Higher-Dimensional Types in the Mechanization of Homotopy Theory

Kuen-Bang Hou (Favonia)

CMU-CS-17-101

February 2017

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

This research was sponsored by the National Science Foundation under grant numbers CCF-1116703, CCF-1351344, DMS-1324746 and IIS-1065251; Air Force Office of Scientific Research under grant numbers FA-95501210370 and FA-95501510053; and US Army Research Office under grant numbers W911-NF0910273 and W911-NF1310154. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: mechanized reasoning, higher-dimensional types, homotopy theory

For all my friends who chose to leave early
Hōo guá sóoˈū kuattīng thêˈtsá līˈkhui ê pîngˈiú

Abstract

Mechanized reasoning has proved effective in avoiding serious mistakes in software and hardware, and yet remains unpopular in the practice of mathematics. My thesis is aimed at making mechanization easier so that more mathematicians can benefit from this technology. Particularly, I experimented with *higher-dimensional types*, an extension of ordinary types with a hierarchy of stacked relations, and managed to mechanize many important results from classical homotopy theory in the proof assistant Agda. My work thus suggests higher-dimensional types may help mechanize mathematical concepts.

KiˈhâihuàthuilíÍˈking ēˈtàng pībián nńgthé kap ngēˈthé tiong giâmtiōng ê tshòˈngōo, tsóng--sī tī sòohȧk giánkiù sitbū iáu hántit iōng. Pún lūnbûn ê tsongtsí sī beh sú kiˈhâihuà khah khuài hōo sòohȧkka súˈiōng. Siôngsè lâi kóng, guá súˈiōng koˈuî luīhîng (higher-dimensional types), iā tsiūsī tī itpuann ê luīhîng tíngkuân kaˈthiam toˈkaiˈtsân ê kuanhē, tsiong kóotián tônglûn lílūn (classical homotopy theory) ê kuíˈnā hāng tiōngˈiàu ê sîngkó, tī tsìngbîng húˈtsōo kangkhū Agda lāiˈté sūnlī kiˈhâihuà. Guá ê giánkiù sîngkó hiánsī koˈuî luīhîng kiámtshái ēˈtàng pangtsōo sòohȧk khàiliām ê kiˈhâihuà.

Acknowledgments

Above all I want to thank my advisor and dear friend Robert "Bob" Harper, who opened my eyes to the true beauty of programming and made my years at Carnegie Mellon an enjoyable journey. In retrospect, I made a wise move to take the course 15-814 *Type Systems for Programming Languages* by Bob in the first semester of my Ph.D. study while being completely ignorant of programming languages. I remember vividly the days I was sitting in Bob's office, struggling to prove the termination of System F before he announced the secret in the next class meeting. Moments like this made it clear that we share many principles and are a good match as an advisor and an advisee. I am grateful for having this wonderful experience.

Over the years, people in the CMU Homotopy Type Theory (HoTT) group and the Principles of Programming (PoP) group have been sharing with me their kindness, great insights and sense of humor. In particular, Steve Awodey has been playing an important role in supporting the CMU HoTT group and leading various seminars and workshops. The cohomology project in section 3.5, for example, was suggested by Steve, my committee member Ulrik Buchholtz and others. Here is an incomplete list of other wonderful people who have helped me along the way: Umut Acar, Carlo Angiuli, Jeremy Avigad, Stephen Brookes, Evan Cavallo, Karl Crary, Henry DeYoung, Floris van Doorn, Anna Gommerstadt, Danny Gratzer, Jan Hoffmann, Chris Kapulkin, Ryan Kavanagh, Dan Licata, Chris Martens, Edward Morehouse, Stefan Muller, Clive Newstead, Cyrus Omar, Frank Pfenning, Egbert Rijke, Cody Roux, Julian Shun, Robert J. Simmons, Kristina Sojakova, Jon Sterling, Joseph Tassarotti, Ian Ernest Voysey, Todd Wilson and Samuel Westrick.

In addition to friends at CMU, I had met many knowledgable and down-to-earth researchers while participating in the special year at Institute for Advanced Study during 2013 and various conferences and workshops. I feel lucky to have the opportunities to work with Guillaume Brunerie, Jesper Cockx, Eric Finster, Daniel R. Grayson, Nicolai Kraus, Peter LeFanu Lumsdaine, Michael Shulman and others. In particular, Michael patiently taught me the Seifert–van Kampen theorem and helped me revise this thesis; Guillaume, Dan (Licata) and Peter kindly helped me finish the mechanization of the Blakers–Massey theorem; and Guillaume, Daniel and Chris (Kapulkin) generously taught me covering spaces in the classical theory. I also remember clearly the inspiring moments I had with Thorsten Altenkirch, Andrej Bauer, Gershom Bazerman and Urs Schreiber. In addition, I want to thank Simon Huber for helping me visualize their paper (not included in this thesis).

My journey at CMU would be a dull one without many friends playing board games or cramming for SCS musicals together; in particular, Julian Shun and I have planned many fun events and Danny Zhu and I have carried out several interesting performances. In addition, I want to thank my previous advisors at CMU, Jeannette Wing and Avrim Blum, who have kindly supported my transition. The computer science department also has an amazing staff, and we graduate students will remember how wonderful Deborah A. Cavlovich is.

I have to apologize to all people who should have been thanked but were nonetheless ignored due to my oversight. Finally, I want to thank my beloved partner for taking care of me and tolerating my impatience during the thesis writing; I hope this very sentence, the one you are reading right now, is marking the end of the suffering of my partner.

Preface

This thesis serves as a progress report of my five-year journey in mechanizing mathematical theorems using higher-dimensional types. It has been quite a unique experience to expand the frontier of the human knowledge with numerous friendly fellows, and the goal of this report is to help anyone who also hungers for knowing the unknown.

Therefore, much emphasis is on the possibilities in the future, the reflections from the past, but not the perfection of the present; although I wish my writing could be a guide for someone's next journey, I have been avoiding asserting that the paths not travelled must be technically inferior. The rationale behind many of my decisions will be provided for the benefits of future explorers, but I will also be delighted to know if, one day, my arguments against alternative paths no longer stand.

To better serve the subsequent journeys, including my own, I have been trying to identify abstract principles from the current technical development, hoping to better suit other related but different contexts—especially when the actual theories and tools employed in my work become outdated. The disadvantage is that some concepts may appear duplicated, with one being more abstract and one more concrete. I hereby would like to ask every reader to bear with me and maintain an open mind about other choices we could make, especially during the discussion of the principles of higher-dimensional types.

Hope you will find this report useful, and good luck with your journeys!

Favonia, 2017/02/13

Contents

List of Figures

List of Tables

Overview

This thesis demonstrates how a particular abstraction may help the mechanization of mathematics in computer systems; in particular, it consists of several important theorems mechanized in the proof assistant AGDA. The abstraction in discussion here is *higher-dimensional types,* or types extended with higher-dimensional structures. The mechanization is focused on *homotopy theory,* the study of topological spaces up to continuous deformation. To begin with, here is my thesis statement:

> Higher-dimensional types provide novel abstraction that facilitates the mechanization of homotopy theory.

This thesis is organized as follows: Chapter 1 discusses the motivation, the idea and the current development of higher-dimensional types. Chapter 2 introduces the type theory that will be used in chapter 3 to present my work with higher-dimensional types. Chapter 4 discusses some technical details of using the proof assistant AGDA for mechanization. Finally, chapter 5 concludes the thesis with a brief note.

Chapter 1

Mechanization with Higher-Dimensional Types

1.1 Mechanization and Mathematics

The active pursuit of formal mathematical languages starting from the 19th century could be seen as an attempt to evade logical gaps or inconsistencies in more and more abstract developments of mathematics. Through resolving arduous issues in the foundation of mathematics, we largely regained the confidence of absoluteness and correctness. However, I would like to argue that computer systems today are ready to provide yet another layer of protection, further strengthening our faith in modern mathematics consisting of even more abstract developments than one century ago.

Mechanized reasoning, or *computer-aided reasoning,* refers to the use of computer systems to encode, verify or execute logical reasoning. It has been extensively applied to different kinds of software and hardware, probably because many software or hardware bugs with grave consequences keep emerging. Notable verification projects include the kernel `seL4` [79], the C compiler `CompCert` [85] and floating-point units [36, 63, 64, 124].

The worry about correctness in modern mathematics is also real; for example:

- The four-dimensional case of the Busemann–Petty problem was disproved and then proved by the same mathematician Gaoyong Zhang [149, 150].

- A "theorem" by Jan-Erik Roos [121] was found false [107] only after almost 40 years. A proof of a more restricted version was published afterwards [122].

- The Schur multiplier of the Mathieu group M_{22} [103, 104] has been incorrectly calculated as three [33] and then six [34] and is currently believed to be 12 [76].

As a response to the worry about correctness, there have been many mechanization projects, notably:

- the Flyspeck project [61], which mechanized the Kepler conjecture in the proof assistants HOL Light and Isabelle/HOL; and

- the mechanization [57] of the odd order (Feit–Thompson) theorem in the proof assistant Coq; and

- a fully mechanized proof [56] of the four-color theorem in the proof assistant Coq, unlike previous ones which require manual verification of the combinatorial arguments and computer programs specifically written for this theorem.

However, mechanized proofs never enjoyed wide acceptance despite valid correctness concerns and impressive mechanization projects. As Donald MacKenzie pointed out [98, p. 98],

> Mathematicians also showed little interest, however, in the less highly automated but more capable "proof checkers". These are systems that are provided with a full (or nearly full) formal proof, constructed by a human being, and that check whether this proof is indeed a correct one. [...] For example, L. S. van Benthem Jutting, a student of de Bruijn, translated into AUTOMATH and automatically checked the proofs from Edmund Landau's text, *Grundlagen der Analysis.*
>
> Such achievements, however, aroused little enthusiasm among mathematicians. [...]

One major cause, in my opinion, is the fear of mechanized proofs being lengthy and involving excess details. Donald MacKenzie also made a similar observation [98, p. 99]:

> The difficulty and length of formal proofs are certainly a major cause of the absence of any widespread adoption by mathematicians of automated proof checkers. [...]

I believe this concern is partially due to current technical limitations and can be addressed by using *abstraction*, which should be able to help suppress unwanted details, support high-level reasoning, and therefore reduce the obstacles to mechanization.

In an ideal world, mechanized proofs are not just accepted by but also understandable to human beings. There are already signs that this could be the future. For example, during the Institute for Advanced Study (IAS) special year on the univalent foundations of mathematics [137], many proofs were first done mechanically and then "unmechanized" to engage a wider audience. Also, recently mathematician Charles Rezk read my mechanized proof of the Blakers–Massey theorem, wrote a proof in a more traditional style [117] which in turn inspired new research in mathematics [8]. The significance of these events is that ideas can flow from mechanized proofs to conventional, non-mechanized proofs, contrary to common wisdom.

1.2 Preference for Types

Before diving into the ocean of higher-dimensional types, I should describe what types are and why types were preferred: Generally speaking, *type theories* are foundations of mathematics alternative to set theories; *types* express mathematical concepts as sets in set theory; *terms* in type theories are the syntax of programs, proofs or realizers "fulfilling" the types, depending on how exactly the type theories are designed. In the literature, there are two distinct philosophies regarding types:

Ontological types. A type is an ontological classification assigned to associated terms. Every term has a type and its type determines its available operations. Operational semantics, if any, is given after defining the types.

Behavioral types. The operational semantics of terms is first given, and then a type is a mathematical verification property on operational semantics. A term satisfies a type if its operational behavior satisfies the type.

The ontological sense is also called the *intrinsic* interpretation of types or the Church style, while the behavioral sense is the *ex*trinsic interpretation or the Curry style. Note that it is entirely possible for a type theory to have both kinds of types, and in such a type theory the behavioral types are sometimes called *type refinements* as they refine the ontological types. Moreover, although one can claim that behavioral type theory has one single ontological type, it is also possible to build a new ontological type theory based on behavioral type theory by turning selected properties into ontological classifications. Therefore, no particular typing principle is more "fundamental" than the other.

The focus of my thesis was set on ontological types, mainly because the fate of behavioral types with higher-dimensional structures was unclear. Future mechanization projects, however, should revisit the applicability of behavioral types. The reasons why such behavioral types were considered challenging will be discussed in later sections.

The remaining issue is why types were chosen instead of sets, or more precisely, ZF material sets.[1] At first glance, sets are arguably more prevalent in traditional mathematics, which is exactly what this thesis is aiming at, and it seems odd to choose a different foundation. However, there seem to be at least two advantages of types for the purpose of mechanization:

1. Types provide formal abstraction.

2. Type theory is (selectively) proof relevant.

[1]In this thesis I view *structural* sets as types providing the structure of set theory, leaving the term "sets" to *material* sets as in the ZF theory.

Formal abstraction. Abstraction is a powerful mental tool to grasp the complexity of the real world. For example, it would be off-putting to work on number theory if each step had to be carried out in terms of the axioms of the ZF theory. As a result, mathematicians usually do not directly work on set-theoretic definitions of numerals, but on some abstract notion of natural numbers. Types provide a formal language to express (and *enforce*) such abstractions so that it becomes impossible, not just morally undesirable, to inspect the encoding of numerals. With types, such abstraction that was once purely conventional can be easily checked by computers, which fits nicely to the practice of mechanization.

Proof relevance. Secondly, proof relevance is the principle of making (formal) proofs or realizers first-class citizens in theory. Formal proofs or realizers become mathematical objects that can be examined and manipulated directly. This is convenient for mechanizing existing proofs in informal mathematics because it is common for a later proof to cite some construction within a previous proof. With proof relevance, the construction which manifested as a mathematical object is immediately available for later usage (in the system for mechanization). It also suits computer science better because it creates a seamless integration between theorem proving and programming.

Note that it is possible to suppress some proofs and mimic the proof-irrelevant reasoning in type theory if necessary; thus proof relevance is actually more general than otherwise.

In sum, due to formal abstraction and proof relevance, types appear to be a better choice than sets for mechanization, and due to historical accidents, ontological types were chosen over behavioral ones.

1.3 Higher-Dimensional Types

In either the ontological or behavioral typing, a collection of terms "associated" with each type, or simply *elements* of a type, may be identified; it is not uncommon to assume that types may be represented solely by its elements.

This thesis demonstrates that it might be desirable to drop this assumption and equip a type with higher structures on top of its elements. The first approximation is to include a binary relation among elements as an inherent part of a type. An immediate example is a quotient, which can be viewed as a collection of elements equipped with an equivalence relation. A suitable framework of these "types with relations" will require all constructs to respect inherent relations in types; in other words, functions are functors that map related elements to related elements.

If, in some framework, a piece of evidence of relations can be internalized as a term, then we may consider yet another binary relation among pieces of evidence that relate the same two elements. The next layer would be a binary relation among pieces of evidence relating two pieces of evidence relating the same two elements. This may go up to infinity; in the most general case we have countably infinite indexed binary relations stacked on top of elements.

Such a hierarchy of relations enables us to represent with ease mathematical objects inherently with many layers of relations. I call these stacked relations *higher-dimensional structures*, where *dimension* refers to the distance from the elements at the zeroth dimension; the relation among elements is at the first dimension, the next level at the second, and so on. A *higher-dimensional type* is a type equipped with higher-dimensional structures, and a *higher-dimensional type theory* is a type theory compatible with types with non-trivial higher-dimensional structures.

Note that these higher-dimensional structures need not be equivalence relations. In fact, there are motivating examples to consider non-symmetric relations. Unfortunately, higher-dimensional structures other than equivalence relations are still underdeveloped at the time of writing.

1.4 Applications of Higher-Dimensional Types

What are the immediate applications of higher-dimensional types? In this section I will review four known applications: univalent set-theoretic reasoning, patch theory, homotopy theory and quotients.

Univalent reasoning. Informal mathematics usually assumes that if a group (as an algebraic structure) satisfies some property, so do other isomorphic groups. For example, if a group is abelian, so are other isomorphic ones. The *univalence principle* formally asserts that every two isomorphic structures are the same and thus every property must respect structural equivalence (such as group equivalence). Moreover, because there can be more than one equivalence between two structures, there can be more than one way in which two structures can be identified. If one adopts the proof relevance principle as we did, there will be different identification proofs between two structures.

An easy way to represent such a universe of structures with identification is to assign structural equivalence to the higher-dimensional structures of the universe. This way we have a univalent universe of structures.

Patch theory. It has been shown [10] that we can formulate patch theories as higher-dimensional types. Here a *patch* (such as "adding a file") represents a change to a repository, and it only applies in certain *patch contexts* (such as "the file being added does not exist"). There are also *patch laws*, the equations the patches must obey. Carlo Angiuli, Edward Morehouse, Dan Licata and Robert Harper demonstrated how to wrap all these components of a patch theory into one single higher-dimensional type, where elements are patch contexts, relations between elements are patches, and relations between patches are patch laws.

Notice that if we can drop the widely-assumed symmetry of higher-dimensional structures, irreversible patches can be modeled more elegantly.[2] This research direction remains open at the time of writing.

Homotopy theory. Homotopy theory is the study of topological spaces up to continuous deformation (homotopy), and these homotopies naturally fit into higher-dimensional structures. The idea here is to use type theory as the language for (well-behaved) topological spaces up to homotopy equivalence. With homotopies internalized as an inherent part of types, every element (as a formal proof) must respect homotopies, which leads to a more general theorem that holds in various models with homotopy structures. This line of research has been fruitful [16, 19–21, 29, 51, 78, 87, 94, 106, 135, 138, 146], and my thesis work is following this approach.

Remark 1.4.1. This methodology is called *synthetic homotopy theory* because it uses no open sets, as *synthetic geometry* uses no coordinates. Instead, new spaces are built from primitive spaces and combinators.[3]

There are also considerable efforts [7, 17, 44, 123] in mechanizing topological objects *without* internalizing homotopies or using higher-dimensional types. These could eventually lead to, for example, full verification of the program KENZO [136] which implements many algorithms for computing algebraic structures of topological spaces. The advantage of this approach is that the proofs may incorporate non-homotopy-preserving methods but then the disadvantage is that some constructions such as loop spaces might need more work and the resulting proofs admit fewer models.

Quotients. Finally, higher-dimensional structures enable us to define quotients, a prevalent construct in mathematics. A prominent example is Cauchy real numbers, which are usually defined as the quotient of Cauchy sequences [138]. Roughly speaking, Cauchy reals are intended to be the completion of rational numbers under Cauchy sequences, sequences in which for any distance, there always exists a suffix whose numbers do not differ by more than that given distance. A proper definition of real numbers thus needs to identify two Cauchy sequences "approaching the same number", and higher-dimensional types are able to bundle such identification.

These four examples show the potential of higher-dimensional types in providing better abstractions for mechanization of mathematics.

[2]In Angiuli *et al.* [10] the authors tackle irreversibility by encoding complete patch histories as patch contexts.

[3]More precisely, the homotopy classes of topological spaces.

	Judgmental equality	Internalized equality	Identification
Common usage ↳ such as [67]	definitional equality	extensional identity	intensional identity
Common usage ↳ such as [138]	judgmental equality	*(not used)*	propositional equality and identity
Martin-Löf (1975)	definitional equality	*(not used)*	identity
Martin-Löf ↳ (1982, 1984)	definitional equality *(intensional)*	propositional equality *(extensional)*	*(not used)*
NuPRL [41]	equality	equality	*(not used)*

Table 1.1: Rosetta Stone for identification and equality.

NuPRL terminology does not line up with others' well.
Readers are recommended to consult the documentation of NuPRL directly.

1.5 Examples of Higher-Dimensional Type Theories

At the time of writing there are already many higher-dimensional type theories as defined in this thesis. In the following these icons are used to mark the features of individual type theories: {⧈} means *cubical*, {≡} means having *internalized equality* as types and {🕓} means *guarded* (see below).

Martin-Löf ontological type theory with identification types. Martin-Löf ontological type theory with identification types (the variant at year 1975 [100]) is already higher-dimensional in the sense that every type comes with a hierarchy of iterated identification. In practice, however, additional axioms or other extensions are needed to force the existence of types with non-trivial higher-dimensional structures. More details will be discussed in chapter 2.

Remark 1.5.1. The name *identification types* is used to emphasize that they are fundamentally different from *equality types* and signify the action of identifying two things. Other common names are shown in table 1.1.

Remark 1.5.2. There is a misconception that behavioral types have no interesting higher-dimensional structures was partially due to the failure of mimicking iterated identification: behavioral typing almost always validates equality types, equality and identification types appear similar, but equality types are always trivial. However, higher-dimensional structures in behavioral type theories need not be equality types, as shown in some type theories listed below.

Univalent type theory with higher inductive types (UniTT+HIT) (syntax of homotopy type theory). This type theory extends the above theory with the *univalence axiom* and *higher inductive types* and is the theory in use for the thesis work. Both extensions are motivated by homotopy theory and will be discussed in chapter 2. Suitable mechanization tools are introduced in chapter 4.

Remark 1.5.3. I use the name *univalent type theory* instead of *homotopy type theory* for two reasons: Firstly, depending on the speaker and the context, *homotopy type theory* may refer to the entire research area, the type theory, or the type theory combined with a particular interpretation into homotopy theory. Secondly, although all of my thesis work indeed follows this particular interpretation (see chapters 2 and 3), some applications of the type theory in computer science [10] seem unrelated to homotopy theory. People also call this type theory *the book HoTT,* referring to the type theory defined in the appendix of the book *Homotopy Type Theory: Univalent Foundations for Mathematics*.

Homotopy type system [141] {≡}. This type theory proposed by Vladimir Voevodsky extends the univalent type theory UniTT with a new layer with internalized equality. The two-layer structure consists of *fibrant* types which respect homotopy equivalence, and non-fibrant types which may not. The new layer of non-fibrant types makes it possible to "break" homotopy equivalence and define for example simplicial types, which seem impossible without this feature. There is a proof checker prototype by Daniel R. Grayson [58].

Two-level system [5]. This type theory by Thorsten Altenkirch, Paolo Capriotti and Nicolai Kraus is heavily inspired by the homotopy type system by Vladimir Voevodsky, but replaces the outer (non-fibrant) layer by Martin-Löf type theory with a version of identification obeying the principle of uniqueness of identification proofs (UIP). Although I will not talk about this two-layer system, the principle of UIP will be discussed in section 2.3. There is no implementation of this type theory to the best of my knowledge.

Real-cohesive homotopy type theory [125, 128]. This type theory, mainly by Michael Shulman and Urs Schreiber, extends UniTT further with synthetic topological reasoning. Formally, a new context containing *crisp variables* was introduced to take into account possibly discontinuous mappings. This new context, along with several new modal operators, makes it possible to use the direct arguments of synthetic homotopy theory to yield non-homotopical conclusions. Unfortunately, there is still no implementation for this type theory to date.

Two-dimensional type theory [92]. This was one of the early efforts by Dan Licata and Robert Harper to fill in missing computational contents of the univalence axioms and higher inductive types. The type theory was, in retrospect, a step toward ontological cubical type theory, but it was unclear then how to extend the technique to higher dimensions, and there is no known implementation at the time of writing.

Cubical type theories [6, 39, 74, 90, 110] {⧉}. Progress in modeling univalent type theory UniTT in cubical sets [23] has inspired Thorsten Altenkirch and Ambrus Kaposi; Guillaume Brunerie and Dan Licata; Cyril Cohen, Thierry Coquand, Simon Huber and Anders Mörtberg; Valery Isaev; and Ian Orton and Andrew M. Pitts to build several ontological type theories with explicit cubical structures. Higher-dimensional structures are made explicit at the judgmental level. There are an experimental implementation [45] and a formalization in NuPRL [24] in progress.

Guarded cubical type theory [25] {⧉,◷}. This type theory by Lars Birkedal, Aleš Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters and Andrea Vezzosi combines features of the above cubical type theories and the guarded dependent type theory [25] such as later types and delayed substitution. It also has a prototype checker [60].

Computational higher-dimensional type theory [9] {⧉}. Another line of research led by Carlo Angiuli, Robert Harper and Todd Wilson, also inspired by cubes, constructs a behavioral cubical type theory by extending the NuPRL semantics to higher dimensions. An experimental implementation is in progress as well [131].

1.6 Relationship with the Univalent Foundations

Recently, Vladimir Voevodsky has been actively promoting the *univalent foundations* as new foundations of mathematics. It emphasizes the need of mechanization in mathematical practice and promotes ∞-groupoids and homotopy types[4] as the organizing principles. Currently the main development following Vladimir Voevodsky's ideas is the UniMath library [145] of the proof assistant Coq. The history and philosophy of the foundations can be found in the 2014 Paul Bernays Lectures [142] given by Vladimir Voevodsky and also the review paper by Álvaro Pelayo and Michael A. Warren [112].

Remark 1.6.1. The sameness of homotopy types and ∞-groupoids was already pointed out by the great mathematician Alexander Grothendieck [59]:

> [...] One comment is that presumably, the category of ∞-groupoids (which is still to be defined) is a "model category" for the usual homotopy category; this would be at any rate one plausible way to make explicit the intuition referred to before, that a homotopy type is "essentially the same" as an ∞-groupoid up to ∞-equivalence. [...]

This thesis is very close to Vladimir Voevodsky's vision about the univalent foundations program. We share the view that ideally mechanization should be part of mathematical proving. We both promote higher-dimensional structures and none of us insist

[4]See chapter 2 on page 11 for the definition of homotopy types.

on a particular type theory. Moreover, this thesis aims to improve the current technology of mechanization, and Vladimir Voevodsky once expressed “a lot of wishes in terms of getting this proof assistant to work better” [143].

That said, I am hesitant to accept the philosophical belief that (in terms of this thesis) higher-dimensional structures should be symmetric relations (as ∞-groupoids), and thus I cannot place my thesis under the univalent foundations. This is partially due to my background in computer science: Computation in most systems is inherently an irreversible process, usually modeled as a non-symmetric partial order among elements. Important ideas in computer science such as normalization, bisimulation and fixed points all expect the non-symmetric nature of computation. The study of patch theory [10] also suggests non-symmetric higher-dimensional structures could be useful. While this thesis is focusing on mechanization of mathematics, I do hope that higher-dimensional types can find their applications in computer science as well.

Chapter 2

Univalent Type Theory

Homotopy theory is an important part of algebraic topology, which studies topological spaces up to *homotopy equivalence.* Intuitively, two spaces are homotopy equivalent, or of the same *homotopy type,* if one space can deform to the other continuously. Although the holy grail of topology is to classify all spaces up to the stricter homeomorphism, it turns out that many interesting topological properties respect homotopy equivalence, and so this coarser partition often suffices.

The book *Homotopy Type Theory: Univalent Foundations for Mathematics* [138] presents a type theory that is capable of capturing homotopy-theoretic concepts. It drew many ideas from different researches, among them the following two threads are the most influential:

- Steve Awodey and Michael A. Warren gave a model of type theory in abstract homotopy theory [16, 146]. Benno van den Berg and Richard Garner published a paper addressing the coherence issue [21].
- Vladimir Voevodsky gave a model of type theory in the concrete homotopy theory of simplicial sets and proposed the novel *univalence axiom* [139, 140] (also see [78]).

Collectively I summarize these models as the *identification-as-path* interpretation, signifying their treatment of higher-dimensional structures. These ideas, along with many other considerations, lead to a *univalent type theory* (UniTT) with *higher inductive types* (HIT), or UniTT+HIT for short.[1] UniTT is essentially a variant of the Martin-Löf type theory plus the univalence axiom. Higher inductive types were invented at the Oberwolfach meeting in 2011 during the discussions between Andrej Bauer, Peter LeFanu Lumsdaine, Michael Shulman and Michael A. Warren [95] and then further developed by many people [138, note of chap. 6]. The following is a brief and incomplete introduction to UniTT+HIT; a more comprehensive account can be found in the book *Homotopy Type Theory: Univalent Foundations for Mathematics* [138].

[1] See remark 1.5.3 on page 8 for the reason why this is not called *homotopy type theory.*

2.1 Martin-Löf Type Theory

Per Martin-Löf has published many different type theories, and the most relevant one here is probably the paper [100] at 1975. The UniTT is morally the formal system in that paper (as an ontological type theory) extended with a schema for inductive types and a ramified hierarchy of cumulative universes. For the rest of this section, this particular variant is referred to as *the* Martin-Löf type theory.

The Martin-Löf type theory is a type theory aimed at serving as a foundation for constructive mathematics. It provides basic building blocks such as sum types ($\sum$ types), function types ($\prod$ types), identification types, inductive types and universes; each captures a fundamental concept in mathematics and together they form a powerful language. Here is a complete list of the available components:

Sum types A sum type $\sum_{x:A} B(x)$ is the union of a family of types B indexed by another type A. An element of the sum type $\sum_{x:A} B(x)$ is a pair of the index a of type A and an element of the type instance $B(a)$. The binary product type of two types A and B, written $A \times B$, is defined to be the sum type $\sum_{\_:A} B$ where B does not depend on A.

Function types Again with a family of types B indexed by a type A, an element of the function type $\prod_{x:A} B(x)$ is a λ-function $\lambda(x{:}A).b$ sending an element a of type A to the element $b[a/x]$ of the corresponding type instance $B(a)$. The arrow type from type A to type B, written $A \to B$, is defined to be the function type $\prod_{\_:A} B$ where B does not depend on A.

The unit type The unit type $\mathbb{1}$ with only one element `unit`.[2]

Identification types Given a type A and two elements a and b of type A, an identification type $a =_A b$ collects identifications between elements a and b.

Inductive types Inductive types represent inductively defined structures, which include the empty type, binary coproduct types, natural numbers and many others. See section 2.4 for more details.

Universes A universe is, informally, a special type of all types. A ramified hierarchy of cumulative universes $\mathcal{U}_0, \mathcal{U}_1, \ldots$ is introduced to avoid Girard's paradox [54, 73].

For brevity, the type A in λ-expressions $\lambda(x{:}A).b$, identification types $a =_A b$, sum types $\sum_{x:A} B(x)$, function types $\prod_{x:A} B(x)$ may be omitted when clear from the context. However, the argument type may be additionally marked as $f(x{:}A)$ for clarity.

[2] If the unit type were defined as an inductive type as described later in section 2.4, it would not validate the (judgmental) uniqueness rule.

Remark 2.1.1. A family of types indexed by type A in this thesis is defined to be a function from A to some universe $\mathcal{U}_i$, and thus the above description of function types is circular and not a real definition. The description of sums also unnecessarily depends on the concept of functions. See the formal rules below, which avoid such circularity.

Remark 2.1.2. Function types are also called *dependent product types* where the arrow types are called *function types.* I chose to avoid the word *product* because the sum types were once called dependent product types (as a generalization of binary products) and this has led to serious confusion. I cautiously limited the usage of the word *dependent* because its meaning was also muddled. The terminology in this thesis is still undesirable because the word *sum* does not match the word *function,* but at least it is clear.

Given an ambient logical framework with substitutions and structural rules properly set up, each type constructor (except universes) can be formalized using five kinds of rules: type *formation*, *introduction* and *elimination* of elements of that type, *computation* of matching introducers (constructors) and eliminators (also called β-rules), and sometimes *uniqueness* of elements (also called η-rules). The ambient framework consists of three forms of judgments with the following English reading:

- Γ `ctx`. The variable context Γ, a finite list of variables names and their types, is valid in the sense that the type of every variable belongs to some universe with respect to preceding variables (that is, the prefix of Γ before that variable).
- $\Gamma \vdash a : A$. The context Γ is valid, and with respect to that context, the A is a type in some universe $\mathcal{U}_i$ and the element a is of the type A.
- $\Gamma \vdash a \equiv b : A$. The context Γ is valid, and with respect to that context, the A is a type in some universe $\mathcal{U}_i$ in the context Γ, the elements a and b are of the type A, and they are considered synonyms.

With these three forms of judgments, sum types $\sum_{x:A} B$, function types $\prod_{x:A} B$ and the unit type $\mathbb{1}$ can be formalized as figs. 2.1 to 2.3, respectively; other types will be discussed in later sections. The point is that the entire Martin-Löf type theory can be defined using rules.

2.2 Identification

Shown in fig. 2.4, identification types can be formalized in a similar fashion as sums, functions and the unit (but without the uniqueness rule). As hinted in the previous chapter, the presence of identification types immediately grants us certain kinds of higher-dimensional types. Given any type A, the relation at the first dimension can be identification types $a =_A b$ indexed by two elements a and b, and the relation at the second dimension can be identification types $p =_{a=_A b} q$ indexed by two identifications p and q

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, x{:}A \vdash B : \mathcal{U}_i}{\Gamma \vdash \sum_{x:A} B : \mathcal{U}_i}\ \textstyle\sum\text{-FORMATION}$$

$$\frac{\Gamma, x{:}A \vdash B : \mathcal{U}_i \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B[a/x]}{\Gamma \vdash \langle a; b\rangle : \sum_{x:A} B}\ \textstyle\sum\text{-INTRODUCTION}$$

$$\frac{\Gamma, z{:}\sum_{x:A} B \vdash C : \mathcal{U}_i \qquad \Gamma, x{:}A, y{:}B \vdash c : C[\langle x; y\rangle/z] \qquad \Gamma \vdash s : \sum_{x:A} B}{\Gamma \vdash \texttt{elim}_{\sum}[z.C](x.y.c; s) : C[s/x]}\ \textstyle\sum\text{-ELIMINATION}$$

$$\frac{\begin{array}{c}\Gamma, z{:}\sum_{x:A} B \vdash C : \mathcal{U}_i \\ \Gamma, x{:}A, y{:}B \vdash c : C[\langle x; y\rangle/z] \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B[a/x]\end{array}}{\Gamma \vdash \texttt{elim}_{\sum}[z.C](x.y.c; \langle a; b\rangle) \equiv c[a, b/x, y] : C[\langle a; b\rangle/z]}\ \textstyle\sum\text{-COMPUTATION}$$

$$\frac{\Gamma \vdash s : \sum_{x:A} B}{\Gamma \vdash s \equiv \langle \texttt{fst}_{A;x.B}(s); \texttt{snd}Ax.B(s)\rangle : \sum_{x:A} B}\ \textstyle\sum\text{-UNIQUENESS}$$

$$\begin{aligned}
\texttt{fst}_{A;x.B} &:\equiv \lambda\Big(s{:}\textstyle\sum_{x:A} B\Big).\texttt{elim}_{\sum}\Big[\_.A\Big](x.\_.x; s)\\
\texttt{snd}_{A;x.B} &:\equiv \lambda\Big(s{:}\textstyle\sum_{x:A} B\Big).\texttt{elim}_{\sum}\Big[s.B[\texttt{fst}_{A;x.B}(s)/x]\Big](\_.y.y; s)
\end{aligned}$$

Figure 2.1: Rules of sum.

