	B is a predicate
$(a : A) \to B(a)$	$\prod_{a \in A} B_a$	$\forall\, a\,.\, B(a)$
$(a : A) \times B(a)$	$\bigsqcup_{a \in A} B_a$	$\exists\, a\,.\, B(a)$
$x = y$	$x = y$	$x = y$

Table 1: Interpretations of type theory

that the identity types may play an important role further on). The obvious notion of equality for (dependent) functions is called *function extensionality*.

Definition 7 (Function extensionality)**.** *The principle of function extensionality says that two (possibly dependent) functions $f, g : (a : A) \to B(a)$ are equal if for every $(a : A)$, we have that $f(a) = g(a)$.*

We cannot prove function extensionality without modifying our type theory (see e.g. [Str93, §3.7]). Fortunately, when we later add the univalence axiom to get *homotopy* type theory, we will actually be able to prove it. If we do not wish to go that far, simply assuming function extensionality as an axiom is completely fine.

On a somewhat related note, we should also mention the dependent analogue of function application (Lemma 4). The original construction was given for plain functions but can now also be given for dependent functions. Given $f : (a : A) \to B(a)$ and $x, y : A$ we can define $\mathsf{ap\text{-}d}_f : (p : x = y) \to f(x) =_p^{B \circ f} f(y)$. Just like regular ap, it is defined by a simple application of the J-rule.

Over and out With these definitions, the reader should, at least in principle, have all they need to go out and do type theory. The system we have defined here can be used to provide a foundation of constructive mathematics. If we want more power, we may need to spice things up with a few extra axioms and/or constructions. For instance, we can add choice/LEM to get classical mathematics, but there are also other alternatives – one of the hottest ones will be covered in the following section. Before we get there, let us summarise the constructions we have seen so far and what they (roughly) translate to in set theory and logic. If you know a bit of type theory, you may have seen it 100 times before and, if you know even a bit more than that, you may have typed it up equally many times: I present to you, Table 1.

2 Homotopy type theory

If the reader feels confused after reading the above section on identity types, they need not worry: identity types in MLTT continue to confuse (and intrigue) even the most seasoned logicians today, years after their invention. In traditional mathematics, as pointed out earlier, identity proofs are meta-mathematical objects which we never have to manipulate directly (proof theorists may ignore this sentence). By allowing proofs to live in the same world as 'ordinary' mathematical objects (such as sets, numbers, etc.), we run the risk of these bringing with them strange artefacts. For instance, let us consider a simple task like defining the pre-image of a map $f : A \to B$ over some element $b : B$. In type theory, we could define it to be the dependent sum $f^{-1}(b) := (a : A) \times (f(a) = b)$. We would like to treat the elements of this type as (a 'subset' of the) elements of A, but we cannot immediately do so: its elements are pairs (a, p) where $a : A$ and $p : f(a) = b$. Suppose we have two different proofs $p, q : f(a) = b$ – which of the pairs (a, p) and (a, q) should we use to represent the element a? This definition really only makes sense if we can prove that $(a, p) = (a, q)$. Now, we could certainly try: the first components are equal and we prove this by refl_a. We now need to show that $p = q$ – how do we do that? We would need the following principle to hold for the type B.

Definition 8 (UIP)**.** *A type A satisfies the principle of uniqueness of identity proofs (UIP) if for any $x, y : A$ and $p, q : x = y$, we have that $p = q$.*

When we speak of the *general UIP*, we will mean the principle that UIP holds for all types. If the general UIP were to hold, our definition of 'subtypes' such as $f^{-1}(b)$ would be unproblematic. Thus, the general UIP appears to be crucial to the development of ordinary mathematics and, for a long time, type theorists were stuck trying to figure out its status [Str93; Coq92]. In the 'standard model' of type theory – the set model (see e.g. [Hof97]) – where types are interpreted as sets, the general UIP certainly holds. However...

Theorem 9 (Hoffman and Streicher [HS98])**.** *The general UIP is independent of type theory.*

The proof of Theorem 9 proceeds by considering a model of type theory in *groupoids* rather than sets, i.e. in categories where every morphism is invertible. The reader who is not impressed by category theory may equivalently think of groupoids as groups with a *partially defined* multiplication. The intuition behind this model is, *very* simplified, that elements x and y of a type can be taken to correspond to objects $[\![x]\!]$ and $[\![y]\!]$ of a suitable category, while an identity $p : x = y$ is interpreted as a morphism $[\![p]\!] \in \mathsf{Hom}([\![x]\!], [\![y]\!])$. In this model, these Hom-sets are non-trivial, thereby refuting UIP. So, we seem to be forced to make a decision: we either restrict the models of our type theory by adding the general UIP as an axiom or we simply bite the bullet and accept that 'subtypes' like the pre-image of a function are out of reach.

Well, if I could decide... It turns out that we actually are able to get quite far without the general UIP – the principle still holds for many types of interest, namely those with *decidable equality*. Let us spell out what this means. We say that a type A is *decidable* if there is an element of type $A + \neg A$ where '$+$' denotes the coproduct (roughly 'disjoint union') and $\neg A := A \to \bot$. The fact that this statement does not hold for all types may surprise you but, remember, type theory is a constructive language and this statement is essentially the law of excluded middle. We say that a type A has *decidable equality* if for any $x, y : A$, the identity type $x = y$ is decidable.

Theorem 10 (Hedberg [Hed98])**.** *UIP holds for any type with decidable equality.*

Fortunately for us, many of the types that we are interested in have decidable equality: the empty type, the unit type, the natural numbers, the rationals, and so on. As type theorists, we often think of types satisfying UIP as sets (as opposed to thinking of all types as such). These types are, just like sets, 'discrete' – more on this soon. Nonetheless, the fact that many types can be shown to satisfy UIP means that our type theory is still able to capture much of classical reasoning.

So, how are we to think of types which do not satisfy UIP? Sure, we can think of them as groupoids according to the groupoid model, but this does not quite capture the whole story: while the groupoid model captures the independence of UIP, it validates 'higher' instances of UIP. That is, it validates, for instance, the statement that for all $x, y : A$, the identity type $x = y$ satisfies UIP. This is, however, not provable either: for a counter model, simply consider 2-groupoids instead of 1-groupoids. We can continue to play this game ad infinitum, so let us kill ω birds with one stone and simply say that all higher instances of general UIP fail in the model of ∞-*groupoids* [BG11]. These are ∞-categories where all morphisms, including the higher ones, are invertible. For the uninitiated, ∞-groupoids are widely used in modern algebraic topology and homotopy theory as models of homotopy types (i.e. spaces up to weak homotopy equivalence). We will often simply refer to them as *spaces*. This interpretation of type theory is rather remarkable – just by giving up UIP, we obtain a type theory with models not only in *sets* but in this far richer class of mathematical objects. The fact that UIP holds for those types which we are used to thinking of as sets makes sense in this setting: here, sets are nothing but discrete spaces. In fact, it is standard terminology in (homotopy) type theory to refer to types satisfying UIP as *sets*.

2.1 Reinterpreting types as spaces

HoTT is obtained from type theory by adding so-called higher inductive types [LS20] and univalence [Voe10] to type theory. These concepts are far more intuitive if we have a good grasp of how type theory is interpreted in spaces, so let us devote some time to explaining this. Let us change the order

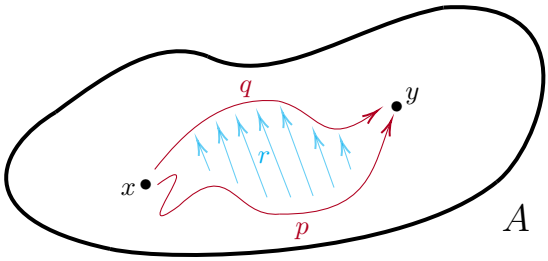


Figure 1: The type A as a space with points, paths and a homotopy

of presentation from §1.2 and start with plain types, function types and identity types. Dependent types and their sums and functions will be introduced after this. In order to understand these, we will need to use a bit more homotopy-theoretic jargon, so we will save them for last in order to avoid scaring anyone away immediately.

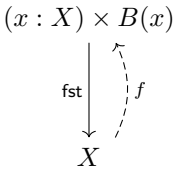
Types, functions and identity types When modelling types as spaces, they are not plain topological spaces but rather (and this is somewhat simplified) equivalence classes of these. Nevertheless, for intuition, we will think of each type $X : \mathcal{U}$ as a concrete (nice) topological space. With this interpretation, an element of type $x : X$ corresponds to a point in the space X. From now on, we will often call elements of types *points*. A function $f : X \to Y$ should be understood as a continuous function between spaces. In other words, *all functions are continuous* (this is not magic – you are simply working in a category where these are all the maps that exist).

The interpretation of identity types is where things really get interesting. Given two points $x, y : A$, an identity proof $p : x = y$ corresponds to a (continuous) path from x to y in A. The reason this idea is so powerful is that we can iterate it in order to provide an interpretation of 'higher' identity proofs: these are simply 'higher paths'. That is, if we interpret two identity proofs $p, q : x = y$ as paths, then an identity proof $r : p = q$ is interpreted as a 'path of paths', i.e. a homotopy of paths. This is illustrated in Figure 1. Naturally, we can keep going to get an interpretation of higher paths as higher homotopies and, suddenly, our type theory has turned into homotopy theory. By function extensionality, this interpretation also means that an equality $f = g$ really is a homotopy of functions.