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, x{:}A \vdash B : \mathcal{U}_i}{\Gamma \vdash \prod_{x:A} B : \mathcal{U}_i}\ \textstyle\prod\text{-FORMATION} \qquad \frac{\Gamma, x{:}A \vdash b : B}{\Gamma \vdash \lambda(x{:}A).b : \prod_{x:A} B}\ \textstyle\prod\text{-INTRODUCTION}$$

$$\frac{\Gamma \vdash f{:}\prod_{x:A} B \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B[a/x]}\ \textstyle\prod\text{-ELIMINATION}$$

$$\frac{\Gamma, x{:}A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda(x{:}A).b)(a) \equiv b[a/x] : B[a/x]}\ \textstyle\prod\text{-COMPUTATION}$$

$$\frac{\Gamma \vdash f : \prod_{x:A} B}{\Gamma \vdash f \equiv (\lambda(x{:}A).f(x)) : \prod_{x:A} B}\ \textstyle\prod\text{-UNIQUENESS}$$

Figure 2.2: Rules of I/O.

$$\frac{\Gamma\ \mathtt{ctx}}{\Gamma \vdash \mathbb{1} : \mathcal{U}_0}\ \mathbb{1}\text{-FORMATION} \qquad \frac{\Gamma\ \mathtt{ctx}}{\Gamma \vdash \mathtt{unit} : \mathbb{1}}\ \mathbb{1}\text{-INTRODUCTION}$$

$$\frac{\Gamma, x{:}\mathbb{1} \vdash C : \mathcal{U}_i \qquad \Gamma \vdash c : C[\mathtt{unit}/x] \qquad \Gamma \vdash u : \mathbb{1}}{\Gamma \vdash \mathtt{elim}_{\mathbb{1}}[x.C](c;u) : C[u/x]}\ \mathbb{1}\text{-ELIMINATION}$$

$$\frac{\Gamma, x{:}\mathbb{1} \vdash C : \mathcal{U}_i \qquad \Gamma \vdash c : C[\mathtt{unit}/x]}{\Gamma \vdash \mathtt{elim}_{\mathbb{1}}[x.C](c;\mathtt{unit}) \equiv c : C[\mathtt{unit}/x]}\ \mathbb{1}\text{-COMPUTATION}$$

$$\frac{\Gamma \vdash u : \mathbb{1}}{\Gamma \vdash u \equiv \mathtt{unit} : \mathbb{1}}\ \mathbb{1}\text{-UNIQUENESS}$$

Figure 2.3: Rules of being alone.

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash a =_A b : \mathcal{U}_i}\ =\text{-FORMATION}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathtt{refl}_a : a =_A a}\ =\text{-INTRODUCTION}$$

$$\frac{\begin{array}{c}\Gamma, x{:}A, y{:}A, z{:}(x{=}y) \vdash C : \mathcal{U}_i \qquad \Gamma, x{:}A \vdash c : C[x, x, \mathtt{refl}_x/x, y, z] \\ \Gamma \vdash a : A \qquad \Gamma \vdash b : A \qquad \Gamma \vdash p : a =_A b\end{array}}{\Gamma \vdash \mathtt{elim}_{=}[x.y.z.C](x.c;a;b;p) : C[a, b, p/x, y, z]}\ =\text{-ELIMINATION}$$

$$\frac{\begin{array}{c}\Gamma, x{:}A, y{:}A, z{:}(x{=}y) \vdash C : \mathcal{U}_i \\ \Gamma, x{:}A \vdash c : C[x, x, \mathtt{refl}_x/x, y, z] \qquad \Gamma \vdash a : A\end{array}}{\Gamma \vdash \mathtt{elim}_{=}[x.y.z.C](x.c;a;a;\mathtt{refl}_a) \equiv c[a/x] : C[a, a, \mathtt{refl}_a/x, y, z]}\ =\text{-COMPUTATION}$$

Figure 2.4: Rules of identification.

identifying the same two elements a and b, and so on. See the work [20, 94] by Benno van den Berg, Richard Garner and Peter LeFanu Lumsdaine for more discussions about the structures formed by identification types. Also see remark 1.6.1 on page 9 for Alexander Grothendieck's insight. In retrospect, the identification-as-path interpretation relate the following three concepts:

1. higher-dimensional structures, and
2. iterated identification types, and
3. (homotopy classes of) paths and homotopies in homotopy theory.

In other words, types are understood as spaces, elements of a type as points in a space, identifications as paths in a space, identifications of identifications as paths between paths (path homotopies), functions as continuous maps, families of types as fibrations, and so on.[3] To see functions of type $A \to B$ indeed carry over all the higher-dimensional structures in the domain A, preserving all identifications in A, one can define the *application to identification* $\mathtt{ap}_f(p)$ for any function $f : A \to B$ and any identification $p : a =_A b$:

$$\mathtt{ap}_f(p) :\equiv \mathtt{elim}_{=}\big[x.y.\_.f(x) =_B f(y)\big](x.\mathtt{refl}_{f(x)}; a; b; p), \tag{2.1}$$

which witnesses the identification in the image. The correspondence between type theory and homotopy theory will be further explored in chapter 3.

Note that the elimination rule in fig. 2.4 indicates that it is sufficient to prove the reflexivity case despite all possible non-trivial identifications! The intuition is that, in the first premise of the elimination rule

$$\Gamma, x{:}A, y{:}A, z{:}(x{=}y) \vdash C : \mathcal{U}_i$$

two end points of the identification, x and y, are "free" and thus the identification z can be continuous shrunk to reflexivity. In other words, proofs for the reflexivity case can be continuously extended to cover all possible identifications. In general, we need at least one free end point for this extension argument to apply. Indeed, there are two equivalent eliminators for identification with either one end fixed (without showing the accompanying computation rules) [138, §1.12.1]:

$$\frac{\begin{array}{c}\Gamma \vdash a : A \qquad \Gamma, x{:}A, y{:}(a{=}x) \vdash C : \mathcal{U}_i \\ \Gamma \vdash c : C[a, \mathtt{refl}_a / x, y] \qquad \Gamma \vdash b : A \qquad \Gamma \vdash p : a =_A b\end{array}}{\Gamma \vdash \mathtt{elim}_{a=}[x.y.C](c; b; p) : C[b, p/x, y]} \text{ =-ELIMINATION (LEFT END FIXED)}$$

$$\frac{\begin{array}{c}\Gamma \vdash a : A \qquad \Gamma, x{:}A, y{:}(x{=}a) \vdash C : \mathcal{U}_i \\ \Gamma \vdash c : C[a, \mathtt{refl}_a / x, y] \qquad \Gamma \vdash b : A \qquad \Gamma \vdash p : b =_A a\end{array}}{\Gamma \vdash \mathtt{elim}_{=a}[x.y.C](c; b; p) : C[b, p/x, y]} \text{ =-ELIMINATION (RIGHT END FIXED)}$$

[3]Again, the terminology borrowed from topology should be understood as homotopy classes of such mathematical objects.

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash \mathtt{Eq}_A(a; b) : \mathcal{U}_i}\ \textsc{Eq-formation}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \langle\rangle : \mathtt{Eq}_A(a; a)}\ \textsc{Eq-introduction} \qquad \frac{\Gamma \vdash p : \mathtt{Eq}_A(a; b)}{\Gamma \vdash a \equiv b : A}\ \textsc{Eq-elimination}$$

$$\frac{\Gamma \vdash p : \mathtt{Eq}_A(a; b)}{\Gamma \vdash p \equiv \langle\rangle : \mathtt{Eq}_A(a; b)}\ \textsc{Eq-uniqueness}$$

Figure 2.5: Rules of internalized equality.

These forms, called *based identification eliminators,* are commonly attributed to Christine Paulin-Mohring [111]. Due to their equivalences I might not distinguish them from the original eliminator during a complex proof in later chapters. Soon we will see yet another elimination rule for identification types that does not validate such extension and in fact kill all non-trivial higher-dimensional structures.

2.2.1 Equality Types

Another similar-looking type constructor, *equality types,* was frequently compared to identification types, but they are nonetheless fundamentally different. Equality types, unlike the identification types, were justified by the meaning explanation, and the comparison on formal presentations could be out of context.

In any case, equality types internalize judgmental equality as types, denoting the sameness in meaning (as synonyms) that may be used in arbitrary substitution. Therefore, a piece of evidence of such type can and should be reflected as a judgmental equality, and its content is and should always be trivial. Identification proofs, on the other hand, may be non-trivial and there can be more than one identification between two elements. Technically, one could base higher-dimensional structures upon equality types, but the entire hierarchy would become trivial.

Internalized equality may be formalized as rules in fig. 2.5, with notable omission of the computation rule; the omission is due to the judgmental equality in the elimination rule not depending on the (trivial) proof of the equality type, and it being senseless even to try to talk about equality between two judgments.

It can be shown that if all the formal rules of internalized equality and identification in figs. 2.4 and 2.5 are naïvely placed into the same type theory, then the two notions collapse. That is, identification can also be reflected as judgmental equality. To see this, consider the following function of type $(x =_A y) \to \mathtt{Eq}_A(x; y)$, which turns identification into internalized equality:

$$\lambda(p{:}x{=}y).\mathtt{elim}_{x=}\Big[y.\_.\mathtt{Eq}_A(x;y)\Big](\langle\rangle;y;p).$$

Combined with the elimination rule of the equality type in fig. 2.5, any identification proof can then be reflected as a judgmental equality. I believe this collapse should be viewed as a symptom of the deeper problem that the meaning explanation, where equality types emerged and where formal rules are secondary, is philosophically incompatible with various mathematical interpretations of a type theory with non-trivial identification (presented in the above form). In other words, the meanings of these two types are different but they happen to have similar formal presentations. Having separate layers with distinct interpretations of types, like [141] by Vladimir Voevodsky, is one way to mitigate the conflict.

2.3 Univalence Principle

The name *univalence,* according to Vladimir Voevodsky [144], came from 1. the Russian translation of *faithful functor*—*унивалентный функтор* (in its normative case)—which can be transliterated as *univalentnyj funktor,*[4] and 2. the wordplay of *universal* without *versal,* referring to the supposed universal property, which consists of existence and uniqueness, without the existence part when modeling universes by fibrations. Therefore, I believe the intention is to have a faithful viewpoint on mathematics through foundations honoring some uniqueness principle in the universe.

Technically speaking, the uniqueness principle says that equivalent things should be identified. This principle can be traced back to the *universe extension* for the groupoid model constructed by Martin Hofmann and Thomas Streicher [67]. The current form for the identification-as-path interpretation originates from Vladimir Voevodsky *et al.*'s work on simplicial sets [78, 139, 140]. The principle, manifested as an extra axiom, asserts that type identification in a universe is equivalent to equivalence between types.[5] This principle recognizes new instances of identification between two types in the universe that cannot be derived from the formal rules in fig. 2.4, and has profound implications in type theory as described below.

Before higher-dimensional structures gained wide appreciation, there was a time where the uniqueness of identification proofs (UIP) remained open, and Thomas Streicher [132] had proposed an additional rule for identification types—*K,* which is equivalent to the UIP.[6] Following our notation, the rule and its accompanying computation

[4]This transliteration follows the standards ISO 9:1995 [75] and GOST 7.79 System A [151]. There are other systems such as the *scientific transliteration* but the differences can be ignored here.

[5]More precisely, it asserts the canonical function from type identification to type equivalence is itself an equivalence. See section 2.3.2.

[6]I assume the rule K was named after the original elimination rule, which was named J.

rule can be presented as follows:

$$\frac{\begin{array}{c}\Gamma, x{:}A, y{:}(x{=}x) \vdash C : \mathcal{U}_i \\ \Gamma, x{:}A \vdash c : C[x, \mathtt{refl}_x/x, y] \qquad \Gamma \vdash a : A \qquad \Gamma \vdash p : a =_A a\end{array}}{\Gamma \vdash \mathtt{K}[x.y.C](x.y.c; a; p) : C[a, p/x, y]} \text{ =-ELIMINATION (K)}$$

$$\frac{\Gamma, x{:}A, y{:}(x{=}x) \vdash C : \mathcal{U}_i \qquad \Gamma, x{:}A \vdash c : C[x, \mathtt{refl}_x/x, y] \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathtt{K}[x.y.C](x.y.c; a; \mathtt{refl}_a) \equiv c[a/x] : C[a, p/x, y]} \text{ =-COMPUTATION (K)}$$

Notice how the first premise of the elimination rule

$$\Gamma, x{:}A, y{:}(x{=}x) \vdash C : \mathcal{U}_i$$

locks up two ends of y and thus does not enjoy the identification extension argument on page 16 as the original elimination rule does.

The groupoid model later constructed by Martin Hofmann and Thomas Streicher [67] showed that the UIP and the new rule are not provable, answering the open problem. The univalence axiom refutes the UIP because the newly recognized instances of identification in the universe are not unique between types. A quick example is to observe that there are two provably distinct equivalences between Booleans (the identity function and the negation function) and hence two provably distinct identification proofs between the Boolean type by the univalence principle.

For the same reason the univalence axiom refutes the law of excluded middle (LEM) in full generality, that is, for any type we know either it is inhabited or any element of it leads to contradiction:

$$\mathtt{LEM} : \prod_{A:\mathcal{U}_i} A + (A \to \mathbb{0}).$$

(See section 2.4 for coproduct types $A + B$ and the empty type $\mathbb{0}$.) The reason is that such LEM precludes non-trivial higher-dimensional structures [66] and thus contradicts with the rich structures postulated by the univalence axiom. Nonetheless, a limited version of the LEM may be safely introduced [138, §3.4].

Somewhat surprisingly, the univalence axiom also implies functional extensionality and a proof may be found in [138, §4.9].

The exact formulation of the univalence principle in UniTT is quite involved, as we have to define *equivalence* before stating that equivalence is equivalent to identification between types. The remainder of this section will give a precise formulation of equivalence and then the univalence axiom, the last rule in fig. 2.6.

2.3.1 Equivalence *in stuvac*

Informally, an equivalence between two types A and B is a function admitting an inverse function; that is, it should contain the following four components:

$$\frac{\Gamma\ \texttt{ctx}}{\Gamma \vdash \mathcal{U}_i : \mathcal{U}_{i+1}}\ \mathcal{U}\text{-INTRODUCTION} \qquad \frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash A : \mathcal{U}_{i+1}}\ \mathcal{U}\text{-CUMULATION}$$

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i}{\Gamma \vdash \texttt{univalence}_i(A;B) : \texttt{is-equiv}\Big(\lambda(p{:}A{=_{\mathcal{U}_i}}B).\texttt{coerce-equiv}(p)\Big)}\ \mathcal{U}\text{-UNIVALENCE}$$

Figure 2.6: Rules of a faithful матрёшка doll with infinite layers.

1. a function f of type $A \to B$, and
2. its supposed inverse function g of type $B \to A$, and
3. a proof α of type $\prod_{a:A} g(f(a)) =_A a$ showing that g is a left inverse of f, and
4. a proof β of type $\prod_{b:B} f(g(b)) =_B b$ showing that g is a right inverse of f.

This seems to suggest the following (undesirable) definition of equivalence types:

$$A \simeq' B :\equiv \sum_{f:A\to B} \texttt{is-equiv}'(f)$$

$$\texttt{is-equiv}'\big(f{:}A{\to}B\big) :\equiv \sum_{g:B\to A}\left(\left(\prod_{a:A} g(f(a)) =_A a\right) \times \left(\prod_{b:B} f(g(b)) =_B b\right)\right).$$

Unfortunately, this definition works poorly because there could be different equivalence proofs of $A \simeq' B$ for the same function $f : A \to B$. Ideally, we wish to define $A \simeq B$ such that there is at most one choice of g, α and β for the same f; in other words, being an equivalence should be a mathematical property that carries no additional data beyond its existence. One way to achieve this is to demand that α and β are *coherent* in the sense that the following two ways to identify $f(g(f(a)))$ and $f(a)$,

$$\begin{aligned} \texttt{ap}_f(\alpha(a)) &: f(g(f(a))) =_B f(a) \\ \beta(f(a)) &: f(g(f(a))) =_B f(a) \end{aligned}$$

are themselves identified; more precisely, we demand an additional proof ϵ of type

$$\prod_{a:A} \texttt{ap}_f(\alpha(a)) =_{f(g(f(a)))=_B f(a)} \beta(f(a))$$

demonstrating that α and β are coherent. It can be proved that there is a unique choice of g, α, β *and* ϵ for each function f up to identification. Putting these together, here is one working definition of equivalence:

$$A \simeq B :\equiv \sum_{f:A\to B} \texttt{is-equiv}(f)$$

$$\texttt{is-equiv}\big(f{:}A{\to}B\big) :\equiv \sum_{(g:B\to A)} \sum_{(\alpha:\prod_a g(f(a))=a)} \sum_{(\beta:\prod_b f(g(b))=b)} \prod_{a:A} \texttt{ap}_f(\alpha(a)) = \beta(f(a)).$$

This definition is called *half adjoint equivalence* because the coherence condition is one of the two adjunction conditions; see [138, chap. 4] for other good definitions of equivalence.

Remark 2.3.1. Even though $\texttt{is-equiv}'(f)$ is not a good definition of equivalence, it represents the standard recipe of establishing an equivalence: given any g, α and β that may violate the coherence condition, it is possible to tweak only α or only β to satisfy the coherence condition. In practice, we rarely care about the value of α or β, and such repairing is handled by some lemma or some library code.

Remark 2.3.2. A curious fact is that if we explicitly added the other adjunction condition

$$\eta : \prod_{b:B} \texttt{ap}_g(\beta(b)) =_{g(f(g(b)))=_A g(a)} \alpha(g(b))$$

as the sixth component of equivalence, then we would have to worry about the coherence between these two coherence conditions themselves; it seems odd numbers of coherence conditions give rise to the right definitions. This observation has been circulated during the Institute for Advanced Study special year.

2.3.2 Univalence Axiom

The univalence principle, manifested as an axiom, asserts equivalence between equivalence and identification. Note that there is already a canonical function from identification to equivalence; more precisely, there is a coercion function between any two identified types,

$$\texttt{coerce}\big(p{:}A{=}_{\mathcal{U}_i}B\big) : A \to B$$
$$\texttt{coerce}\big(p{:}A{=}_{\mathcal{U}_i}B\big) :\equiv \texttt{elim}_{A=}[x.\_.A \to x](\lambda(y{:}A).y; B; p)$$

which can be shown to be an equivalence by considering the reflexivity case:

$$\texttt{coerce-equiv}\big(p{:}A{=}_{\mathcal{U}_i}B\big) : A \simeq B$$
$$\texttt{coerce-equiv}\big(p{:}A{=}_{\mathcal{U}_i}B\big) :\equiv \langle \texttt{coerce}(p);$$
$$\texttt{elim}_{A=}[x.y.\texttt{is-equiv}(\texttt{coerce}(y))](\texttt{idf-is-equiv}(A); B; p)\rangle$$

$$\texttt{idf-is-equiv}\big(A{:}\mathcal{U}_i\big) : \texttt{is-equiv}(\lambda(x{:}A).x)$$

$$\texttt{idf-is-equiv}\big(A{:}\mathcal{U}_i\big) :\equiv \Big\langle \lambda x.x; \Big\langle \lambda x.\texttt{refl}_x; \langle \lambda x.\texttt{refl}_x; \lambda x.\texttt{refl}_{\texttt{refl}_x}\rangle \Big\rangle \Big\rangle.$$

The univalence axiom then states that the function `coerce-equiv` : $(A = B) \to A \simeq B$ is itself an equivalence for any types A and B in $\mathcal{U}_i$:

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i}{\Gamma \vdash \texttt{univalence}_i(A; B) : \texttt{is-equiv}\Big(\lambda(p{:}A =_{\mathcal{U}_i} B).\texttt{coerce-equiv}(p)\Big)}\ \mathcal{U}\text{-}\textsc{univalence}\ .$$

Practically, the most used part of the axiom is the inverse function of `coerce-equiv`, which recognizes equivalence as identification, and the application of this inverse function is commonly noted as "by the univalence axiom". However, it should be clear now that the univalence axiom means more than the existence of a function from equivalence to identification.

2.4 Inductive Types

An *inductive type* categorizes inductively defined structures generated by a finite collection of constructors and it admits inductive reasoning. For example, coproduct types $A + B$ are freely generated by the two injections from A and B, the empty type is generated by no constructors, and the type of natural numbers is generated by the "zero" and "successor" constructors; Booleans, finite lists and many others are also examples of inductive types, and they undoubtedly play an important role in mathematics. Proving theorems about these types involves case analysis on their generators, which is called *induction.* Each inductive type can also be formalized in the same way as we did for sum types, function types, the unit type and identification types in figs. 2.1 to 2.4; as examples, the rules of coproduct types, the empty type and the natural number type are shown in figs. 2.7 to 2.9.

The rules of the empty type in fig. 2.8 are somewhat degenerate because there are no introduction rules and thus no computation rules. On the other hand, the presentation of natural numbers in fig. 2.9 is vastly more complicated due to their recursive nature: To prove a theorem about natural numbers (that is, using the eliminator of $\mathbb{N}$), one considers the zero (`zero`) case and the successor (`succ`) case, for every natural number must be generated by either `zero` or `succ`. This is commonly known as *mathematical induction.* Moreover, in the case of `succ`, we shall be able to access the proof for the predecessor, and thus the eliminator has to recur, as shown in the computation rule for `succ` in fig. 2.9. Due to this the eliminator of an inductive type such as $\mathbb{N}$, written $\texttt{elim}_{\mathbb{N}}[x.C](c_{\texttt{zero}}; x.y.c_{\texttt{succ}}; n)$, is sometimes called a *recursor.* It is the recursive nature of `succ` that adds complexity to these formal rules.

A general schema for inductive types and their extensions has been extensively studied [18, 43, 47, 49, 99, 108, 111] but is out of the scope of this short introduction. The key point to remember is that almost all imaginable inductive structures can be defined following the same pattern of the rules of natural numbers.

Remark 2.4.1. A common misconception is that the $\mathcal{W}$-type [102], which represents well-founded trees, can encode and thus replace these inductive types. It is only a half-truth:

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma \vdash B : \mathcal{U}}{\Gamma \vdash A + B : \mathcal{U}}\ \textsc{+-formation} \qquad \frac{\Gamma \vdash B : \mathcal{U} \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathtt{inl}(a) : A + B}\ \textsc{+-introduction}\ (\mathtt{inl})$$

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathtt{inr}(b) : A + B}\ \textsc{+-introduction}\ (\mathtt{inr})$$

$$\frac{\begin{array}{c}\Gamma, x{:}A{+}B \vdash D : \mathcal{U} \qquad \Gamma, x{:}A \vdash d_{\mathtt{inl}} : D[\mathtt{inl}(x)/x] \\ \Gamma, x{:}B \vdash d_{\mathtt{inr}} : D[\mathtt{inr}(x)/x] \qquad \Gamma \vdash c : A + B\end{array}}{\Gamma \vdash \mathtt{elim}_{+}[x.D](x.d_{\mathtt{inl}}; x.d_{\mathtt{inr}}; c) : D[c/x]}\ \textsc{+-elimination}$$

$$\frac{\begin{array}{c}\Gamma, x{:}A{+}B \vdash D : \mathcal{U} \qquad \Gamma, x{:}A \vdash d_{\mathtt{inl}} : D[\mathtt{inl}(x)/x] \\ \Gamma, x{:}B \vdash d_{\mathtt{inr}} : D[\mathtt{inr}(x)/x] \qquad \Gamma \vdash a : A\end{array}}{\Gamma \vdash \mathtt{elim}_{+}[x.D](x.d_{\mathtt{inl}}; x.d_{\mathtt{inr}}; \mathtt{inl}(a)) \equiv d_{\mathtt{inl}}[a/x] : D[\mathtt{inl}(a)/x]}\ \textsc{+-computation}\ (\mathtt{inl})$$

$$\frac{\begin{array}{c}\Gamma, x{:}A{+}B \vdash D : \mathcal{U} \qquad \Gamma, x{:}A \vdash d_{\mathtt{inl}} : D[\mathtt{inl}(x)/x] \\ \Gamma, x{:}B \vdash d_{\mathtt{inr}} : D[\mathtt{inr}(x)/x] \qquad \Gamma \vdash b : B\end{array}}{\Gamma \vdash \mathtt{elim}_{+}[x.D](x.d_{\mathtt{inl}}; x.d_{\mathtt{inr}}; \mathtt{inr}(b)) \equiv d_{\mathtt{inr}}[b/x] : D[\mathtt{inr}(b)/x]}\ \textsc{+-computation}\ (\mathtt{inr})$$

Figure 2.7: Rules of being constructive or non-constructive.

$$\frac{\Gamma\ \mathtt{ctx}}{\Gamma \vdash \mathbb{0} : \mathcal{U}_0}\ \mathbb{0}\textsc{-formation} \qquad \frac{\Gamma, x{:}\mathbb{0} \vdash C : \mathcal{U}_i \qquad \Gamma \vdash e : \mathbb{0}}{\Gamma \vdash \mathtt{elim}_{\mathbb{0}}[x.C](e) : C[e/x]}\ \mathbb{0}\textsc{-elimination}$$

Figure 2.8: There are zero rules.

$$\frac{\Gamma\ \texttt{ctx}}{\Gamma \vdash \mathbb{N} : \mathcal{U}_0}\ \mathbb{N}\text{-FORMATION} \qquad\qquad \frac{\Gamma\ \texttt{ctx}}{\Gamma \vdash \texttt{zero} : \mathbb{N}}\ \mathbb{N}\text{-INTRODUCTION}\ (\texttt{zero})$$

$$\frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \texttt{succ}(n) : \mathbb{N}}\ \mathbb{N}\text{-INTRODUCTION}\ (\texttt{succ})$$

$$\frac{\begin{array}{c}\Gamma, x{:}\mathbb{N} \vdash C : \mathcal{U}_i \\ \Gamma \vdash c_{\texttt{zero}} : C[\texttt{zero}/x] \qquad \Gamma, x{:}\mathbb{N}, y{:}C \vdash c_{\texttt{succ}} : C[\texttt{succ}(x)/x] \qquad \Gamma \vdash n : \mathbb{N}\end{array}}{\Gamma \vdash \texttt{elim}_{\mathbb{N}}[x.C](c_{\texttt{zero}}; x.y.c_{\texttt{succ}}; n) : C[n/x]}\ \mathbb{N}\text{-ELIMINATION}$$

$$\frac{\begin{array}{c}\Gamma, x{:}\mathbb{N} \vdash C : \mathcal{U}_i \\ \Gamma \vdash c_{\texttt{zero}} : C[\texttt{zero}/x] \qquad \Gamma, x{:}\mathbb{N}, y{:}C \vdash c_{\texttt{succ}} : C[\texttt{succ}(x)/x]\end{array}}{\Gamma \vdash \texttt{elim}_{\mathbb{N}}[x.C](c_{\texttt{zero}}; x.y.c_{\texttt{succ}}; \texttt{zero}) \equiv c_{\texttt{zero}} : C[\texttt{zero}/x]}\ \mathbb{N}\text{-COMPUTATION}\ (\texttt{zero})$$

$$\frac{\begin{array}{c}\Gamma, x{:}\mathbb{N} \vdash C : \mathcal{U}_i \qquad \Gamma \vdash c_{\texttt{zero}} : C[\texttt{zero}/x] \\ \Gamma, x{:}\mathbb{N}, y{:}C \vdash c_{\texttt{succ}} : C[\texttt{succ}(x)/x] \qquad \Gamma \vdash n : \mathbb{N}\end{array}}{\begin{array}{l}\Gamma \vdash\ \texttt{elim}_{\mathbb{N}}[x.C](c_{\texttt{zero}}; x.y.c_{\texttt{succ}}; \texttt{succ}(n)) \\ \qquad \equiv c_{\texttt{succ}}[n, \texttt{elim}_{\mathbb{N}}[x.C](c_{\texttt{zero}}; x.y.c_{\texttt{succ}}; n)/x, y] : C[\texttt{succ}(n)/x]\end{array}}\ \mathbb{N}\text{-COMPUTATION}\ (\texttt{succ})$$

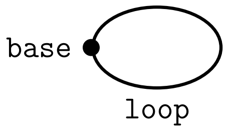
Figure 2.9: Rules of betrayal of revisionism.

in *the* Martin-Löf type theory we consider, the encodings of inductive types in $\mathcal{W}$-type would not validate all judgmental equality rules we expect from the general schema, not even for natural numbers. See [14, 48, 97, 129] for more discussions.

2.5 Higher Inductive Types

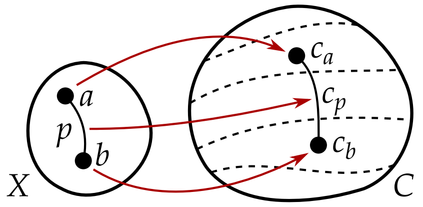
One limitation of inductive types is that they only have trivial higher-dimensional structures; with the identification-as-path interpretation, it is natural to ask how to present in type theory some basic spaces with non-trivial higher-dimensional structures from homotopy theory. Higher inductive types are one promising answer.

An (ordinary) inductive type, as discussed in section 2.4, is a type defined by element generators. For example, the two generators of natural numbers, `zero` and `succ`, all work on elements. Higher inductive types generalize ordinary inductive types by allowing not only element generators but also generators for identification between elements, identification between identification proofs, or more iterated identification. In other words, we may stipulate generators for the stacked relations of the higher-dimensional types being defined. The simplest example is the circle ($\mathbb{S}^1$), which can be characterized by a point and a loop illustrated in this diagram:



The `base` is an element generator and the `loop` is a generator for the identification between `base` and itself. We can also formalize such a higher inductive type as rules, but only after defining the following two pieces of apparatus: *identification over an identification* and *dependent application to identification.* Identification over an identification is for the elimination rule, and dependent application is for the computation rules.

Identification over an identification [89]. Imagine a higher inductive type X with two element generators a and b and an identification generator p of type $a =_X b$, and we are interested in some theorem C about X, which is formally a family of types indexed by the higher inductive type X.



Following the schema for inductive types, if we wish to prove the theorem C, the inputs to the supposed eliminator $\texttt{elim}_X[x.C(x)](c_a; c_b; c_p; x)$ should include these three components:

1. c_a of type $C(a)$ for the generator a, and
2. c_b of type $C(b)$ for the generator b, and
3. c_p of type **??** for the generator p.

What should be the type of c_p? The element c_p functions as a heterogeneous identification between c_a and c_b of different types, and thus what we are looking for is an identification type between c_a and c_b *over the base identification p*. This can actually be defined by applying the identification eliminator to p; the key observation is that the heterogeneous identification reduces to an ordinary one when the base identification p is reflexivity. Formally, let us write $\alpha =^{x.B}_{\gamma} \beta$ as the heterogeneous identification type between α of type $B[u/a]$ and β of type $B[v/x]$ across the family $\lambda(x{:}A).B$ over the base identification γ of type $u =_A v$; the type can be defined as follows:

$$\alpha =^{x.B}_{\gamma} \beta :\equiv \texttt{elim}_{u=}[x.\_.B \to \mathcal{U}_i](\lambda\beta.(\alpha =_{B[u/x]} \beta); v; \gamma)(\beta)$$

which states that the type reduces to an ordinary identification type when γ is reflexivity. Now we can say the type of the input c_p should be $c_a =^{x.C(x)}_{p} c_b$. See [89] for further generalizations of this idea to higher dimensions.

Dependent application to identification. Once the eliminator is defined with identification over an identification, the next step is to specify its accompanying computation rules for all generators. The value of the eliminator of an element generator can be simply referred to as function application, but that of an identification generator demands some operator to make reference to its functorial behavior. Here I write $\texttt{apd}_f(p)$ as the result of one-dimensional application of $f : \prod_{x:A} B$ to the base identification p of type $a =_A b$, which will be of type $f(a) =^{x.B}_{p} f(b)$. Following the above example about proving the theorem C of the higher inductive type X, we wish for the following rule:

$$\Gamma \vdash \texttt{apd}_{\lambda x.\texttt{elim}_X[x.C(x)](c_a;c_b;c_p;x)}(p) \equiv c_p : c_a =^{x.C(x)}_{p} c_b.$$

It is actually unclear whether this judgmental equality rule should be included due to theoretical and practical concerns. In any case, before continuing the discussion I should finish the definition of the operator apd:

$$\texttt{apd}_f(p) :\equiv \texttt{elim}_{a=}[x.y.f(a) =^{x.B}_{y} f(x)](\texttt{refl}_{f(a)}; b; p).$$

With heterogeneous identification and dependent application at hand, we are ready to formalize the circle as rules in fig. 2.10. Note that the computation rule for loop, unlike

$$\frac{\Gamma\ \texttt{ctx}}{\Gamma \vdash \mathbb{S}^1 : \mathcal{U}_0}\ \mathbb{S}^1\textsc{-formation} \qquad \frac{\Gamma\ \texttt{ctx}}{\Gamma \vdash \texttt{base} : \mathbb{S}^1}\ \mathbb{S}^1\textsc{-introduction}\ (\texttt{base})$$

$$\frac{\Gamma\ \texttt{ctx}}{\Gamma \vdash \texttt{loop} : \texttt{base} =_{\mathbb{S}^1} \texttt{base}}\ \mathbb{S}^1\textsc{-introduction}\ (\texttt{loop})$$

$$\frac{\begin{array}{c}\Gamma, x{:}\mathbb{S}^1 \vdash C : \mathcal{U}_i \\ \Gamma \vdash c_{\texttt{base}} : C[\texttt{base}/x] \qquad \Gamma \vdash c_{\texttt{loop}} : c_{\texttt{base}} =^{x.C}_{\texttt{loop}} c_{\texttt{base}} \qquad \Gamma \vdash s : \mathbb{S}^1\end{array}}{\Gamma \vdash \texttt{elim}_{\mathbb{S}^1}[x.C](c_{\texttt{base}}; c_{\texttt{loop}}; s) : C[s/x]}\ \mathbb{S}^1\textsc{-elimination}$$

$$\frac{\begin{array}{c}\Gamma, x{:}\mathbb{S}^1 \vdash C : \mathcal{U}_i \\ \Gamma \vdash c_{\texttt{base}} : C[\texttt{base}/x] \qquad \Gamma \vdash c_{\texttt{loop}} : c_{\texttt{base}} =^{x.C}_{\texttt{loop}} c_{\texttt{base}}\end{array}}{\Gamma \vdash \texttt{elim}_{\mathbb{S}^1}[x.C](c_{\texttt{base}}; c_{\texttt{loop}}; \texttt{base}) \equiv c_{\texttt{base}} : C[\texttt{base}/x]}\ \mathbb{S}^1\textsc{-computation}\ (\texttt{base})$$

$$\frac{\begin{array}{c}\Gamma, x{:}\mathbb{S}^1 \vdash C : \mathcal{U}_i \\ \Gamma \vdash c_{\texttt{base}} : C[\texttt{base}/x] \qquad \Gamma \vdash c_{\texttt{loop}} : c_{\texttt{base}} =^{x.C}_{\texttt{loop}} c_{\texttt{base}}\end{array}}{\Gamma \vdash \texttt{loop}_{\beta} : \texttt{apd}_{\lambda x.\texttt{elim}_{\mathbb{S}^1}[x.C](c_{\texttt{base}}; c_{\texttt{loop}}; x)}(\texttt{loop}) =_{c_{\texttt{base}} =^{x.C}_{\texttt{loop}} c_{\texttt{base}}} c_{\texttt{loop}}}\ \mathbb{S}^1\textsc{-computation}\ (\texttt{loop})$$

Figure 2.10: Rules of 360 degrees.

the one for base, is stated as identification instead of judgmental equality due to various (temporary) concerns. The version with judgmental equality would be the following:

$$\frac{\begin{array}{c}\Gamma, x{:}\mathbb{S}^1 \vdash C : \mathcal{U}_i \\ \Gamma \vdash c_{\texttt{base}} : C[\texttt{base}/x] \qquad \Gamma \vdash c_{\texttt{loop}} : c_{\texttt{base}} =^{x.C}_{\texttt{loop}} c_{\texttt{base}}\end{array}}{\Gamma \vdash \texttt{apd}_{\lambda x.\texttt{elim}_{\mathbb{S}^1}[x.C](c_{\texttt{base}};c_{\texttt{loop}};x)}(\texttt{loop}) \equiv c_{\texttt{loop}} : c_{\texttt{base}} =^{x.C}_{\texttt{loop}} c_{\texttt{base}}}\ \mathbb{S}^1\text{-COMPUTATION } (\texttt{loop})$$

One theoretical concern is that there are actually many ways to define heterogeneous identification and apd, and there was no good reason for computation rules to favor particular definitions. However, progress in modeling higher inductive types such as [96] may eventually remove these doubts. Another practical concern is that no popular proof assistants (notably Agda and Coq) were able to support such judgmental equality for type checking and the current ones adopted the more conservative presentation. Again, this obstacle may eventually be lifted by new computer software, for example the new Agda features discussed in section 4.1.6. See [138, chap. 6] for a more detailed discussion about the hesitation to accept the judgmental equality rule. See also [4, 129] for efforts in giving general syntax of higher inductive types.

Note that, if the motive of an eliminator (C in $\texttt{elim}[x.C](\ldots)$) actually does not depend on the higher inductive type, which is common when building a family of types indexed by that type, then there is a derivable, much simpler eliminator without using dependent application apd or identification over an identification. Take the circle $\mathbb{S}^1$ for example, the following rules are admissible:

$$\frac{\begin{array}{c}\Gamma \vdash C : \mathcal{U}_i \\ \Gamma \vdash c_{\texttt{base}} : C \qquad \Gamma \vdash c_{\texttt{loop}} : c_{\texttt{base}} =_C c_{\texttt{base}} \qquad \Gamma \vdash s : \mathbb{S}^1\end{array}}{\Gamma \vdash \texttt{elim-nd}_{\mathbb{S}^1}[C](c_{\texttt{base}};c_{\texttt{loop}};s) : C}\ \mathbb{S}^1\text{-ELIMINATION (NON-DEPENDENT)}$$

$$\frac{\Gamma \vdash C : \mathcal{U}_i \qquad \Gamma \vdash c_{\texttt{base}} : C \qquad \Gamma \vdash c_{\texttt{loop}} : c_{\texttt{base}} =_C c_{\texttt{base}}}{\Gamma \vdash \texttt{elim-nd}_{\mathbb{S}^1}[C](c_{\texttt{base}};c_{\texttt{loop}};\texttt{base}) \equiv c_{\texttt{base}} : C}\ \mathbb{S}^1\text{-COMP. (NON-DEP.) } (\texttt{base})$$

$$\frac{\Gamma \vdash C : \mathcal{U}_i \qquad \Gamma \vdash c_{\texttt{base}} : C \qquad \Gamma \vdash c_{\texttt{loop}} : c_{\texttt{base}} =_C c_{\texttt{base}}}{\Gamma \vdash \texttt{loop-nd}_\beta : \texttt{ap}_{\lambda x.\texttt{elim-nd}_{\mathbb{S}^1}[C](c_{\texttt{base}};c_{\texttt{loop}};x)}(\texttt{loop}) =_{c_{\texttt{base}} =_C c_{\texttt{base}}} c_{\texttt{loop}}}\ \mathbb{S}^1\text{-COMP. (NON-DEP.) } (\texttt{loop})$$

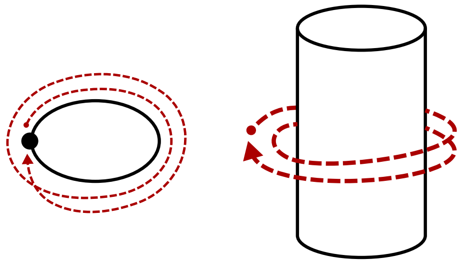
Due to its simplicity, non-dependent elimination for higher inductive types is used whenever possible.

The derivation of non-dependent eliminators from general ones is based on the following two equations, which state that heterogeneous identification and dependent application reduce to ordinary ones when the family is constant:

$$(u =^{\_.B}_{p} v) = (u =_B v)$$
$$\texttt{apd}_{f:A\to B}(p) = \texttt{ap}_f(p)$$

which can be proved by identification elimination on p. In practice, however, it makes little difference whether the non-dependent eliminator is derived or not.

To see what we can do with the circle ($\mathbb{S}^1$) defined above, let's count all identifications between `base` and itself, which under the identification-as-path interpretation are paths from `base` to itself. A good metaphor of such a path is a rope circling around a pillar; for example, the path on the left, below, can be regarded as the rope on the right. Our magical rope can cross itself but not the pillar, and two configurations are identified if one can be transformed into the other without cutting the pillar.



It is then equivalent to count distinct ways to tie a pillar. A key observation is that clockwise and counterclockwise loops will cancel each other after pulling the rope tight, and the remaining part must be composed of only clockwise loops or only counterclockwise ones. Hence, we can establish a correspondence to integers based on the final configurations of rope tying: staying at `base` (leaving the pillar untouched) corresponds to zero, circling clockwisely n times corresponds to positive n, and circling counterclockwisely n times corresponds to negative n. (This integer representation is called the *winding number*.) Let $A \simeq B$ be the equivalence type between types A and B. This intuitive theorem can be formally stated as

$$(\texttt{base} =_{\mathbb{S}^1} \texttt{base}) \simeq \mathbb{Z}$$

and is one of the early milestones of UniTT+HIT [93]. This also lends some justification to the formal rules in fig. 2.10.

Many more higher inductive types can be defined in the same manner and a wide range of theorems have been proved and mechanized. More can be found in chapter 3 and the book *Homotopy Type Theory: Univalent Foundations for Mathematics* [138]. While a precise criterion of higher inductive types remains open, all instances in this thesis are widely accepted as essential. See section 3.6 for a summary of the use of higher inductive types in my thesis. In general, the research so far has affirmed that higher inductive types are capable of capturing many important homotopy-theoretic concepts, which would be difficult or impossible without them.

Chapter 3

Homotopy Theory in Univalent Type Theory

The introduction of this chapter incorporates some text from my manuscript [69].

Many components in the univalent type theory with higher inductive types (UniTT+HIT) have homotopy-theoretic reading, following the *identification-as-path* interpretation briefly described in section 2.2 summarized in table 3.1. In this chapter I will demonstrate more how UniTT+HIT may capture homotopy-theoretic concepts.

Using the interpretation, a wide range of homotopy-theoretic results have been developed in UniTT+HIT and mechanized in proof assistants such as AGDA [29, 109], COQ [19, 134, 135] and LEAN [106], for example homotopy groups of spheres [27, 88, 93, 138], the Seifert–van Kampen theorem [72], the Eilenberg–Mac Lane spaces [91], the Blakers–Massey theorem [71], the Mayer–Vietoris sequences [35], the Cayley–Dickson construction [32], the double groupoids [115] and many more [89, 118, 138].

Over the years I have contributed to this program the following results:

- covering spaces in UniTT+HIT and AGDA (joint work with Robert Harper with help from Guillaume Brunerie, Daniel R. Grayson and Chris Kapulkin) [69]; and
- the Seifert–van Kampen theorem in AGDA (based on the proof by Michael Shulman in UniTT+HIT) [72]; and
- the Blakers–Massey theorem in AGDA (based on the proof by Peter LeFanu Lumsdaine, Eric Finster and Dan Licata in UniTT+HIT) [71]; and
- a lemma for reformulating ordinary Eilenberg–Steenrod cohomology theories in UniTT+HIT and AGDA (joint work with Ulrik Buchholtz).

All these examples show that UniTT+HIT, as a higher-dimensional type theory, is a promising framework for mechanizing homotopy theory, which constitutes a strong evidence for my thesis statement; see section 3.6 for a reflection on the use of the new features introduced in UniTT+HIT. The new techniques presented in our work on the

	Type theory	Homotopy theory
A	type	space
$a : A$	element	point
$f : A \to B$	function	continuous mapping
$A \to B$	arrow type	function space
$A \to \mathcal{U}$	family of types	fibration
$B(a)$	instance of a family of types	fiber
$b(x) : B(x)$	conditional element	section
$\sum_{x:A} B(x)$	sum type	total space
$\prod_{x:A} B(x)$	function type	space of sections
$a =_A b$	identification	path

Table 3.1: The identification-as-path interpretation.

Blakers–Massey theorem even sparked new research back in the community of mathematics [8, 117] as mentioned in chapter 1; this shows that the new framework is not only suitable for mechanizing existing proofs but also for inventing new ones.

My work has been surrounding two important constructs in homotopy theory: *homotopy groups* and *cohomology groups;* both are powerful algebraic tools to distinguish homotopically different spaces. Homotopy groups are about the ways to fold the n-dimensional sphere into a space, and cohomology groups are about the maps from n-dimensional cycles in a space to an abelian group, where the nth homotopy or cohomology group refers to the group for the dimension n. The first homotopy groups, which are also called *fundamental groups,* are fairly well understood, but higher homotopy groups are in general extremely difficult to compute; this difficulty partly led to the popularity of cohomology groups, which are usually easier to calculate for the spaces of interest.

As we will see, covering spaces can be succinctly expressed in UniTT+HIT and have deep connection with fundamental groups; the Seifert–van Kampen enables a divide-and-conquer technique to compute the fundamental group of a larger space from those of smaller ones; the Blakers–Massey theorem provides one of the few available devices to calculate higher homotopy groups. These three results represent important instruments for homotopy groups in the classical theory.

The last result takes the newer direction to expand this collaborative program to cohomology theory, which still remains largely unexplored in UniTT+HIT. Ulrik Buchholtz's and my contribution is a step toward linking several different cohomology theories in UniTT+HIT.

This chapter is organized as follows: section 3.1 will cover the basics of homotopy theory in UniTT+HIT, sections 3.2 to 3.5 will go through my four pieces of work, and finally sections 3.6 and 3.7 provide some ending thoughts.

Remark 3.1. I choose to present my contribution in UniTT+HIT instead of mechanized proofs because of a more beautiful typesetting in XƎLaTeX when mixed with English words in proportional fonts. Nonetheless, see chapter 4 for a direct translation of all notations in this thesis to AGDA code. There is no intrinsic reason beyond aesthetic consideration to present my work in UniTT+HIT, because I always start working in proof assistants such as AGDA for this thesis. This is possible because of the synthetic nature of the identification-as-path interpretation.

Remark 3.2. Terminology and notation in this chapter will be closer to the AGDA mechanization than the book *Homotopy Type Theory: Univalent Foundations for Mathematics* to better connect the mechanized proofs. A notable exception is $\mathtt{refl}_x$, which corresponds to `idp` (identity path) in the AGDA mechanization. This unfortunate mismatch is due to the timing that reflexivity identification was introduced before the identification-as-path interpretation, and the AGDA mechanization starts out with emphasis on homotopy theory.

3.1 Basics

This section incorporates some text from my manuscript [69].

This goal of this section is to prepare readers for the presentation of the homotogy-theoretic results in following sections. I will also review some notational conventions adopted in this chapter.

3.1.1 Sums and Records

Nested sum types will be presented as records types with labels (like `label`). A label is also used as the projection function that projects out the corresponding component from a record. Record types are directly supported in AGDA and are often more desirable than the raw, nested sum types. The notational convention adopted here matches the AGDA practice perfectly. See chapter 4 for more discussion.

3.1.2 A Short Note on Functions

Multi-argument application $f(a_1)(a_2) \dots (a_n)$ is written $f(a_1, a_2, \dots, a_n)$. Moreover, the parentheses may be omitted if the argument already comes with delimiters (such as $f\langle a; b\rangle$). Nested function types with the same domain such as $\prod_{a:A}\prod_{b:A} B$ may be abbreviated as $\prod_{a,b:A} B$; similarly $\lambda(x,y{:}A).b$ stands for $\lambda(x{:}A).\lambda(y{:}A).b$.

3.1.3 Identification Revisited

The concatenation is (in the diagram order) written $p \cdot q$ and the inverse identification as p^{-1}. These functions can be easily defined by identification elimination:

$$\begin{aligned} p \cdot q &:\equiv \mathtt{elim}_{=b}[x.\_.x =_A c](q; a; p) \\ p^{-1} &:\equiv \mathtt{elim}_{a=}[x.\_.x =_A a](\mathtt{refl}_a; b; p) \end{aligned}$$

and they satisfy all groupoid laws.

For a family of types B indexed by a type A, an identification $p : a =_A b$ will force an identification and an equivalence between corresponding fibers $B(a)$ and $B(b)$. The identification, written $\mathsf{ap}_f(p)$, is already defined in eq. (2.1) on page 16; the accompanying equivalence (as a function) is called *transport*, written $\mathtt{transport}[x.B(x)](p; u)$, meaning the transport of $u : B(a)$ along $p : a =_A b$ across the family B to the fiber $B(b)$; formally,

$$\begin{aligned} &\mathtt{transport}[x.B(x)](p; u) : B(b) \\ &\mathtt{transport}[x.B(x)](p; u) :\equiv \mathtt{coerce}(\mathsf{ap}_f(p))(u). \end{aligned}$$

It is also functorial in p in the sense that it preserves reflexivity and concatenation.

Note that there are actually two alternative definitions of heterogeneous identification introduced in section 2.5 based on transport:

$$\begin{aligned} u =_p^{x.B} v &:\equiv' \mathtt{transport}[x.B](p; u) = v \\ u =_p^{x.B} v &:\equiv'' u = \mathtt{transport}[x.B](p^{-1}; v). \end{aligned}$$

The intuition is to decompose such an identification into the "horizontal" part induced by the base identification and the "vertical" part within the same fiber, as visualized in fig. 3.1. All these definitions (including the one in this thesis) are equivalent to each other. In practice, the asymmetric nature of transports often clouds the essence of heterogeneous identification and it is best to keep the notion abstract. That said, we will need this conversion function later in this chapter:

$$\mathtt{to\text{-}transp} : u =_p^{x.B} v \to \mathtt{transport}[x.B](p; u) = v.$$

3.1.4 A Very Short Note on Universe Levels

In this chapter I will suppress all universe levels, pretending there is only one universe written $\mathcal{U}$.

3.1.5 Truncation Levels

Truncation levels denote the dimension *above which* a type is trivial: a type is at level -2 if it is *contractible,* which means it is equivalent to the unit type and is trivial at all dimensions;

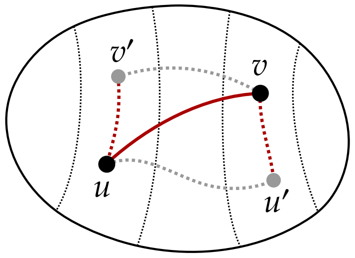


Figure 3.1: Heterogeneous identification and transports.

The element u' is u transported to the fiber the element v lies in, and the element v' vice versa.

a type is at level $(n + 1)$ if its identification types lie at level n. A type at level -1 is called a *mere proposition*, where any two elements are identified, and a type at level 0 is called a *set*, where any two parallel identifications are identified. Equivalences between sets are called *isomorphisms*. It can be shown that types at different truncation levels form a cumulative hierarchy, in addition to the existing one based on their universe levels (which are suppressed in this chapter).

Formally, a new type of truncation levels, `TLevel`, is introduced; see fig. 3.2. It is essentially natural numbers but starting at −2. For convenience, we will overload the numeric literals to represent natural numbers, truncation levels and integers, and assume an obvious implementation of addition written +. We can then define all the properties mentioned above as follows: $\texttt{is-contr}(A)$ means that the type A is contractible in the sense that there is a center from which there is an identification to any element, and $\texttt{has-level}_n(A)$ means the type A has truncation level n.

$$\texttt{is-contr}(A{:}\mathcal{U}) :\equiv \sum_{a:A} \prod_{x:A} a =_A x.$$

$$\texttt{has-level}_n(A{:}\mathcal{U}) :\equiv \texttt{elim}_{\texttt{TLevel}}[\_.\mathcal{U}](\texttt{is-contr}(A); x.y. \prod_{a:A} \prod_{b:A} \texttt{has-level}_x(a = b); n)$$

$$\texttt{is-prop}(A{:}\mathcal{U}) :\equiv \texttt{has-level}_{-1}(A) \equiv \prod_{x:A} \prod_{y:A} \texttt{is-contr}(x =_A y)$$

$$\texttt{is-set}(A{:}\mathcal{U}) :\equiv \texttt{has-level}_0(A) \equiv \prod_{x:A} \prod_{y:A} \texttt{is-prop}(x =_A y).$$

Remark 3.1.1. The overloading of numeric literals perfectly matches our Agda mechanization, but the addition of truncation levels in mechanized proofs is

$$m \mathbin{\hat{+}} n :\equiv (m + n) + 2$$

instead of the plain + to prevent overflowing; for example $(-2) + (-2)$ would fall out of range. However, in this chapter, I will still use the traditional + to better match the classical theory. This discrepancy will affect lemma 3.1.3 and theorem 3.4.1.

An *n-type* is a type at truncation level n. The type `Set` is the type of all 0-types, or $\sum_{A:\mathcal{U}} \texttt{is-set}(A)$. An *n-truncation* of a type A is, intuitively, the *best* n-type approximation

$$\frac{\Gamma\ \mathtt{ctx}}{\Gamma \vdash \mathtt{TLevel} : \mathcal{U}}\ \mathtt{TLevel}\text{-}\textsc{formation} \qquad \frac{\Gamma\ \mathtt{ctx}}{\Gamma \vdash -2 : \mathtt{TLevel}}\ \mathtt{TLevel}\text{-}\textsc{introduction}\ (-2)$$

$$\frac{\Gamma \vdash t : \mathtt{TLevel}}{\Gamma \vdash \mathtt{succ}(t) : \mathtt{TLevel}}\ \mathtt{TLevel}\text{-}\textsc{introduction}\ (\mathtt{succ})$$

$$\frac{\begin{array}{c}\Gamma, x{:}\mathtt{TLevel} \vdash C : \mathcal{U} \qquad \Gamma \vdash c_{-2} : C[-2/x] \\ \Gamma, x{:}\mathtt{TLevel}, y{:}C \vdash c_{\mathtt{succ}} : C[\mathtt{succ}(x)/x] \qquad \Gamma \vdash t : \mathtt{TLevel}\end{array}}{\Gamma \vdash \mathtt{elim}_{\mathtt{TLevel}}[x.C](c_{-2}; x.y.c_{\mathtt{succ}}; t) : C[t/x]}\ \mathtt{TLevel}\text{-}\textsc{elimination}$$

$$\frac{\begin{array}{c}\Gamma, x{:}\mathtt{TLevel} \vdash C : \mathcal{U} \\ \Gamma \vdash c_{-2} : C[-2/x] \qquad \Gamma, x{:}\mathtt{TLevel}, y{:}C \vdash c_{\mathtt{succ}} : C[\mathtt{succ}(x)/x]\end{array}}{\Gamma \vdash \mathtt{elim}_{\mathtt{TLevel}}[x.C](c_{-2}; x.y.c_{\mathtt{succ}}; -2) \equiv c_{-2} : C[-2/x]}\ \mathtt{TLevel}\text{-}\textsc{computation}\ (-2)$$

$$\frac{\begin{array}{c}\Gamma, x{:}\mathtt{TLevel} \vdash C : \mathcal{U} \qquad \Gamma \vdash c_{-2} : C[-2/x] \\ \Gamma, x{:}\mathtt{TLevel}, y{:}C \vdash c_{\mathtt{succ}} : C[\mathtt{succ}(x)/x] \qquad \Gamma \vdash t : \mathtt{TLevel}\end{array}}{\begin{array}{l}\Gamma \vdash\ \mathtt{elim}_{\mathtt{TLevel}}[x.C](c_{-2}; x.y.c_{\mathtt{succ}}; \mathtt{succ}(t)) \\ \qquad \equiv c_{\mathtt{succ}}[t, \mathtt{elim}_{\mathtt{TLevel}}[x.C](c_{-2}; x.y.c_{\mathtt{succ}}; t)/x, y] : C[\mathtt{succ}(t)/x]\end{array}}\ \mathtt{TLevel}\text{-}\textsc{comp.}\ (\mathtt{succ})$$

Figure 3.2: Revival of non-naturalism.

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma \vdash n : \texttt{TLevel}}{\Gamma \vdash \|A\|_n : \mathcal{U}} \ \|-\|\text{-}\textsc{formation}$$

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma \vdash n : \texttt{TLevel}}{\Gamma \vdash \|A\|_n\texttt{-level} : \texttt{has-level}_n\|A\|_n} \ \|-\|\text{-}\textsc{formation (leval)}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash n : \texttt{TLevel}}{\Gamma \vdash |a|_n : \|A\|_n} \ \|-\|\text{-}\textsc{introduction}$$

$$\frac{\begin{array}{c}\Gamma, x{:}\|A\|_n \vdash C : \mathcal{U} \qquad \Gamma, x{:}\|A\|_n \vdash C\texttt{-level} : \texttt{has-level}_n(C(x)) \\ \Gamma, x{:}A \vdash c_{|-|} : C[|x|_n/x] \qquad \Gamma \vdash t : \|A\|_n\end{array}}{\Gamma \vdash \texttt{elim}_{\|-\|}[x.C; x.C\texttt{-level}](x.c_{|-|}; t) : C[t/x]} \ \|-\|\text{-}\textsc{elimination}$$

$$\frac{\begin{array}{c}\Gamma, x{:}\|A\|_n \vdash C : \mathcal{U} \qquad \Gamma, x{:}\|A\|_n \vdash C\texttt{-level} : \texttt{has-level}_n(C(x)) \\ \Gamma, x{:}A \vdash c_{|-|} : C[|x|_n/x] \qquad \Gamma \vdash a : A\end{array}}{\Gamma \vdash \texttt{elim}_{\|-\|}[x.C; x.C\texttt{-level}](x.c_{|-|}; |a|_n) \equiv c_{|-|}[a/x] : C[|a|_n/x]} \ \|-\|\text{-}\textsc{computation}$$

Figure 3.3: Rules of the least truncated.

of the type A, written $\|A\|_n$, where the projection of $a : A$ into the truncation is written $|a|_n$. See fig. 3.3 for the formal rules of truncation; the truncation $\|A\|_n$ is an n-type with the universal property that there is a unique extension of any function of type $A \to B$ to $\|A\|_n$ for any n-type B, as shown in the following diagram. The n-truncation of an n-type is equivalent to the n-type itself.

$$\begin{tikzcd} A \arrow[r] \arrow[d, "|-|_n"'] & B \\ \|A\|_n \arrow[ur, dashed] & \end{tikzcd} \quad \text{for any } n\text{-type B.}$$

A (-1)-truncation of a type matches the notion of existence in the classical logic, representing the idea that we know the existence of a proof but not its content. The existence of (-1)-truncation in UniTT+HIT makes it trivial to mimic proof-irrelevance as hinted on page 4.[1] The adverb *merely* may be used to indicate the use of (-1)-truncation; for example, there *merely* exists an element of type A if we have an element of type $\|A\|_{-1}$.

[1]In many systems supporting proof-irrelevance, there is an erasure operator to drop irrelevant parts. It may be non-trivial to support automatic erasure of (-1)-truncated proofs, but we then have lemmas 3.1.6 and 3.1.7 that utilize (-1)-truncated contents, which would be impossible in systems enforcing a naïve segregation policy.

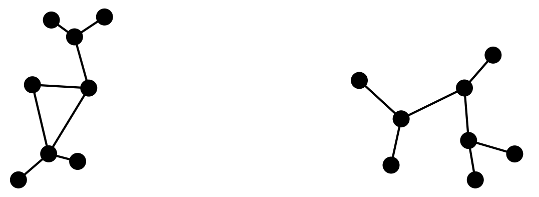


Figure 3.4: Examples of connected spaces without structures above dimension 1.

The space on the left is not 1-connected because paths between points are not unique. Conversely, a 1-connected space is always 0-connected.

3.1.6 Connectivity

Connectivity is the dual of truncation level in the sense that an n-connected type is trivial (sufficiently filled) *below or at* the dimension n. See fig. 3.4 for a visualization of 0-connected and 1-connected spaces. In this thesis we critically rely on the fact that, for any two elements in an n-connected type, there is an $(n-1)$-truncated identification in between. Technically, an n-connected type is defined to be a type whose n-truncation is contractible, meaning that it can only have non-trivial structures above dimension n:

$$\texttt{is-connected}_n(A{:}\mathcal{U}) :\equiv \texttt{is-contr}\,\|A\|_n.$$

3.1.7 Homotopy Fibers

Given a function f from type A to type B, there is a family of types $B \to \mathcal{U}$, called *(homotopy) fibers* of f, which at an element $b : B$ is the *generalized preimage* under f, as it reduces to the ordinary preimage when both A and B are sets. Formally, let the fiber be

$$\texttt{hfiber}_{f:A\to B}(b{:}B) :\equiv \sum_{a:A} f(a) =_B b,$$

which makes the following dimagram commute:

$$\begin{tikzcd}
\sum_{b:B}\texttt{hfiber}_f(b) \arrow[r, equal] \arrow[dr, "\texttt{fst}"'] & A \arrow[d, "f"] \\
 & B.
\end{tikzcd}$$

This diagram demonstrates that families of types and arrow types are essentially the same; a function of an arrow type can be turned into a family of types by considering $\texttt{hfiber}_f$, and a family of types can be treated as a projection function from its total space. The only difference is the judgmental equality they admit. This duality makes it possible for users of UniTT+HIT to choose what is most convenient.

Similarly to connectivity of types, we can also say a *function f* from A to B has connectivity n if f is *surjective below or at dimension $n+1$.* Note that being surjective at dimensions $k+1$ implies being injective at dimension k, because the identification between structures at dimension k, which will be at dimension $k+1$, will be reflected back to the domain by surjectivity. This means a function with connectivity n is actually *isomorphic for dimension below or at n;* as an example, it induces group isomorphisms between homotopy groups of the domain and the codomain.

This can be equivalently defined as *connectivity of its fiber;* that is,

$$\texttt{has-conn-fibers}_n(f{:}A{\to}B) :\equiv \prod_{b:B} \texttt{is-connected}_n(\texttt{hfiber}_f(b)).$$

The intuition is that, if f is an equivalence, then each fiber will be contractible, or loosely speaking "∞-connected". Having n-connected fibers means f behaves like an equivalence *up to dimension n,* matching our intuition discussed above.

Connectivity of types can also be defined in terms of connectivity of functions: a type A is equivalent to the (unique) fiber of the function $\lambda(\_{:}A).\texttt{unit}$ from A to $\mathbb{1}$, and thus the connectivity of A can be defined to be the connectivity of the function from A to $\mathbb{1}$. It also means matching the trivial type $\mathbb{1}$ up to dimension n, which is exactly our intuition of connectivity of types.

Although it is clear that f is *surjective* if it is (-1)-connected, a simpler definition for this special case is given due to its frequent usage: a function is *surjective* if we know each fiber is *merely* inhabited; that is,

$$\texttt{is-surj}(f{:}A{\to}B) :\equiv \prod_{b:B} \left\| \texttt{hfiber}_f(b) \right\|_{-1}.$$

It can be shown that $\texttt{has-conn-fibers}_{-1}$ and $\texttt{is-surj}$ are equivalent, justifying the above special definition.

3.1.8 Set Quotients

Let A be a type and $R : A \to A \to \mathcal{U}$ a family of types doubly indexed by A. We write A/R as the set quotient of A by R, $[a]$ as the equivalence class of $a : A$, and $\texttt{quot-rel}(r)$ for $r : R(a, b)$ as a witness of $[a] =_{A/R} [b]$. The family R need not be an equivalence relation itself, but the set quotient in type theory effectively takes the reflexive, symmetric and transitive closure of R. Formalization as a higher inductive type is in fig. 3.5.

Remark 3.1.2. We did not require R to be a family of mere propositions as in the book [138] because in theory it makes little difference and in practice it is convenient not to be concerned about truncation levels.

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma \vdash R : A \to A \to \mathcal{U}}{\Gamma \vdash A/R : \mathcal{U}}\ /\text{-}\textsc{formation}$$

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma \vdash R : A \to A \to \mathcal{U}}{\Gamma \vdash A/R\text{-}\mathtt{level} : \mathtt{is\text{-}set}(A/R)}\ /\text{-}\textsc{formation (level)}$$

$$\frac{\Gamma \vdash R : A \to A \to \mathcal{U} \qquad \Gamma \vdash a : A}{\Gamma \vdash [a] : A/R}\ /\text{-}\textsc{introduction}\ ([-])$$

$$\frac{\begin{array}{c}\Gamma \vdash R : A \to A \to \mathcal{U} \\ \Gamma \vdash a : A \qquad \Gamma \vdash b : A \qquad \Gamma \vdash r : R(a,b)\end{array}}{\Gamma \vdash \mathtt{quot\text{-}rel}(r) : [a] =_{A/R} [b]}\ /\text{-}\textsc{introduction}\ (\mathtt{quot\text{-}rel})$$

$$\frac{\begin{array}{c}\Gamma, x{:}A/R \vdash C : \mathcal{U} \\ \Gamma, x{:}A/R \vdash C\text{-}\mathtt{level} : \mathtt{is\text{-}set}(C(x)) \qquad \Gamma, x{:}A \vdash c_{[-]} : C[[x]/x] \\ \Gamma, x{:}A, y{:}A, z{:}R(x,y) \vdash c_{\mathtt{quot\text{-}rel}} : c_{[-]} =^{x.C}_{\mathtt{quot\text{-}rel}(z)} c_{[-]}[y/x] \qquad \Gamma \vdash q : A/R\end{array}}{\Gamma \vdash \mathtt{elim}_{/}[x.C; x.C\text{-}\mathtt{level}](x.c_{[-]}; x.y.z.c_{\mathtt{quot\text{-}rel}}; q) : C[q/x]}\ /\text{-}\textsc{elimination}$$

$$\frac{\begin{array}{c}\Gamma, x{:}A/R \vdash C : \mathcal{U} \\ \Gamma, x{:}A/R \vdash C\text{-}\mathtt{level} : \mathtt{is\text{-}set}(C(x)) \qquad \Gamma, x{:}A \vdash c_{[-]} : C[[x]/x] \\ \Gamma, x{:}A, y{:}A, z{:}R(x,y) \vdash c_{\mathtt{quot\text{-}rel}} c_{[-]} =^{x.C}_{\mathtt{quot\text{-}rel}(z)} c_{[-]}[y/x] \qquad \Gamma \vdash a : A\end{array}}{\Gamma \vdash \mathtt{elim}_{/}[x.C; x.C\text{-}\mathtt{level}](x.c_{[-]}; x.y.z.c_{\mathtt{quot\text{-}rel}}; [a]) \equiv c_{[-]}[a/x] : C[[a]/x]}\ /\text{-}\textsc{computation}\ ([-])$$

$$\frac{\begin{array}{c}\Gamma, x{:}A/R \vdash C : \mathcal{U} \\ \Gamma, x{:}A/R \vdash C\text{-}\mathtt{level} : \mathtt{is\text{-}set}(C(x)) \qquad \Gamma, x{:}A \vdash c_{[-]} : C[[x]/x] \\ \Gamma, x{:}A, y{:}A, z{:}R(x,y) \vdash c_{\mathtt{quot\text{-}rel}} : c_{[-]} =^{x.C}_{\mathtt{quot\text{-}rel}(z)} c_{[-]}[y/x] \\ \Gamma \vdash a : A \qquad \Gamma \vdash b : A \qquad \Gamma \vdash r : R(a,b)\end{array}}{\begin{array}{r}\Gamma \vdash \mathtt{quot\text{-}rel}_{\beta} : \mathtt{apd}_{\lambda x.\mathtt{elim}_{/}[x.C; x.C\text{-}\mathtt{level}](x.c_{[-]}; x.y.z.c_{\mathtt{quot\text{-}rel}}; x)}(\mathtt{quot\text{-}rel}(r)) \\ =_{c_{[-]}[a/x] =^{x.C}_{\mathtt{quot\text{-}rel}(r)} c_{[-]}[b/x]} c_{\mathtt{quot\text{-}rel}}[a,b,r/x,y,z]\end{array}}\ /\text{-}\textsc{comp.}\ (\mathtt{quot\text{-}rel})$$

Figure 3.5: Rules of squashing the second dimension.

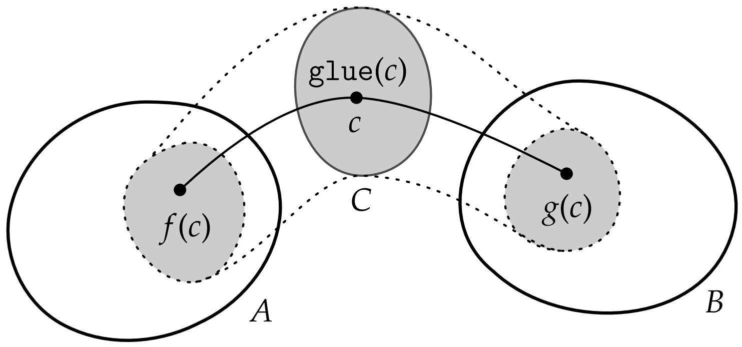


Figure 3.6: Pushouts as two spaces glued together.

3.1.9 Fundamental Groups and Groupoids

As mentioned earlier, iterated identification forms the structure of ∞-groupoids. The 0-truncation of identification thus gives ordinary groupoids, called *fundamental groupoids.* If we only focus on truncated identification at some particular element, then they reduce to *fundamental groups.* As an example, we have seen the argument in section 2.5 that the fundamental group of $\mathbb{S}^1$ is $\mathbb{Z}$.

Formally, for any type A, the fundamental groupoid of A written $\Pi_1(A)$ is formed by $\lambda(a{,}b{:}A).\|a =_A b\|_0$ being the arrows of the groupoid[2] along with concatenation as composition and reflexivity as the unit. Given a distinguished element $a : A$, the *fundamental group* of the type A at a, written $\pi_1(A, a)$, is the set $\|a = a\|_0$ with the same operators.

Technically speaking, the concatenation, reflexivity, inversion operators and $\mathtt{ap}_f$ on 0-truncated identifications are different from those on untruncated identification, but regardless I will reuse the symbols for a cleaner presentation; however, the distinction is still important and the transport along a 0-truncated identification across a family of sets, written $\mathtt{transport}_0[x.B(x)](p; a)$, will have a subscript 0 denoting the truncation level.