When working in HoTT, the idea of identity proofs as paths is so rooted that it has taken careful proof reading to eliminate all premature instances of the word 'path' in the preceding sections. Finally, I can relax: from this point on, we will consistently refer to identity proofs as *paths*. We can now interpret the logical laws of the identity paths proved in §1.2 as path operations. The law of reflexivity, which is witnessed by $\mathsf{refl}_x : x = x$, is simply to be interpreted as

the constant path never leaving x. The law of symmetry, i.e. Lemma 2, is path reversal. Given a path $p : x = y$, we will write $p^{-1} : y = x$ for its inverse path. Finally, the law of transitivity, i.e. Lemma 3, is simply path composition. Given paths $p : x = y$ and $q : y = z$, we will write $p \cdot q : x = z$ for their composition. We can also interpret the J-rule in terms of paths, but let us first revisit the remaining concepts from type theory.

Dependent types and their functions and sums If types are spaces, one may reasonably think of dependent types as parameterised spaces or *fibrations*. Let $X : \mathcal{U}$ be a type and let $B : X \to \mathcal{U}$ be a dependent type. If we interpret B as a fibration, we may interpret $(x : X) \times B(x)$ as its *total space*. This interpretation is justified: $(x : X) \times B(x)$ consists of pairs of points $x : X$ in the base space of the fibration together with a point in the fibre $B(x)$. On a similar note, we interpret $(x : X) \to B(x)$ as the *space of sections* of the fibration. The reason for this interpretation is that a dependent function $f : (x : X) \to B(x)$ equally well may be described as a function $f : (x : X) \to (x : X) \times B(x)$ such that, for all $x : X$, we have that $\mathsf{fst}(f(x)) = x$ as illustrated in the following perhaps familiar diagram.



Path induction One of the perhaps most mysterious rules for newcomers to type theory is the J-rule (Definition 1). The principle tells us that sections of fibrations $B : (x : X) \times (x_0 = x) \to \mathcal{U}$ are uniquely determined by a choice of point in the fibre $B(x_0, \mathsf{refl}_{x_0})$. This principle can be generalised. To this end, say that a type X is *contractible* if it is pointed, i.e. inhabited by some $\star_X : X$, and equipped with a section $(x : X) \to \star_X = x$, i.e a proof that $\star_X$ is the unique point. Now, you hopefully find it reasonable that for an arbitrary contractible type X and fibration $B : X \to \mathcal{U}$, a section $(x : X) \to B(x)$ is uniquely described by the choice of a point in $B(\star_X)$. For any type X and $x_0 : X$, the total space $(x : X) \times (x_0 = x)$ should be contractible, and thus we have reduced the J-rule, or *path induction* as we will now start calling it, to a slogan everyone remembers from school: *the total space of the path space fibration is contractible*. This is rather remarkable – even though the original formulation of J was grounded purely in logico-mathematical considerations, it turns out to perfectly match one of the founding principles of homotopy theory. This concludes the homotopical interpretation of type theory. Let us

summarise, in Table 2, our new interpretation of types as spaces in an updated version of *the table*.

Type theory	Homotopy theory	Set theory	Logic
$A : \mathcal{U}$	A is a space	A is a set	A is a proposition
$x : A$	x is a point in A	$x \in A$	x is a proof of A
$A \to B$	$A \to B$ (cont.)	$A \to B$	A implies B
$B : A \to \mathcal{U}$	B is a fibration	B_a is a set family	B is a predicate
$(a : A) \to B(a)$	Sections of B	$\prod_{a \in A} B_a$	$\forall\, a\,.\, B(a)$
$(a : A) \times B(a)$	Total space of B	$\bigsqcup_{a \in A} B_a$	$\exists\, a\,.\, B(a)$
$x = y$	Path space $P(x, y)$	$x = y$	$x = y$

Table 2: Interpretations of type theory, V2

2.2 Getting HoTT

Although we have provided an interpretation of types as spaces, we have not quite described how spaces are manifested within type theory. That is, we have not really described how to *do homotopy theory* in type theory. The truth is that we are not quite there yet – we will need some additional machinery in order to internalise notions from the homotopical model. This additional machinery is what extends our type theory to make it *homotopy* type theory (or, recall, HoTT). We will add two things to our type theory: higher inductive types and univalence.

Higher inductive types One of the key ways of introducing new types is by means of *inductive types*. However, as pointed out earlier, types formed in this way often (but not always, as we know is the case for identity types) correspond to sets. As inductive types are defined in terms of their points, the homotopical model will often see them as discrete spaces. Nevertheless, if we wish to be able to do homotopy theory in our type theory, we better be able to capture other spaces than discrete ones – at the very least, we would need to be able to capture basic cell complexes. To this end, we introduce *higher inductive types* (HITs) [LS20]. HITs are obtained by letting inductive types allow not only for point constructors (such as $0 : \mathbb{N}$ and $\mathsf{suc} : \mathbb{N} \to \mathbb{N}$) but also for *path constructors* which, as the name suggests, allow us to specify paths between, for instance, the points introduced by the point constructors. The canonical example of a HIT is probably the circle, $\mathbb{S}^1$, so let us define it. We define it to be the (higher) inductive type with the following constructors.

- $\mathsf{base} : \mathbb{S}^1$
- $\mathsf{loop} : \mathsf{base} = \mathsf{base}$

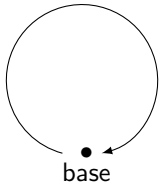


Figure 2: $\mathbb{S}^1$ as a HIT

This definition may look strange at first, but all it says is that $\mathbb{S}^1$ the space generated by one point $\mathsf{base} : \mathbb{S}^1$ and a path (or 1-cell) loop identifying this point with itself. In Figure 2, you will find a picture of the situation. Indeed, this idea corresponds well to the definition of the so-called minimal simplicial circle in classical homotopy theory [Lod92]. However, unlike the traditional minimal simplicial circle which often is rather restrictive (it is not fibrant), the implementation of $\mathbb{S}^1$ as a HIT provides a perfectly workable representation of the circle. The significance of this is that type theory often allows us to work directly with (homotopy types of) spaces in their most naïve form – this is certainly a simplifying feature.

Like plain inductive types, HITs come with recursion and induction principles. Let us start by investigating the recursion principle for $\mathbb{S}^1$. As functions are to be interpreted as continuous functions between spaces, we simply need to understand what data is encoded by a continuous map out of $\mathbb{S}^1$. Viewing $\mathbb{S}^1$ as a cell complex, the answer to this question is simple: a map $f : \mathbb{S}^1 \to X$ is described by a point $x : X$ (the image of base, the 0-cell) and a loop $p : x = x$ (the image of loop, the 1-cell). That is, given this data, the recursion principle of $\mathbb{S}^1$ tells us that we may define $f : \mathbb{S}^1 \to X$ by

$$\begin{aligned} f(\mathsf{base}) &:= x \\ \mathsf{ap}_f(\mathsf{loop}) &:= p \end{aligned}$$

where ap_f denotes function application – recall Lemma 4. Let us define a concrete function, the degree 2 function on $\mathbb{S}^1$, to showcase the recursion principle. In this case, the recursion principle simply states that the following definition is valid.

$$\begin{aligned} 2 \cdot (-) &: \mathbb{S}^1 \to \mathbb{S}^1 \\ 2 \cdot \mathsf{base} &:= \mathsf{base} \\ \mathsf{ap}_{2\cdot(-)}(\mathsf{loop}) &:= \mathsf{loop} \cdot \mathsf{loop} \end{aligned}$$

The induction principle is no different but, to state it, we require the dependent identity type – which will now be called the dependent *path* type – from Definition 6. In order to define a dependent function $g : (x : \mathbb{S}^1) \to B(x)$, we need to provide, just like before, a point $x : B(\mathsf{base})$ and a *dependent* path

$p : x =^{B}_{\mathsf{loop}} x$. We write

$$g(\mathsf{base}) := x$$
$$\mathsf{ap\text{-}d}_g(\mathsf{loop}) := p.$$

Note that the only difference from $\mathbb{S}^1$-recursion is that we have to use the dependent analogue of ap.

HITs can be used to express more than just typical cell complexes. They can, among other things, be used to capture set quotients. For instance, we can define $\mathbb{Z}/2\mathbb{Z}$ to be the HIT generated by

- $[-] : \mathbb{N} \to \mathbb{Z}/2\mathbb{Z}$
- $\mathsf{quot} : (n : \mathbb{N}) \to [n] = [2 + n]$.

In general, this naïve definition of set quotients does not work – it may turn out that the resulting type is no longer a set (i.e. no longer satisfies UIP). To remedy this, we can simply add another (higher) path constructor guaranteeing that the HIT is a set. A way of doing this more generally is by defining an operation called the *set truncation* $\|-\|_0 : \mathcal{U} \to \mathcal{U}$ which takes a type and forces it to be a set. For any A, this type can be defined as the HIT with the following constructors.

- $|-| : A \to \|A\|_0$
- $\mathsf{squash} : (x, y : \|A\|_0) \times (p, q : x = y) \to \mathsf{ap}_{|-|}(p) = \mathsf{ap}_{|-|}(q)$

Notice that this is not just a HIT – it is a HIT with a *higher* and *recursive* path constructor. We allow for such HITs too.

To showcase why this construction is important, consider defining $A/\!\sim$ for some A with an equivalence relation $\sim$ on it. One naïve definition would mimic that of $\mathbb{Z}/2\mathbb{Z}$: define the type $\widehat{A/\!\sim}$ to be the HIT with the following generators.

- $[-] : A \to \widehat{A/\!\sim}$
- $\mathsf{quot} : (x, y : A) \times (x \sim y) \to [x] = [y]$

In general, this type does not capture $A/\!\sim$. Consider, for instance, when A is the trivial type and $\sim$ is the trivial relation. In this case, the above type really captures $\mathbb{S}^1$ (their constructors are in one-to-one correspondence), which is not a set. For this reason, the appropriate definition is $A/\sim := \|\widehat{A/\sim}\|_0$.