3.1.10 Pushouts and Friends

A *(homotopy) pushout* is a type made of two types A and B glued together by adding identifications (as bridges) across the two sides; the additional identifications, written `glue`, are indexed by another type C, along with functions $f : C \to A$ and $g : C \to B$ denoting the end points of $\mathtt{glue}(c)$. See fig. 3.6 for a visualization. In type theory, the pushout is written $A \sqcup_{C;f;g} B$, with inclusions from A and B written as `left` and `right`, respectively; see fig. 3.7 for its formal rules. The f and g in $A \sqcup_{C;f;g} B$ may be omitted when clear from the context. Category-theoretically speaking, a pushout is a colimit of the span

[2]More precisely, this is a *pre*groupoid because the object type A may not be a 1-type. A proper fundamental groupoid should have $\|A\|_1$ as its object type and $\lambda(a{,}b{:}\|A\|_1).a =_{\|A\|_1} b$ as its arrow family. However, for any $a, b : A$ there is an equivalence between $\|a =_A b\|_0$ and $|a|_1 =_{\|A\|_1} |b|_1$ and in practice $\|a =_A b\|_0$ seems easier to work with. See Rezk completion (or stack completion) in [138, §9.9, 3].

$$\begin{tikzcd} A & C \arrow[l, "f"'] \arrow[r, "g"] & B \end{tikzcd}$$

or equivalently the following is a commuting square (called the *pushout square*) with initiality toward the bottom-right corner.

$$\begin{tikzcd} C \arrow[r, "g"] \arrow[d, "f"'] & B \arrow[d, "\texttt{right}"] \\ A \arrow[r, "\texttt{left}"'] & A \sqcup_{C;f;g} B \end{tikzcd}$$

Many types of homotopy-theoretic interest can be obtained by the pushout construction, including the circle, truncations, set quotients, suspensions, cofibers or even the expressive cellular complexes. See [46, 81, 119] for implementing truncations and set quotients using only pushouts. The following are suspensions and cofibers:

Suspension types. The *suspension* of a type A is intuitively the type A being suspended with "strings" pinned at two poles (`north` and `south` below). Formally, they can be defined as follows:

$$\begin{aligned} \texttt{susp}(A) &:\equiv \mathbb{1} \sqcup_A \mathbb{1} \\ \texttt{north} &:\equiv \texttt{left}(\texttt{unit}) : \texttt{susp}(A) \\ \texttt{south} &:\equiv \texttt{right}(\texttt{unit}) : \texttt{susp}(A) \\ \texttt{merid}(a{:}A) &:\equiv \texttt{glue}(a) : \texttt{north} = \texttt{south} \end{aligned}$$

$$\begin{tikzcd} A \arrow[r] \arrow[d] & \mathbb{1} \arrow[d, "\lambda\_.\texttt{south}"] \\ \mathbb{1} \arrow[r, "\lambda\_.\texttt{north}"'] & \texttt{susp}(A) \end{tikzcd}$$

Cofiber types. The *cofiber* of a function $f : A \to B$ is the dual of *fiber,* intuitively the domain B with the image of f contracted; formally, it consists of these elements:[3]

$$\begin{aligned} \texttt{cofiber}(f) &:\equiv \mathbb{1} \sqcup_{A;\lambda\_.\texttt{unit};f} B \\ \texttt{cfbase} &:\equiv \texttt{left}(\texttt{unit}) : \texttt{cofiber}(f) \\ \texttt{cfcod}(b{:}B) &:\equiv \texttt{right}(b) : \texttt{cofiber}(f) \\ \texttt{cfglue}(a{:}A) &:\equiv \texttt{glue}(a) : \texttt{cfbase} = \texttt{cfcod}(f(a)) \end{aligned}$$

[3]The prefix `cf` stands for "cofiber" and `cod` stands for "codomain".

$$\frac{\Gamma \vdash f : C \to A \qquad \Gamma \vdash g : C \to B}{\Gamma \vdash A \sqcup_{C;f;g} B : \mathcal{U}} \ \sqcup\text{-}\textsc{formation}$$

$$\frac{\Gamma \vdash f : C \to A \qquad \Gamma \vdash g : C \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash \texttt{left}(a) : A \sqcup_{C;f;g} B} \ \sqcup\text{-}\textsc{introduction}\ (\texttt{left})$$

$$\frac{\Gamma \vdash f : C \to A \qquad \Gamma \vdash g : C \to B \qquad \Gamma \vdash b : B}{\Gamma \vdash \texttt{right}(b) : A \sqcup_{C;f;g} B} \ \sqcup\text{-}\textsc{introduction}\ (\texttt{right})$$

$$\frac{\Gamma \vdash f : C \to A \qquad \Gamma \vdash g : C \to B \qquad \Gamma \vdash c : C}{\Gamma \vdash \texttt{glue}(c) : \texttt{left}(f(c)) =_{A \sqcup_{C;f;g} B} \texttt{right}(g(c))} \ \sqcup\text{-}\textsc{introduction}\ (\texttt{glue})$$

$$\frac{\begin{array}{c} \Gamma, x{:}(A \sqcup_{C;f;g} B) \vdash D : \mathcal{U} \\ \Gamma, x{:}A \vdash d_{\texttt{left}} : D[\texttt{left}(x)/x] \qquad \Gamma, x{:}B \vdash d_{\texttt{right}} : D[\texttt{right}(x)/x] \\ \Gamma, x{:}C \vdash d_{\texttt{glue}} : d_{\texttt{left}}[f(x)/x] =^{x.D}_{\texttt{glue}(x)} d_{\texttt{right}}[g(x)/x] \qquad \Gamma \vdash p : A \sqcup_{C;f;g} B \end{array}}{\Gamma \vdash \texttt{elim}_{\sqcup}[x.D](x.d_{\texttt{left}}; x.d_{\texttt{right}}; x.d_{\texttt{glue}}; p) : D[p/x]} \ \sqcup\text{-}\textsc{elimination}$$

$$\frac{\begin{array}{c} \Gamma, x{:}(A \sqcup_{C;f;g} B) \vdash D : \mathcal{U} \\ \Gamma, x{:}A \vdash d_{\texttt{left}} : D[\texttt{left}(x)/x] \qquad \Gamma, x{:}B \vdash d_{\texttt{right}} : D[\texttt{right}(x)/x] \\ \Gamma, x{:}C \vdash d_{\texttt{glue}} : d_{\texttt{left}}[f(x)/x] =^{x.D}_{\texttt{glue}(x)} d_{\texttt{right}}[g(x)/x] \qquad \Gamma \vdash a : A \end{array}}{\begin{array}{l} \Gamma \vdash \texttt{elim}_{\sqcup}[x.D](x.d_{\texttt{left}}; x.d_{\texttt{right}}; x.d_{\texttt{glue}}; \texttt{left}(a)) \\ \qquad\qquad \equiv d_{\texttt{left}}[a/x] : D[\texttt{left}(a)/x] \end{array}} \ \sqcup\text{-}\textsc{computation}\ (\texttt{left})$$

$$\frac{\begin{array}{c} \Gamma, x{:}(A \sqcup_{C;f;g} B) \vdash D : \mathcal{U} \\ \Gamma, x{:}A \vdash d_{\texttt{left}} : D[\texttt{left}(x)/x] \qquad \Gamma, x{:}B \vdash d_{\texttt{right}} : D[\texttt{right}(x)/x] \\ \Gamma, x{:}C \vdash d_{\texttt{glue}} : d_{\texttt{left}}[f(x)/x] =^{x.D}_{\texttt{glue}(x)} d_{\texttt{right}}[g(x)/x] \qquad \Gamma \vdash b : B \end{array}}{\begin{array}{l} \Gamma \vdash \texttt{elim}_{\sqcup}[x.D](x.d_{\texttt{left}}; x.d_{\texttt{right}}; x.d_{\texttt{glue}}; \texttt{right}(b)) \\ \qquad\qquad \equiv d_{\texttt{right}}[b/x] : D[\texttt{right}(b)/x] \end{array}} \ \sqcup\text{-}\textsc{computation}\ (\texttt{right})$$

$$\frac{\begin{array}{c} \Gamma, x{:}(A \sqcup_{C;f;g} B) \vdash D : \mathcal{U} \\ \Gamma, x{:}A \vdash d_{\texttt{left}} : D[\texttt{left}(x)/x] \qquad \Gamma, x{:}B \vdash d_{\texttt{right}} : D[\texttt{right}(x)/x] \\ \Gamma, x{:}C \vdash d_{\texttt{glue}} : d_{\texttt{left}}[f(x)/x] =^{x.D}_{\texttt{glue}(x)} d_{\texttt{right}}[g(x)/x] \qquad \Gamma \vdash c : C \end{array}}{\Gamma \vdash \texttt{glue}_{\beta} : \texttt{apd}_{\lambda x.\texttt{elim}_{\sqcup}[x.D](x.d_{\texttt{left}}; x.d_{\texttt{right}}; x.d_{\texttt{glue}}; x)}(\texttt{glue}(c)) = d_{\texttt{glue}}[c/x]} \ \sqcup\text{-}\textsc{computation}\ (\texttt{glue})$$

Figure 3.7: Rules of gluing.

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d] & B \arrow[d, "\texttt{cfcod}"] \\
\mathbb{1} \arrow[r, "\lambda\_.\texttt{cfbase}"'] & \texttt{cofiber}(f)
\end{tikzcd}$$

To see the duality, note that a fiber over $b : B$ is the following pullback:

$$\begin{tikzcd}
\texttt{hfiber}_f(b) \arrow[r] \arrow[d] & A \arrow[d, "f"] \\
\mathbb{1} \arrow[r, "\lambda\_.b"'] & B
\end{tikzcd}$$

3.1.11 Pointed Types and Wedges

A *pointed type* is a type with a distinguished element, or more precisely a pair of type $\sum_{A:\mathcal{U}} A$. To better present the results, we have specialized projections `carrier` and `pt` for pointed types:

$$\texttt{carrier} : \sum_{A:\mathcal{U}} A \to \mathcal{U} \qquad \texttt{pt} : \prod_{X:\sum_{A:\mathcal{U}} A} \to \texttt{carrier}(X)$$

$$\texttt{carrier} :\equiv \texttt{fst} \qquad \texttt{pt} :\equiv \texttt{snd}$$

Let X and Y be two pointed types. We have *pointed arrow types* from X to Y, written $X \cdot\!\!\to Y$, which collect functions that preserve the distinguished element; it is defined as

$$X \cdot\!\!\to Y :\equiv \sum_{f:\texttt{carrier}(X)\to\texttt{carrier}(Y)} f(\texttt{pt}(X)) = \texttt{pt}(Y).$$

Note that the type $X \cdot\!\!\to Y$ could be made pointed with the constant function $\lambda\_.\texttt{pt}(Y)$, but my Agda experience shows that making pointed arrow types pointed would only make proofs clumsier, especially without the implicit coercion.

Many types defined so far have a natural choice of their distinguished element. For example, the distinguished element of a pushout $A \sqcup_C B$ comes from the distinguished element of A (if pointed); so are those of the types defined in terms of pushouts. For these types, I will recycle the symbols for their pointed counterparts. In particular, the pointed variants $\texttt{cofiber}(f)$ for pointed arrows f and $\texttt{susp}(X)$ for pointed types X will be used in section 3.5.

In addition, we can define the *binary wedge* of two pointed types X and Y, written $X \vee Y$, as a pushout with two distinguished elements identified; that is,

$$\begin{aligned}
X \vee Y &:\equiv \texttt{carrier}(X) \sqcup_{\mathbb{1};\lambda\_.\texttt{pt}(X);\lambda\_.\texttt{pt}(Y)} \texttt{carrier}(Y) \\
\texttt{winl}\big(x{:}\texttt{carrier}(X)\big) &:\equiv \texttt{left}(x) : X \vee Y \\
\texttt{winr}\big(y{:}\texttt{carrier}(Y)\big) &:\equiv \texttt{right}(y) : X \vee Y \\
\texttt{wglue} &:\equiv \texttt{glue}(\texttt{unit}) : \texttt{winl}(\texttt{pt}(X)) = \texttt{winl}(\texttt{pt}(Y))
\end{aligned}$$

$$\begin{tikzcd}
\mathbb{1} \arrow[r, "\lambda\_.\texttt{pt}(Y)"] \arrow[d, "\lambda\_.\texttt{pt}(X)"'] & \texttt{carrier}(Y) \arrow[d, "\texttt{winr}"] \\
\texttt{carrier}(X) \arrow[r, "\texttt{winl}"'] & X \vee Y
\end{tikzcd}$$

There is a canonical function from $X \vee Y$ to $\texttt{carrier}(X) \times \texttt{carrier}(Y)$, which sends $\texttt{winl}(x)$ to $\langle x; \texttt{pt}(Y)\rangle$ and $\texttt{winr}(y)$ to $\langle \texttt{pt}(X); y\rangle$. An intriguing fact is that if $\texttt{carrier}(X)$ is m-connected and $\texttt{carrier}(Y)$ is n-connected, then for any family of $(m+n)$-types P indexed by $\texttt{carrier}(X) \times \texttt{carrier}(Y)$, functions from the wedge $X \vee Y$ to P extend along the canonical function to $\texttt{carrier}(X) \times \texttt{carrier}(Y)$.

$$\begin{tikzcd}
X \vee Y \arrow[r] \arrow[d] & P \\
\texttt{carrier}(X) \times \texttt{carrier}(Y) \arrow[ru, dashed]
\end{tikzcd}$$

This can be made precise as follows; for easier use, the pointed types are decomposed into the carriers and the distinguished elements, P is curried and the function from $X \vee Y$ to P is broken into the individual data fed into the eliminator of wedges.

Lemma 3.1.3 (wedge connectivity). *Suppose A is an m-type with a distinguished element a_0 and B an n-type with b_0 where $m, n \geq -1$. Given the data*

$$\begin{aligned}
P &: A \to B \to \mathcal{U} \\
P\texttt{-level} &: \prod_{a:A} \prod_{b:B} \texttt{has-level}_{m+n}(P(a,b)) \\
f &: \prod_{a:A} P(a, b_0) \\
g &: \prod_{b:B} P(a_0, b) \\
\alpha &: f(a_0) =_{P(a_0,b_0)} f(b_0)
\end{aligned}$$

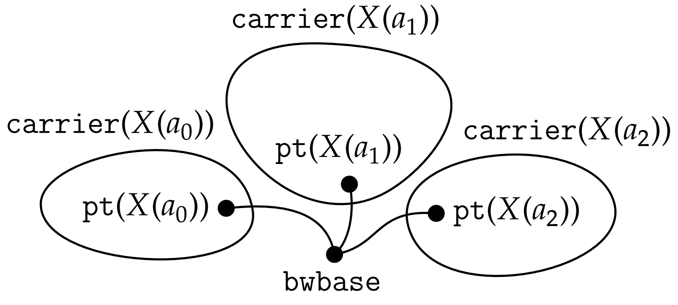
there is a function $p : \prod_{a:A} \prod_{b:B} P(a,b)$ such that there are elements q, r and s of these types:

$$q : \prod_{a:A} p(a, b_0) = f(a)$$
$$r : \prod_{b:A} p(a_0, b) = g(b)$$
$$s : q(a_0)^{-1} \cdot r(b_0) = \alpha.$$

Proof. See [138, lemma 8.6.2]. (The lemma works for $m = -1$ or $n = -1$ despite the range stated in the book.) □

Remark 3.1.4. As noted in remark 3.1.1, the mechanized version replaces m and n by $\mathtt{succ}(m)$ and $\mathtt{succ}(n)$, respectively, replaces $m + n$ by $m \hat{+} n$, and drops the condition $m, n \geq -1$.

In addition to binary wedges, for a family of pointed types X indexed by A, there is a general wedge of X with all distinguished elements in all fibers identified, written $\bigvee_{a:A} X(a)$, with `bwbase` being the center to which the distinguished elements are identified, `bwin` being the inclusion and `bwglue` being the identification from `bwbase`.[4]



This can again be defined as a pushout (or the cofiber of $\lambda a.\langle a; \mathtt{pt}(X(a))\rangle$):

$$\begin{aligned}
\textstyle\bigvee_{a:A} X(a) &:\equiv \mathbb{1} \sqcup_{A;\lambda\_.\mathtt{unit};\lambda a.\langle a;\mathtt{pt}(X(a))\rangle} \textstyle\sum_{a:A} \mathtt{carrier}(X(a)) \\
\mathtt{bwbase} &:\equiv \mathtt{left}(\mathtt{unit}) : \textstyle\bigvee_{a:A} X(a) \\
\mathtt{bwin}\big(a{:}A, x{:}\mathtt{carrier}(X(a))\big) &:\equiv \mathtt{right}\big(\langle a; x\rangle\big) : \textstyle\bigvee_{a:A} X(a) \\
\mathtt{bwglue}(a{:}A) &:\equiv \mathtt{glue}(a) : \mathtt{bwbase} = \mathtt{bwin}(a, \mathtt{pt}(X(a))).
\end{aligned}$$

$$\begin{tikzcd}
A \arrow[r, "\lambda a.\langle a;\mathtt{pt}(X(a))\rangle"] \arrow[d] & \sum_{a:A} \mathtt{carrier}(X(a)) \arrow[d, "\lambda s.\mathtt{bwin}(\mathtt{fst}(s),\mathtt{snd}(s))"] \\
\mathbb{1} \arrow[r, "\lambda\_.\mathtt{bwbase}"'] & \bigvee_{a:A} X(a)
\end{tikzcd}$$

We will see the usage of this type in section 3.5.

[4]The prefix `bw` stands for "big wedge".

$$\frac{\Gamma\ \texttt{ctx}}{\Gamma \vdash \mathbb{2} : \mathcal{U}_0}\ \mathbb{2}\text{-FORMATION} \qquad \frac{\Gamma\ \texttt{ctx}}{\Gamma \vdash \texttt{true} : \mathbb{2}}\ \mathbb{2}\text{-INTRODUCTION } (\texttt{true})$$

$$\frac{\Gamma\ \texttt{ctx}}{\Gamma \vdash \texttt{false} : \mathbb{2}}\ \mathbb{2}\text{-INTRODUCTION } (\texttt{false})$$

$$\frac{\begin{array}{c}\Gamma, x{:}\mathbb{2} \vdash C : \mathcal{U}_i \\ \Gamma \vdash c_{\texttt{true}} : C[\texttt{true}/x] \qquad \Gamma \vdash c_{\texttt{false}} : C[\texttt{false}/x] \qquad \Gamma \vdash b : \mathbb{2}\end{array}}{\Gamma \vdash \texttt{elim}_{\mathbb{2}}[x.C](c_{\texttt{true}}; c_{\texttt{false}}; b) : C[b/x]}\ \mathbb{2}\text{-ELIMINATION}$$

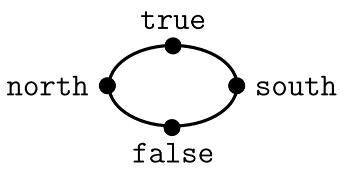
$$\frac{\begin{array}{c}\Gamma, x{:}\mathbb{2} \vdash C : \mathcal{U}_i \\ \Gamma \vdash c_{\texttt{true}} : C[\texttt{true}/x] \qquad \Gamma \vdash c_{\texttt{false}} : C[\texttt{false}/x]\end{array}}{\Gamma \vdash \texttt{elim}_{\mathbb{2}}[x.C](c_{\texttt{true}}; c_{\texttt{false}}; \texttt{true}) \equiv c_{\texttt{true}} : C[\texttt{true}/x]}\ \mathbb{2}\text{-COMPUTATION } (\texttt{true})$$

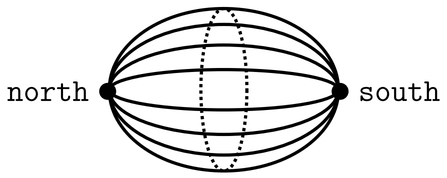
$$\frac{\begin{array}{c}\Gamma, x{:}\mathbb{2} \vdash C : \mathcal{U}_i \\ \Gamma \vdash c_{\texttt{true}} : C[\texttt{true}/x] \qquad \Gamma \vdash c_{\texttt{false}} : C[\texttt{false}/x]\end{array}}{\Gamma \vdash \texttt{elim}_{\mathbb{2}}[x.C](c_{\texttt{true}}; c_{\texttt{false}}; \texttt{false}) \equiv c_{\texttt{false}} : C[\texttt{false}/x]}\ \mathbb{2}\text{-COMPUTATION } (\texttt{false})$$

Figure 3.8: Rules of `1/0`.

3.1.12 Booleans as the Zeroth Sphere

The good old Boolean type $\mathbb{2}$ can be formalized as rules in fig. 3.8. An interesting fact is that the suspension of the Boolean type, $\texttt{susp}(\mathbb{2})$, is equivalent to $\mathbb{S}^1$; moreover, the suspension of $\mathbb{S}^1$, with the circle being the equator, gives the 2-dimensional sphere.





In general, the suspension of the n-dimensional sphere gives the $(n + 1)$-dimensional sphere, and this leads to the following definition of spheres as iterated suspensions; for convenience we may just call $\mathbb{2}$ as $\mathbb{S}^0$ because $\mathbb{S}^1$ is the suspension of $\mathbb{2}$.

$$\mathbb{S}^n :\equiv \texttt{elim}_{\mathbb{N}}[\_.\mathcal{U}](\mathbb{2}; \_.s.\texttt{susp}(s); n).$$

Remark 3.1.5. Therefore, there are two equivalent definitions of $\mathbb{S}^1$! In our Agda mechanization, the circle is *defined* to be the suspension of the Boolean type, which means the rules in fig. 2.10 (including judgmental equality) are actually implemented by other rules. Due to this, I intentionally use the same symbol for both $\mathbb{S}^1$'s in this thesis.

3.1.13 Constancy Extension Lemmas

Following the discussion about truncation level, a (-1)-truncated proof can only be used within a context at level -1, but often the context is at the level 0. For example, in group theory many theorems are at the set level but a surjectivity condition only asserts a witness in the domain at the -1 level. Fortunately, such a level gap can be filled by a constancy condition; the intuition is that if a function does not depend on the value of the input but only its existence, a (-1)-truncated input should suffice. This is formulated as the following lemma:

Lemma 3.1.6 (extension by weak constancy)**.** *Let A be a type and B a set. For any function $f : A \to B$ with f-`is-const` $: \prod_{x,y:A} f(x) =_B f(y)$, there exists a function $g : \|A\|_{-1} \to B$ such that $f \equiv g \circ |{-}|_{-1}$.*

Because I proved this lemma, Nicolai Kraus *et al.* have significantly generalized the result and considered the cases from mere propositions to types at arbitrary levels; see [80, 82, 83]. The following is a short proof of lemma 3.1.6:

Proof of lemma 3.1.6. Given a function f from A to B satisfying the constancy condition, define $C :\equiv A/R$ where

$$R(a, b) :\equiv f(a) =_B f(b).$$

One can then show that the function f factors through C. Since C is provably a mere proposition, the function f can be extended to the (-1)-truncation of A. The judgmental equality is derived from the computation rules of truncation and set quotients. □

There is another version of lemma 3.1.6 that generalizes the truncation projection $|{-}|_{-1}$ to any surjective function and replace the weak consistency by what I call *relative constancy.* As far as I understand, this lemma is new. Here is its formulation:

Lemma 3.1.7 (extension by relative constancy)**.** *Let A and B be two types and C be a family of sets indexed by B. For any surjective function $f : A \to B$ and any function $g : \prod_{a:A} C(f(a))$ such that*

$$g\text{-}\texttt{is-const} : \prod_{a_0,a_1:A} \prod_{(p:f(a_0)=f(a_1))} g(a_0) =_p^{x.C(x)} g(a_1),$$

there exists a function $h : \prod_{b:B} C(b)$ such that $\prod_{a:A} h(f(a)) = g(a)$.

This is saying that if g gives the same result for all pairs equated by f, then g extends to B. We will find the usage of this lemma in section 3.3. Pictorially, this can be summarized as follows:

$$\begin{tikzcd} A \arrow[r, "g"] \arrow[d, two heads, "f"'] & \sum_{b:B} C(b) \\ B \arrow[ur, dashed, "h"'] & \end{tikzcd}$$

Proof. A direct proof is to consider the fibers of f for each b, and invoke lemma 3.1.6 to break the truncation restriction in the surjectivity condition. More precisely, for any $b : B$, let

$$\hat{h} : \prod_{b:B} \mathtt{hfiber}_f(b) \to C(b)$$

$$\hat{h} :\equiv \lambda(b{:}B).\lambda\big(s{:}\mathtt{hfiber}_f(b)\big).\mathtt{transport}[x.C(x)]\big(\mathtt{snd}(s); g(\mathtt{fst}(s))\big)$$

and apply lemma 3.1.6 to the function

$$\hat{h}(b) : \mathtt{hfiber}_f(b) \to C(b)$$

where the constancy condition is given by applying identification elimination on the second component of s_1:

$$\lambda\big(s_0,s_1{:}\mathtt{hfiber}_f(b)\big).\mathtt{elim}_{f(\mathtt{fst}(s_1))=}\Big[b.p.\textstyle\prod_{q:\mathtt{fst}(s_0))=b} \hat{h}\big(b, \langle \mathtt{fst}(s_0); q\rangle\big) = \hat{h}\big(b, \langle \mathtt{fst}(s_0); p\rangle\big)\Big]\Big($$
$$\mathtt{to\text{-}transp}\big(g\mathtt{\text{-}is\text{-}const}(\mathtt{fst}(s_0), \mathtt{fst}(s_1), q)\big); b; \mathtt{snd}(s_1)\Big)(\mathtt{snd}(s_0)) : \prod_{s_0,s_1} \hat{h}(b, s_0) = \hat{h}(b, s_1).$$

Lemma 3.1.6 will give a function $h' : \|\mathtt{hfiber}_f(b)\|_{-1} \to C(b)$; the desired function h is then the composition of the surjectivity (as a function) and h':

$$h :\equiv h' \circ f\mathtt{\text{-}is\text{-}surj} : \prod_{b:B} C(b).$$

The computational content comes from the fact that all elements in $\|\mathtt{hfiber}_f(f(a))\|_{-1}$, as a mere proposition, are identified and among them the particular element $|\langle a; \mathtt{refl}_{f(a)}\rangle|_{-1}$ will make the function h' run. □

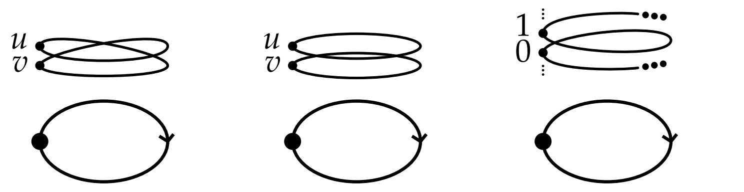
3.1.14 Implicit Coercion

To further reduce notational clutter, I will adopt more implicit coercion when no confusion would occur. For example, a group may be implicitly coerced into its underlying set, or a pointed type X into its underlying type $\mathtt{carrier}(X)$.

3.2 Covering Spaces

This is joint work with Robert Harper and this section incorporates text from my manuscript [69].

Covering spaces are one of the important constructs in homotopy theory, and given the connection between type theory and homotopy theory, a natural question to ask is whether such a notion can be stated in type theory as well. It turns out that we can express covering spaces (up to homotopy) concisely as follows.



set: $\{u, v\}$	set: $\{u, v\}$	set: $\mathbb{Z}$
action: $u \mapsto v$	action: $u \mapsto u$	action: $i \mapsto (i+1)$
$v \mapsto u$	$v \mapsto v$	

Figure 3.9: Covering spaces and sets equipped with a group action.

Note that the actions are represented by how they act on the generator `loop`.

Definition 3.2.1. A *covering space* of a type A is a family of sets indexed by A.

That is, the type of covering spaces of A is simply $A \to$ `Set`. The classical definition is significantly longer [65], though to be fair the classical theory does not take homotopy equivalence classes as we do in UniTT+HIT through the identification-as-path interpretation.

How do we know this definition really defines covering spaces? A characteristic feature of covering spaces of a type A in the classical theory is that they are represented by sets with a group action of the fundamental group of A. Therefore, we may justify our definitions by proving this theorem, as we will in section 3.2.1. See fig. 3.9 for some examples of the correspondence between covering spaces and such sets. Moreover, considering the category of pointed covering spaces where morphisms are fiberwise functions, we also know there should be an initial covering space (named the *universal* covering space) and it should be represented by the fundamental group itself through the representation theorem stated above.[5] We also managed to show these results as demonstrated in section 3.2.2.

3.2.1 Representation Theorem

The first main result of this section is that covering spaces of a 0-connected, pointed type A are represented by *sets equipped with a group action of the fundamental group of A*, which is to say there is an equivalence between covering spaces and such sets. The intuition is that everything in UniTT+HIT must respect identification, and the fact that the base type A is 0-connected indicates that there is a (-1)-truncated identification between any two elements and thus a (-1)-truncated isomorphism between any two fibers. Therefore, it is represented by one copy of these isomorphic sets and a description of how they are isomorphic, encoded as an action of the fundamental group.

[5]See section 3.2.2 for a more precise statement.

Formally, a set with a group action of G is called a *G-set*, a functor from the group G (treated as a category with one object and elements in G as morphisms) to the category of sets up to isomorphism; a *group set* is a G-set without the group G being specified. In type theory, a *G-set* is a record with the following components:

- `El`: a set.
- α: a (right) group action of type $\texttt{El} \to G \to \texttt{El}$.
- α-`unit`: a proof of the property that α preserves the group identity:

$$\prod_{x:\texttt{El}} \alpha(x, \texttt{unit}(G)) =_{\texttt{El}} x.$$

- α-`comp`: a proof of the property that α preserves the group composition:

$$\prod_{x:\texttt{El}} \prod_{g_1,g_2:G} \alpha(x, \texttt{comp}(G)(g_1, g_2)) =_{\texttt{El}} \alpha(\alpha(x, g_1), g_2).$$

The representation theorem is then about covering spaces of A being represented by $\pi_1(A, a)$-sets, which can be formally stated as follows:

Theorem 3.2.2 (representation by group sets)**.** *For any 0-connected type A with an element a, we have $(A \to \texttt{Set}) \simeq \pi_1(A, a)$-`Set`.*

An argument (provided by Steve Awodey) proceeds as follows: In the context of $A \to \texttt{Set}$, because the codomain `Set` is itself a 1-type (as the type of all n-types is an $(n+1)$-type [138, theorem 7.1.11]), structures at dimension higher than 1 in the domain A are irrelevant, which means that $(A \to \texttt{Set}) \simeq (\|A\|_1 \to \texttt{Set})$. (This can also be argued from the universal property of the 1-truncation of A.) Moreover, the 1-truncation of a pointed, 0-connected type A can be represented by its fundamental group $\pi_1(A, a)$ where a is the distinguished element,[6] and so the type $\|A\|_1 \to \texttt{Set}$ is really the collection of functors from $\pi_1(A, a)$ (as a category) to `Set`, or simply $\pi_1(A, a)$-sets. However, this argument relies on several components that were not available at the time of development; the following is a more elementary type-theoretic proof:

Proof. The standard methodology to show equivalence in UniTT+HIT is to establish two functions inverse to each other. That is, we want to establish two functions from covering spaces $A \to \texttt{Set}$ to group sets $\pi_1(A)$-`Set` and vice versa, and show that the round-trips are the identity function.

The direction from covering spaces $A \to \texttt{Set}$ to group sets $\pi_1(A)$-`Set` is relatively straightforward: the group set should capture a representative fiber with isomorphisms

[6]The equivalence between $\|A\|_1$ and the Eilenberg–Mac Lane space $K(\pi_1(A, a), 1)$ was mechanized by Floris van Doorn in the library of LEAN [106]. We are not aware of any publication regarding this result in UniTT+HIT at the time of writing.

between fibers. Since the base type A is 0-connected, every fiber is equally qualified, and so we choose the one over the distinguished element a. Moreover, recall that the isomorphism forced by an identification in the base type, as discussed in section 3.1, is the transport function. Putting these together, we can define a $\pi_1(A)$-set from a covering space $F : A \to \mathtt{Set}$ by taking

$$\begin{aligned} \mathtt{El} &:\equiv F(a) \\ \alpha &:\equiv \lambda x.\lambda g.\mathtt{transport}_0[x.F(x)](g; x) \end{aligned}$$

with properties α-`unit` and α-`comp` derived from functoriality of transports. The reason that we only have to record the automorphisms of $F(a)$ forced by loops at a (instead of all isomorphisms between all fibers) is because A is 0-connected; intuitively, one can continuously merge all the points in a 0-connected type into a, leaving only loops at a, which force those automorphisms.

The other direction, from group sets $X : \pi_1(A)$-`Set` to covering spaces, is more technically involved. A good guide is to focus on a group set generated from some covering space $F' : A \to \mathtt{Set}$ through the above process; if the theorem is true, we should be able to recreate a covering space F equivalent to F'. A key observation is that every element in any fiber of F' is a result of transporting some element in $F'(a)$ to that fiber, and X was exactly defined to be the source fiber $F'(a)$. Thus, one idea is to populate the new family F with *formal transports from X quotiented by the supposed functoriality of transports and the agreement with α*, in the hope to mimic the real transports in F'. Formally, we say F is a family of set quotients

$$F :\equiv \lambda b.(X \times \|a =_A b\|_0)/\sim_b$$

by some relation $\sim_b$ to be defined. It turns out that the functoriality of transports and the agreement with α can be succinctly summarized as

$$\langle \alpha(x,\ell); p\rangle \sim_b \langle x; \ell \cdot p\rangle$$

for any $x : X$, $\ell : \|a = a\|_0$ and $p : \|a = b\|_0$. This completes the construction of the new covering space F.

The next step is to show that these two functions are indeed inverse to each other. The most interesting part lies in proving that the covering space F reconstructed above is indeed equivalent to the original F': Following the standard recipe of equivalence, two functions back and forth are needed for the equivalence between two covering spaces. While the direction from F to F' is simply realizing the formal transports, the other direction is somewhat unclear—given an element y in the fiber $F'(b)$, how shall we locate an element x in $F(a)$ and compute a truncated identification p such that y will be the result of transporting x along p?

Recall that the connectivity of A implies that there is a (-1)-truncated identification between any two elements. That is, for any element $b : A$ we have a truncated identification $p : \|a =_A b\|_{-1}$. One attempt is then to transport y along the *inverse* of p to some element x in $F(a)$, for transporting x back along p should cancel the opposite transportation and recover y; the pair $\langle x; p\rangle$ in $F(b)$ then corresponds to y.

The only problem is that the identification is (-1)-truncated but the above construction is at the set level. This is where lemma 3.1.6 comes to rescue: we can show that different choices of identifications between a and b result into pairs related by the quotient relation imposed on $F(b)$, and then by lemma 3.1.6 we can extend the above construction to (-1)-truncated identifications.

In the remainder of this section, we will carefully construct the function from F' to F sketched above. For any point $b : A$, we have a function $f_b : F'(b) \to (a =_A b) \to F(b)$ as

$$f_b :\equiv \lambda y.\lambda p.\left[\left\langle \mathtt{transport}_0[x.F'(x)]\left(|p|_0{}^{-1}; y\right); |p|_0 \right\rangle\right],$$

which transports y to some point in $F(a)$. It is the second argument to the function f_b that is at the mismatched truncation level. We want to show lemma 3.1.6 applies to $f_b(y, -)$ for any $y : F'(b)$ so that a (-1)-truncated identification suffices. To satisfy the constancy condition in lemma 3.1.6, it is sufficient to demonstrate that for any two identifications p, q of type $a =_A b$

$$\left\langle \mathtt{transport}_0[x.F'(x)]\left(|p|_0{}^{-1}; y\right); |p|_0 \right\rangle \sim_b \left\langle \mathtt{transport}_0[x.F'(x)]\left(|q|_0{}^{-1}; y\right); |q|_0 \right\rangle$$

where $\sim_b$ is the quotient relation of $F(b)$ and thus $f_b(y, p) =_{F(b)} f_b(y, q)$. This can be proved by the groupoid laws of identification and the definition of $\sim_b$; we have

$$\begin{aligned}
&\left\langle \mathtt{transport}_0[x.F'(x)]\left(|p|_0{}^{-1}; y\right); |p|_0 \right\rangle \\
&= \left\langle \mathtt{transport}_0[x.F'(x)]\left(|p|_0{}^{-1}; y\right); |p|_0 \cdot |q|_0{}^{-1} \cdot |q|_0 \right\rangle \\
&\sim_b \left\langle \alpha\left(\mathtt{transport}_0[x.F'(x)]\left(|p|_0{}^{-1}; y\right), |p|_0 \cdot |q|_0{}^{-1}\right); |q|_0 \right\rangle \\
&\equiv \left\langle \mathtt{transport}_0[x.F'(x)]\left(|p|_0 \cdot |q|_0{}^{-1}; \mathtt{transport}_0[x.F'(x)]\left(|p|_0{}^{-1}; y\right)\right); |q|_0 \right\rangle \\
&= \left\langle \mathtt{transport}_0[x.F'(x)]\left(|p|_0{}^{-1} \cdot |p|_0 \cdot |q|_0{}^{-1}; y\right); |q|_0 \right\rangle \\
&= \left\langle \mathtt{transport}_0[x.F'(x)]\left(|q|_0{}^{-1}; y\right); |q|_0 \right\rangle.
\end{aligned}$$

This means $f_b(y, -)$ is (pairwise) constant, and thus by lemma 3.1.6 there exists an extension $g_{b,y} : \|a =_A b\|_{-1} \to F'(b)$ to the constant function $f_b(y, -)$. Putting these together, we have the following function of type $F'(b) \to F(b)$:

$$\lambda y.g_{b,y}\left(p(a, b)\right)$$

where $p(x, y)$ is the (-1)-truncated identification between x and y derived from the connectivity of A. This concludes the two functions between $F'(b)$ and $F(b)$; the remaining parts of the equivalence proof are a routine calculation. □

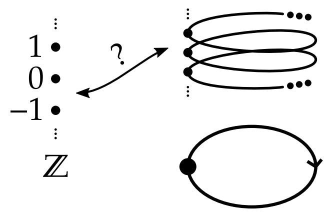


Figure 3.10: The lack of a canonical equivalence.

3.2.2 Universal Covering Spaces

In addition to the representation theorem, we also mechanize several well-known properties about a special covering space, the *universal covering space,* which is intuitively the most general or the most "unfolded" covering space over a type. It has two equivalent definitions, one based on connectivity and one based on initiality (and hence the name *universal*). In addition to the two definitions, when the base type is 0-connected it is also represented by the fundamental group—which is itself a $\pi_1(A, a)$-set—through the representation theorem in section 3.2.1; this argument was implicitly used in for example the calculation about the fundamental group of the circle in [93] and here we show a general result.

In this subsection the base type is fixed to be a type A with a distinguished point a.

Definition 3.2.3 (pointed covering space)**.** A *pointed covering space* is a covering space whose fiber over a is pointed.

Definition 3.2.4 (universal covering space)**.** A *universal covering space* is a pointed covering space whose total space is 1-connected.

The reason we stipulated a point in the specific fiber over the specific point is to make available a canonical choice among fiberwise equivalents. Considering the helix in fig. 3.10, the universal covering space over the circle whose fundamental group is $\mathbb{Z}$, there are multiple different equivalences between $\mathbb{Z}$ and any fiber of the helix, and there is no canonical choice—until we pin down a particular point in the helix and demand it be mapped to zero. To fit the notion of fiberwise equivalences, distinguished points of different covering spaces should be in the matching fibers, and thus we further demand the distinguished point lie in the fiber over the point a.

As hinted above, the following definition should be equivalent.

Definition 3.2.5 (alternative definition of universal covering space)**.** A *universal covering space* is a covering space that is (homotopy) initial in the category of pointed covering spaces with point-preserving fiberwise functions as morphisms.

The main observation to unify all these properties and simplify the proving is that the covering space consisting of 0-truncated identifications from the distinguished point

$$P :\equiv \lambda b. \left\| a =_A b \right\|_0$$

with its own distinguished point $\texttt{refl}_a$ in $P(a)$ *is* the universal covering space. This means that it suffices to show the covering space P is the one and only pointed covering space satisfying the two definitions of universal covering spaces, and that it is represented by the fundamental group. In fact, its correspondence to the fundamental group is trivial because its fiber over the distinguished point a is exactly (the underlying set of) the fundamental group, and it is not difficult to prove the group action is the concatenation. The rest of the section is dedicated to the two equivalent definitions.

First of all, we will show P is the one and only 1-connected covering space.

Lemma 3.2.6. *The total space of P is* 1*-connected.*

Proof. To show that the total space is 1-connected, by definition it suffices to show that the 1-truncation of $\sum_{b:A} P(b)$ is contractible, which means the 1-truncation is inhabited and there is an identification to any element in that truncation. The truncated pair $|\langle a; \texttt{refl}_a\rangle|_1$ is clearly an inhabitant, and the identification between $|\langle a; \texttt{refl}_a\rangle|_1$ and some other truncated pair $|\langle b; p\rangle|_1$ can be established by applying the identification elimination on p. $\square$

Lemma 3.2.7. *Any pointed covering space whose total space is* 1*-connected is equivalent to P.*

Proof. Let F be a pointed covering space whose total space is 1-connected. Once again we will follow the recipe of equivalence by establishing two functions inverse to each other. The direction from P to F can be done fiberwise by transports; that is, for any $b : A$, we can define a function from $P(b)$ to $F(b)$ as evaluating the transport of the distinguished point in $F(a)$ along the input in $P(b)$ (which is a truncated identification from a to b) to the fiber $F(b)$. Formally, it is

$$\lambda p.\texttt{transport}_0[x.F(x)]\big(p; a_F^*\big)$$

where a_F^* is the distinguished point of F over a. The other direction is to exploit the 1-connectivity: for any point y in the total space of F, there is a 0-truncated identification from the distinguished point $\langle a; a_F^*\rangle$ to y in the total space, which can then be "projected down" to the base type as a 0-truncated identification from the point a to the point over which y is. The projection of the 0-truncated identification is done by applying $\texttt{ap}_{\texttt{fst}}$. It can then be shown that these two functions are inverse to each other. $\square$

Lemmas 3.2.6 and 3.2.7 tell us P is the only 1-connected universal covering space. Thus the remaining step is to prove that P is the initial object in the category up to homotopy. Note that we did not explicitly define the category but directly talked about its morphisms.

Lemma 3.2.8. *For any pointed covering space F, there exists one and only one point-preserving fiberwise function from P to F.*

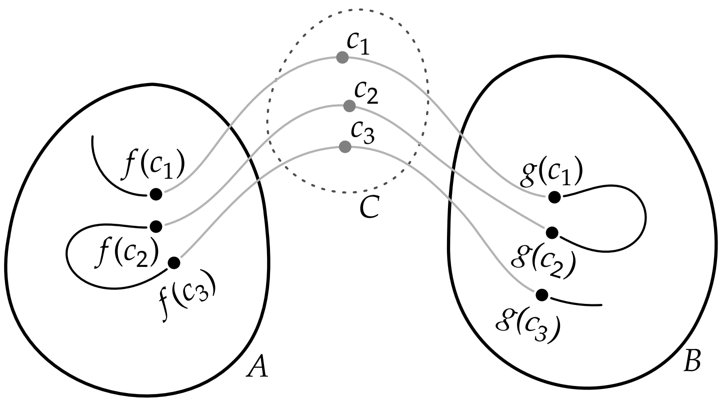


Figure 3.11: Identifications in a pushout.

Proof. The existence is again by transporting the distinguished point of F along the points in P, which are themselves 0-truncated identifications. The uniqueness is by applying the identification elimination on points in the total space P, which suggests we only have to consider the case $\texttt{refl}_a$, the distinguished point of P. However, a point-preserving function must send $\texttt{refl}_a$ to the distinguished point of F, and thus all such functions must agree. □

Now we can conclude this section by the following theorem:

Theorem 3.2.9. *The covering space P with $\texttt{refl}_a$ as its distinguished point is the universal covering and is represented by the fundamental group of the base type if the base type is* 0*-connected.*

Proof. By Lemmas 3.2.6 to 3.2.8 and the definition of P. □

3.3 The Seifert–van Kampen Theorem

This is based on the joint work with Michael Shulman [72] but the text was rewritten.

The *Seifert–van Kampen theorem* computes the fundamental groupoid of a pushout $P :\equiv A \sqcup_{C;f;g} B$ from those groupoids of individual parts A, B and C; the fundamental group of a pushout is then a corollary of the theorem. The significance comes from the fact that many types of homotopy-theoretic interest can be constructed from pushouts.

Recall that a fundamental groupoid is essentially the 0-truncated identification. The intuition is that a (0-truncated) identification in a pushout must be an identification touching either side alternatively, as shown in fig. 3.11. To make this precise, a family of sets $\texttt{code} : P \to P \to \mathcal{U}$ will be defined in a way that each fiber is obviously a type of alternative sequences of identifications connected by bridges generated by C, and then I will show why $\texttt{code}(a, b)$ and $\|a =_P b\|_0$ are isomorphic.

The family of sets $\texttt{code}$ is actually the pushout in the category of groupoids, and thus the theorem can be summarized nicely as the commutativity between the fundamental groupoid functor and pushout construction. However, the combinatorial description is probably more useful for calculation, and I did not mechanize that code indeed forms a pushout partly due to insufficient development of groupoid theory in our Agda codebase. Mechanizing this fact should not be too difficult with proper library support.

We also show an improved version, where the apex of the pushout span C is equipped with a surjective function h from some set D to C, and the connecting bridges shown in fig. 3.11 are restricted to the ones indexed by $h(d)$ for some $d : D$. The motivation is to make more explicit the hidden complexity of the naïve code from higher-dimensional structures in C. For example, when C is $\mathbb{S}^1$, one can select $\mathbb{1}$ as D and $\lambda\_.\mathtt{base}$ as h, which forces all bridges in code to be indexed by base and further reveals the higher-dimensional structures generated by loop.

It turns out the improved theorem works even if D might not be a set, as long as h stays surjective; in particular, the naïve version can be recovered by choosing C itself as D and the identity function of C as h. In the following sections, however, we will still discuss the naïve version first and then show the improved version.

Since the naïve version can be recovered from the improved one, only the latter remains in the final version of mechanization. Overall, the structure of the Agda mechanization follows closely the informal argument. The combinatorial description code is defined in 600 lines of Agda code, and the entire mechanization is of roughly 1,200 lines.

Remark 3.3.1. There are many different theorems called Seifert–van Kampen in the classical literature, for example some assuming $C :\equiv \mathbb{1}$ and calculating the fundamental groups. Our version, especially the improved one, should match most versions up to homotopy (or be more powerful).

Remark 3.3.2. The section uses lemma 3.1.7 to reorganize the proofs in our paper [72] so that only standard higher inductive types are used: pushouts, truncation and set quotients. The mechanization was also redone accordingly.

3.3.1 The Naïve Version

Let $f : C \to A$ and $g : C \to B$ be given functions and $P :\equiv A \sqcup_{C;f;g} B$ be their pushout. The first step is to define a family of sets $\mathtt{code} : P \to P \to \mathcal{U}$ that should match the 0-truncated identification. By applying the univalence axiom and the (non-dependent) pushout elimination on both arguments, we need to consider several base cases and the coherence conditions among them induced by glue. There are four base cases:

- $\mathtt{code}(\mathtt{left}(a_0), \mathtt{left}(a_1))$ representing identifications from A to A, and
- $\mathtt{code}(\mathtt{left}(a_0), \mathtt{right}(b_1))$ representing identifications from A to B, and
- $\mathtt{code}(\mathtt{right}(b_0), \mathtt{left}(a_1))$ representing identifications from B to A, and
- $\mathtt{code}(\mathtt{right}(b_0), \mathtt{right}(b_1))$ representing identifications from B to B,

each being a set quotient of raw alternative sequences by some quotient relation making raw sequences behave like identifications. The coherence conditions, by the univalence axiom, include four equivalences,

$$\prod_{a:A}\prod_{c:C}\texttt{code}(\texttt{left}(a),\texttt{left}(f(c))) \simeq \texttt{code}(\texttt{left}(a),\texttt{right}(g(c)))$$
$$\prod_{b:A}\prod_{c:C}\texttt{code}(\texttt{right}(b),\texttt{left}(f(c))) \simeq \texttt{code}(\texttt{right}(b),\texttt{right}(g(c)))$$
$$\prod_{a:A}\prod_{c:C}\texttt{code}(\texttt{left}(f(c)),\texttt{left}(a)) \simeq \texttt{code}(\texttt{right}(g(c)),\texttt{left}(a))$$
$$\prod_{b:A}\prod_{c:C}\texttt{code}(\texttt{left}(f(c)),\texttt{right}(b)) \simeq \texttt{code}(\texttt{right}(g(c)),\texttt{right}(b)),$$

and a final coherence condition on these equivalences concerning the commutativity of the following square for all c_0, c_1 of type C:

$$\begin{tikzcd}
\texttt{code}(\texttt{left}(f(c_0)),\texttt{left}(f(c_1))) \arrow[r, equal] \arrow[d, equal] & \texttt{code}(\texttt{left}(f(c_0)),\texttt{right}(g(c_1))) \arrow[d, equal] \\
\texttt{code}(\texttt{right}(g(c_0)),\texttt{left}(f(c_1))) \arrow[r, equal] & \texttt{code}(\texttt{right}(g(c_0)),\texttt{right}(g(c_1))).
\end{tikzcd}$$

Our plan is to define these quotients separately and then show they satisfy all coherence conditions imposed by `glue`.

Four base cases. It turns out to be easier to define two raw sequences at once as mutually recursive inductive types. More precisely, let $\texttt{precode}_{\texttt{AA}} : A \to A \to \mathcal{U}$ be the family of raw sequences for the first case and $\texttt{precode}_{\texttt{AB}} : A \to B \to \mathcal{U}$ for the second case; they can be defined with these three generators by considering the last bridge crossed (if any):

- A generator meaning an identification *without* crossing:

$$\texttt{start}_{\texttt{AA}} : \prod_{a_0:A}\prod_{a_1:A} \|a_0 = a_1\|_0 \to \texttt{precode}_{\texttt{AA}}(a_0, a_1)$$

- Two generators meaning the last bridge is indexed by c:

$$\texttt{append}_{\texttt{AA}} : \prod_{a_0:A}\prod_{c:C}\prod_{a_1:A} \texttt{precode}_{\texttt{AB}}(a_0, g(c)) \to \|f(c) = a_1\|_0 \to \texttt{precode}_{\texttt{AA}}(a_0, a_1)$$
$$\texttt{append}_{\texttt{AB}} : \prod_{a_0:A}\prod_{c:C}\prod_{b_1:B} \texttt{precode}_{\texttt{AA}}(a_0, f(c)) \to \|g(c) = b_1\|_0 \to \texttt{precode}_{\texttt{AB}}(a_0, b_1)$$

Then, the families of set quotients $\mathtt{code}(\mathtt{left}(-), \mathtt{left}(-))$ and $\mathtt{code}(\mathtt{left}(-), \mathtt{right}(-))$ are constructed with respect to some relations

$$\mathtt{precode\text{-}rel}_{\mathtt{AA}} : \prod_{a_0:A} \prod_{a_1:A} \mathtt{precode}_{\mathtt{AA}}(a_0, a_1) \to \mathtt{precode}_{\mathtt{AA}}(a_0, a_1) \to \mathcal{U}$$
$$\mathtt{precode\text{-}rel}_{\mathtt{AB}} : \prod_{a_0:A} \prod_{b_1:B} \mathtt{precode}_{\mathtt{AB}}(a_0, b_1) \to \mathtt{precode}_{\mathtt{AB}}(a_0, b_1) \to \mathcal{U}.$$

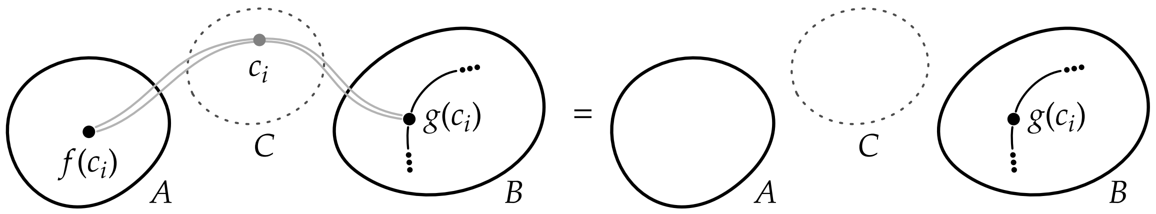
The intuition is that crossing a bridge and then immediately traveling back should be identified as not moving; this is obviously true for real identification and is enough to make the raw sequences behave as identification. Let

$$[a_0, p_0, c_0, q_0, c_0', \dots, p_n, a_1] \quad \text{and} \quad [a_0, p_0, c_0, q_0, c_0', \dots, q_n, b_1]$$

be the convenient flat-out notation for alternative sequences, where data at even positions, p_i's and q_i's, are 0-truncated identifications, and data at odd positions refer to end points of adjacent identifications. The above intuition can be phrased as the following two equations:

$$[\dots, p_i, c_i, \mathtt{refl}_{g(c_i)}, c_i, p_{i+1}, \dots] = [\dots, p_i \cdot p_{i+1}, \dots]$$
$$[\dots, q_i, c_i', \mathtt{refl}_{f(c_i')}, c_i', q_{i+1}, \dots] = [\dots, q_i \cdot q_{i+1}, \dots]$$

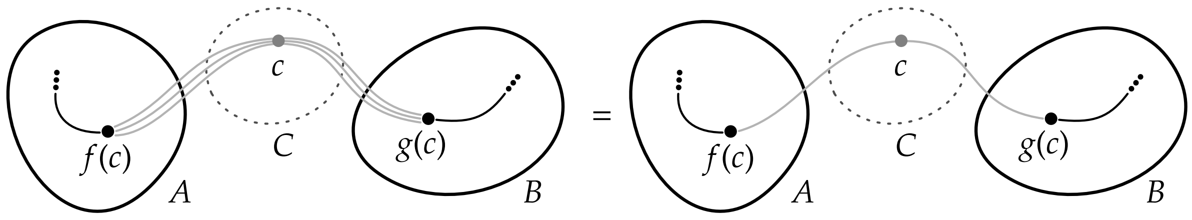
This can be viewed as removing the crossing of the same bridge twice in a row:



By the elimination rules of identification and truncation, it suffices to consider the cases where p_i or q_i is reflexivity; in other words, it is sufficient to remove any two consecutive reflexivity identifications:

$$[\dots, q, c, \mathtt{refl}_{f(c)}, c, \mathtt{refl}_{g(c)}, c, p, \dots] = [\dots, q, p, \dots]$$
$$[\dots, p, c, \mathtt{refl}_{g(c)}, c, \mathtt{refl}_{f(c)}, c, q, \dots] = [\dots, p, q, \dots]$$

This can be visualized as the cases where the same bridge is crossed three times in a row:



Formally, this quotient can be implemented by populating the quotient relations with the following four generators.

- Two generators for the removal of two reflexivity identifications at the very end:

$$\begin{aligned}\texttt{refl-refl}_{\texttt{AA}} : \prod_{a_0:A}\prod_{c:C}\prod_{(\alpha:\texttt{precode}_{\texttt{AA}}(a_0,f(c)))}\\ \texttt{precode-rel}_{\texttt{AA}}(\texttt{append}_{\texttt{AA}}(c,\texttt{append}_{\texttt{AB}}(c,\alpha,\texttt{refl}_{g(c)}),\texttt{refl}_{f(c)}),pc)\\ \texttt{refl-refl}_{\texttt{AB}} : \prod_{a_0:A}\prod_{c:C}\prod_{(\beta:\texttt{precode}_{\texttt{AB}}(a_0,g(c)))}\\ \texttt{precode-rel}_{\texttt{AB}}(\texttt{append}_{\texttt{AB}}(c,\texttt{append}_{\texttt{AB}}(c,\beta,\texttt{refl}_{f(c)}),\texttt{refl}_{g(c)}),pc)\end{aligned}$$

- Two generators for congruence:

$$\texttt{cong}_{\texttt{AA}} : \prod_{a_0:A}\prod_{c:C}\prod_{a_1:A}\prod_{(\beta_0:\texttt{precode}_{\texttt{AB}}(a_0,g(c)))}\prod_{(\beta_1:\texttt{precode}_{\texttt{AB}}(a_0,g(c)))}\prod_{(p:\|f(c)=b_1\|_0)}$$

$$\texttt{precode-rel}_{\texttt{AB}}(a_0,g(c),\beta_0,\beta_1) \to \texttt{precode-rel}_{\texttt{AA}}(a_0,a_1,\texttt{append}_{\texttt{AA}}(c,\beta_0,p),\texttt{append}_{\texttt{AA}}(c,\beta_1,p))$$

$$\texttt{cong}_{\texttt{AB}} : \prod_{a_0:A}\prod_{c:C}\prod_{b_1:B}\prod_{(\alpha_0:\texttt{precode}_{\texttt{AA}}(a_0,f(c)))}\prod_{(\alpha_1:\texttt{precode}_{\texttt{AA}}(a_0,f(c)))}\prod_{(p:\|g(c)=b_1\|_0)}$$

$$\texttt{precode-rel}_{\texttt{AA}}(a_0,f(c),\alpha_0,\alpha_1) \to \texttt{precode-rel}_{\texttt{AB}}(a_0,b_1,\texttt{append}_{\texttt{AB}}(c,\alpha_0,p),\texttt{append}_{\texttt{AB}}(c,\alpha_1,p))$$

The families of set quotients are then defined as follows:

$$\begin{aligned}\texttt{code}(\texttt{left}(a_0),\texttt{left}(a_1)) &:\equiv \texttt{precode}_{\texttt{AA}}(a_0,a_1)/\texttt{precode-rel}_{\texttt{AA}}(a_0,a_1)\\ \texttt{code}(\texttt{left}(a_0),\texttt{right}(b_1)) &:\equiv \texttt{precode}_{\texttt{AB}}(a_0,b_1)/\texttt{precode-rel}_{\texttt{AB}}(a_0,b_1).\end{aligned}$$

The other two families of sets, $\texttt{code}(\texttt{right}(-),\texttt{left}(-))$ and $\texttt{code}(\texttt{right}(-),\texttt{right}(-))$ are defined using the same construction.

Four equivalences. As a reminder, we need to establish

$$\begin{aligned}&\prod_{a:A}\prod_{c:C}\texttt{code}(\texttt{left}(a),\texttt{left}(f(c))) \simeq \texttt{code}(\texttt{left}(a),\texttt{right}(g(c)))\\ &\prod_{b:A}\prod_{c:C}\texttt{code}(\texttt{right}(b),\texttt{left}(f(c))) \simeq \texttt{code}(\texttt{right}(b),\texttt{right}(g(c)))\\ &\prod_{a:A}\prod_{c:C}\texttt{code}(\texttt{left}(f(c)),\texttt{left}(a)) \simeq \texttt{code}(\texttt{right}(g(c)),\texttt{left}(a))\\ &\prod_{b:A}\prod_{c:C}\texttt{code}(\texttt{left}(f(c)),\texttt{right}(b)) \simeq \texttt{code}(\texttt{right}(g(c)),\texttt{right}(b)).\end{aligned}$$

The first equivalence between $\mathtt{code}(\mathtt{left}(a), \mathtt{left}(f(c)))$ and $\mathtt{code}(\mathtt{left}(a), \mathtt{right}(g(c)))$ is simply appending reflexivity as follows:

$$[\dots, f(c)] \mapsto [\dots, c, \mathtt{refl}_{g(c)}, g(c)]$$
$$[\dots, c, \mathtt{refl}_{f(c)}, f(c)] \mapsfrom [\dots, g(c)]$$

which respects all the equations imposed by the quotient relations. Round trips are the identity function because the quotient relations will kill two consecutive reflexivity identifications at the end. The other three equivalences are essentially the same.

The commuting square. The equivalences in the diagram are all about adding reflexivity at the beginning or at the end. In this case, the commutativity basically states that appending an identification and then prepending another is the same as doing them in the other order, which can be verified by a routine calculation.

With these components one can assemble individual pieces into a coherent family of sets code, concluding the construction of the combinatorial description. We are ready to state the theorem.

Theorem 3.3.3 (naïve Seifert–van Kampen)**.**

$$\prod_{u,v:P} \|u =_P v\|_0 \simeq \mathtt{code}(u, v).$$

Proof. Again, we follow the standard recipe of equivalence by showing two functions inverse to each other. The encoding function is defined using transport:

$$\mathtt{encode} :\equiv \lambda(u, v{:}P).\lambda\big(p{:}\|u =_P v\|_0\big).\mathtt{transport}_0[v.\mathtt{code}(u, v)]\big(p; \mathtt{encode\text{-}refl}(u)\big)$$

assuming we have the proof for the reflexivity case:

$$\mathtt{encode\text{-}refl} : \prod_{u:P} \mathtt{code}(u, u).$$

By pushout elimination, it suffices to provide these components:

- An element c_A of type $\prod_{a:A} \mathtt{code}(\mathtt{left}(a), \mathtt{left}(a))$.
- An element c_B of type $\prod_{b:B} \mathtt{code}(\mathtt{right}(b), \mathtt{right}(b))$.
- $\prod_{c:C} c_A(f(c)) =^{u.\mathtt{code}(u,u)}_{\mathtt{glue}(c)} c_B(g(c))$.

The first two are easily satisfied by $[a, \mathtt{refl}_a, a]$ and $[b, \mathtt{refl}_b, b]$. The third component can be equivalently expressed in terms of transport:

$$\prod_{c:C} \mathtt{transport}[u.\mathtt{code}(u,u)]\Big(\mathtt{glue}(c); \big[f(c), \mathtt{refl}_{f(c)}, f(c)\big]\Big) = \big[g(c), \mathtt{refl}_{g(c)}, g(c)\big].$$

The transport along `glue` will essentially extract the equivalence data used in building the family of sets `code`; this transport, in particular, is moving from the upper left corner to the bottom right corner in the commuting square

$$\begin{tikzcd}
\mathtt{code}(\mathtt{left}(f(c_0)), \mathtt{left}(f(c_1))) \arrow[r, equal] \arrow[d, equal] \arrow[dr, equal] & \mathtt{code}(\mathtt{left}(f(c_0)), \mathtt{right}(g(c_1))) \arrow[d, equal] \\
\mathtt{code}(\mathtt{right}(g(c_0)), \mathtt{left}(f(c_1))) \arrow[r, equal] & \mathtt{code}(\mathtt{right}(g(c_0)), \mathtt{right}(g(c_1))).
\end{tikzcd}$$

Therefore, this transport is prepending and appending reflexivity on both ends. In other words, it suffices to show

$$\prod_{c:C} \big[g(c), \mathtt{refl}_{g(c)}, c, \mathtt{refl}_{f(c)}, c, \mathtt{refl}_{g(c)}, g(c)\big] = \big[g(c), \mathtt{refl}_{g(c)}, g(c)\big]$$

which is obvious from the quotient relations.

The other direction that decodes sequences is done by connecting fragmented identifications into one. This again respects the quotient relations because the quotient relations are designed to mimic real identification.

The round trip from truncated identification is again by elimination of identification and truncation. The round trip from code, on the other hand, is by induction on the alternative sequences; the critical step is to prove the following two equations where $+\!\!+$ is the code concatenation:

$$\mathtt{encode}(\mathtt{decode}([\dots, c, p, a])) = \mathtt{encode}(\mathtt{decode}([\dots, g(c)])) +\!\!+ [f(c), p, a]$$
$$\mathtt{encode}(\mathtt{decode}([\dots, c, q, b])) = \mathtt{encode}(\mathtt{decode}([\dots, f(c)])) +\!\!+ [g(c), q, b]$$

These are true because

$$\begin{aligned}
&\texttt{encode}\Big(\texttt{decode}\big([\dots, c, p, a]\big)\Big)\\
&\qquad\boxed{\text{the definition of decoding function}}\\
\equiv\ &\texttt{encode}\Big(\texttt{decode}\big([\dots, g(c)]\big) \cdot \texttt{glue}(c)^{-1} \cdot \texttt{ap}_f(p)\Big)\\
&\qquad\boxed{\text{functoriality of transport}}\\
=\ &\texttt{transport}_0[v.\texttt{code}(u,v)]\Big(\texttt{ap}_f(p); \texttt{transport}_0[v.\texttt{code}(u,v)]\big(\texttt{glue}(c)^{-1}; \texttt{encode}(\texttt{decode}([\dots, g(c)]))\big)\Big)\\
&\qquad\boxed{\text{transport along } \texttt{glue}(c) \text{ reveals the inverse function of the equivalence}}\\
=\ &\texttt{transport}_0[v.\texttt{code}(u,v)]\Big(\texttt{ap}_f(p); \texttt{encode}(\texttt{decode}([\dots, g(c)])) +\!\!+ [f(c), \texttt{refl}_{f(c)}, f(c)]\Big)\\
&\qquad\boxed{\begin{array}{l}\text{transport along } \texttt{ap}_f(p) \text{ at the second position extends the code;}\\ \text{this can be shown by considering the case } p \text{ is reflexivity}\end{array}}\\
=\ &\texttt{encode}\Big(\texttt{decode}([\dots, g(c)])\Big) +\!\!+ [f(c), p, a]
\end{aligned}$$

and the other case is similar. This concludes the equivalence and thus the theorem. □

3.3.2 Improvement with an Index Type

The improvement of Seifert–van Kampen presented now is motivated by a similar improvement in the classical theory, where C is equipped with a *set of base points D*, even though it turns out to be unnecessary for our proof to assume D to be actually a set, and the original version can be recovered as a result.

What is important is that D merely contains at least one element in each connected component of C, or equivalently there is a surjective function $h : D \to C$. Through lemma 3.1.7 on page 48, proving any theorem about C is essentially a matter of proving the theorem about D and showing relative constancy.

The new $\texttt{code} : P \to P \to \mathcal{U}$ is again defined by applying (non-dependent) pushout elimination on both arguments. The difference from the old one is that bridges are now indexed by $h(d)$ for some $d : D$ instead of a general $c : C$. More precisely, the generators $\texttt{append}_{\texttt{AA}}$ and $\texttt{append}_{\texttt{AB}}$ are replaced by

$$\begin{aligned}
\texttt{append}_{\texttt{AA}} &: \prod_{a_0:A}\prod_{d:D}\prod_{a_1:A} \texttt{precode}_{\texttt{AB}}(a_0, g(h(d))) \to \|f(h(d)) = a_1\|_0 \to \texttt{precode}_{\texttt{AA}}(a_0, a_1)\\
\texttt{append}_{\texttt{AB}} &: \prod_{a_0:A}\prod_{d:D}\prod_{b_1:B} \texttt{precode}_{\texttt{AA}}(a_0, f(h(d))) \to \|g(h(d)) = b_1\|_0 \to \texttt{precode}_{\texttt{AB}}(a_0, b_1).
\end{aligned}$$

A similar change is made to the other two base cases $\texttt{precode}_{\texttt{BA}}$ and $\texttt{precode}_{\texttt{BB}}$. Let

$$[a_0, p_0, d_0, q_0, d_0', \dots, p_n, a_1] \quad\text{and}\quad [a_0, p_0, d_0, q_0, d_0', \dots, q_n, b_1]$$

be the new flat-out notation with d's instead of c's. In order to establish the required equivalences later, the new quotients receive a new equation

$$[\dots, d_0, \mathtt{ap}_f(p), d_1, \mathtt{refl}_{g(h(d_1))}, d_1, \dots] = [\dots, d_0, \mathtt{refl}_{f(h(d_0))}, d_0, \mathtt{ap}_g(p), d_1, \dots]$$

which says the image of an identification p in C on different sides are identified. This was derivable in the original code by identification elimination on p, but not obviously derivable now because none of the end points of $p : \|h(d_0) = h(d_1)\|_0$ are free and thus identification elimination is not directly applicable. Formally, the equation is implemented by a new generator for the relation $\mathtt{precode\text{-}rel}_{\mathtt{AB}}$,

$$\begin{aligned}\mathtt{switch}_{\mathtt{AB}} : \prod_{a_0:A}\prod_{d_0:D}\prod_{d_1:D}\prod_{(\beta:\mathtt{precode}_{\mathtt{AB}}(a_0, g(h(d_0))))}\prod_{(p:\|h(d_0)=h(d_1)\|_0)} \\ \to \mathtt{precode\text{-}rel}_{\mathtt{AB}}(\mathtt{append}_{\mathtt{AB}}(d_1, \mathtt{append}_{\mathtt{AA}}(d_0, \beta, \mathtt{ap}_f(p)), \mathtt{refl}_{g(h(d_1))}), \\ \mathtt{append}_{\mathtt{AB}}(d_0, \mathtt{append}_{\mathtt{AA}}(d_0, \beta, \mathtt{refl}_{f(h(d_0))}), \mathtt{ap}_g(p)))\end{aligned}$$

and a similar one for $\mathtt{precode\text{-}rel}_{\mathtt{BB}}$ with a different starting side.

The four equivalences are essentially the same, which append or prepend reflexivity to code, except that we have to show relative constancy required by lemma 3.1.7: the requirement is that the supposed equivalences indexed by C must respect any identification $r : h(d_0) =_C h(d_1)$. Considering the equivalence between $\mathtt{code}(\mathtt{left}(a), \mathtt{left}(f(c)))$ and $\mathtt{code}(\mathtt{left}(a), \mathtt{right}(g(c)))$, this boils down to showing commutativity of appending reflexivity and transporting along the r itself; that is,

$$\begin{aligned}&\mathtt{transport}\Big[c.\mathtt{code}(\mathtt{left}(a), \mathtt{left}(f(c)))\Big]\Big(r; [\dots, f(h(d_0))]\Big) +\!\!+ \Big[g(h(d_1)), \mathtt{refl}_{g(h(d_1))}, g(h(d_1))\Big] \\ =\; &\mathtt{transport}\Big[c.\mathtt{code}(\mathtt{left}(a), \mathtt{right}(g(c)))\Big]\Big(r; [\dots, d_0, \mathtt{refl}_{g(h(d_0))}, g(h(d_0))]\Big).\end{aligned}$$

This is proved by

$$\begin{aligned}&\mathtt{transport}\Big[c.\mathtt{code}(\mathtt{left}(a), \mathtt{left}(f(c)))\Big]\Big(r; [\dots, f(h(d_0))]\Big) +\!\!+ \Big[g(h(d_1)), \mathtt{refl}_{g(h(d_1))}, g(h(d_1))\Big] \\ &\quad \boxed{\text{by identification elimination on } r \text{ and the quotient relation}} \\ =\; &\Big[\dots, d_0, \mathtt{refl}_{f(h(d_0))}, d_0, \mathtt{ap}_f|r|_0, g(h(d_1))\Big] +\!\!+ \Big[g(h(d_1)), \mathtt{refl}_{g(h(d_1))}, g(h(d_1))\Big] \\ \equiv\; &\Big[\dots, d_0, \mathtt{refl}_{f(h(d_0))}, d_0, \mathtt{ap}_f|r|_0, d_1, \mathtt{refl}_{g(h(d_1))}, g(h(d_1))\Big] \\ &\quad \boxed{\text{by } \mathtt{switch}_{\mathtt{AB}}} \\ =\; &\Big[\dots, d_0, \mathtt{refl}_{f(h(d_1))}, d_0, \mathtt{refl}_{g(h(d_1))}, d_0, \mathtt{ap}_g|r|_0, g(h(d_1))\Big] \\ &\quad \boxed{\text{by identification elimination on } r \text{ and the quotient relation}} \\ =\; &\mathtt{transport}\Big[c.\mathtt{code}(\mathtt{left}(a), \mathtt{right}(g(c)))\Big]\Big(r; \Big[\dots, d_0, \mathtt{refl}_{g(h(d_0))}, g(h(d_0))\Big]\Big).\end{aligned}$$

Other equivalences can be established in a similar way.

As for the commutativity condition, the constancy requirement is free because the commutativity itself is a mere proposition. The construction of the new code $\texttt{code}(u, v)$ is thus completed.

Theorem 3.3.4 (Seifert–van Kampen with an index type)**.**

$$\prod_{u,v:P} \|u =_P v\|_0 \simeq \texttt{code}(u, v).$$

Proof. The proof is almost the same except for two spots:

- The new decoding function needs to respect the new equations in the quotient relation such as $\texttt{switch}_{\texttt{AB}}$; essentially we want the following identification for any $p : c_0 = c_1$:

$$\texttt{ap}_{\texttt{left}}(\texttt{ap}_f(p)) \cdot \texttt{glue}(c_1) = \texttt{glue}(c_0) \cdot \texttt{ap}_{\texttt{right}}(\texttt{ap}_g(p))$$

  which is exactly the naturality of $\texttt{glue}$. This can be shown by identification elimination on p.