The truncation HIT can be generalised to an n-truncation $\|-\|_n : \mathcal{U} \to \mathcal{U}$ for $n \geq 0$, but let us leave the introduction of these to the papers included in this thesis. In fact, there is also a lower level of truncation: the (-1)-truncation or *propositional* truncation. We have so far seen how HITs can be

used to internalise both homotopy-theoretic and set-theoretic notions. What about notions from the last entry in Table 2, i.e. logic? For the purpose of internalising logic, the propositional truncation is crucial. It is defined just like set truncation but with the path constructor in one dimension lower. For any type A, we define $\|A\|_{-1}$ to be the HIT generated by the following constructors.

- $|-| : A \to \|A\|_{-1}$
- $\mathsf{squash} : (x, y : \|A\|_{-1}) \to x = y$

The idea is that $\|A\|_{-1}$ is an exact copy of A but where all (possibly zero) elements have been identified – it is a *proposition*, to use HoTT terminology. This is useful when internalising e.g. logical quantifiers. Recall that given a predicate $B : A \to \mathcal{U}$, we can encode existential quantification by letting it be the dependent sum $(a : A) \times B(a)$. As mentioned earlier, this encodes a very strong constructive notion of existence which allows us to have several proofs (witnesses) of the same existentially quantified statement. In practice, this notion of existence is often too strong, and if we just wish to do elementary logic, we need to identify these different proofs. This is precisely the job done by the propositional truncation: it allows us to define an appropriate internal notion of (logical) existence by $\exists\,(a : A) \times B(a) := \|(a : A) \times B(a)\|_{-1}$.

In conclusion, HITs have many important roles when encoding 'standard' mathematics into type theory. They have, however, not yet quite reached their full potential. In particular, we are lacking tools for defining fibrations over them. Suppose, for instance, that we were to define some interesting fibration $B : \mathbb{S}^1 \to \mathcal{U}$ (say, the one encoding the universal covering of $\mathbb{S}^1$). The recursion principle for $\mathbb{S}^1$ tells us that this boils down to providing two pieces of data: a type X and a loop $p : X = X$. While we do always have a loop $\mathsf{refl}_X : X = X$, setting $p := \mathsf{refl}_X$ amounts to saying that B is constant. If we wish to construct more interesting fibrations than constant ones, we need new ways to construct paths between types. This is one of many applications of the univalence axiom.

Univalence One feature of our homotopical model of type theory is that its objects really are spaces up to homotopy. In other words, we have identified any two spaces X and Y which are (weakly) homotopy equivalent. This is something that is not quite reflected in our type theory. To remedy this, we may add Voevodsky's *univalence axiom* [Voe10], which says, somewhat simplified, that any two equivalent types are equal.

In order to internalise this, we first need to define a suitable notion of equivalence of types. Concretely, what we need to do is to define a predicate $\mathsf{isEquiv} : (X \to Y) \to \mathcal{U}$. One obvious attempt at a definition would be to say that a map $f : X \to Y$ is an equivalence if we can find a map $g : Y \to X$ such that it and f cancel out. Such a map g is called a *quasi-inverse* (following [UF13, §4.1]) and we can define the type of quasi-inverses of f as follows.

$$\mathsf{qinv}(f) := (g : Y \to X) \times ((x : X) \to g(f(x)) = x) \times ((y : Y) \to f(g(y)) = y)$$

For most practical purposes, quasi-inverses will be how we think of equivalences in HoTT. However, it turns out that setting $\mathsf{isEquiv}(f) := \mathsf{qinv}(f)$ does not quite make for an appropriate definition. The issue is that the type $\mathsf{qinv}(f)$ is not a proposition (i.e. it could contain two or more distinct elements). Even though the inverse function g promised by $\mathsf{qinv}(f)$ is unique, the additional pieces of data can be filled out in distinct ways. That is, we can provide different homotopies witnessing the cancellations of f and g. One option would, of course, be to simply use the propositional truncation from the previous section, setting $\mathsf{isEquiv}(f) := \|\mathsf{qinv}(f)\|_{-1}$. While this definition *will* turn out to be equivalent to the standard one, we will go with a more standard definition (which does not require HITs). In what follows, we define the fibre of a map $f : X \to Y$ over $y : Y$ by $\mathsf{fib}_f(y) := (x : X) \times (f(x) = y)$. Recall that a type X is contractible if it is an inhabited proposition, i.e there is some $x_0 : X$ together with a section $(x : X) \to x_0 = x$. It turns out that the property of being a proposition is closed under Π-types (sections) and that $\mathsf{isContr}(X)$ always is a proposition. This gives for a rather ingenious type-theoretic definition of equivalences due to Voevodsky [Voe10].

Definition 11 (Equivalences)**.** *We say that a map $f : X \to Y$ is an equivalence if for every $y : Y$, we have that $\mathsf{fib}_f(y)$ is contractible. That is, we define*

$$\mathsf{isEquiv}(f) := (y : Y) \to \mathsf{isContr}(\mathsf{fib}_f(y)).$$

We write $X \simeq Y$ for the type of equivalences, i.e. the pair of points (f, e_f) where $f : X \to Y$ and $e_f : \mathsf{isEquiv}(f)$. We often simply write $f : X \simeq Y$ and take e_f implicit.

In practice, we almost always treat equivalences in terms of quasi-invertible functions: we have a bi-implication $\mathsf{qinv}(f) \leftrightarrow \mathsf{isEquiv}(f)$ (consult [UF13, §4] for a more in-depth discussion). Definition 11 is, however, often crucial, as many proofs make use of the fact that $\mathsf{isEquiv}(f)$ is a proposition.

At this point, a good guess for what the formal statement of the univalence axiom could be is that it simply postulates a function $\mathsf{ua} : X \simeq Y \to X = Y$. This is almost it – ua will certainly be postulated into existence, but we need to know a little bit more about it than just the fact that it exists. Concretely, we would like to know what its inverse looks like. For this reason, the univalence axiom is introduced the other way around. This is done by first noting that, for all types X and Y, there is a map $\mathsf{coe} : X = Y \to X \simeq Y$ defined by $\mathsf{coe}(p) := \mathsf{transport}^{X \simeq (-)}(p, \mathsf{idEquiv}_X)$ (or, if you prefer to define it directly by path induction on p, it is simply given by $\mathsf{coe}(\mathsf{refl}_X) := \mathsf{idEquiv}_X$). Here, $\mathsf{idEquiv}_X$ denotes the identity $X \simeq X$.

Definition 12 (The univalence axiom)**.** *The univalence axiom is the statement that for any two types X and Y, the function $\mathsf{coe} : X = Y \to X \simeq Y$ is itself an equivalence (with an inverse called ua).*

The exact statement of univalence will not be too important for us here – the existence of $\mathsf{ua} : X \simeq Y \to X = Y$ is the core idea. We obtain HoTT by

adding, along with HITs, the univalence axiom to type theory. From now on, this will be the setting we work in.

One of the key features we obtain from the univalence axiom is the ability to transport proofs and constructions between equivalent structures. Whenever we have two types X and Y s.t. $X \simeq Y$, *any* predicate $B : \mathcal{U} \to \mathcal{U}$ which holds for X will also hold for Y (and vice versa). This is both a blessing and a curse: a corollary of this principle is that we only are able to define predicates which are homotopy invariant. This prevents us from defining several constructions in the arsenal of the working algebraic topologist. For instance, we can never hope to capture exactly what it means for a type to be a manifold[2]: if we had a predicate $\mathsf{isManifold} : \mathcal{U} \to \mathcal{U}$ expressing that an arbitrary type is a manifold, univalence would tell us that whenever $X \simeq Y$, we have an implication $\mathsf{isManifold}(X) \to \mathsf{isManifold}(Y)$ which certainly looks strange – this property should not be stable under weak homotopy equivalence.

As mentioned briefly earlier, univalence is crucial for the construction of non-trivial fibrations over HITs. Let us return to the example of defining a non-trivial fibration $B : \mathbb{S}^1 \to \mathcal{U}$. Like before, the data we need to provide will consist of a space X and an identification $p : X = X$. Unlike before, however, we now have other options than simply setting $p := \mathsf{refl}_X$. Instead, we can invoke univalence, which will allow us to replace the demand of a path $p : X = X$ with the demand of an equivalence $e : X \simeq X$ (so that we simply can set $p := \mathsf{ua}(e)$). Let us consider a concrete example due to [LS13] for $X = \mathbb{Z}$. In this case, we can set e to be the equivalence $1 + (-) : \mathbb{Z} \simeq \mathbb{Z}$. That is, we can define $B : \mathbb{S}^1 \to \mathcal{U}$ by

$$\begin{aligned} B(\mathsf{base}) &:= X \\ \mathsf{ap}_B(\mathsf{loop}) &:= \mathsf{ua}(1 + (-)). \end{aligned}$$

This fibration is non-trivial. To see this, note that we get a map $\mathsf{transport}^B(\mathsf{loop}, -) : B(\mathsf{base}) \to B(\mathsf{base})$. If our fibration were constant, it would be equal to the identity. For our fibration, however, this is not the case. We choose a point a in $B(\mathsf{base})$ – i.e. in $\mathbb{Z}$ – and compute

$$\mathsf{transport}^B(\mathsf{loop}, a) = \mathsf{coe}(\mathsf{ua}(1 + (-)))(a) = 1 + a. \tag{1}$$

Above, the first equality is a simple manipulation of transports (using that coe was also described as a transport) and the second equality holds by definition of the univalence axiom. You may recognise the fibration constructed from the classical construction of the universal covering of $\mathbb{S}^1$: each fibre is $\mathbb{Z}$ and the canonical loop lifts to addition by 1. We can proceed to show that its total space is contractible and obtain that $\Omega(\mathbb{S}^1) \simeq \mathbb{Z}$, where $\Omega(\mathbb{S}^1) := (\mathsf{base} = \mathsf{base})$. Thus, we have proved our first 'real' homotopy-theoretic result in HoTT. This proof, which is originally due to [LS13], is remarkably simple compared to its classical counterpart (when the classical proof is given from scratch, that is).