- When showing the round trip from the code is the identity function, it is critical to extract the equivalences in $\texttt{code}(u, v)$ when transporting along $\texttt{glue}$. This is again done by lemma 3.1.7, where the required constancy condition is free because an identification between the round trip and the identity function is a mere proposition.

□

3.4 The Blakers–Massey Theorem

This is joint work with Eric Finster, Dan Licata and Peter LeFanu Lumsdaine and this section incorporates text from the paper [71].

In previous sections we have seen covering spaces and the Seifert–van Kampen theorem for calculating the fundamental groups. Higher homotopy groups, however, defy a straightforward generalization of those tools. Here we present one of the few known tools for computing the higher homotopy groups—the Blakers–Massey theorem, which characterizes the higher homotopy groups of a pushout $A \sqcup_{C;f;g} B$ in a certain range, depending on the connectivity of f and g.

The theorem states the connectivity of identification generator $\texttt{glue}$, which links the structures in the type C and those in the pushout (shifted by one dimension), is the sum of the connectivity levels of f and g; that is, if f induces isomorphisms up to dimension m and g up to dimension n, then the type C and the pushout share the homotopy groups up to dimensions $m + n$. Because information about the homotopy groups of the apex type C chosen to make a pushout is often known, this is a useful way to obtain information about the homotopy groups of the pushout itself. For example, it has as a special case the Freudenthal suspension theorem, which was used in past mechanization to calculate the n^{th} homotopy group of the n-dimensional sphere [88] and to verify the correctness of a construction of Eilenberg–Mac Lane spaces [91], and implies stability of the homotopy groups of spheres (in a certain range, increasing both the dimension of the sphere and the homotopy group by one gives the same group).

The Blakers–Massey theorem heavily relies on the connectivity of functions, yet it is rather clumsy to handle using `hfiber`. Using the family-as-function correspondence in section 3.1.7 on page 38, we can transform the pushout $A \sqcup_{C;f;g} B$ into an equivalent pushout

$$A \sqcup_{\sum_{a:A} \sum_{b:B} Q(a,b); p_A; p_B} B$$

where

$$\begin{aligned} &Q : A \to B \to \mathcal{U} \\ &Q :\equiv \lambda(a{:}A).\lambda(b{:}B).\sum_{c:C}\Big(\big(f(c) = a\big) \times \big(g(c) = b\big)\Big) \end{aligned}$$

$$\begin{aligned} &p_A : \sum_{a:A}\sum_{b:B} Q(a,b) \to A \qquad && p_B : \sum_{a:A}\sum_{b:B} Q(a,b) \to B \\ &p_A :\equiv \texttt{fst} && p_B :\equiv \texttt{fst} \circ \texttt{snd}. \end{aligned}$$

With the new family Q, the fibers of f and g can be expressed as follows:

$$\begin{aligned} \texttt{hfiber}_f(a) &\simeq \sum_b Q(a,b) \\ \texttt{hfiber}_g(b) &\simeq \sum_a Q(a,b) \end{aligned}$$

and thus the connectivity of f and g, which was defined as connectivity of their fibers, is now stated as connectivity of $\sum_b Q(a,b)$ and $\sum_a Q(a,b)$. This way we can contain the overwhelming `hfiber` construction. As a convenient notation, we define $\texttt{qglue}_{a;b}$ of type $Q(a,b) \to \texttt{left}(a) = \texttt{right}(b)$ to be

$$\texttt{qglue}_{a;b} :\equiv \lambda(q{:}Q(a,b)).\texttt{glue}\big(\langle a; \langle b; q\rangle\rangle\big)$$

and the subscript may be dropped if clear from the context. The remainder of this section will directly start from a family of Q indexed by A and B along with the canonical projections p_A and p_B defined above.

3.4.1 Formulation

Theorem 3.4.1 (Blakers–Massey theorem)**.** *Let A and B be types, and Q a family $A \to B \to \mathcal{U}$. Suppose $m, n \geq -1$, and for each $a : A$ the type $\sum_{b:B} Q(a,b)$ is m-connected, and dually* *for each $b : B$ the type $\sum_{a:A} Q(a,b)$ is n-connected. Then for each $a : A$ and $b : B$, the map* $\mathtt{qglue}_{a;b} : Q(a,b) \to \mathtt{left}(a) = \mathtt{right}(b)$ *is $(m+n)$-connected, where* $\mathtt{left}(a) = \mathtt{right}(b)$ *is an identification type of the pushout $A \sqcup_{\sum_{a:A}\sum_{b:B} Q(a,b);p_A;p_B} B$ and p_A and p_B are the canonical projections from $\sum_{a:A}\sum_{b:B} Q(a,b)$ to A and B, respectively.*

Remark 3.4.2. Again, as noted in remark 3.1.1, the mechanized version replaces m and n by $\mathtt{succ}(m)$ and $\mathtt{succ}(n)$, respectively, replaces $m + n$ by $m \hat{+} n$, and drops the condition $m, n \geq -1$. See also section 4.3.

For the rest of this section, fix A, B, Q, m, n as in the theorem, and define P as the pushout _∑_a:A∑_b:BQ(a,b);p_A;p_B for brevity. Unwinding the definition of connectivity of a function, it is the statement:

$$\prod_{a_0:A}\prod_{b:B}\prod_{(r:\mathtt{left}(a_0)=\mathtt{right}(b))} \mathtt{is\text{-}connected}_{m+n}\Big(\mathtt{hfiber}_{\mathtt{qglue}_{a_0;b}}(r)\Big) \tag{3.1}$$

We single out a_0 with a subscript, because we will fix it throughout the rest of the proof. Recalling that

$$\mathtt{is\text{-}connected}_{m+n}(D) :\equiv \mathtt{is\text{-}contr}\,\|D\|_{m+n},$$

this unwound form of the theorem can be thought of as saying that for every identification $r : \mathtt{left}(a_0) = \mathtt{right}(b)$, there is an element $q : Q(a_0, b)$ (in the domain of $\mathtt{qglue}_{a_0;b}$) that is a kind of explicit representation or canonical form for r, up to level $m + n$.

Overall, the (perhaps rather mysterious) connectivity hypotheses are used twice: once rather weakly, to supply some extra auxiliary assumptions, and once more substantially to apply the wedge connectivity lemma. It is the wedge connectivity lemma that gives rise to the additivity of connectivity in the conclusion.

3.4.2 Definition of Code

To prove eq. (3.1), we want to apply identification elimination to the identification r, so that we only have to consider the case where r is reflexivity. However, by the argument on page 16, this requires either $\mathtt{left}(a_0)$ or $\mathtt{right}(b)$ to be "free"—generalized over the whole pushout. So we want to generalize the original goal to the statement

$$\prod_{p:P}\prod_{(r:\mathtt{left}(a_0)=p)} \mathtt{is\text{-}contr}\Big(\mathtt{code}(p, r)\Big) \tag{3.2}$$

for some family of types

$$\mathtt{code} : \prod_{p:P}(\mathtt{left}(a_0) = p) \to \mathcal{U}$$

such that

$$\mathtt{code}(\mathtt{right}(b_1), r) \equiv \Big\|\mathtt{hfiber}_{\mathtt{qglue}_{a_0;b_1}}(r)\Big\|_{m+n}.$$

Recall that $\texttt{is-connected}_n(D)$ is defined as $\texttt{is-contr}\,\|D\|_n$, and so, with this definition of $\texttt{code}$ for $\texttt{right}$, $\texttt{is-contr}(\texttt{code}(\texttt{right}(b_1), r))$ is exactly the original goal. The $\texttt{code}(p, r)$ can be thought of as a type of explicit characterizations or canonical forms for (the $(m+n)$-level information in) an identification r with an end point anywhere in the pushout.

The family $\texttt{code}$ will be defined by applying the pushout eliminator on $p : P$, so as demonstrated many times in previous sections, we need to feed these three components:

- $\texttt{code}\Big(\texttt{left}(a_1), (r{:}\texttt{left}(a_0){=}\texttt{left}(a_1))\Big)$ for any $a_1 : A$; and
- $\texttt{code}\Big(\texttt{right}(b_1), (r{:}\texttt{left}(a_0){=}\texttt{right}(b_1))\Big)$ for any $b_1 : B$, which is defined as above to be $\Big\|\texttt{hfiber}_{\texttt{qglue}_{a_0;b_1}}(r)\Big\|_{m+n}$; and
- $\texttt{apd}_{\texttt{code}}\Big(\texttt{qglue}_{a_1,b_1}(q)\Big)$ for any $a_1 : A$, $b_1 : B$ and $q : Q(a_1, b_1)$.

The difficulty is to find the analogue of the theorem for the $\texttt{left}(a_1)$ case, that is, when both the end points of r are in A. Our trick is to make our assumptions more symmetric, by supposing we have some distinguished $b_0 : B$ and $q_{0;0} : Q(a_0, b_0)$ while defining $\texttt{code}$; like a_0, b_0 and $q_{0;0}$ will be fixed through most of the rest of the argument. We will show that we can discharge these extra assumptions toward the end of the section.

The list of three needed components for defining $\texttt{code}$ remain the same, as does the definition in the $\texttt{right}$ case, but for the other cases we will now make use of newly added arguments b_0 and $q_{0;0}$. In terms of diagrams, the above definition of the $\texttt{right}$ case $\texttt{code}(\texttt{right}(b_1), (r{:}\texttt{left}(a_0){=}\texttt{right}(b_1)))$, which was chosen to make our generalization imply the original theorem, can be drawn as follows.[7] It is the type of all $q_{0;1}$'s such that

$$\begin{tikzcd}
\texttt{left}(a_0) \arrow[dr, "r"] & & & \texttt{left}(a_0) \arrow[dr, "\texttt{qglue}(q_{0;1})"] & \\
& \texttt{left}(b_1) & = & & \texttt{left}(b_1)
\end{tikzcd} \qquad (3.3)$$

For the $\texttt{left}$ case, we define

$$\texttt{code}\Big(\texttt{left}(a_1), (r{:}\texttt{left}(a_0){=}\texttt{left}(a_1))\Big) :\equiv \Big\|\texttt{hfiber}_{\lambda q_{1;0}.\texttt{qglue}(q_{0;0})\cdot\texttt{qglue}(q_{1;0})^{-1}}(r)\Big\|_{m+n}.$$

[7]Truncations are ignored in diagrams.

This represents (the truncation of) the type of all $q_{1;0}$'s such that

$$\begin{tikzcd}
\mathtt{left}(a_0) \arrow[d, "r"] & \mathtt{left}(a_0) \arrow[r, "\mathtt{qglue}(q_{0;0})"] & \mathtt{right}(b_0) \arrow[dl, "\mathtt{qglue}(q_{1;0})^{-1}"] \\
\mathtt{left}(a_1) & \mathtt{left}(a_1) &
\end{tikzcd}$$

The remaining missing piece is $\mathtt{apd}_{\mathtt{code}}(\mathtt{qglue}(q_{1;1}))$, that is, to show the above types are equivalent when there is $q_{1;1} : Q(a_1, b_1)$ such that $\mathtt{qglue}(q_{1;1})$ connects $\mathtt{left}(a_1)$ to $\mathtt{right}(b_1)$. This boils down to an equivalence between the type $\mathtt{code}(\mathtt{left}(a_1))$ *transported along* $\mathtt{qglue}(q_{1;1})$, and the type $\mathtt{code}(\mathtt{right}(b_1))$. Pictorially, the type $\mathtt{code}(\mathtt{left}(a_1))$ after transportation maps each $r : \mathtt{left}(a_0) = \mathtt{right}(b_1)$ to a type of all $q_{1;0}$'s such that

$$\begin{tikzcd}
\mathtt{left}(a_0) \arrow[dr, "r"] & \mathtt{right}(b_0) & \mathtt{left}(a_0) \arrow[r, "\mathtt{qglue}(q_{0;0})"] & \mathtt{right}(b_0) \arrow[dl, "\mathtt{qglue}(q_{1;0})^{-1}"] \\
\mathtt{left}(a_1) & \mathtt{right}(b_1) & \mathtt{left}(a_1) \arrow[r, "\mathtt{qglue}(q_{1;1})"'] & \mathtt{right}(b_1).
\end{tikzcd} \tag{3.4}$$

The goal is to show, for any r, there is an equivalence between the truncation of the type of all $q_{0;1}$'s satisfying eq. (3.3) and that of all $q_{1;0}$'s satisfying eq. (3.4). It turns out that this equivalence is non-trivial and heavily relies on the connectivity conditions in the Blakers–Massey theorem.

We can slightly simplify the needed equivalence by eliminating the middle r; ignoring the truncation for the moment, essentially we want to prove that for any $q_{0;1}$ there is a $q_{1;0}$ such that the equation

$$\begin{tikzcd}
\mathtt{left}(a_0) \arrow[dr, "\mathtt{qglue}(q_{0;1})"] & \mathtt{right}(b_0) & \mathtt{left}(a_0) \arrow[r, "\mathtt{qglue}(q_{0;0})"] & \mathtt{right}(b_0) \arrow[dl, "\mathtt{qglue}(q_{1;0})^{-1}"] \\
\mathtt{left}(a_1) & \mathtt{right}(b_1) & \mathtt{left}(a_1) \arrow[r, "\mathtt{qglue}(q_{1;1})"'] & \mathtt{right}(b_1)
\end{tikzcd}$$

is true, and vice versa. Then we show that these two functions are inverse to each other, which establishes an equivalence between all $q_{0;1}$'s and $q_{1;0}$'s. In this section we will only demonstrate the direction from $q_{1;0}$ to $q_{0;1}$ as the other is symmetric. Putting back the truncations, we wish to show

$$\prod_{q_{1;0}:Q(a_1,b_0)} \left\| \Sigma_{q_{0;1}:Q(a_0,b_1)} \mathtt{qglue}(q_{0;1}) = \mathtt{qglue}(q_{0;0}) \cdot \mathtt{qglue}(q_{1;0})^{-1} \cdot \mathtt{qglue}(q_{1;1}) \right\|_{m+n}.$$

The idea is to *reorder* all the hypotheses (including abstracting over the fixed a_0, b_0 and $q_{0;0}$) to match the wedge connectivity theorem. After the reordering the lemma is

$$\prod_{a_1:A} \prod_{b_0:B} \prod_{(q_{1;0}:Q(a_1,b_0))} \prod_{a_0:A} \prod_{(q_{0;0}:Q(a_0,b_0))} \prod_{b_1:B} \prod_{(q_{1;1}:Q(a_1,b_1))}$$
$$\left\| \Sigma_{q_{0;1}:Q(a_0,b_1)} \mathtt{qglue}(q_{0;1}) = \mathtt{qglue}(q_{0;0}) \cdot \mathtt{qglue}(q_{1;0})^{-1} \cdot \mathtt{qglue}(q_{1;1}) \right\|_{m+n}$$

and after grouping those arguments depending on a_0 or b_1 it is

$$\prod_{a_1:A} \prod_{b_0:B} \prod_{(q_{1;0}:Q(a_1,b_0))} \prod_{(u:\Sigma_{a_0:A} Q(a_0,b_0))} \prod_{(v:\Sigma_{b_1:B} Q(a_1,b_1))}$$
$$\left\| \Sigma_{q_{0;1}:Q(a_0,b_1)} \mathtt{qglue}(q_{0;1}) = \mathtt{qglue}(\mathtt{snd}(u)) \cdot \mathtt{qglue}(q_{1;0})^{-1} \cdot \mathtt{qglue}(\mathtt{snd}(v)) \right\|_{m+n}.$$

This may be visualized as

$$\begin{tikzcd}
\mathtt{left}(\mathtt{fst}(u)) \arrow[dr, "\mathtt{fst}(p(u,v))"] & \mathtt{right}(b_0) & & \mathtt{left}(\mathtt{fst}(u)) \arrow[r, "\mathtt{qglue}(\mathtt{snd}(u))"] & \mathtt{right}(b_0) \arrow[dl, "\mathtt{qglue}(q_{1;0})^{-1}"'] \\
\mathtt{left}(a_1) & \mathtt{right}(\mathtt{fst}(v)) & = & \mathtt{left}(a_1) \arrow[r, "\mathtt{qglue}(\mathtt{snd}(v))"'] & \mathtt{right}(\mathtt{fst}(v)).
\end{tikzcd}$$

In intuition the variable u determines the upper arm and the variable v determines the lower arms; with the wedge connectivity theorem and proper truncations, we only have to consider the cases either $q_{0;0}$ (or $\mathtt{snd}(u)$) collides with $q_{1;0}$ or $q_{1;1}$ (or $\mathtt{snd}(v)$) collides with $q_{1;0}$, as long as we give a coherent choice when both collide. Formally, we will apply lemma 3.1.3 on page 45 (wedge connectivity) to the wedge $U \vee V$ where the two pointed types U and V are

$$U :\equiv \left\langle \left(\sum_{a:A} Q(a, b_0) \right); \langle a_1; q_{1;0} \rangle \right\rangle \qquad \text{(the upper arm)}$$

$$V :\equiv \left\langle \left(\sum_{b:B} Q(a_1, b) \right); \langle b_0; q_{1;0} \rangle \right\rangle \qquad \text{(the lower arm)}$$

and the motive $P : U \to V \to \mathcal{U}$ is

$$P :\equiv \lambda u.\lambda v.\left\|\mathtt{hfiber}_{\mathtt{qglue}_{\mathtt{fst}(u);\mathtt{fst}(v)}}(\mathtt{qglue}(\mathtt{snd}(u)) \cdot \mathtt{qglue}(q_{1;0})^{-1} \cdot \mathtt{qglue}(\mathtt{snd}(v)))\right\|_{m+n}.$$

The connectivities of U and V are exactly the original hypotheses of Blakers–Massey, and the truncation level of P is forced by the explicit truncation. $\langle a_1; q_{1;0}\rangle$ is the distinguished element of U and $\langle b_0; q_{1;0}\rangle$ is the distinguished element of V, which signifies the collision of either $q_{0;0}$ or $q_{1;1}$ with $q_{1;0}$. The element f, which represents the case where the upper arm $q_{0;0}$ collides to the diagonal $q_{1;0}$, is the truncated pair $|\langle q_{1;1}; s\rangle|_{m+n}$ where s witnesses the groupoid law

$$\mathtt{qglue}(q_{1;1}) = \mathtt{qglue}(q_{1;0}) \cdot \mathtt{qglue}(q_{1;0})^{-1} \cdot \mathtt{qglue}(q_{1;1}).$$

On the other hand, the element g, which represents the case where the lower arm $q_{1;1}$ collides to the diagonal $q_{1;0}$, is the truncated pair $|\langle q_{1;1}; t\rangle|_{m+n}$ where t witnesses the groupoid law

$$\mathtt{qglue}(q_{0;0}) = \mathtt{qglue}(q_{0;0}) \cdot \mathtt{qglue}(q_{1;0})^{-1} \cdot \mathtt{qglue}(q_{1;0}).$$

We further demand that s and t reduce to `refl` when its inputs are `refl`, which are automatically true if they are naturally defined by identification elimination. The last argument α is to show that f and g agree on $q_{1;0}$, which follows from the fact that $q_{1;0}$ is picked when both arms collide with the diagonal and that s and t agree when everything is `refl`. In sum, these define a function from $q_{1;0}$'s to $q_{0;1}$'s.

The other direction can be defined similarly, and through the same technique one can show they are inverse to each other. This eventually fills the final piece of type

$$\mathtt{apd}_{\mathtt{code}}\left(\mathtt{qglue}_{a_1;b_1}(q_{1;1})\right)$$

and concludes the construction of `code`. In total, this construction is the largest part of the proof, and consists of approximately 400 lines of Agda code. In the verification that the two functions are inverse, the mechanization involves some clever abstraction over portions of the proof that can be shared between both the f and g cases of lemma 3.1.3 (wedge connectivity), which simplifies showing the final α coherence assumption of this lemma.

It remains to show the contractibility of $\mathtt{code}(p, r)$, for each $p : P$ and $r : \mathtt{left}(a_0) = p$.

3.4.3 Contractibility of Code

A straightforward way to show contractibility of a type is to follow the definition of `is-contr`, which demands a *center*, an element of that type, and a *contraction*, an identification from the center to every other element.

Centers. So we want to give an element of $\mathtt{code}(p,r)$, for each $p : P$ and $r : \mathtt{left}(a_0) = p$. By identification elimination on r, it is enough to give an element of $\mathtt{code}(\mathtt{left}(a_0), \mathtt{refl}_{a_0})$. By the construction of $\mathtt{code}$, this type reduces to

$$\left\|\mathtt{hfiber}_{\lambda q_{1;0}.\mathtt{qglue}(q_{0;0})\cdot\mathtt{qglue}(q_{1;0})^{-1}}(\mathtt{refl}_{a_0})\right\|_{m+n}$$

which is inhabited by the truncated pair

$$\left|\langle q_{0;0}; \mathtt{inv\text{-}r}(\mathtt{qglue}(q_{0;0}))\rangle\right|_{m+n}$$

where $\mathtt{inv\text{-}r}(p{:}a{=}b) : p \cdot p^{-1} = \mathtt{refl}_a$ is a proof that any identification concatenated with its inverse is homotopic to $\mathtt{refl}$, which is a groupoid law that can be defined as

$$\mathtt{inv\text{-}r}\big(p{:}a{=}b\big) :\equiv \mathtt{elim}_{a=}[b.p.p \cdot p^{-1} = \mathtt{refl}_a](\mathtt{refl}_{\mathtt{refl}_a}; b; p).$$

Putting these together, we obtain the desired centers:

$$\mathtt{code\text{-}center} : \prod_{p:P} \prod_{r:\mathtt{left}(a_0)=p} \mathtt{code}(p,r)$$

$$\mathtt{code\text{-}center} = \lambda(p{:}P).\lambda r.\mathtt{elim}_{\mathtt{left}(a_0)=}\Big[p.r.\mathtt{code}(p,r)\Big]\Big(\big|\langle q_{0;0}; \mathtt{inv\text{-}r}(\mathtt{qglue}(q_{0;0}))\rangle\big|_{m+n}; r\Big)$$

Contractions. We now want an identification to each $\mathtt{code}(p,r)$ from $\mathtt{code\text{-}center}(p,r)$. The type $\sum_{p:P} \mathtt{left}(a_0) = p$ of pairs of such p and r is contractible, so it is enough to give a contraction for any specific pair. We give it for the pair $\langle \mathtt{right}(b_0); \mathtt{qglue}(q_{0;0})\rangle$; but to do this, we step back to an intermediate generality, and show

$$\prod_{b_1:B} \prod_{(r:\mathtt{left}(a_0)=\mathtt{right}(b_1))} \prod_{(c:\mathtt{code}(\mathtt{right}(b_1),r))} \mathtt{code\text{-}center}\big(\mathtt{right}(b_1), r\big) = c$$

i.e. the case where p is $\mathtt{right}(b_1)$ for some $b_1 : B$, and r is arbitrary. The re-generalization of r is needed for the identification elimination below.

By construction, $\mathtt{code}(\mathtt{right}(b_1), r)$ is just $\|\mathtt{hfiber}_{\mathtt{qglue}}(r)\|_{m+n}$. Using elimination rules of truncations, $\mathtt{hfiber}$ (as pairs), and identifications, we may assume that c is of the form $|\langle q_{0;1}; \mathtt{refl}_{\mathtt{qglue}(q_{0;1})}\rangle|_{m+n}$ for some $q_{0;1} : Q(a_0, b_1)$, and r is $\mathtt{qglue}(q_{0;1})$. So it remains to show

$$\mathtt{code\text{-}center}\big(\mathtt{right}(b_1), \mathtt{qglue}(q_{0;1})\big) = \big|\langle q_{0;1}; \mathtt{refl}_{\mathtt{qglue}(q_{0;1})}\rangle\big|_{m+n}$$

This is mostly a routine calculation; roughly speaking, $\mathtt{code\text{-}center}(\mathtt{right}(b_1), \mathtt{qglue}(q_{0;1}))$ can be turned into a transport of the $\mathtt{code\text{-}center}$ at $(\mathtt{left}(a_0), \mathtt{refl}_{\mathtt{left}(a_0)})$ along the identification $\mathtt{qglue}(q_{0;1})$, which extracts the equivalence we put into the construction of $\mathtt{code}$, and then the rest carries out the reduction to $|\langle q_{0;1}; \mathtt{refl}_{\mathtt{qglue}(q_{0;1})}\rangle|_{m+n}$. (Full details can be found in section 4.3.)

This gives a contraction from $\mathtt{code\text{-}center}$ on $\mathtt{code}(\mathtt{right}(b_1), r)$, for any b_1 and r. In particular, it gives us a contraction on $\mathtt{code}(\mathtt{right}(b_0), (\mathtt{qglue}(g_{0;0})))$, and hence, by the contractibility of the type of pairs $\langle p; r\rangle$, on each type $\mathtt{code}(p,r)$, as required for the our second generalization of Blakers–Massey.

3.4.4 Theorem

We have shown our second generalization assuming fixed $a_0 : A$, $b_0 : B$ and $q_{0;0} : Q(a_0, b_0)$ (which were implicit parameters in the above constructions). Making these explicit, we have shown

$$\prod_{a_0} \prod_{b_0} \prod_{(q_{0;0}:Q(a_0,b_0))} \prod_{p:P} \prod_{(r:\texttt{left}(a_0)=p)} \texttt{is-contr}\Big(\texttt{code}(a_0, b_0, q_{0;0}, p, r)\Big)$$

We need to show that this implies the first generalization (eq. (3.2)) without access to $b_0 : B$ nor $q_{0;0} : Q(a_0, b_0)$:

$$\prod_{a_0} \prod_{p:P} \prod_{(r:\texttt{left}(a_0)=p)} \texttt{is-contr}\Big(\texttt{code}(a_0, p, r)\Big) \tag{3.5}$$

which, by definition of `code`, immediately implies the original Blakers–Massey by taking p of the form $\texttt{right}(b)$.

So the only remaining gap is the extra assumptions b_0 and $q_{0;0}$ in the second generalization. Note that eq. (3.5) is a mere proposition, and we assumed that m is at least -1, so the context is *a fortiori* m-truncated by cumulativity of truncation levels. Our connectivity assumptions say that $\sum_{b:B} Q(a_0, b)$ is m-connected, so in particular its m-truncation is inhabited. Because the context is an m-type, we can eliminate the truncation to obtain some element of it; i.e. a pair $\langle b_0; q_{0;0}\rangle$ as desired. This closes the gap and finishes the proof of the main theorem.

3.5 Study of Ordinary Eilenberg–Steenrod Cohomology

This is joint work with Ulrik Buchholtz.

In this section we change the subject from homotopy groups to *cohomology groups,* the structure of *functions from cycles* in a type. In a very broad sense, the study of cohomology groups, *cohomology theory,* still belongs to homotopy theory because it respects homotopy equivalence and has deep connections with proper homotopy theory. There are several ways to define a theory of such groups, some combinatorial and some axiomatic; amazingly, the classical theory states that the two approaches are essentially equivalent. Our goal is to recreate such an equivalence in UniTT+HIT and AGDA.

In this section we focus on a particular class of types, *CW complexes,* which come with an explicit description of how the type is built from lower dimensional structures to higher dimensions. We will then introduce *cellular cohomology theory,* a combinatorial cohomology theory specifically defined for CW complexes. After that, we will define *ordinary Eilenberg–Steenrod cohomology theory,* an axiomatic framework for cohomology theory, and then state our equivalence conjecture. A critical lemma toward the conjecture is then stated and proved.

Notation	Types	Pointed types	Groups
$A \to B$	arrows	pointed arrows *(only in diagrams)*	group homomorphisms
$\prod_{a:A} B$	functions	*(not reused)*	direct products
$A \simeq B$	equivalence	point-preserving equivalence	group isomorphism
$\mathbb{0}$	the empty type	*(not reused)*	the trivial group
$\mathbb{1}$	the unit type	the unit type	*(not reused)*
$A \times B$	binary products	*(not reused)*	binary direct products
$\mathtt{susp}(A)$	suspensions	pointed suspensions	*(not reused)*
$\mathtt{cofiber}(f)$	cofibers	pointed cofibers	*(not reused)*
A/B	set quotients	pointed cofibers of inclusions	group quotients

Table 3.2: Abuse of notation in section 3.5.

Throughout this section, I reuse the symbol of arrows ($\to$), functions ($\prod$), equivalence ($\simeq$), the unit ($\mathbb{1}$), binary products ($\times$) and others for pointed types and groups as in table 3.2. Pointed arrows are still marked as $X \cdot\!\!\to Y$ (except in diagrams in the category of pointed types) because pointedness of functions plays an important role in the definition of *degrees* that will be discussed later.

3.5.1 CW complexes

A CW complex is an inductively defined type built by attaching cells, starting from points at the zeroth dimension, lines at the first dimension, faces at the second, and so on. The description consists of A_n as the set of cells at dimension n, along with functions α_n denoting how cells are attached. The following recursive definition was adapted from Ulrik Buchholtz's work in Lean [31].

Let X_n be the construction up to dimension n. A cell $a : A_{n+1}$ at dimension $n + 1$ is specified by its boundary in X_n, denoted by a function from $\mathbb{S}^n$ to the type X_n; the type X_{n+1} is then the result after attaching all cells at dimension $n + 1$ to X_n. More formally, α_{n+1} is a function from $A_{n+1} \times \mathbb{S}^n$ to X_n describing the boundary of each cell. Inductively, the starting type X_0 is the set A_0, and the type X_{n+1} is defined to be the pushout $X_n \sqcup_{A_{n+1} \times \mathbb{S}^n; \alpha_{n+1}; \mathtt{fst}} A_{n+1}$:

$$\begin{tikzcd}
A_{n+1} \times \mathbb{S}^n \arrow[r, "\texttt{fst}"] \arrow[d, "\alpha_{n+1}"'] & A_{n+1} \arrow[d] \\
X_n \arrow[r] & X_{n+1}
\end{tikzcd}$$

As a first approximation, only finite-dimensional CW complexes are considered, which means the building process stops at some finite dimension. Pictorially, a (finite) CW complex is the following iterated pushout, starting from the type $X_0 :\equiv A_0$ and ending at some dimension.

$$\begin{tikzcd}
 & A_{n+1} \times \mathbb{S}^n \arrow[r, "\texttt{fst}"] \arrow[d, "\alpha_{n+1}"] & A_{n+1} \arrow[dr] & A_{n+2} \times \mathbb{S}^{n+1} \arrow[r, "\texttt{fst}"] \arrow[d, "\alpha_{n+2}"] & A_{n+2} \arrow[dr] & & \\
\cdots \arrow[r] & X_n \arrow[rr] & & X_{n+1} \arrow[rr] & & X_{n+2} \arrow[r] & \cdots
\end{tikzcd}$$

As a remark, a *pointed* CW complex additionally requires A_0 to be pointed (and hence X_0 and all following pushouts).

3.5.2 Cellular Cohomology

Cohomology theory concerns about *functions from cycles,* and one of the best ways to introduce it is through its dual, *homology theory,* which is about the cycles themselves. Given an explicit description of a type, simplicial or cellular, suitable algebraic structures can be defined for cycles. We will focus on the case where a CW complex X is given, along with its cellular description.

To begin with, a one-dimensional cycle in homology theory (and cohomology theory) is a *linear combination of (oriented) lines* forming a collection of (possibly overlapping) cycles in graph theory. Each coefficient in a homology-theoretic cycle (as a linear combination) tracks the number of occurrences of a line in these graph-theoretic cycles, where traversals in the opposite direction are counted negatively. Because a linear combination is determined by its coefficients, the order of lines is irrelevant; only the orientation of lines is of concern, and two traversals of opposite directions cancel each other. One can then define the addition, subtraction and negation on these homology-theoretic cycles, where a negated homology-theoretic cycle form the same graph-theoretic cycles in the opposite direction. In the following, homology-theoretic cycles are simply called *cycles.*

Let $\tilde{\partial}_1$ be the function mapping a line from a to b to the linear combination $a - b$, which represents the (oriented) boundary of the line. Whether a collection of lines forms a collection of graph-theoretic cycles reduces to whether the summation of $\tilde{\partial}_1$ of these lines are exactly zero, as every point in a graph-theoretic cycle has equal numbers of "ins" and "outs", contributing zero to the linear combination. If we extend the function $\tilde{\partial}_1$ on lines to linear combinations of lines as ∂_1, then *a linear combination of lines is a cycle* *if and only if it is in the kernel of ∂_1.* In our setting, the linear combinations of lines and points are $\mathbb{Z}[A_1]$ and $\mathbb{Z}[A_0]$ as below.

$$\begin{tikzcd} \mathbb{Z}[A_1] \arrow[r, "\partial_1"] & \mathbb{Z}[A_0] \end{tikzcd}$$

We would like to identify cycles up to cells at higher dimensions; in particular, if there is a two-dimensional cell filling the difference between two cycles (which is itself a cycle), then those two cycles should be regarded as the same. Intuitively, a cell fills a cycle if its boundary matches the cycle. Similar to ∂_1, one can define the boundary function ∂_2 for any two-dimensional cell $a \in A_2$ by summing up lines traversed by $\alpha_2\langle a; -\rangle$. A cycle can be filled if and only if it is in the image of ∂_2. Consider the following diagram:

$$\begin{tikzcd} \mathbb{Z}[A_2] \arrow[r, "\partial_2"] & \mathbb{Z}[A_1] \arrow[r, "\partial_1"] & \mathbb{Z}[A_0] \end{tikzcd}$$

The subject of our interest, *cycles up to identifications,* is exactly the quotient of cycles (the kernel of ∂_1) by boundaries of cells at the next dimension (the image of ∂_2). This quotient forms the *first cellular homology group* of type X, and groups for higher dimensions can be defined in a similar way. How exactly the boundary functions ∂_n at higher dimensions should be defined from A_n and α_n will be discussed later.

Cellular cohomology groups take the dual of the sequence of linear combinations above before calculating the quotients of kernels by images;[8] it applies the contravariant functor $\hom(-, G)$ for some given abelian group G to the entire sequence. The resulting diagram is

$$\begin{tikzcd} \hom\big(\mathbb{Z}[A_2], G\big) & \hom\big(\mathbb{Z}[A_1], G\big) \arrow[l, "\hom(\partial_2, G)"'] & \hom\big(\mathbb{Z}[A_0], G\big) \arrow[l, "\hom(\partial_1, G)"'] \end{tikzcd}$$

and the *first* co*homology group* is the quotient of the kernel of $\hom(\partial_2, G)$ by the image of $\hom(\partial_1, G)$. Groups at higher dimensions are defined in a similar way, and we write $H^n(X; G)$ as the nth cellular cohomology group with parameter G.

Remark 3.5.1. To be precise, I use *reduced* cellular cohomology groups; the difference between reduced and unreduced ones will be discussed at the end of this section.

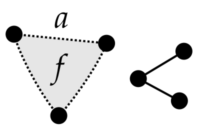
Boundary functions. One reasonable definition of ∂_{n+1} on a cell f at dimension $n+1$ is to individually calculate the coefficient $\texttt{coeff}(f, a) : \mathbb{Z}$ of each cell a within the boundary of the cell f; that is, the boundary function is of the following form (where the $\sum$ below is the summation in linear algebra, not sum types):

$$\partial_{n+1}(f) :\equiv \sum_{a:A_n} \texttt{coeff}(f, a)\, a.$$

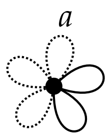
[8]The sequence is called *chain complex* and the dualized sequence is called *cochain complex.*

In order to make possible summation over a possibly infinite A_n, it seems we have to assume the final `coeff`$(f,-)$ has finite support; this corresponds to the *closure-finiteness* condition in the classical theory, which is part of the definition of CW complexes and in fact what the "C" in the "CW" stands for. The classical condition says the boundary of each cell should be covered by a finite union of cells at lower dimensions, and so our assumption is well-motivated and may be necessary. However, we believe it is possible to avoid this assumption due to the way we define CW complexes in UniTT+HIT and the nature of synthetic homotopy theory.

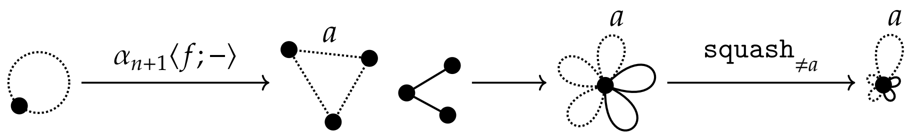
Intuitively, the `coeff`(f,a) should capture the (signed) occurrences of a in the boundary of f. Considering the CW complex



with a two-cell f of boundary consisting of line a and the other two, the value `coeff`(f,a) should be 1 under suitable orientation. The trick is to identify all points and obtain



where the boundary of f is then composed of loops at the center; the winding number of the loop a is then the coefficient we are looking for. More precisely, the coefficient is the winding number of the function



from $\mathbb{S}^1$ to $\mathbb{S}^1$, where `squash`$_{\neq a}$ kills every loop except the one indexed by a; the squashing function is definable when the index type has a decidable equality. In terms of the syntax of UniTT+HIT, the above drawing is visualizing the following chain:

$$\mathbb{S}^1 \xrightarrow{\alpha_2\langle f;-\rangle} X_1 \xrightarrow{\texttt{cfcod}} X_1/X_0 \simeq \bigvee_{\_:A_1} \mathbb{S}^1 \xrightarrow{\texttt{squash}_{\neq a}} \mathbb{S}^1$$

where X_1/X_0 is the cofiber of the inclusion from X_0 to X_1.

To generalize this idea to arbitrary dimension $n \geq 1$, we follow the same steps to obtain a function from $\mathbb{S}^n$ to $\mathbb{S}^n$,

$$\mathbb{S}^n \xrightarrow{\alpha_{n+1}\langle f;-\rangle} X_n \xrightarrow{\texttt{cfcod}} X_n/X_{n-1} \simeq \bigvee_{\_:A_n} \mathbb{S}^n \xrightarrow{\texttt{squash}_{\neq a}} \mathbb{S}^n,$$

and then inspect its generalized winding number, which is called *degree.* Again, the type X_n/X_{n-1} is the cofiber of the inclusion from X_{n-1} to X_n. Intuitively, the identity function will have degree 1 and the reversing function will have degree -1; functions that cover the sphere twice will have degree 2 (or -2 if in the opposite direction). Formally, the coefficient is defined as

$$\texttt{coeff}(f,a) :\equiv \texttt{deg}\Big(\texttt{squash}_{\neq b} \circ e \circ \texttt{cfcod} \circ \lambda x.\alpha_{n+1}\langle a;x\rangle\Big)$$

where e is the equivalence between X_n/X_{n-1} and $\bigvee_{\_:A_n} \mathbb{S}^n$; see the proof of lemma 3.5.6 for more details about the equivalence e. What remains is a definition of degrees in UniTT+HIT.

One classical definition of degrees relies on homology theories, which are not available in UniTT+HIT yet. Another approach, which is our current choice, is to apply the homotopy group functor to the function in question. Let the functor $\pi_n(X)$ be the nth homotopy group of a pointed type X. We can compute the degree of a pointed arrow $f : \mathbb{S}^n \cdot\!\!\to \mathbb{S}^n$ by inspecting where the following group homomorphism is sending the generator $1 : \mathbb{Z}$:

$$\mathbb{Z} \simeq \pi_n(\mathbb{S}^n) \xrightarrow{\pi_n(f)} \pi_n(\mathbb{S}^n) \simeq \mathbb{Z}$$

The caveat is that the function constructed in the previous paragraph might not preserve the distinguished element `north`, that is, might not be a pointed arrow. Fortunately, the homotopy group functor works on sets, and the canonical map from the set of pointed endo-arrows at the sphere to the set of endo-arrows,

$$\texttt{elim-nd}_{\|-\|}\Big[\big\|\mathbb{S}^n \to \mathbb{S}^n\big\|_0\Big]\big(f.|\texttt{fst}(f)|_0\big) : \big\|\mathbb{S}^n \cdot\!\!\to \mathbb{S}^n\big\|_0 \to \big\|\mathbb{S}^n \to \mathbb{S}^n\big\|_0,$$

is an equivalence, which means pointedness is free in this situation. This finishes our definition of boundary functions in UniTT+HIT.

Reduced cellular cohomology theory The cellular cohomology theory that is relevant here is the *reduced* cellular cohomology theory for pointed types; a characteristic difference is that a reduced theory will assign the trivial group as the zeroth cohomology group of the unit type—the most trivial pointed type—rather than the group $\mathbb{Z}$; it is more stylish to have trivial groups for trivial types.

The unit type in its natural CW complex representation is one point, which means $\mathbb{Z}[A_n]$ is isomorphic to $\mathbb{Z}$ for $n = 0$ but trivial in all other dimensions. An *unreduced* theory starts with the sequence

$$\cdots \longrightarrow \mathbb{Z}[A_1] \xrightarrow{\partial_1} \mathbb{Z}[A_0] \longrightarrow \mathbb{1}$$

with the dual

$$\cdots \longleftarrow \hom\big(\mathbb{Z}[A_1], G\big) \xleftarrow{\hom(\partial_1, G)} \hom\big(\mathbb{Z}[A_0], G\big) \longleftarrow \hom(\mathbb{1}, G)$$

which gives $\mathbb{Z}$ as the zeroth homology (and cohomology) group of $\mathbb{1}$. On the other hand, the *reduced* homology theory augmented the sequence with ϵ to $\mathbb{Z}$ as

$$\cdots \longrightarrow \mathbb{Z}[A_1] \xrightarrow{\partial_1} \mathbb{Z}[A_0] \xrightarrow{\epsilon} \mathbb{Z}$$

where ϵ sums up integer coefficients in $\mathbb{Z}[A_0]$; its dual,

$$\cdots \longleftarrow \hom\big(\mathbb{Z}[A_1], G\big) \xleftarrow{\hom(\partial_1, G)} \hom\big(\mathbb{Z}[A_0], G\big) \xleftarrow{\hom(\epsilon, G)} \hom(\mathbb{Z}, G),$$

gives $\mathbb{1}$ as the zeroth cohomology group of $\mathbb{1}$. The ending $\mathbb{Z}$ effectively kills one degree of freedom in $\mathbb{Z}[A_0]$ after the kernel-image quotienting.[9] In general, the reduced and unreduced ones only differ by a $\mathbb{Z}$ at the zeroth dimension.

3.5.3 Eilenberg–Steenrod Cohomology

Unlike the above explicit construction, there is an *axiomatic* framework for cohomology. People including Guillaume Brunerie, Eric Finster, Peter LeFanu Lumsdaine, Dan Licata, Michael Shulman and others have brought into UniTT+HIT the standard abstract framework for cohomology theories—Eilenberg–Steenrod axioms [35, 50, 126]. An (ordinary)[10] reduced cohomology theory in UniTT+HIT may be defined as a contravariant functor h from pointed types to a sequence of abelian groups satisfying the following axioms. We write $h^n(X)$ to denote the nth group in the sequence for a pointed type X.

Before presenting these axioms, however, we need to define what it means to *satisfy set-level axiom of choice*, a condition stating that $\prod$ quantifiers and 0-truncation commute. This will be used in one of the cohomology axioms shown later.

Definition 3.5.2 (set-level axiom of choice)**.** A type A *satisfies the set-level axiom of choice* if, for any family of types B indexed by A, the *unchoosing function*

$$\lambda f.\lambda(i{:}I).\mathtt{elim}_{\|-\|}\Big[\_.\big\|W(i)\big\|_0;\ \_.\big\|W(i)\big\|_0\text{-}\mathtt{level}\Big]\Big(f'.|f'(i)|_0; f\Big) : \Big\|\prod_{i:I} W(i)\Big\|_0 \to \prod_{i:I}\big\|W(i)\big\|_0$$

[9]The kernel-image quotienting is called *homology* in the classical literature, so the homology of a chain complex is homology, and the homology of a cochain complex is cohomology. I refrain from reusing this word in this thesis to avoid further confusing people who are not familiar with algebraic topology.

[10]A cohomology theory is *ordinary* if it satisfies the last Eilenberg–Steenrod axiom. See below.

is an equivalence.

See [138, ex. 7.8] for more discussion about the axiom of choice and [126] for its role in cohomology theory in UniTT+HIT. Essentially, one could present the Eilenberg–Steenrod axioms *without* the axiom of choice, but it would be difficult for pointed arrows whose codomains are Eilenberg–Mac Lane spaces, an important example of cohomology theories, to satisfy these axioms *within* UniTT+HIT. In any case, here are the axioms we use in UniTT+HIT:

Suspension. There is an isomorphism between $h^{n+1}(\mathtt{susp}(X))$ and $h^n(X)$, and the choice of isomorphisms is natural in X.

Exactness. For any pointed arrow $f : X \cdot\!\!\to Y$, the following sequence is exact, which means the kernel of $h^n(f)$ is exactly the image of $h^n(\mathtt{cfcod})$.

$$h^n(\mathtt{cofiber}(f)) \xrightarrow{h^n(\mathtt{cfcod})} h^n(Y) \xrightarrow{h^n(f)} h^n(X).$$

Wedge. Let I be a type satisfying the set-level axiom of choice. For any family of pointed types X indexed by I, the group morphism

$$\iota^* : h^n\left(\bigvee_{i:I} X(i)\right) \to \prod_{i:I} h^n(X_i)$$

induced by inclusions $\mathtt{bwin}(i,-) : X(i) \to \bigvee_{i:I} X(i)$ is a group isomorphism.

Dimension. For any integer $n \neq 0$, the group $h^n(\mathbb{2})$ is trivial.

The word *ordinary* refers to satisfying the dimension axiom. Interesting examples violating this axiom (but satisfying the rest), such as K-theories (originating from Alexander Grothendieck's work; see [12]) and complex cobordism [11], were discovered after the introduction of the framework, and are called *extra*ordinary cohomology theories. Our result only handles ordinary ones.

3.5.4 Equivalence Conjecture and Partial Results

The current goal is to show the following statement:

Conjecture 3.5.3. *For any ordinary reduced cohomology theory h, any pointed finite-dimensional CW complex X and any $n : \mathbb{Z}$, $h^n(X)$ is isomorphic to $H^n(X; h^0(\mathbb{2}))$.*

This basically states that, on CW complexes, two notions of cohomology coincide. The significance is that it connects an explicit construction with a rather abstract framework over a wide range of types. To prove this conjecture, we may need additional assumptions to recover some power of the classical reasoning, for example that sets of cells satisfy the set-level axiom of choice or that there is a decidable equality among cells.

Our approach is to break this conjecture into two parts. Note that for $n \geq m$ we write X_n/X_m as the cofiber of the inclusion from X_m to X_n.

Definition 3.5.4 (separable). A pointed type X is *separable* if $\mathtt{pt}(X) = x$ is decidable for any $x : \mathtt{carrier}(X)$. That is, whether an element is the distinguished element or not is decidable.

Lemma 3.5.5 (reformulation of ordinary cohomology groups). *For any ordinary reduced cohomology theory h and any pointed finite-dimensional CW complex X such that*

1. *all cell index sets A_n satisfy the set-level axiom of choice; and*
2. *the cell index set A_0 at the zeroth dimension is separable,*

there is a choice of coboundary functions δ_n forming the sequence

$$\cdots \longleftarrow h^n(X_n/X_{n-1}) \xleftarrow{\delta_n} h^{n-1}(X_{n-1}/X_{n-2}) \longleftarrow \cdots \longleftarrow h^1(X_1/X_0) \xleftarrow{\delta_1} h^0(\mathbb{2}) \times h^0(X_0) \xleftarrow{\delta_0} h^0(\mathbb{2})$$

such that $h^n(X)$ is isomorphic to the quotient of the kernel of δ_{n+1} by the image of δ_n for any $n \geq 0$.

The above theorem states that any ordinary cohomology groups are also the kernel-image quotients of some sequence, similar to cellular cohomology groups on page 79. It is then sufficient to show that the sequences they start with are equivalent. The rightmost groups, $h^0(\mathbb{2})$ and $\hom(\mathbb{Z}, h^0(\mathbb{2}))$, are isomorphic due to the universal property of $\mathbb{Z}$ as a free abelian group. The groups in the sequences are isomorphic because of this result:

Lemma 3.5.6 (groups in sequences agree). *For any ordinary reduced cohomology theory h and any pointed finite-dimensional CW complex X satisfying the preconditions of lemma 3.5.5, there exist an isomorphism*

$$k_n : h^n(X_n/X_{n-1}) \simeq \hom\big(\mathbb{Z}[A_n], h^0(\mathbb{2})\big)$$

for any $n \geq 1$ and an isomorphism for the zeroth dimension

$$k_0 : h^0(\mathbb{2}) \times h^0(X_0) \simeq \hom\big(\mathbb{Z}[A_0], h^0(\mathbb{2})\big).$$

Proof. The cases for $n \geq 1$ are based on the type equivalence

$$X_n/X_{n-1} \simeq \bigvee_{\_:A_n} \mathbb{S}^n.$$

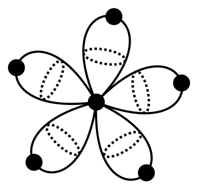
The desired result is derivable from this type equivalence, because the cohomology group of the right hand side is the direct product $\prod_{\_:A_n} h^0(\mathbb{2})$ by the Eilenberg–Steenrod axioms, and then the direct product is isomorphic to the group $\hom\big(\mathbb{Z}[A_n], h^0(\mathbb{2})\big)$ by the universal property of the free abelian group $\mathbb{Z}[A_n]$. That is,

$$h^n(X_n/X_{n-1}) \simeq h^n\Big(\bigvee_{\_:A_n} \mathbb{S}^n\Big) \simeq \prod_{\_:A_n} h^0(\mathbb{2}) \simeq \hom\big(\mathbb{Z}[A_n], h^0(\mathbb{2})\big).$$

It is not difficult to obtain the type equivalence $X_n/X_{n-1} \simeq \bigvee_{\_:A_n} \mathbb{S}^n$ but it is worthwhile to look into the insights behind it. The key observation is that the concatenation of two pushout squares is still a pushout square; in particular, one can read this diagram as a whole or as two squares:

$$\begin{tikzcd}
A_n \times \mathbb{S}^{n-1} \arrow[r, "\texttt{fst}"] \arrow[d, "\alpha_n"'] & A_n \arrow[d] \\
X_{n-1} \arrow[r] \arrow[d] & X_n \arrow[d, "\texttt{cfcod}"] \\
\mathbb{1} \arrow[r] & X_n/X_{n-1}
\end{tikzcd}$$

which shows that $X_n/X_{n-1} \simeq$ `cofiber`(`fst`:$A_n{\times}\mathbb{S}^{n-1}{\to}A_n$) where the cofiber is obtained by reading the diagram as a whole. This cofiber may be visualized as the following drawing, where the satellites are A_n, the middle dotted parts are $A_n{\times}\mathbb{S}^{n-1}$ and the center is `cfbase`; the gluing `cfglue` then fills in each petal.



The gluing `cfglue` turns each petal into a suspension of $\mathbb{S}^{n-1}$, that is, $\mathbb{S}^n$; this shows that the cofiber is equivalent to the wedge of $\mathbb{S}^n$.

The zeroth dimension needs special attention because, although a pointed set is intuitively a wedge of $\mathbb{S}^0 \equiv \mathbb{2}$, one of the points—here the distinguished element—is used as the center in the above diagram. Therefore, we have to add one copy of $h^0(\mathbb{2})$ to the left hand side to make ends meet. We also need a separator for the pointed type A_n to pick out the center. The rest of the isomorphism k_0 is the same as in other dimensions. □

The remaining issue is whether the group morphisms within two sequences are equivalent; this remains unproven at the time of writing:

Conjecture 3.5.7. *Suppose h is an ordinary reduced cohomology theory, X is a pointed finite-dimensional CW complex X satisfying the preconditions of lemmas 3.5.5 and 3.5.6, and cellular boundary functioins ∂_n are definable on X. Let δ_n be the morphisms given by lemma 3.5.5 and k_n be the isomorphisms given by lemma 3.5.6. For any $n \geq 1$, we have a commuting square*

$$\begin{tikzcd}
h^n(X_{n+1}/X_n) \arrow[d, equal, "k_{n+1}"'] & & h^{n-1}(X_n/X_{n-1}) \arrow[ll, "\delta_{n+1}"'] \arrow[d, equal, "k_n"] \\
\hom\Big(\mathbb{Z}[A_{n+1}], h^0\mathbb{2}\Big) & & \hom\Big(\mathbb{Z}[A_n], h^0\mathbb{2}\Big) \arrow[ll, "\hom(\partial_{n+1}, h^0(\mathbb{2}))"]
\end{tikzcd}$$

and similarly for the case $n = 0$ with suitable groups and group isomorphisms.

The remainder of this section is dedicated to the proof of lemma 3.5.5. From now on, let's fix an ordinary Eilenberg–Steenrod cohomology theory h and a CW complex X satisfying the conditions listed in lemma 3.5.5. The central idea is to construct as many cofibers as possible from its cellular description, and then apply the exactness axiom on these cofibers to obtain long exact sequences. From the obtained long exact sequences we can then calculate the groups of our interest.

Before constructing those cofibers, it is essential to observe that there is a lemma complimentary to lemma 3.5.6 and that there is a long exact sequence for every cofiber:

Lemma 3.5.8. *For any $m \neq n : \mathbb{Z}$ such that $n \geq 1$, $h^m(X_n/X_{n-1})$ is trivial. Moreover, for any $m : \mathbb{Z}$ such that $m \neq 0$, $h^m(X_0)$ is also trivial.*

Proof. The proof of lemma 3.5.6 actually works for mismatched dimensions without modification. For the cases $n \geq 1$, it gives an isomorphism

$$h^m(X_n/X_{n-1}) \simeq \hom\Big(\mathbb{Z}[A_n], h^{m-n}(\mathbb{2})\Big).$$

However, this time the mismatch between m and n enables the dimension axiom, which implies the right hand side is trivial. The case for the zeroth dimension is similar. □

Lemma 3.5.9. *For any pointed arrow $f : X \cdot\!\!\rightarrow Y$, there exist a natural choice of γ_n such that* *the following is a long exact sequence:*

$$\begin{tikzcd}
 & & \cdots \arrow[dll, "\gamma_n"'] \\
h^n(\mathtt{cofiber}(f)) \arrow[r, "h^n(\mathtt{cfcod})"] & h^n(Y) \arrow[r, "h^n(f)"] & h^n(X) \arrow[dll, "\gamma_{n+1}"'] \\
h^{n+1}(\mathtt{cofiber}(f)) \arrow[r, "h^{n+1}(\mathtt{cfcod})"] & h^{n+1}(Y) \arrow[r, "h^{n+1}(f)"] & h^{n+1}(X) \arrow[dll, "\gamma_{n+2}"'] \\
\cdots & &
\end{tikzcd}$$

Proof sketch. The key observation is that a cofiber of a cofiber, or more precisely,

$$\mathtt{cofiber}(\mathtt{cfcod}{:}Y \to \mathtt{cofiber}(f))$$

is equivalent to $\mathtt{susp}(X)$. The correspondence continues forever as

$$\begin{tikzcd}
X \arrow[r, dashed] \arrow[d] & Y \arrow[r] \arrow[d, dashed] & \mathbb{1} \arrow[d] & & \\
\mathbb{1} \arrow[r] & \mathtt{cofiber}(f) \arrow[r, dashed] \arrow[d] & \mathtt{susp}(X) \arrow[r] \arrow[d, dashed] & \mathbb{1} \arrow[d] & \\
 & \mathbb{1} \arrow[r] & \mathtt{susp}(Y) \arrow[r, dashed] \arrow[d] & \mathtt{susp}(\mathtt{cofiber}(f)) & \\
 & & \mathbb{1} & \ddots & \ddots
\end{tikzcd}$$

where every square is a pushout square. The long exact sequence is obtained by applying h to the snake sequence in the middle (marked as thick red dashed arrows) and then invoking the suspension axiom to remove those suspensions. The suspension axiom lowers the dimension by one while unwraping one layer of suspension, leading to the dimension stepping (as h^n is contravariant) in the statement. □

Remark 3.5.10. Lemma 3.5.9 plus the naturality of γ_n is actually equivalent to the suspension and the exactness axioms. As a result many presentations of Eilenberg–Steenrod axioms assert long exact sequences instead.

The way to construct numerous cofiber squares is to consider the following grid diagram where every grid is a pushout square.

$$\begin{tikzcd}
X_0 \arrow[r] \arrow[d] & X_1 \arrow[r] \arrow[d] & \cdots & \arrow[r] & X_n \arrow[r] \arrow[d] & X_{n+1} \arrow[r] \arrow[d] & \cdots \\
\mathbb{1} \arrow[r] & X_1/X_0 \arrow[r] & \cdots & \arrow[r] & X_n/X_0 \arrow[r] & X_{n+1}/X_0 \arrow[r] & \cdots \\
 & \ddots & \ddots & & \vdots & \vdots & \vdots \\
 & & & \mathbb{1} \arrow[r] & X_n/X_{n-1} \arrow[r] \arrow[d] & X_{n+1}/X_{n-1} \arrow[r] \arrow[d] & \cdots \\
 & & & & \mathbb{1} \arrow[r] & X_{n+1}/X_n \arrow[r] \arrow[d] & \cdots \\
 & & & & & \mathbb{1} \arrow[r] & \cdots
\end{tikzcd}$$

Any square (consisting of one or more grids) having the unit type $\mathbb{1}$ at the bottom left is a cofiber square and generates a long exact sequence by lemma 3.5.9. Various conclusions can be drawn by choosing different cofiber squares:

- Zoom in on a grid on the diagonal:

$$\begin{tikzcd}
X_n/X_{n-1} \arrow[r] \arrow[d] & X_{n+1}/X_{n-1} \arrow[d] \\
\mathbb{1} \arrow[r] & X_{n+1}/X_n.
\end{tikzcd}$$

  Through lemma 3.5.9, this grid generates the following exact sequence:

$$\begin{tikzcd}
\mathbb{0} \arrow[d, equal] & \mathtt{ker}(\delta_{n+1}) \arrow[d, equal] & \\
h^n(X_{n+1}/X_n) \arrow[r] & h^n(X_{n+1}/X_{n-1}) \arrow[r, tail] & h^n(X_n/X_{n-1}) \arrow[dll, bend left, "\delta_{n+1}"'] \\
h^{n+1}(X_{n+1}/X_n) \arrow[r, two heads] & h^{n+1}(X_{n+1}/X_{n-1}) \arrow[r] \arrow[d, equal] & h^{n+1}(X_n/X_{n-1}). \arrow[d, equal] \\
 & \mathtt{coker}(\delta_{n+1}) & \mathbb{0}
\end{tikzcd}$$

  We choose the coboundary function δ_{n+1} to be the middle function as required. Because $h^n(X_{n+1}/X_n)$ is trivial, from the exactness we know $h^n(X_{n+1}/X_{n-1})$ is isomorphic to the kernel of δ_{n+1} and the group homomorphism from it is injective. Dually, we know $h^{n+1}(X_{n+1}/X_{n-1})$ is isomorphic to the cokernel of δ_{n+1} and the group homomorphism to it is surjective.

- Let's turn our focus to this square:

$$\begin{tikzcd} X_m \arrow[r] \arrow[d] & X_{m+1} \arrow[d] \\ \mathbb{1} \arrow[r] & X_{m+1}/X_m \end{tikzcd}$$

  which by lemma 3.5.9 gives this exact sequence:

$$h^n(X_{m+1}/X_m) \to h^n(X_{m+1}) \to h^n(X_m) \to h^{n+1}(X_{m+1}/X_m).$$

  When $n \neq m$ or $m{+}1$, both $h^n(X_{m+1}/X_m)$ and $h^{n+1}(X_{m+1}/X_m)$ are trivial by lemma 3.5.8; therefore, by the exactness of the above sequence, $h^n(X_{m+1}) \simeq h^n(X_m)$. In other words, cells at dimensions much higher or much lower than n are irrelevant to the cohomology group at dimension n. This implies that there are at most three different values of $h^n(X_m)$ up to isomorphism:

  1. $h^n(X_{n-1}) \simeq h^n(X_{n-2}) \simeq \cdots \simeq h^n(X_0) \simeq \mathbb{0}$. The intuition is that X_m for any $m < n$ does not have any interesting information at dimension n.
  2. $h^n(X_n)$. X_n has the cells at dimension n, but lacks the cells at dimension $(n+1)$ which may identify some cycles at dimension n.
  3. $h^n(X_{n+1}) \simeq h^n(X_{n+2}) \simeq \cdots \simeq h^n(X)$. Cells at dimension $(n+2)$ or above play no role in the nth cohomology group.

  It is thus sufficient to study $h^n(X_{n+1})$ for the nth cohomology group of X.

- The next step is to investigate the square

$$\begin{tikzcd} X_{n-2} \arrow[r] \arrow[d] & X_{n+1} \arrow[d] \\ \mathbb{1} \arrow[r] & X_{n+1}/X_{n-2} \end{tikzcd}$$

  which generates the exact sequence

$$h^{n-1}(X_{n-2}) \to h^n(X_{n+1}/X_{n-2}) \to h^n(X_{n+1}) \to h^n(X_{n-2}).$$

  From the previous cofiber square we know both $h^{n-1}(X_{n-2})$ and $h^n(X_{n-2})$ are trivial, and again by the exactness $h^n(X_{n+1}/X_{n-2}) \simeq h^n(X_{n+1})$. Therefore, we have

$$h^n(X_{n+1}/X_{n-2}) \simeq h^n(X_{n+1}) \simeq h^n(X_{n+2}) \simeq \ldots \simeq h^n(X).$$

  This means it is sufficient to calculate $h^n(X_{n+1}/X_{n-2})$.

Combining these observations, we have the following commuting square for $n \geq 2$:

$$\begin{tikzcd}
\mathtt{coker}(\delta_n) \simeq h^n(X_n/X_{n-2}) & h^n(X_{n+1}/X_{n-2}) \simeq h^n(X) \arrow[l] \\
\mathrm{hom}\big(\mathbb{Z}[A], G\big) \simeq h^n(X_n/X_{n-1}) \arrow[u, two heads] & h^n(X_{n+1}/X_{n-1}) \simeq \mathtt{ker}(\delta_{n+1}). \arrow[l, tail] \arrow[u]
\end{tikzcd}$$

We can further infer that the top homomorphism is injective and the right one is surjective by applying lemma 3.5.9 to the following two cofiber squares, respectively,

$$\begin{tikzcd}
X_n/X_{n-2} \arrow[r] \arrow[d] & X_{n+1}/X_{n-2} \arrow[d] & X_{n-1}/X_{n-2} \arrow[r] \arrow[d] & X_{n+1}/X_{n-2} \arrow[d] \\
\mathbb{1} \arrow[r] & X_{n+1}/X_n & \mathbb{1} \arrow[r] & X_{n+1}/X_{n-1}
\end{tikzcd}$$

and finally obtain the desired isomorphism

$$h^n(X) \simeq \mathtt{ker}(\delta_{n+1})/\mathtt{im}(\delta_n)$$

by the following lemma from group theory:

Lemma 3.5.11. *Let $Q \subseteq P$ be two subgroups of G where Q is normal. If we have a group K and a commuting diagram as follows, where the group homomorphism from P to G is the canonical inclusion and the one from G to G/Q is the quotienting, then $K \simeq P/Q$.*

$$\begin{tikzcd}
G/Q & K \arrow[l, tail] \\
G \arrow[u, two heads] & P \arrow[l, tail] \arrow[u, two heads]
\end{tikzcd}$$

By choosing P to be the kernel and Q the image, we have the desired formula that K is the quotient. (Q is normal because it is a subgroup of an abelian group.) This shows that for any $n \geq 2$, $h^n(X)$ is isomorphic to the kernel-image quotient of adjacent coboundary functions δ_n. The cases for $n = 0$ or 1 can also be derived from the grid diagram similarly but with special care of the separable pointed set $X_0 :\equiv A_0$ as demonstrated in the proof of lemma 3.5.6. This concludes the proof of lemma 3.5.5.

3.6 Use of Univalence and Higher Inductive Types

It is worthwhile to review how the new features of UniTT+HIT are used in the homotopy-theoretic results we have seen in this chapter.

Univalence axiom. It is challenging to trace the univalence axiom due to its prevalent uses, but here are two essential ones:

- Functional extensionality.

  Currently the functional extensionality is proved by the univalence axiom and it is essential for any non-trivial reasoning about functions.

- Non-trivial identifications between types or non-trivial families of types indexed by higher inductive types.

  When defining such a family (as a function from some higher inductive type to the universe), often the elimination rule of the higher inductive type demands a non-trivial type identification for its identification generator, which can only be given by the univalence axiom.

For example, to establish a non-trivial covering space F indexed by the circle $\mathbb{S}^1$, we need to provide a non-trivial loop at $F(\texttt{base}) : \mathcal{U}$, and the univalence axiom enables us to use any automorphism. Next, when comparing two covering spaces, we may use functional extensionality to compare their fibers instead. Finally, to identify two fibers, the univalence axiom makes it sufficient to show an equivalence between them.

The above example shows that the development of covering spaces in section 3.2 heavily relies on the univalence axiom. For the Seifert–van Kampen in section 3.3, the `code` is a non-trivial family indexed by the pushout, and thus requires the univalence axiom; so does the `code` for Blakers–Massey in section 3.4. Moreover, all of these sections, including the cohomology groups in section 3.5, use functional extensionality.

Currently, the univalence axiom is also used as a shortcut to show $P(B)$ from $P(A)$ for two equivalent types A and B. It seems most of them can be rewritten without the univalence axiom and thus are probably not essential.

Higher inductive types. The essential higher inductive types in use are pushouts, set-quotients and truncations, and all of them can be reduced to pushouts [46, 81, 119]. Other higher inductive types, such as the circle, suspensions, spheres, cofibers and wedges, are already defined in terms of pushouts. Therefore, even though the precise scope of higher inductive types is still up in the air, my thesis only uses pushouts and pushouts are regarded as a core higher inductive type.

Truncations are fundamental to all developments in this chapter; for example, connectivity and surjectivity are defined using truncation, and every previous section in this chapter uses at least one of them. Set-quotients are already part of the definitions of reconstructed covering spaces in section 3.2, the `code` for the Seifert–van Kampen theorem and the cohomology groups. In addition to various higher inductive types used in the proofs, the Seifert–van Kampen and Blakers–Massey theorems directly concern pushouts; without higher inductive types, it also seems difficult to find an interesting type[11] with non-trivial higher cohomology groups.

In conclusion, higher inductive types not only play a role in various parts of the proofs in my thesis, but also appear directly in the theorem statements. Without them (for example in other developments such as [7, 17, 44, 123]) there is no obvious way to present interesting topological spaces (up to homotopy) *directly as types.* The univalence axiom, in addition to its direct use in proving type identification, provides the handy functional extensionality and makes higher inductive types more usable.

3.7 Notes on Homotopy Theory

Our proof of the Blakers–Massey theorem in section 3.4 is mathematically new and has led to new research in mathematics: my Agda mechanization has been translated by Charles Rezk back into the more conventional mathematical language [117] and then, based on the translation, Mathieu Anel, Georg Biedermann, Eric Finster and André Joyal have generalized the theorem to Goodwillie calculus by abstracting over the notion of connectivity and truncation [8]. It will be interesting to mechanize the new results back into some computer system.

There are many other results in homotopy and cohomology theories awaiting us! For example, the study of covering spaces presented in this thesis is just a starting point. The representation theorem in the classical theory is actually a correspondence between two categories, not just the objects. Also, the connectivity condition on the base type may be dropped if we replace fundamental groups by fundamental groupoids. There should also be a connection between *transitive* π_1-sets and connected covering spaces. Moreover, there are other possible generalizations such as n-covering spaces over a type as families of n-types indexed by that type, which to our knowledge do not immediately correspond to well-known structures in classical homotopy theory.

There are also various generalizations of the Seifert–van Kampen theorem into higher dimensions, many by Ronald Brown *et al.*, which aid the calculation of higher-dimensional structures based on groupoids [26]. One may wonder whether they can be stated in UniTT+hit; the double groupoids mechanized by Jakob von Raumer [115] can be seen as a step in this direction.

[11]Except possibly the universe itself, where the univalence axiom acts as an identification generator.

Chapter 4

Mechanization in Agda

Because the UniTT+hit introduced in chapter 2 is based on the Martin-Löf type theory, existing tools designed for similar flavors of Martin-Löf type theories may meet our need with little modification. Moreover, new proof assistants that are aware of UniTT+hit are also being developed. Here are some well-known computer programs for mechanization in UniTT+hit:

- Coq [134] is a proof assistant based on the dependent type theory Calculus of Constructions [42]. It emphasizes on the usage of tactics scripts to complete the proofs, even though it is still possible to write out the proofs directly.
- Agda [109] is a proof assistant based on a Martin-Löf type theory. It is also a functional programming language with sophisticated pattern matching. Compared to Coq, it encourages the users to write out proofs directly.
- Lean [106] is a newer proof assistant that is also based on dependent type theory as Coq and Agda are. It "aims to bridge the gap between interactive and automated theorem proving" [106].
- Cubical [40] is an experimental proof assistant that is based on the cubical set model [23] which provides a constructive interpretation of the univalence axiom.

All my thesis work is mechanized in the proof assistant Agda and this chapter is dedicated to highlighting theoretical and practical details when mechanizing proofs in UniTT+hit in Agda. To the best of my knowledge, Agda is still lacking a thorough consistency proof of the language (or even the core language); the confidence in the formal proofs written in Agda seem to be rooted in its perceived similarity to Martin-Löf type theories. The common belief is that, even if the proof checker turns out to be faulty, the mechanized proofs probably still stand.

This thesis will also assume this *correctness from perceived similarity;* see table 4.1 for how UniTT+hit and Agda appear similar (except the universe lifting). In this chapter, I will highlight some differences between UniTT+hit and Agda and discuss some practical considerations.