[2]This is a half-truth: there is work on defining an appropriate notion of a *homotopy* manifold in HoTT by [Buc+24].

We never have to verify the continuity of any of the maps involved, and we never have to worry about the possible existence of pathological loops in $\mathbb{S}^1$.

Univalence has many other important features. Perhaps most importantly, it can be used to *prove* function extensionality [Voe10]. In addition to this, it is, among other things, crucially used in the characterisation of path spaces of truncations and can be used to prove Aczel's *structure identity principle* (SIP) [Acz11] which roughly says that any two isomorphic structures (such as rings, groups, etc.) are equal. The SIP implies that any predicate that we can form in HoTT about, say, groups automatically is invariant under isomorphism, thereby making formal an informal principle employed by mathematicians on a daily basis. There is, of course, a lot more that can be said about the SIP and other applications of univalence, but let us be content with this so far – the reader who also reads the five papers included in this thesis should come across plenty of other applications of univalence.

2.3 Cubical type theory

One of the neat things about type theory is its close connection to *computation*. In general, we expect any type-theoretic construction to compute. As we can interpret types as program specifications, we should always be able to extract a concrete algorithm from any type-theoretic construction we make. For instance, if we construct a function $f : \mathbb{Z} \to \mathbb{Z}$ in type theory, we should be able to implement it as a computer program. If we wish to know what, say, $f(0)$ computes to, we should never have to find this out 'manually' (by mathematical proof); we should be able to simply run the program we constructed. We say that all of our constructions have *computational content*.

One issue with HoTT is that we have simply postulated the univalence axiom. What we never did, however, was to explain its computational content. This means that univalence becomes a computational black box – if our function f is described using univalence, we can no longer hope to be able to automatically extract an algorithm from it. One example is Equation (1). If we were only interested in the case when a is a fixed number, say $a := 0$, we should not have to do any work: the left-hand side should *compute* to 1, so the proof should be given by refl_1. However, as the left-hand side is described in terms of univalence, this does not work – the computation gets stuck.

It is still open whether 'standard' HoTT (systems like the one described here) can be equipped with a computational univalence axiom. One thing we can do, however, is to alter the type theory slightly. The resulting type theory, a *cubical type theory* (CuTT), will have slightly less general models (in its natural model, spaces are modelled as cubical sets rather than what we have so far called 'spaces' [BCH14; Coh+18; Ang+19]) but will have a computational univalence axiom. This flavour of HoTT is also currently better suited for implementing HITs [CHM18; CH19]. This will be the type theory all computer formalisations in this thesis are built on (more on this in §3). Nevertheless, most

of the thesis is written in an informal style which is agnostic with respect to flavour of HoTT. One should point out that there are several different versions of CuTT; the one we present here is closest to the one by [Coh+18] and, from now on, 'CuTT' will always mean this particular version.

The key difference between plain HoTT and CuTT is the treatment of the identity type. To get some intuition, let us start by giving an alternative definition of identity types in HoTT. In HoTT, we can define the type I, representing the unit interval, to be the HIT with the following constructors.

- $i_0, i_1 : I$
- $i_{0=1} : i_0 = i_1$

Now, consider a function $f : I \to A$. Such a function encodes precisely two points $f(i_0), f(i_1) : I$ together with a path $\mathsf{ap}_f(i_{0=1}) : f(i_0) = f(i_1)$. Thus, for any points $x, y : A$, a function $f : I \to A$ with $f(i_0) := x$ and $f(i_1) := y$ captures precisely the identity type $x = y$. This is, in spirit, the approach taken when defining the identity type in CuTT: here, the identity type is not the usual inductive one (although it is consistent to add it) but one described in terms of a primitive interval (pre-)type I. This type comes with two points i_0 and i_1 but is not a HIT – that would be rather circular, as the notion of a HIT presupposes a notion of identity. This type lives in a special universe, which essentially makes it untouchable – it is only there to provide us with a definition of paths and is not there to be reasoned about (again, this would lead to circularity). So, a path $x = y$ in CuTT is *literally* a function $f : I \to A$ such that $f(i_0) := x$ and $f(i_1) := y$. For instance, we can define $\mathsf{refl}_x : x = x$ to be the constant path $\mathsf{refl}_x(i) := x$. The interval does come with some primitive operations which are needed to actually make it useful. In particular, it comes with a path reversal operation $\sim : I \to I$ and meet and join operations $\vee, \wedge : I \times I \to I$. These satisfy the following (strict) equalities

$$\begin{array}{rrr} \sim i_0 := i_1 & i_0 \wedge j := i_0 & i_0 \vee j := j \\ \sim i_1 := i_0 & i_1 \wedge j := j & i_1 \vee j := i_1 \\ \sim (\sim i) := i & i \wedge j := j \wedge i & i \vee j := j \vee i \end{array}$$

as well as the appropriate distribution and associativity laws turning I into a de Morgan algebra. For instance, $\sim$ is what gives rise to symmetry: for a path $p : x = y$, we can define its inverse path by $p^{-1}(i) := p(\sim i)$. Another example is the principle of path induction, which is a theorem in CuTT (rather than a built-in rule). This is proved by showing that the total space of the path fibration is contractible, which can be done by providing paths defined in terms of $\wedge$. For full transparency, we should mention that the proof of path induction also uses a $\mathsf{transport}$ – an operation which is in fact another primitive of CuTT. Nevertheless, this is more detail than we will ever need. The fact that we have path induction means that we, for all intents and purposes, can treat CuTT

as any other implementation of HoTT and use the same informal language we have used so far.

One remarkable thing about CuTT is that it makes function extensionality trivial. Indeed, a proof that two functions $f, g : A \to B$ are equal boils down to providing a path $p : I \to (A \to B)$ s.t. $p(i_0) := f$ and $p(i_1) := g$. If we uncurry this, it is equivalent to saying that we need to provide a function $p : I \times A \to B$ s.t. $p(i_0, x) := f(x)$ and $p(i_1, x) := g(x)$. These two ways of viewing the path type $f = g$ amounts precisely to function extensionality.

There are more primitives than the ones introduced so far. For instance, there is the notorious Glue type which is used to (constructively) *prove* univalence, thereby endowing it with computational content. We will never explicitly rely on this construction either.

Two things that do become simpler in CuTT are dependent paths and HITs. When defining dependent identity in §1.2, we pointed out that plain paths really are special cases of their dependent analogue; yet, we used non-dependent identity to define dependent identity. In CuTT, dependent paths are simple. Let $B : I \to \mathcal{U}$ be any 'line' of types (i.e. a path of types $B(i_0) = B(i_1)$ in the universe $\mathcal{U}$) and let $x : B(i_0)$ and $y : B(i_1)$. A dependent path from x to y over B is simply a dependent function $p : (i : I) \to B(i)$ s.t. $p(i_0) := x$ and $p(i_1) := y$ hold strictly. In the special case when B is constant, such a (no longer dependent) function p is a regular path. Removing the need for $\mathsf{transport}$s in the definition of dependent paths often strips away a fair bit of bureaucracy and makes dependent paths very natural. For instance, the identity type of a dependent sum $(x : X) \times B(x)$ is easy to understand: proving an equality $(x, b_x) = (y, b_y)$ is done by providing a dependent function $p : (i : I) \to (x : X) \times B(x)$ s.t. $p(i_0)$ and $p(i_1)$ reduce to respective pair. In order to define $p(i)$ for $i : I$, we thus need to provide an element $p_1(i) : X$ s.t. $p_1(i_0) := x$ and $p_1(i_1) := y$ as well as an element $p_2(i) : B(p_1(i))$ s.t. $p_2(i_0) := b_x$ and $p_2(i_1) := b_y$. This is exactly the principle described in §1.2, but this time it holds definitionally.

Arguably, HITs also become more natural in CuTT and the only fully fledged computer program implementations of these rely on some form of CuTT (e.g. `Cubical Agda` [VMA21]). In CuTT there is (in theory) no real difference between ordinary inductive types and HITs. As paths are described as functions out of the interval, we can see path constructors appearing in HITs as the special case of constructors taking input from the interval. Naïvely, the HIT defining $\mathbb{S}^1$ could simply be thought of as a plain inductive type with constructors

- $\mathsf{base} : \mathbb{S}^1$
- $\mathsf{loop} : I \to \mathbb{S}^1$

but with the caveat that $\mathsf{loop}(i_0) := \mathsf{base}$ and $\mathsf{loop}(i_1) := \mathsf{base}$. This makes it easier to implement HITs in proof assistants.

3 Proof assistants

One thing that is easy to forget is that, despite all the buzzwords from homotopy theory we have used here, type theory is a programming language. This means that a proof written in type theory can effectively be understood by a computer. Indeed, it is, to some extent, used in the foundations of almost every programming language. Some functional languages, like `Haskell`, are very close to the type theory we have described in §1. On the other hand, if a computer can understand type theory, then it can understand statements and proofs in logic (by the Curry–Howard correspondence). For a proof in type theory to be correct, it simply needs to type check – something computers are very good at verifying.

Concretely, this means that type theory works incredibly well as a language for so-called *proof assistants*. These are essentially programming languages developed for the particular task of checking the correctness of proofs. If we use type theory, 'verifying a proof' simply means type checking it. Hence, a type-theoretic proof assistant is essentially a programming language whose syntax only allows us to write correct code. There are several examples of proof assistants and almost all of them use some form of type theory. This even applies to proof assistants such as `Lean` [Mou+15], despite the fact that it primarily intended for 'traditional' set-theoretic mathematics (another point for type theory as a foundation of mathematics!). In all papers presented in this thesis, we have used the proof assistant `Agda` [Agda], or rather its cousin `Cubical Agda` [VMA21], to verify all key results. We say that these results have been *(computer) formalised* (or, sometimes, *mechanised*). It may seem like I am downplaying this part of the thesis by introducing it only here, in the last paragraphs of the introduction, but this is not the message I want the reader to take away. The computer formalisations in this thesis are as much a contribution as the mathematical results.

3.1 Agda

The computer formalisation of this thesis uses the proof assistant `Cubical Agda` [VMA21] which adds certain features from CuTT to the proof assistant `Agda` [Agda]. Let us start by discussing the latter, as it is the basis of `Cubical Agda`. `Agda` is a type-theoretic language incredibly close to MLTT which, recall, is what we mean by type theory in this thesis. In fact, the reader has already been introduced to some of `Agda`'s syntax without realising: for instance, the notation $(x : X) \to B(x)$ for dependent functions is not traditionally used in type theory, but in `Agda` it is. To give a very brief overview of how the language works, let us go through how the basic concepts from §1.2 are implemented in `Agda`.

Universes Agda is equipped with a hierarchy of universes, usually written Set ℓ where ℓ is a universe level (i.e. an index). Here, we will rename Set to Type and simply omit the index, with the convention that Type without a universe level denotes the lowest universe Type ℓ-zero. If we wish to define a new type, the syntax is as follows.

```
myType : Type
myType = someDefinition
```

Functions Functions are introduced pretty much exactly like we would write them using pen and paper. For instance (presupposing a definition of the natural numbers and a squaring operation), here is how we can define the function $x \mapsto x^2 + x$:

```
myFun : ℕ → ℕ
myFun x = x ² + x
```

We can also define the same function using lambda notation:

```
myFunLambda : ℕ → ℕ
myFunLambda = λ x → x ² + x
```

These two definitions will be equal *by definition* in Agda. Another way of introducing functions is by pattern matching, but for that we need to see how inductive types are dealt with.

Inductive types, dependent types and dependent functions and sums All the inductive types from §1.2 can easily be captured using Agda's primitive data syntax. The syntax is almost identical to our presentation.

```
data ℕ : Type where
  zero : ℕ
  suc : ℕ → ℕ
```

```
data Bool : Type where
  true : Bool
  false : Bool
```

```
data 𝟙 : Type where
  * : 𝟙
```

```
data ⊥ : Type where
```

We clarify that the last type, $\bot$, is the data type with no constructors, i.e. it is the empty type. The types above all come automatically equipped with their recursion/induction principles – Agda supports *pattern matching* (compatible with HoTT [CDP14]) which is its way of capturing these notions. To showcase how this works, while simultaneously showing how to define dependent types, let us define the isEven-predicate from §1.2.

```
isEven : ℕ → Type
isEven zero = 𝟙
```

```
isEven (suc zero) = ⊥
isEven (suc (suc n)) = isEven n
```

Above, we have pattern matched on a variable m of type $\mathbb{N}$ and `Agda` thereby introduces a case-split: either $m := 0$ or $m := \mathsf{suc}(n)$ for some $n : \mathbb{N}$. We can now carry out our first proof in `Agda` – the proof that $2 + n$ is even if n is even.

```
isEven2+ : (n : ℕ) → isEven n → isEven (suc (suc n))
isEven2+ n p = p
```

This is an instance of a dependent function. For completeness, let us also prove the riveting theorem 'there exists an even number' to showcase the implementation of the dependent sum type. We provide 2 (which, in `Agda`, is short for suc (suc zero)) as a witness.

```
existsEven : Σ[ n ∈ ℕ ] (isEven n)
existsEven = 2 , *
```

Identity types We can easily define Martin-Löf's inductive identity type in `Agda`:

```
data _≡_ (x : A) : A → Type where
  refl : x ≡ x
```

We use the triple bar notation to avoid clashing with '=' which already is used in `Agda`'s syntax. Just like the other inductive types we have implemented, the identity type automatically comes equipped with its induction principle, which again can be used by pattern matching (i.e. by asking `Agda` to 'case split' on a variable of type $x \equiv y$). Here are all of the elementary definitions concerning identity types from §1.2:

```
_⁻¹ : x ≡ y → y ≡ x
refl ⁻¹ = refl
```

```
_·_ : x ≡ y → y ≡ z → x ≡ z
refl · q = q
```

```
ap : (f : A → B) → x ≡ y → f x ≡ f y
ap f refl = refl
```

```
transport : (B : A → Type) (p : x ≡ y) → B x → B y
transport B refl x = x
```

Despite how confusing the J-rule (path induction) may be, the above definition should showcase how easy it is to use in `Agda`. As discussed, the J-rule does not work whenever the end-point is not free. A special case of this proscription was presented as Non-Theorem 1. We could attempt to prove it in `Agda` by

```
haligh-a-Lie : (p : x ≡ x) → p ≡ refl
haligh-a-Lie refl = refl
```

but, fortunately, `Agda` will complain:

```
I'm not sure if there should be a case for the constructor refl,
because I get stuck when trying to solve the following unification
problems (inferred index ≟ expected index):
  x₁ ≟ x₁
```

Computation/normalisation When we prove in Agda, we often rely on constructions computing/normalising. For instance, if we were interested in showing that for any path $p : x \equiv y$, we have that $p \cdot p^{-1} = \mathsf{refl}_x$, we would prove this as follows (of course, using J).

```
rCancel : (p : x ≡ y) → p ∙ p ⁻¹ ≡ refl
rCancel refl = refl
```

The reason this proof works is that after pattern matching on p, Agda will rewrite the target type to $\mathsf{refl} \cdot \mathsf{refl}^{-1} \equiv \mathsf{refl}$ – a type which Agda now will start simplifying or *normalising*. Agda will see that we *defined* refl^{-1} to be refl and, in turn, $\mathsf{refl} \cdot \mathsf{refl}$ to again be refl. Hence, by computing all the functions involved, Agda simplifies to the goal to make it $\mathsf{refl} \equiv \mathsf{refl}$, which we can prove by refl.

There are also ways in which Agda's computational features can provide less fundamental aid in our theorem proving. Suppose, for instance, that we are interested in computing some function – call it $\mathsf{ackermann} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ – at some concrete input. Say we wish and we wish to prove that this output is equal to some other number – call it $2 \uparrow^3 4 - 3$. One way of doing this is by some form of traditional 'manual' proof: the way we do it in classical mathematics. Another way would be to simply compute the function (this works if it is constructively defined). Agda's computational feature allows for this proof approach.

```
trustMe : ackermann (5 , 1) ≡ 2 ↑³ 4 - 3
trustMe = refl
```

The above proof (when it finishes type checking) is a verified computation or a 'proof by computation'. This is a powerful proof technique available in constructive proof assistants like Agda which will be explored in some of the papers in this thesis.

3.2 Cubical Agda

You may have noticed that the constructions covered in our brief introduction to Agda all were constructions from plain type theory. However, we did not say anything about the two key notions from HoTT: HITs and univalence. This is because they do not have native support in plain Agda.[3] In Cubical Agda, however, they are supported – this is my reason for having chosen to carry out my formalisations in this language. This is not to say that HITs and univalence (and HoTT in general) cannot be meaningfully reasoned about in vanilla Agda – some of the largest libraries of computer formalised univalent

[3]There are however still ways of implementing HITs, e.g. using 'Licata's trick' [Lic11].

mathematics use this version, e.g `HoTT-Agda` [Bru+18], `agda-unimath` [Rij+] and `TypeTopology` [Ec]. In plain `Agda`, we simply postulate these concepts into existence. While this is perfectly sufficient for formalising results from HoTT, it does come with the caveat that it breaks computation. My choice of `Cubical Agda` is not ideological – I simply appreciate its computational aspects. Let us showcase the new features. We will be brief as this topic is also covered in Paper II.

Paths Although we can still define the usual inductive path type à la Martin-Löf in `Cubical Agda`, the primary way of capturing paths in this proof assistant is via the interval (pre-)type described in §2.3. We keep using the same notation for path types but note that, in the following pages, they are defined cubically. We can mimic the construction of the constant path and path reversal by copying the definitions in §2.3 pretty much verbatim:

```
refl : (x : A) → x ≡ x
refl x i = x
```

```
_⁻¹ : x ≡ y → y ≡ x
(p ⁻¹) i = p (~ i)
```

In addition to the above, we can also prove function extensionality in 1 line.

```
funExt : {f g : (x : A) → B x} → ((x : A) → f x ≡ g x) → f ≡ g
funExt p i x = p x i
```

Above, the curly brackets indicate implicit arguments. It would be natural to ask what happened to the definition of path composition; we do not talk about this (it uses a primitive called `hcomp` which takes some time getting used to and is never used explicitly in this thesis).

HITs We can define HITs in `Cubical Agda` just like we defined plain inductive types in `Agda`. For instance, here is the definition of $\mathbb{S}^1$.

```
data 𝕊¹ : Type where
  base : 𝕊¹
  loop : base ≡ base
```

We have access to its recursion and induction principles immediately via pattern matching. For instance, we can define the canonical (-1)-degree map by

```
inv𝕊¹ : 𝕊¹ → 𝕊¹
inv𝕊¹ base = base
inv𝕊¹ (loop i) = loop (~ i)
```

and prove that it is an involution

```
inv𝕊¹Involution : (x : 𝕊¹) → inv𝕊¹ (inv𝕊¹ x) ≡ x
inv𝕊¹Involution base = refl
inv𝕊¹Involution (loop i) = refl
```

Univalence Arguably, *the* key feature of `Cubical Agda` is that it natively supports the univalence axiom. Actually, using the word 'axiom' is wrong – here, it is a theorem. Its main component, the function lifting equivalences to paths, can be constructed in `Agda` using the following code

```
ua : A ≃ B → A ≡ B
ua e i = Glue B (λ { (i = i0) → (A , e) ; (i = i1) → (B , idEquiv B) })
```

This definition is, of course, not very illuminating. It uses strange new syntax together with a new (built-in) Glue type. Like most users of `Cubical Agda`, we will simply accept this definition and move on. It is very uncommon that we actually have to understand this definition in order to make use of univalence. Instead, let us showcase this function by defining the fibration giving rise to the universal covering of the $\mathbb{S}^1$ described in §2.2. The fibration, which we here call helix, can be defined by

```
helix : 𝕊¹ → Type
helix base = ℤ
helix (loop i) = ua +1Equiv i
```

The equality proved in Equation (1) with $a := 0$ can now, thanks to the computational content of univalence, be proved simply by refl.

```
helixComputes : transport (λ i → helix (loop i)) 0 ≡ 1
helixComputes = refl
```

Papers I and, in particular, Paper II will discuss statements similar to the above which also can be proved by computation in `Cubical Agda`. This concludes this very brief introduction to `Cubical Agda`.

4 The context and contributions of this thesis

The publication of the HoTT book [UF13] kick-started the large-scale project of developing (both old and new) mathematics in the framework of HoTT, including the formalisation of these results in HoTT-based proof assistants like `(Cubical) Agda`. These results are not just interesting because they are more directly available for computer formalisation – they also generalise their classical counterparts. Indeed, HoTT can be interpreted in any suitably structured ∞-topos [Shu19] and thus provides an immediate generalisation of traditional mathematics with its usual model in (the very particular ∞-topos of) sets.

So far, we have seen quick progress in several traditional domains of mathematics, such as algebraic geometry [ZM23; ZH24; CCH24; Che+24], (higher) group theory [BDR18; Swa22; Bez+25], set theory [Jon+23; GS24; Gra+24], graph theory [PG24] and other subjects. The area within which we have seen probably the most activity, however, is that of (synthetic[4]) homotopy theory.

[4]Here, 'synthetic' means that we are manipulating the homotopy types of spaces directly.

We have already seen an example of how HoTT can be used to reason about homotopy theory in §2.1, in particular when discussing the construction of the universal covering of $\mathbb{S}^1$ (following Licata and Shulman [LS13]). In the last 10 years, we have seen an impressive amount of work dedicated to the study of homotopy theory in HoTT. There have been significant contributions made to e.g. the study of homotopy groups/types of spheres [LS13; Bru16; BR18; Bak23; Cag+24], cohomology [Bru16; BF18; Doo18; Wär23; CF23], homology [Gra18; CS23], path spaces of pushouts [KR21; Wär24], H-spaces [BR18; Buc+23], acyclic spaces [BdR25], and so on.[5] Much of this research is computer formalised in proof assistants such as `Agda`, `Cubical Agda`, `Coq` and `Lean`. My intention with this thesis is to make a contribution to these efforts.

The first three papers of this thesis are dedicated to filling in some of the gaps in the literature by giving a formal account of details missing in some key proofs and constructions in Brunerie's landmark PhD thesis on computing $\pi_4(\mathbb{S}^3)$ in HoTT. In particular, Paper I introduces a complete construction of cohomology rings, a primary contribution being the previously missing proof of the associativity of the cup product, and Paper II presents, relying on the results from Paper I, a complete computer formalisation of Brunerie's proof in `Cubical Agda`. Paper III attacks the closely related problem of proving the symmetric monoidality of smash products, something which Brunerie [Bru16] only sketched in his thesis and left as future work.

The last two papers instead explore new avenues for synthetic homotopy theory in HoTT by addressing open ends left by other papers in the field. Paper IV investigates to which extent it is possible to use Buchholtz and Favonia's definition of cellular cohomology [BF18] to define cellular *homology*. In particular, the paper includes a construction of cellular homology in terms of a constructive version of the cellular approximation theorem. This approach can, completely analogously, be used to obtain Buchholtz and Favonia's cohomology theory, and thus the paper provides a unified approach to cellular homology and cohomology in HoTT. Paper V, on the other hand, consists of a synthetic development of the Steenrod squares in HoTT. The paper contains proofs of the fact that these cohomology operations satisfy a collection of defining axioms (such as the Cartan formula and stability). The proofs of these properties were left open by Brunerie [Bru17], whose definition of the Steenrod squares we use. This paper is perhaps particularly interesting because it heavily relies on concepts which are, to some extent, especially natural in the language of HoTT, such as unordered pairs and joins.

Loose ends and future work There are several research problems suggested or made open by the work in this thesis. This particularly applies to Papers IV and V, but these problems naturally touch upon material from the first three papers. Paper IV was, in fact, intended as a foundation for an ongoing project together with Mörtberg and Pujet on using implementations of cellular

[5] There are of course many more topics studied and papers produced, but I cannot mention them all here.

homology and cohomology in `Cubical Agda` to solve computational problems such as that of computing the Brunerie number in Paper II or the numbers in §6 of Paper I. On the other hand, the proof of the (cellular) Hurewicz theorem (and, in particular, its formalisation) is part of an ongoing project, led by Barton and Milner, on the formalisation of Barton and Campion's [Bar22] proof of the Serre finiteness theorem in `Cubical Agda`.

For Paper V, the continuation is very clear: we would like to be able to define not only the Steenrod squares but the higher Steenrod p-powers for any prime p. This is an interesting problem which really puts the expressivity of HoTT to the test. Indeed, the proofs are already involved in the case of $p = 2$. If we are to generalise these proofs, we need to define p-fold (unordered) smash products or joins. These are constructions we simply do not currently know how to define.

Summaries of included papers

Paper I

Paper I, *Computational Synthetic Cohomology Theory in Homotopy Type Theory*, concerns the development of cohomology and summarises, to a large extent, the first half of my PhD. The paper generalises work by Brunerie et al. [BLM22] (included in my licentiate thesis [Lju23]) and explains the mathematics underpinning the first complete computer formalisation of cohomology rings in HoTT by Lamiaux et al. [LLM23]. Concretely, it extends the definition of the cup product by Brunerie et al. to cohomology with arbitrary ring coefficients – a definition which is new to HoTT and significantly simplifies pre-existing definitions. As a result, we are able to provide a complete construction of cohomology rings in HoTT. This problem was previously open due to the perceived difficulty of proving associativity. We also verify that our cohomology theory satisfies a version of the Eilenberg–Steenrod axioms and construct the Gysin sequence (generalising the construction by Brunerie [Bru16]).

In addition to this, we carry out some concrete computations of cohomology groups and rings. For instance, we compute the cohomology ring of $\mathbb{R}P^\infty$ in $\mathbb{Z}/2\mathbb{Z}$-coefficients – something which is crucially used in Paper V.

We finally discuss some new 'Brunerie numbers', i.e. numbers which are definable in `Cubical Agda` and contain non-trivial information about, in this case, cohomological constructions. We benchmark these in order to better understand the feasibility of 'proof by computation' in `Cubical Agda`. All results in the paper have been formalised.

Paper II

Paper II, *Formalising and Computing the Fourth Homotopy Group of the* 3*-Sphere in Cubical Agda*, is an extended version of a paper by Ljungström and Mörtberg [LM23]. This paper is written in a mock-`Cubical Agda` language and

presents a computer formalisation of Brunerie's 2016 proof of $\pi_4(\mathbb{S}^3) \cong \mathbb{Z}/2\mathbb{Z}$. This formalisation contains results and constructions such as Whitehead products, (a special case of) the James construction, the Hopf invariant, cohomology rings (borrowed from Paper I) and much more. As Brunerie's thesis contained some minor but seemingly difficult gaps, providing a formalisation of his results constituted a, to some, important open problem in HoTT.

Another open problem (partially) solved in this paper is that of computing the Brunerie number. This is a number $n : \mathbb{N}$ s.t. $\pi_4(\mathbb{S}^3) \cong \mathbb{Z}/n\mathbb{Z}$. The formalisation of Brunerie's thesis showed that this number has absolute value 2, but actually extracting the value of n purely by computation in e.g. `Cubical Agda` – the problem of computing the Brunerie number – turns out to be very difficult. All attempts to compute Brunerie's original definition of this number have simply run out of memory. We provide a new and simplified definition of the Brunerie number which we *can* compute. This is done by slightly altering the definition of homotopy groups in terms of joins, which in turn allows us to better understand the interaction between certain maps and Whitehead products (one such product is what gives rise to the original Brunerie number).

Paper III

Paper III, *Symmetric monoidal smash products in homotopy type theory*, addresses another long-standing open problem in HoTT, namely that of verifying that the smash product is (1-coherent) symmetric monoidal. This was previously open due to the difficult coherence problems which appear in the proof of, in particular, the pentagon axiom (see e.g. [Bru18]).

The paper shows how a seemingly unrelated result by Cavallo [Cav21] regarding pointed functions and homogeneous types can be used to automatically infer certain higher coherences when proving equalities of maps defined over smash products. We start by providing an informal heuristic that captures this idea. We then use this heuristic to provide a proof of the fact that smash products are symmetric monoidal and, in particular, satisfy the pentagon axiom. We end the paper with a formal statement intended to capture the heuristic. The statement drastically reduces the amount of data needed when constructing homotopies over iterated smash products and should be useful when working with large smash products in general. All results have been formalised in `Cubical Agda`.

Paper IV

Paper IV, *Cellular Methods in Homotopy Type Theory* concerns the development of the basic theory of CW complexes and cellular homology in HoTT. In this paper, we revisit a definition of cellular cohomology in HoTT by Buchholtz

and Favonia [BF18] and attempt to develop the analogous homology theory instead. In particular, in order to obtain functoriality, we need to prove a suitable version of the cellular approximation theorem. The usual formulation of this theorem is not constructive and, for this reason, a large part of the contribution in this paper is the constructivisation of this classical result. On a similar note, we prove a special case of the CW-approximation theorem which says that the notion of an n-connected finite CW complex coincides with the notion of an n-connected type (whenever this type is a finite CW complex) – another seemingly non-constructive result. This allows us to prove the Hurewicz theorem for our homology theory.

In addition to the above, we prove the Eilenberg–Steenrod axioms for our theory. In order to state them, we also need to provide certain pushouts of cellular maps with a CW structure. Most results have been formalised in `Cubical Agda`.

Paper V

In Paper V, *The Steenrod squares via unordered joins*, we lay out the basic theory of the Steenrod squares in HoTT. The Steenrod squares are a family of cohomology operations in $\mathbb{Z}/2\mathbb{Z}$ cohomology introduced in HoTT by [Bru17]. While Brunerie provided a definition, the question of whether these squares satisfy any of their classical characterising properties has remained open. In this paper, we revisit Brunerie's construction and analyse it in terms of so-called unordered joins. Using these, we are able to prove a certain Fubini-like theorem from which we can deduce several of the Steenrod squares' properties (the Cartan formula and Adem relations, among other things). The constructions we provide are very much native to the language of HoTT and come with both advantages and disadvantages which are discussed at several points in the paper. Key technical results are supported by a formalisation in `Cubical Agda`.

Bibliography

[Acz11] Peter Aczel. On Voevodsky's Univalence Axiom. Talk given at the Third European Set Theory Conference. 2011. URL: https://staff.cs.manchester.ac.uk/~petera/Recent-Slides/Edinburgh-2011-slides_pap.pdf.

[Agda] The Agda Development Team. The Agda Programming Language. 2023. URL: http://wiki.portal.chalmers.se/agda/.

[Ang+19] Carlo Angiuli et al. "Syntax and Models of Cartesian Cubical Type Theory". Preprint. Feb. 2019. URL: https://github.com/dlicata335/cart-cube.

[Bak23] Raymond Baker. "Eckmann-hilton and the Hopf Fibration In Homotopy Type Theory". Honor's thesis. University of Colorado Boulder, Apr. 2023.

[Bar22] Reid Barton. Finite presentability of homotopy groups of spheres. Talk at the Seminar on Homotopy Type Theory at CMU, presenting joint work with Tim Campion. 2022. URL: https://www.cmu.edu/dietrich/philosophy/hott/seminars/previous.html.

[BCH14] Marc Bezem, Thierry Coquand and Simon Huber. "A Model of Type Theory in Cubical Sets". *19th International Conference on Types for Proofs and Programs (TYPES 2013)*. Ed. by Ralph Matthes and Aleksy Schubert. Vol. 26. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 107–128. ISBN: 978-3-939897-72-9. DOI: 10.4230/LIPIcs.TYPES.2013.107.

[BDR18] Ulrik Buchholtz, Floris van Doorn and Egbert Rijke. "Higher Groups in Homotopy Type Theory". *Proceedings of the 33rd Annual ACM/ IEEE Symposium on Logic in Computer Science*. LICS '18. Oxford, United Kingdom: Association for Computing Machinery, 2018, pp. 205–214. ISBN: 9781450355834. DOI: 10.1145/3209108.3209150.

[BdR25] Ulrik Buchholtz, Tom de Jong and Egbert Rijke. Epimorphisms and Acyclic Types In Univalent Foundations. *The Journal of Symbolic Logic* (Feb. 2025), pp. 1–36. ISSN: 1943-5886. DOI: 10.1017/jsl.2024.76.

[Bez+25] Marc Bezem et al. Symmetry. *https://github.com/UniMath/SymmetryBook*. Commit: 8f95b7d. 25th Mar. 2025.

[BF18] Ulrik Buchholtz and Kuen-Bang Hou Favonia. "Cellular Cohomology in Homotopy Type Theory". *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '18. Oxford, United Kingdom: Association for Computing Machinery, 2018, pp. 521–529. ISBN: 9781450355834. DOI: 10.1145/3209108.3209188.

[BG11] Benno van den Berg and Richard Garner. Types are weak ω-groupoids. *Proceedings of the London Mathematical Society* 102.2 (2011), pp. 370–394. DOI: 10.1112/plms/pdq026.

[BLM22] Guillaume Brunerie, Axel Ljungström and Anders Mörtberg. "Synthetic Integral Cohomology in Cubical Agda". *30th EACSL Annual Conference on Computer Science Logic (CSL 2022)*. Ed. by Florin Manea and Alex Simpson. Vol. 216. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 11:1–11:19. ISBN: 978-3-95977-218-1. DOI: 10.4230/LIPIcs.CSL.2022.11.

[BM75] Donald W. Barnes and John M. Mack. "Zermelo-Fraenkel Set Theory". *An Algebraic Introduction to Mathematical Logic*. New York, NY: Springer New York, 1975, pp. 52–61. ISBN: 978-1-4757-4489-7. DOI: 10.1007/978-1-4757-4489-7_6.

[BR18] Ulrik Buchholtz and Egbert Rijke. The Cayley-Dickson Construction in Homotopy Type Theory. *Higher Structures* 2.1 (2018), pp. 30–41.

[Bru+18] Guillaume Brunerie et al. Homotopy Type Theory in Agda. 2018. URL: https://github.com/HoTT/HoTT-Agda.

[Bru16] Guillaume Brunerie. "On the homotopy groups of spheres in homotopy type theory". PhD thesis. Université Nice Sophia Antipolis, 2016. arXiv: 1606.05916.

[Bru17] Guillaume Brunerie. "The Steenrod squares in homotopy type theory". Abstract at *23rd International Conference on Types for Proofs and Programs (TYPES 2017)*. 2017. URL: https://types2017.elte.hu/proc.pdf#page=45.

[Bru18] Guillaume Brunerie. "Computer-generated proofs for the monoidal structure of the smash product". *Homotopy Type Theory Electronic Seminar Talks*. Nov. 2018. URL: https://www.uwo.ca/math/faculty/kapulkin/seminars/hottest.html.

[Buc+23] Ulrik Buchholtz et al. Central H-spaces and banded types. 2023. arXiv: 2301.02636.

[Buc+24] Ulrik Buchholtz et al. Tangent bundles and Euler classes. Extended abstract at *Workshop on Homotopy Type Theory/Univalent Foundations (HoTT/UF 2024)*. 2024. URL: https://hott-uf.github.io/2024/HoTTUF_2024_paper_22.pdf.

[Cag+24] Pierre Cagne et al. “On symmetries of spheres in univalent foundations”. *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS ’24. Tallinn, Estonia: Association for Computing Machinery, 2024. ISBN: 9798400706608. DOI: 10.1145/3661814.3662115.

[Cav21] Evan Cavallo. Pointed functions into a homogeneous type are equal as soon as they are equal as unpointed functions. Agda formalization, part of the cubical library. 2021. URL: https://agda.github.io/cubical/Cubical.Foundations.Pointed.Homogeneous.html#1616.

[CCH24] Felix Cherubini, Thierry Coquand and Matthias Hutzler. A foundation for synthetic algebraic geometry. *Mathematical Structures in Computer Science* 34.9 (Oct. 2024), pp. 1008–1053. ISSN: 1469-8072. DOI: 10.1017/s0960129524000239.

[CDP14] Jesper Cockx, Dominique Devriese and Frank Piessens. Pattern matching without K. *SIGPLAN Not.* 49.9 (Aug. 2014), pp. 257–268. ISSN: 0362-1340. DOI: 10.1145/2692915.2628139.

[CF23] J. Daniel Christensen and Jarl G. Taxerås Flaten. Ext groups in Homotopy Type Theory. 2023. arXiv: 2305.09639.

[CH19] Evan Cavallo and Robert Harper. Higher Inductive Types in Cubical Computational Type Theory. *Proceedings of the ACM on Programming Languages* 3.POPL (Jan. 2019), 1:1–1:27. ISSN: 2475-1421. DOI: 10.1145/3290314.

[Che+24] Felix Cherubini et al. Projective Space in Synthetic Algebraic Geometry. 2024. arXiv: 2405.13916.

[CHM18] Thierry Coquand, Simon Huber and Anders Mörtberg. “On Higher Inductive Types in Cubical Type Theory”. *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS ’18. Oxford, United Kingdom: ACM, 2018, pp. 255–264. ISBN: 978-1-4503-5583-4. DOI: 10.1145/3209108.3209197.

[Chu32] Alonzo Church. A Set of Postulates for the Foundation of Logic. *Annals of Mathematics* 33.2 (1932), pp. 346–366. ISSN: 0003486X, 19398980. URL: http://www.jstor.org/stable/1968337 (visited on 12/02/2025).

[Coh+18] Cyril Cohen et al. “Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom”. *21st International Conference on Types for Proofs and Programs (TYPES 2015)*. Ed. by Tarmo Uustalu. Vol. 69. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 5:1–5:34. ISBN: 978-3-95977-030-9. DOI: 10.4230/LIPIcs.TYPES.2015.5.

[Coq92] Thierry Coquand. “Pattern matching with dependent types”. Workshop on Logical Frameworks. Preliminary proceedings. Båstad, 1992.

[CS23] J Daniel Christensen and Luis Scoccola. The Hurewicz theorem in homotopy type theory. *Algebraic & Geometric Topology* 23 (July 2023), pp. 2107–2140. DOI: 10.2140/agt.2023.23.2107.

[DAL24] Stefania Damato, Thorsten Altenkirch and Axel Ljungström. Formalising inductive and coinductive containers. Submitted. 2024. arX-iv: 2409.02603.

[Doo18] Floris van Doorn. "On the Formalization of Higher Inductive Types and Synthetic Homotopy Theory". PhD thesis. Carnegie Mellon University, May 2018. arXiv: 1808.10690.

[Ec] Martín H. Escardó and contributors. TypeTopology. Agda development. URL: https://github.com/martinescardo/TypeTopology.

[Gra+24] Daniel Gratzer et al. The category of iterative sets in homotopy type theory and univalent foundations. *Mathematical Structures in Computer Science* 34.9 (Oct. 2024), pp. 945–970. ISSN: 1469-8072. DOI: 10.1017/s0960129524000288.

[Gra18] Robert Graham. Synthetic Homology in Homotopy Type Theory. Preprint. 2018. arXiv: 1706.01540.

[GS24] Håkon Robbestad Gylterud and Elisabeth Stenholm. Univalent Material Set Theory. 2024. arXiv: 2312.13024.

[Hed98] Michael Hedberg. A coherence theorem for Martin-Löf's type theory. *Journal of Functional Programming* 8.4 (1998), pp. 413–436. DOI: 10.1017/S0956796898003153.

[Hof97] Martin Hofmann. "Syntax and Semantics of Dependent Types". *Semantics and Logics of Computation*. Ed. by Andrew M. Pitts and P.Editors Dybjer. Publications of the Newton Institute. Cambridge University Press, 1997, pp. 79–130.

[How80] William Alvin Howard. "The Formulae-as-Types Notion of Construction". *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Ed. by Haskell Curry et al. Academic Press, 1980.

[HS98] Martin Hofmann and Thomas Streicher. "The groupoid interpretation of type theory". *Twenty-Five Years of Constructive Type Theory*. Ed. by Giovanni Sambin and Jan Smith. Vol. 36. Oxford Logic Guides. Oxford University Press, 1998, pp. 83–111.

[Jon+23] Tom de Jong et al. "Set-Theoretic and Type-Theoretic Ordinals Coincide". *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, June 2023, pp. 1–13. DOI: 10.1109/lics56636.2023.10175762.

[KR21] Nicolai Kraus and Jakob von Raumer. "Path spaces of higher inductive types in homotopy type theory". *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '19. Vancouver, Canada: IEEE Press, 2021.

[Lic11] Dan Licata. Running Circles Around (In) Your Proof Assistant; or, Quotients that Compute. post on the Homotopy Type Theory blog: https://homotopytypetheory.org/2011/04/23/running-circles-around-in-your-proof-assistant/. 2011.

[Lju23] Axel Ljungström. "A Cubical Formalisation of Cohomology Theory and $\pi_4(\mathbb{S}^3) \cong \mathbb{Z}/2\mathbb{Z}$". Licentiate thesis. Stockholm University, May 2023.

[Lju24] Axel Ljungström. Symmetric monoidal smash products in homotopy type theory. *Mathematical Structures in Computer Science* (2024), pp. 1–23. DOI: 10.1017/S0960129524000318.

[LLM23] Thomas Lamiaux, Axel Ljungström and Anders Mörtberg. "Computing Cohomology Rings in Cubical Agda". *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2023. Boston, MA, USA: Association for Computing Machinery, 2023, pp. 239–252. ISBN: 9798400700262. DOI: 10.1145/3573105.3575677.

[LM23] Axel Ljungström and Anders Mörtberg. "Formalizing $\pi_4(\mathbb{S}^3) \cong \mathbb{Z}/2\mathbb{Z}$ and Computing a Brunerie Number in Cubical Agda ". *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. Los Alamitos, CA, USA: IEEE Computer Society, 2023, pp. 1–13. DOI: 10.1109/LICS56636.2023.10175833.

[LM24a] Axel Ljungström and Anders Mörtberg. Computational Synthetic Cohomology Theory in Homotopy Type Theory. To appear in *Mathematical Structures in Computer Science*. 2024. arXiv: 2401.16336.

[LM24b] Axel Ljungström and Anders Mörtberg. Formalising and Computing the Fourth Homotopy Group of the 3-Sphere in Cubical Agda. Preprint. 2024. arXiv: 2302.00151.

[Lod92] Jean-Louis Loday. "Cyclic Spaces and S1-Equivariant Homology". *Cyclic Homology*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 223–252. ISBN: 978-3-662-21739-9. DOI: 10.1007/978-3-662-21739-9_7.

[LP25] Axel Ljungström and Loïc Pujet. Cellular Methods in Homotopy Type Theory. Preprint. 2025. URL: https://aljungstrom.github.io/files/cellular2025.pdf.

[LS13] Daniel R. Licata and Michael Shulman. "Calculating the Fundamental Group of the Circle in Homotopy Type Theory". *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '13. New Orleans, LA, USA: IEEE Computer Society, 2013, pp. 223–232. ISBN: 978-0-7695-5020-6. DOI: 10.1109/LICS.2013.28.

[LS20] Peter LeFanu Lumsdaine and Michael Shulman. Semantics of higher inductive types. *Mathematical Proceedings of the Cambridge Philosophical Society* 169.1 (2020), pp. 159–208. DOI: 10.1017/S0305041190001 5X.

[LW25] Axel Ljungström and David Wärn. The Steenrod squares via unordered joins. To appear at the *40th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 2025. URL: https://aljungstrom.github.io/files/steenrod2025.pdf.

[Mar82] Per Martin-Löf. "Constructive Mathematics and Computer Programming". *Logic, Methodology and Philosophy of Science VI*. Ed. by L. Jonathan Cohen et al. Vol. 104. Studies in Logic and the Foundations of Mathematics. Elsevier, 1982, pp. 153–175. DOI: https://doi.org/10.1016/S0049-237X(09)70189-2.

[Mar84] Per Martin-Löf. *Intuitionistic type theory. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980*. Vol. 1. Studies in Proof Theory. Bibliopolis, 1984. ISBN: 88-7088-105-9.

[Mik91] Vladimir Voevodsky Mikhail Kapranov. ∞-groupoids and homotopy types. *Cahiers de Topologie et Géométrie Différentielle Catégoriques* 32.1 (1991), pp. 29–46.

[MM94] Saunders Mac Lane and Ieke Moerdijk. "Topoi and Logic". *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. New York, NY: Springer New York, 1994, pp. 267–346. ISBN: 978-1-4612-0927-0. DOI: 10.1007/978-1-4612-0927-0_8.

[Mou+15] Leonardo de Moura et al. "The Lean Theorem Prover (System Description)". *Automated Deduction – CADE-25*. Ed. by Amy P. Felty and Aart Middeldorp. Berlin, Germany: Springer International Publishing, 2015, pp. 378–388. ISBN: 978-3-319-21401-6. DOI: 10.1007/978-3-319-21401-6_26.

[NPS90] Bengt Nordström, Kent Petersson and Jan M. Smith. *Programming in Martin-Löf's type theory: an introduction*. USA: Clarendon Press, 1990. ISBN: 0198538146.

[Pea89] Giuseppe Peano. *Arithmetices Principia Novo Methodo Exposita*. lat. Augustae Taurinorum: Bocca, 1889. URL: http://eudml.org/doc/203509.

[PG24] Jonathan Prieto-Cubides and Håkon Robbestad Gylterud. On planarity of graphs in homotopy type theory. *Mathematical Structures in Computer Science* 34.4 (2024), pp. 281–321. DOI: 10.1017/S0960129524000100.

[Rij+] Egbert Rijke et al. *The agda-unimath library*. URL: https://github.com/UniMath/agda-unimath/.

[Rus03] Bertrand Russell. *Principles of Mathematics*. Appendix B: The Doctrine of Types. New York, Routledge, 1903.

[Sch24] M. Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen* 92.3 (Sept. 1924), pp. 305–316. ISSN: 1432-1807. DOI: 10.1007/BF01448013.

[Shu19] Michael Shulman. *All* $(\infty, 1)$*-toposes have strict univalent universes.* Preprint. Apr. 2019. arXiv: 1904.07004.

[Sim98] Carlos Simpson. Homotopy types of strict 3-groupoids. 1998. arXiv: math/9810059 [math.CT].

[Str93] Thomas Streicher. "Investigations Into Intensional Type Theory". Habilitation thesis. Ludwig-Maximilians-Universität München, 1993. URL: https://www2.mathematik.tu-darmstadt.de/~streicher/HabilStreicher.pdf.

[Swa22] Andrew W Swan. On the Nielsen-Schreier Theorem in Homotopy Type Theory. *Logical Methods in Computer Science* Volume 18, Issue 1 (Jan. 2022). ISSN: 1860-5974. DOI: 10.46298/lmcs-18(1:18)2022.

[UF13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* Institute for Advanced Study: Self-published, 2013. URL: https://homotopytypetheory.org/book/.

[VMA21] Andrea Vezzosi, Anders Mörtberg and Andreas Abel. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *Journal of Functional Programming* 31 (2021), e8. DOI: 10.1017/S0956796821000034.

[Voe10] Vladimir Voevodsky. "Univalent Foundations Project". A modified version of an NSF grant application. Oct. 2010. URL: http://www.math.ias.edu/vladimir/files/univalent_foundations_project.pdf.

[Voe14] Vladimir Voevodsky. "Univalent Foundations". Talk at Institute for Advanced Study, Princeton. Mar. 2014. URL: https://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/2014_IAS.pdf.

[Wär23] David Wärn. Eilenberg–Maclane spaces and stabilisation in homotopy type theory. *Journal of Homotopy and Related Structures* 18.2 (Sept. 2023), pp. 357–368. ISSN: 1512-2891. DOI: 10.1007/s40062-023-00330-5.

[Wär24] David Wärn. Path spaces of pushouts. 2024. arXiv: 2402.12339.

[ZH24] Max Zeuner and Matthias Hutzler. "The Functor of Points Approach to Schemes in Cubical Agda". en. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. DOI: 10.4230/LIPICS.ITP.2024.38.

[ZM23] Max Zeuner and Anders Mörtberg. “A Univalent Formalization of Constructive Affine Schemes”. *28th International Conference on Types for Proofs and Programs (TYPES 2022)*. Ed. by Delia Kesner and Pierre-Marie Pédrot. Vol. 269. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 14:1–14:24. ISBN: 978-3-95977-285-3. DOI: 10.4230/LIPIcs.TYPES.2022.14.