Types	UniTT+HIT	AGDA implementation in my thesis
Sums	$\sum_{a:A} B$	`Σ A (λ a → B)`
	$\langle a; b\rangle$	`(a , b)`

	$\mathtt{elim}_{\Sigma}[z.C](x.y.c; s)$	`f s where`
		`  f : (z : Σ A (λ a → B)) → C`
		`  f (x , y) = c`

	`fst`	`fst`
	`snd`	`snd`
	$s \equiv \langle \mathtt{fst}(s); \mathtt{snd}(s)\rangle$	valid
Functions	$\prod_{a:A} B$	`(a : A) → B`
	$\lambda(x{:}A).b$	`λ (x : A) → b`
	$f(a)$	`f a` or `f $ a`
	$f \equiv \lambda x.f(x)$	valid
	functional extensionality	`λ=`
The unit	$\mathbb{1}$	`⊤`
	`unit`	`unit`

	$\mathtt{elim}_{\mathbb{1}}[x.C](c; u)$	`f u where`
		`  f : (x : ⊤) → C`
		`  f unit = c`

	$u \equiv \mathtt{unit}$	valid
Identification	$a =_A b$	`a == b`
	$\mathtt{refl}_a$	`idp`

	$\mathtt{elim}_{=}[x.y.z.C](x.c; a; b; p)$	`f a b p where`
		`  f : (x y : A) (z : x == y) → C`
		`  f x .x idp = c`

	$\mathtt{ap}_f(p)$	`ap f p`

	$\mathtt{elim}_{a=}[x.y.C](c; b; p)$	`f b p where`
		`  f : (x : A) (y : a == x) → C`
		`  f .a idp = c`

	$\mathtt{elim}_{=a}[x.y.C](c; b; p)$	`f b p where`
		`  f : (x : A) (y : x == a) → C`
		`  f .a idp = c`

	$p \cdot q$	`p • q`
	p^{-1}	`! p`
	$\mathtt{transport}[x.B](p; u)$	`transport (λ x → B) p u`
	`to-transp`	`to-transp`
Universes	$\mathcal{U}_i$	`Type i`
	$\mathcal{U}_0$	`Type lzero`
	$\mathcal{U}_{i+1}$	`Type (lsucc i)`
	$\mathcal{U}_{i \vee j}$	`Type (lmax i j)`

(continued on the next page)

Table 4.1: Similarity between UniTT+HIT and AGDA.

Types	UniTT+HIT	AGDA implementation in my thesis
	$\mathcal{U}_i : \mathcal{U}_{i+1}$	valid
	$A : \mathcal{U}_i \implies A : \mathcal{U}_{i+1}$	not valid; `A : Type i` only implies `Lift A : Type (lsucc i)`
Coproducts	$A + B$	`A ⊔ B`
	$\texttt{inl}(a)$	`inl a`
	$\texttt{inr}(b)$	`inr b`
	$\texttt{elim}_{+}[x.D](x.d_{\texttt{inl}};x.d_{\texttt{inr}};c)$	`f c where`
		`  f : (x : A ⊔ B) → D`
		`  f (inl x) = d_inl`
		`  f (inr x) = d_inr`
This cell intentionally left blank	$\mathbb{0}$	`⊥`
	$\texttt{elim}_{\mathbb{0}}[x.C](e)$	`f e where`
		`  f : (x : ⊥) → C`
		`  f ()`
Natural numbers	$\mathbb{N}$	`data ℕ : Type lzero`
	$\texttt{zero}$	`O`
	$\texttt{succ}$	`S`
	$\texttt{elim}_{\mathbb{N}}[x.C](c_{\texttt{zero}};x.y.c_{\texttt{succ}};n)$	`f n where`
		`  f : (x : ℕ) → C`
		`  f O = c_O`
		`  f (S x) = c_S where y = f x`
Equivalence	$\texttt{is-equiv}(f)$	`is-equiv f`
	$A \simeq B$	`A ≃ B`
Univalence	$\texttt{coerce}(p)$	`coe p`
	$\texttt{coerce-equiv}(p)$	`coe-equiv p`
	$\texttt{idf-is-equiv}(A)$	`idf-is-equiv A`
	$\texttt{univalence}(A;B)$	`snd ua-equiv`
Dependent identification	$u =_{p}^{x.B} v$	`u == v [ (λ x → B) ↓ p ]`
	$\texttt{apd}_{f}(p)$	`apd f p`
The circle	$\mathbb{S}^1$	`S¹`
	$\texttt{base}$	`base`
	$\texttt{loop}$	`loop`
	$\texttt{elim}_{\mathbb{S}^1}[x.C](c_{\texttt{base}};c_{\texttt{loop}};s)$	`S¹-elim c_base c_loop s`
	$\texttt{loop}_{\beta}$	`loop-β`
	$\texttt{elim-nd}_{\mathbb{S}^1}[x.C](c_{\texttt{base}};c_{\texttt{loop}};s)$	`S¹-rec c_base c_loop s`
	$\texttt{loop-nd}_{\beta}$	`loop-β`
Truncation level	$\texttt{TLevel}$	`data TLevel : Type lzero`

(continued on the next page)

Table 4.1: Similarity between UniTT+HIT and AGDA.

Types	UniTT+HIT	Agda implementation in my thesis
	-2	`⟨-2⟩`
	$\mathtt{succ}$	`S`

	$\mathtt{elim}_{\mathtt{TLevel}}[x.C](c_{-2}; x.y.c_{\mathtt{succ}}; n)$	`f t where`
		`f : (x : ℕ) → C`
		`f O = c_⟨-2⟩`
		`f (S x) = c_S where y = f x`

	$m \mathbin{\hat{+}} n$	`m +2+ n`
	$\mathtt{is\text{-}contr}(A)$	`is-contr A`
	$\mathtt{has\text{-}level}_n(A)$	`has-level n A`
	$\mathtt{is\text{-}prop}(A)$	`is-prop A`
	$\mathtt{is\text{-}set}(A)$	`is-set A`
	n-type	`n -Type i`
	$\mathtt{Set}$	`hSet`
Truncation	$\lVert A \rVert_n$	`Trunc n A`
	$\lVert A \rVert_n\mathtt{\text{-}level}$	`Trunc-level`
	$\lvert a \rvert_n$	`[ a ]`

	$\mathtt{elim}_{\lVert - \rVert}[x.C; x.C\mathtt{\text{-}level}](x.c_{\lvert - \rvert}; t)$	`Trunc-elim {P = λ x → C}`
		`(λ x → C-level) (λ x → c_[]) t`

	$\mathtt{elim\text{-}nd}_{\lVert - \rVert}[C; C\mathtt{\text{-}level}](x.c_{\lvert - \rvert}; t)$	`Trunc-rec C-level (λ x → c_[]) t`
Connectivity	$\mathtt{is\text{-}connected}_n(A)$	`is-connected n A`
Fibers	$\mathtt{hfiber}_f(b)$	`hfiber f b`
	$\mathtt{has\text{-}conn\text{-}fibers}_n(f)$	`has-conn-fibers n f`
	$\mathtt{is\text{-}surj}(f)$	`is-surj f`
Set quotients	A/R	`SetQuot R`
	$A/R\mathtt{\text{-}level}$	`SetQuot-level`
	$[a]$	`q[ a ]`
	$\mathtt{quot\text{-}rel}(r)$	`quot-rel r`

	$\mathtt{elim}_{/}[x.C; x.C\mathtt{\text{-}level}](x.c_{[-]};$
	$x.y.z.c_{\mathtt{quot\text{-}rel}}; q)$
		`SetQuot-elim {P = λ x → C}`
		`(λ x → C-level) (λ x → c_q[])`
		`(λ {x} {y} z → c_quot-rel) q`

	$\mathtt{quot\text{-}rel}_\beta$	`quot-rel-β`

	$\mathtt{elim\text{-}nd}_{/}[C; C\mathtt{\text{-}level}](x.c_{[-]};$
	$x.y.z.c_{\mathtt{quot\text{-}rel}}; q)$
		`SetQuot-rec`
		`C-level (λ x → c_q[])`
		`(λ {x} {y} z → c_quot-rel) q`

	$\mathtt{quot\text{-}rel\text{-}nd}_\beta$	`quot-rel-β`
Pushouts	$A \sqcup_{C;f;g} B$	`Pushout (span A B C f g)`
	$\mathtt{left}(a)$	`left a`
	$\mathtt{right}(b)$	`right b`

(continued on the next page)

Table 4.1: Similarity between UniTT+HIT and Agda.

Types	UniTT+HIT	Agda implementation in my thesis
	$\mathtt{glue}(c)$	`glue c`
	$\mathtt{elim}_{\sqcup}[x.D](x.d_{\mathtt{left}}; x.d_{\mathtt{right}}; x.d_{\mathtt{glue}}; p)$	`Pushout-elim {P = λ x → D} (λ x → d_left) (λ x → d_right) (λ x → d_glue) p`
	$\mathtt{glue}_{\beta}$	`glue-β`
	$\mathtt{elim\text{-}nd}_{\sqcup}[D](x.d_{\mathtt{left}}; x.d_{\mathtt{right}}; x.d_{\mathtt{glue}}; p)$	`Pushout-rec (λ x → d_left) (λ x → d_right) (λ x → d_glue) p`
	$\mathtt{glue\text{-}nd}_{\beta}$	`glue-β`
Suspension	$\mathtt{susp}(A)$	`Susp A`
	$\mathtt{north}$	`north`
	$\mathtt{south}$	`south`
	$\mathtt{merid}(a)$	`merid a`
Cofibers	$\mathtt{cofiber}(f)$	`Cofiber f`
	$\mathtt{cfbase}$	`cfbase`
	$\mathtt{cfcod}(b)$	`cfcod b`
	$\mathtt{cfglue}(a)$	`cfglue a`
Pointedness	$\sum_{A:\mathcal{U}_i} A$	`Ptd i`
	$\mathtt{pt}(X)$	`pt X`
	$\mathtt{carrier}(X)$	`de⊙ X`
	$X \cdot\!\!\to Y$	`X ⊙→ Y`
Binary wedges	$X \vee Y$	`X ∨ Y`
	$\mathtt{winl}(x)$	`winl x`
	$\mathtt{winr}(y)$	`winr y`
	$\mathtt{wglue}$	`wglue`
Wedges	$\bigvee_A X$	`BigWedge X`
	$\mathtt{bwbase}$	`bwbase`
	$\mathtt{bwin}(a, x)$	`bwin a x`
	$\mathtt{bwglue}(a)$	`bwglue a`
Spheres	$\mathbb{S}^n$	`Sphere n`
Booleans	$\mathbb{2}$	`Bool`
	$\mathtt{true}$	`true`
	$\mathtt{false}$	`false`

(continued on the next page)

Table 4.1: Similarity between UniTT+HIT and Agda.

Types	UniTT+HIT	AGDA implementation in my thesis

	$\mathtt{elim}_{\mathbb{2}}[x.C](c_{\mathtt{true}};c_{\mathtt{false}};b)$	`f b where`
		`  f : (x : Bool) → C`
		`  f true = c_true`
		`  f false = c_false`

Table 4.1: Similarity between UniTT+HIT and AGDA.

Remark 4.1. Typesetting AGDA code with English text is challenging, and this is why I chose UniTT+HIT to present my work in chapter 3. The naïve arrangement would lead to

- I wrote `A B` and also `f x`. That is it.

where the space between `A` and `B` or `f` and `x` is wider than the space between sentences, which is visually disturbing to my taste. There are various ways to reconnect the fragments separated by these wide spaces:

- I wrote `A␣B` and also `f␣x`. That is it.
- I wrote `A B` and also `f x`. That is it.
- I wrote `A B` and also `f x`. That is it.

I do not feel there is a good solution, and the difference between printers and LED displays makes the situation even worse. Eventually, I chose to add the light gray background, and I apologize if this is not your favorite style.

4.1 Specifics of AGDA

4.1.1 Function Types

There is no essential difference between functions types in AGDA and those in UniTT+HIT, but AGDA provides much syntax sugar to make the programming easier; see table 4.2. AGDA also supports implicit arguments to reduce clutter, which are indicated by the curly brackets in `{a : A} → B`; this is useful for suppressing arguments that can be inferred from later arguments. For example, the following is an identity function for any type `A` in `Type lzero`:

```
id : {A : Type lzero} → A → A
id a = a
```

where AGDA can usually infer the type `A` from the succeeding argument in a typical usage of `id` such as `id a`.

Syntax	Elaboration
`A → B`	`(_ : A) → B` with an unused variable
`(a : A) (b : B) → C`	`(a : A) → (b : B) → C`
`(a b : A) → C`	`(a : A) → (b : A) → B`
`∀ (a : A) → B`	`(a : A) → B`
`∀ a → B`	`(a : _) → B` with domain unspecified
`λ (a : A) (b : B) → c`	`λ (a : A) → λ (b : B) → c`
`λ (a b : A) → c`	`λ (a : A) → λ (b : A) → c`
`λ a → b`	`λ (a : _) → b`

Table 4.2: Syntax sugar for functions in AGDA.

4.1.2 Universes

Both UniTT+HIT and AGDA have a hierarchy of countably infinite universes, where the *i*th universe $\mathcal{U}_i$ is roughly transcribed as `Set i` in plain AGDA and then aliased to `Type i` in this thesis. Nonetheless, there are significant discrepancies in how the universe hierarchy may be used in two systems.

Universe polymorphism. Universe levels in UniTT+HIT remain meta-variables in rules, separate from the term variables. AGDA, on the other hand, provides a special type for the universe levels and reuses function types for quantification over (finite) levels. The level type is named `ULevel` in my thesis. For example, the following is an identity function that works for any type in any (finite-level) universe.

```
id : ∀ {i : ULevel} (X : Set i) → X → X
id X x = x
```

This feature is called *universe polymorphism* in the AGDA documentation. One problem is that every AGDA type should belong to some universe, but the introduction of universe polymorphism in AGDA creates new types that do not belong to any finite level in the universe hierarchy. To mitigate this issue, in AGDA there is a special ω-level universe, `Setω`[1], which includes all types whose universe levels can be expressed with universally quantified bound variables. For example, the type of the above function `id`

```
∀ {i : Level} (X : Set i) → X → X
```

involves a bound variable `i` and belongs to `Setω`. Note that `Setω` does not belong to any universe and is not a type, nor can it be used directly in an AGDA proof.

[1]There is no space between `Set` and `ω` in the AGDA output messages.

Lifting Agda admits implicit lifting of types to higher-level universes only in limited cases, where UniTT+hit admits such transparent lifting everywhere. Explicit lifting in Agda is possible through defining a new type `Lift A` with a single constructor from `A` such that types `A` and `Lift A` live in different universes but are trivially equivalent. Fortunately, the need of explicit lifting during the mechanization of homotopy theory is rare.

An alternative approach to implementing the universe hierarchy is to separate universe level variables from term variables, and then apply the elaboration algorithm by Robert Harper and Robert Pollack [62]; the algorithm imposes the prenex form restriction on universe level variables as the polymorphism in ML-like programming languages, but provides implicit lifting as in UniTT+hit and was recently implemented in the proof assistant Coq [130]. At the time of writing, there are concerns in the Agda community about the prenex form being too restricted and the potential slowdown of the type checking; see [70] for a discussion about these concerns.

4.1.3 Pattern Matching

In Agda, pattern matching plays the role of elimination rules in UniTT+hit as shown in table 4.1 except for higher inductive types. It comes with flexible syntax for function definitions that matches the rich syntax of data types and record types. The correctness is guarded by two parts: the coverage checker of Agda guarantees that all cases are considered, and the termination checker analyzes the call graphs to determine whether recursive definitions are terminating [1, 84]. Together they maintain the Haskell programming style without obliviously demanding additional proof-theoretic power from the underlying type theory.

Note that the default pattern matching relies on the rule K for identification [55, 105], which is incompatible with UniTT as discussed in section 2.3. The current version of Agda implements a new flag [38] `--without-K` which removes the dependency on the rule K. The mechanization in this thesis was checked against this new flag.

4.1.4 Data Types

Agda supports rich syntax for data types following the similar principles of mutually defined inductive-recursive types in [49, 108], which is able to cover most inductive types in UniTT+hit, if not all. A simple example is that we can define natural numbers ℕ as follows; see table 4.1 for a complete translation of uses of ℕ into Agda.

```
data ℕ : Set where
  O : ℕ
  S : ℕ → ℕ
```

4.1.5 Records

Agda also has built-in syntax for *n*-ary labelled sums, called *records.* Sum types and the unit type in UniTT+hit can be easily defined as record types; also see table 4.1.

```
record Σ {i j} (A : Type i)
  (B : A → Type j) : Type (lmax i j) where

  constructor _,_
  field
    fst : A
    snd : B fst

record ⊤ : Type lzero where
  constructor unit
```

The difference between record types and data types with only one constructor is that (non-recursive) record types admit the uniqueness rule as we see in figs. 2.1 and 2.3; for this reason, the unit and sum types are not defined as inductive types in section 2.1.

4.1.6 Higher Inductive Types

Unfortunately, it is still unclear how to incorporate higher inductive types in UniTT+hit into Agda. Currently there are two ways to encode a higher inductive type in Agda:

The old way. Dan Licata proposed postulating constructors at higher dimensions on top of an ordinary inductive type defined by constructors at the zeroth dimension [86]. There is, however, a constant worry that it might be possible to contradict the postulated identification generator which leads to inconsistency; at the time of writing complicated mechanisms are needed to seal all known loopholes.

To see the issue, consider an interval as a higher inductive type $\mathbb{I}$, which has two element generators `0` and `1` and one identification generator `seg` between `0` and `1`. Dan Licata's method would first define an inductive type generated by `0` and `1` and then postulate that `0` is identified to `1` as follows:

```
data I : Type lzero where
  I0 : I
  I1 : I

postulate
  Iseg : I0 == I1

I-elim : ∀ {l} {C : I → Type l}
```

```
  (I0* : C I0) (I1* : C I1)
  (Iseg* : I0* == I1* [ C ↓ Iseg ])
  → (x : I) → C x
I-elim I0* I1* Iseg* I0 = I0*
I-elim I0* I1* Iseg* I1 = I1*
```

However, in Agda we can prove `I0` and `I1` are different and contradict `seg`:

```
¬Iseg : (I0 == I1) → ⊥
¬Iseg ()
```

This is due to Agda knowing `I0` and `I1` are different constructors. To prevent this, we have to block the pattern matching on `I` or in general any way to show `I0` and `I1` are different.[2] Over time more ways have been found to prove that `I0` and `I1` are different in Agda, and the workaround has evolved into delicate hacks that have been manually applied to each higher inductive type; see [28] for more information.

The new way: rewrite rules. Currently Agda is experimenting with a new extension, called *rewrite rules* [37], that enables adding (certain) judgmental equalities. To define a higher inductive type is to first postulate everything as axioms, and then insert the computation rules as rewrite rules.

```
postulate
  I : Type lzero
  I0 : I
  I1 : I
  Iseg : I0 == I1

module IElim {l} {C : I → Type l}
  (I0* : C I0) (I1* : C I1)
  (Iseg* : I0* == I1* [ C ↓ Iseg ]) where

  postulate
    f : (x : I) → C x
    I0-β : f I0 ↦ I0*
    I1-β : f I1 ↦ I1*
  {-# REWRITE left-β #-}
  {-# REWRITE right-β #-}

  postulate
    Iseg-β : apd f Iseg == Iseg*
```

[2]There are also other issues such as `Iseg*` is improperly ignored during type checking because it is not used; this has led to another hack to block the special treatment of unused variables in Agda.

Rewrite rules seem safer, because they add features to a safe state, unlike the workaround in the previous method which tries to subtract features from an unsafe state. They also make it possible to have judgmental computational rules for higher-dimensional structures (discussed in section 2.5). During the time of writing this thesis, Jesper Cockx and I have completed the migration of the main Agda development of homotopy theory (including the mechanization in this thesis) [29] to rewrite rules.

4.2 Practice of Mechanization

This section is dedicated to practical considerations that are not visible on paper. In my experience, there is a regular struggle between these essential factors to a successful Agda mechanization:

1. the speed of type checking, and
2. the computational content, and
3. the economy or elegance of the syntax.

It is desired to have fast type checking to reduce machine efforts, economic syntax to reduce human efforts, and much computational content (in terms of judgmental equality) to strengthen the mathematical results in addition to their typing. That is, we consider the standard addition + with the judgmental equality $2 + 2 \equiv 4$ *stronger* than the one without. However, in practice it is difficult to achieve the best in all aspects, and the following discusses the interplay between these factors in several examples.

Remark 4.2.1. Although these three criteria are my best attempt to abstract over various trade-offs made in our Agda mechanization, there is no reason to believe that this section directly applies to other mechanization systems or future versions of Agda.

Expansion control. In my observation, the most important factor to the speed of the type checking is how definitions are expanded; during the equality checking, Agda might expand the terms into unnecessarily long expressions, sometimes pushing the computers to their limits. The easiest way to manually contain the expansion in Agda is to mark a definition *abstract,* which effectively strips off any attached computational content from the definition. In other words, the key word `abstract` [2] sacrifices computational content for speed. Usually, properties such as a function being an equivalence, group laws, or that a function respects higher-dimensional generators in the domain, are all good candidates to mark abstract; the rationale is that their actual proofs are usually insignificant. As a side note, there has been an on-going research to automatically optimize the expansion in the logical framework Twelf [114, 116] without sacrificing any computational content. See also [77] for the proposal for Agda to try unifying arguments before expansion.

Records and general sums. When we are bundling data together, there is always a choice between using (possibly nested) sum types and defining new specific record types. A major benefit of defining new record types is that the new type constructor is injective (for unification) and thus can be used to pass implicit parameters. For example, let `X a` be a new record type, a function of type `(a : A) → X a` can make the first argument `a` implicit because the unification of `X a` and `X b` forces the unification of `a` and `b`. A definition using sum types might not be able to achieve this.[3] However, sum types enjoy a large collection of lemmas, which might offset the cost of specifying implicit arguments.

Univalence. Here is another example of the trade-off between these factors. Using univalence, proving lemmas about equivalences often become trivial because identification elimination only considers reflexivity (or effectively only identity functions). However, the price is that all identifications constructed by the univalence axiom are stuck, which in turn demands manual manipulation at the call sites. Despite the (superficial) convenience of the univalence axiom, I chose to limit the usage of it for more computational content.

Note that the difference between a good design and a bad one (in terms of balancing these factors) can make a mechanized proof impossible to type check or too difficult to finish despite it being based on the same paper proof.

The best way to feel the struggle is through real code. Here is a (slightly adapted) example from the real development:[4] we will explore the design space of the mechanized definition of finite-dimensional CW complexes defined in section 3.5.1. Fixing some universe level `i`, the primary goal is to have a family of types

```
Skeleton : ℕ → Type (lsucc i)
```

such that `Skeleton n` is the type of combinatorial descriptions of `n`-dimensional CW complexes whose cell index sets are of universe level `i`. The secondary goal is to find the *best* definition judged by the above three criteria. Because the boundary of a cell is a function from a sphere to the realized CW complex up to the previous dimension, the skeleton family `Skeleton` and the realizer, which is named `Realizer`, are mutually defined. In sum, we want the following two functions:

```
Skeleton : ℕ → Type (lsucc i)
Realizer : {n : ℕ} → Skeleton n → Type i
```

Here is a naïve inductive implementation closely following the descriptions in section 3.5.1.

[3] Agda can infer injectivity in some cases.

[4] The mismatch is due to the convenient helper functions used in real mechanized proofs to save typing and make code beautiful. This section inlines most of them.

```
Skeleton O = Σ (Type i) is-set
Skeleton (S n) =
  Σ (Skeleton n) λ skel →
    Σ (Σ (Type i) is-set) λ cells →
      fst cells → Sphere n → Realizer {n} skel

Realizer {n = O} A = fst A
Realizer {n = S n} (skel , cells , α) = Pushout (span
  (Realizer {n} skel) (fst cells) (fst cells × Sphere n)
  (uncurry α) fst)
```

However, there is a major drawback of this naïve implementation: Suppose we define a helper function to strip off the highest dimension called `cw-init`.

```
cw-init : ∀ {n} → Skeleton (S n) → Skeleton n
cw-init (skel , _ , _) = skel
```

The problem is that, in a typical use of `cw-init` such as `cw-init skel`, the implicit argument `n` will not be inferred from the skeleton `skel`; users often have to write `cw-init {n} skel` to specify the dimension. The reason is that the current Agda is unable to infer injectivity of `Skeleton`, as both cases of `Skeleton` have the same head symbol `Σ`. A direct solution is to replace either the definition of `Skeleton O` or that of `Skeleton (S n)` with a custom record type. Here I define `AttachedSkeleton` to replace the nested `Σ` type used in the successor case:

```
record AttachedSkeleton n (Skel : Type (lsucc i))
  (Real : Skel → Type i) : Type (lsucc i) where
  constructor attached-skeleton
  field
    skel : Skel
    cells : hSet i
    attaching : fst cells → Sphere n → Real skel

Skeleton : ℕ → Type (lsucc i)
Realizer : {n : ℕ} → Skeleton n → Type i

Skeleton O = Σ (Type i) is-set
Skeleton (S n) = AttachedSkeleton n (Skeleton n) Realizer

Realizer {n = O} A = fst A
Realizer {n = S n} (attached-skeleton skel cells α) =
  Pushout (span
    (Realizer skel) (fst cells) (fst cells × Sphere n)
    (uncurry α) fst)
```

This works well in practice and is used in our mechanization; as a bonus in style, it uses dedicated projections instead of the generic `fst` and `snd`. There are two other possible definitions of `AttachedSkeleton` but they seem inferior: One possibility is a data type with only one constructor.

```
data AttachedSkeleton n (Skel : Type (lsucc i))
  (Real : Skel → Type i) : Type (lsucc i) where
  attached-skeleton :
      (skel : Skel)
    → (cells : hSet i)
    → (fst cells → Sphere n → Real skel)
    → AttachedSkeleton n Skel Real
```

Another choice is a recursive record type to avoid abstraction over `Skeleton`:

```
record AttachedSkeleton (n : ℕ) : Type (lsucc i) where
  inductive
  constructor attached-skeleton
  field
    skel : Skeleton n
    cells : hSet i
    attaching : fst cells → Sphere n → Realizer skel
```

Both alternative solutions lack the uniqueness rule that makes `cw-init skel` or other operations compute without further pattern matching. This may seem minor on first thought, but one of the subjects of my study, cohomology groups, is all about adjacent dimensions, and additional boilerplate is needed if operations to extract neighboring information are stuck. Therefore these two solutions score low on the economy of syntax.

To summarize the impact of different choices, let's consider the inclusion function `cw-incl-last` from the realization up to the previous dimension to that up to the current dimension. In my current mechanization using a non-recursive record type, this can be defined as:

```
cw-incl-last : ∀ {n} (skel : Skeleton (S n))
  → (Realizer (cw-init skel) → Realizer skel)
cw-incl-last _ = left
```

In the naïve definition using `Σ`, dimensions need to be explicitly specified.

```
cw-incl-last : ∀ {n} (skel : Skeleton (S n))
  → (Realizer {n} (cw-init {n} skel) → Realizer {S n} skel)
cw-incl-last _ = left
```

Finally, in the solutions using data types or recursive record types, additional pattern matching is required.

```
cw-incl-last : ∀ {n} (skel : Skeleton (S n))
  → (Realizer (cw-init skel) → Realizer skel)
cw-incl-last (attached-skeleton _ _ _) = left
```

My choice is a clear winner in terms of the economy of syntax and the computational content, so the only worry is whether the type checking is significantly slowed down. Fortunately, the current development can be checked in a reasonable amount of time and it is hard to believe that any of the above alternative definitions can considerably improve the checking time.

4.3 Head-to-Head Comparison

It has been asserted (perhaps too many times) that the Agda mechanization is comparable to the UniTT+hit proofs on paper. In this section I shall substantiate this claim by showing part of the real mechanized proof of the Blakers–Massey theorem. Except for the definition of `app=`, the only changes from the actual code are comment removal and line break insertion; even the indentation is preserved. The mechanization starts with

```
module homotopy.BlakersMassey {i j k}
  {A : Type i} {B : Type j} (Q : A → B → Type k)
  m (f-conn : ∀ a → is-connected (S m) (Σ B (λ b → Q a b)))
  n (g-conn : ∀ b → is-connected (S n) (Σ A (λ a → Q a b)))
  where
```

which matches the preconditions of theorem 3.4.1 except that m and n are replaced by `S m` and `S n` (and $m + n$ by `m +2+ n` below). This way we can drop the condition $m, n \geq -1$; see remark 3.1.1. In this section we will focus on the part after establishing the crucial equivalence in section 3.4.2 for $\mathtt{apd}_{\mathtt{code}}(\mathtt{qglue}(q_{1;1}))$, that is,

$$\|\mathtt{hfiber}_{\lambda q_{1;0}.\mathtt{qglue}(q_{0;0})\cdot\mathtt{qglue}(q_{1;0})^{-1}\cdot\mathtt{qglue}(q_{1;1})}(r)\|_{m+n} \simeq \|\mathtt{hfiber}_{\mathtt{qglue}}(r)\|_{m+n}$$

or, in my Agda mechanization,

```
eqv : ∀ {a₀ a₁ b₀ b₁} (q₀₀ : Q a₀ b₀) (q₁₁ : Q a₁ b₁) r
  → Trunc (m +2+ n) (hfiber (λ q₁₀ → bmglue q₀₀
                                 ∙' ! (bmglue q₁₀)
                                 ∙' bmglue q₁₁) r)
  ≃ Trunc (m +2+ n) (hfiber bmglue r)
```

Remark 4.3.1. The primed concatenation `∙'` is similar to the ordinary concatenation `∙`, except that it pattern matches on the second argument instead of the first one; that is, `p ∙' idp` reduces to `p` but `p ∙ idp` will not. This is useful here because later on we will replace `bmglue q₁₁` by `idp` and only the primed version computes. A careful choice between implementations that are identified but computationally different can make a huge difference in mechanization.

The setup begins with the dedicated pushout after the family-as-fibration transformation. This is done by importing the dedicated pushout library

```
open import homotopy.blakersmassey.Pushout Q
```

which contains the following definitions (and more):

```
bmspan : Span {i} {j} {lmax i (lmax j k)}
bmspan = span A B (Σ A λ a → Σ B λ b → Q a b) fst (fst ∘ snd)

BMPushout : Type (lmax i (lmax j k))
BMPushout = Pushout bmspan

bmleft : A → BMPushout
bmleft = left

bmright : B → BMPushout
bmright = right

bmglue : ∀ {a b} → Q a b → bmleft a == bmright b
bmglue {a} {b} q = glue (a , b , q)
```

where the prefix "`bm`" or "`BM`" means "Blakers–Massey". The Agda `BMPushout` corresponds to the pushout P in the UniTT+HIT proof, and `bmglue` corresponds to the function qglue. We then import the equivalence library

```
import homotopy.blakersmassey.CoherenceData Q m f-conn n g-conn
  as Coh
```

under the namespace `Coh`; this corresponds to the equivalence proved in section 3.4.2 for $\mathtt{apd}_{\mathtt{code}}(\mathtt{qglue}(q_{1;1}))$. We then assume there is an element `q₀₀ : Q a₀ b₀` and will discharge it eventually, again perfectly matching the UniTT+HIT proof:

```
module _ {a₀} {b₀} (q₀₀ : Q a₀ b₀) where
```

We proceed by defining the `code` through pushout elimination using the equivalence `Coh.eqv` from the library. Among the following lemmas the only new ingredients are the ones with the suffix `-template`, whose sole purpose is to enable identification elimination on `bmglue q₀₁` later in the proof. Because both end points of `bmglue q₀₁` are not free, we have to abstract over the right end point `bmright b₁` as well. In the following, the argument `r'` to `code-bmleft-template` results from the abstraction over `bmglue q₁₁` in the type of `Coh.eqv`.

```
code-bmleft-template : ∀ a₁ {p} (r' : bmleft a₁ == p)
  → bmleft a₀ == p
  → Type (lmax i (lmax j k))
code-bmleft-template a₁ r' r = Trunc (m +2+ n)
  (hfiber (λ q₁₀ → bmglue q₀₀
                  ∙' ! (bmglue q₁₀)
                  ∙' r') r)

code-bmleft : ∀ a₁ → bmleft a₀ == bmleft a₁
  → Type (lmax i (lmax j k))
code-bmleft a₁ = code-bmleft-template a₁ idp

code-bmright : ∀ b₁ → bmleft a₀ == bmright b₁
  → Type (lmax i (lmax j k))
code-bmright b₁ r = Trunc (m +2+ n) (hfiber bmglue r)
```

Similarly, the argument `r` to `code-bmglue-template` will be given `bmglue q₀₁`; this template even abstracts over the `code` because we are still defining it.

```
code-bmglue-template : ∀ {a₁ p}
  → (code : (r : bmleft a₀ == p) → Type (lmax i (lmax j k)))
  → (r : bmleft a₁ == p)
  → (∀ r' → code-bmleft-template a₁ r r' ≃ code r')
  → code-bmleft-template a₁ idp == code
    [ (λ p → bmleft a₀ == p → Type (lmax i (lmax j k))) ↓ r ]
code-bmglue-template _ idp lemma = λ= (ua ∘ lemma)

code-bmglue : ∀ {a₁ b₁} (q₁₁ : Q a₁ b₁)
  → code-bmleft a₁ == code-bmright b₁
    [ (λ p → bmleft a₀ == p → Type (lmax i (lmax j k)))
      ↓ bmglue q₁₁ ]
code-bmglue {a₁} {b₁} q₁₁ = code-bmglue-template
  (code-bmright b₁) (bmglue q₁₁) (Coh.eqv q₀₀ q₁₁)
```

With all these components, we are ready to define the `code` in Agda. The Agda `code p r` matches the UniTT+hit $\texttt{code}(p, r)$.

```
module Code = BMPushoutElim
  code-bmleft code-bmright code-bmglue

code : ∀ p → bmleft a₀ == p → Type (lmax i (lmax j k))
code = Code.f
```

The remaining part is the contractibility of `code` and the discharge of `q₀₀`. First, we want to state there is a center in any fiber of `code`:

```
  code-center : ∀ {p} r → code p r
```

This can be done by identification elimination on `r` and the following lemma:

```
  code-center-idp : code (bmleft a₀) idp
  code-center-idp = [ q₀₀ , !-inv'-r (bmglue q₀₀) ]
```

where the identification elimination on `r` effectively transports this center into all fibers. However, we have to understand the computational behavior of this transportation (up to identification) because the contractibility of fibers of `code` relies upon knowing the exact values of the transported centers. Therefore, we should not rush into identification elimination but give a good description of the coercion from `code (bmleft a₀) idp` to `code p r`. The definition of the coercion, again, starts with a general template which enables identification elimination on `r`.

```
  coerce-path-template : ∀ {p} r
    → code-bmleft a₀ == code p
      [ (λ p → bmleft a₀ == p → Type (lmax i (lmax j k)))
        ↓ r ]
    → code-bmleft a₀ idp == code p r
  coerce-path-template idp lemma = app= lemma idp

  coerce-path : ∀ {p} r → code (bmleft a₀) idp == code p r
  coerce-path r = coerce-path-template r (apd code r)

  code-center : ∀ {p} r → code p r
  code-center r = coe (coerce-path r) code-center-idp
```

where the lemma `app=` is the inverse of the functional extensionality `λ=`. (The following definition of `app=` was slight adapted from the real library code by incorporating module parameters and removing its indentation.)

```
app= : ∀ {j} {P : A → Type j} {f g : (x : A) → P x}
  → (p : f == g) → ((x : A) → f x == g x)
app= p x = ap (λ u → u x) p
```

The goal is the following lemma

```
  code-coh-lemma : ∀ {b₁} (q₀₁ : Q a₀ b₁)
    → code-center (bmglue q₀₁) == [ q₀₁ , idp ]
```

which implies the contractibility of all fibers of `code` by the same argument in section 3.4.3:

```
code-coh : ∀ {b₁} (r : bmleft a₀ == bmright b₁)
  (s : hfiber bmglue r) → code-center r == [ s ]
code-coh ._ (q₀₁ , idp) = code-coh-lemma q₀₁

code-contr : ∀ {b₁} (r : bmleft a₀ == bmright b₁)
  → is-contr (Trunc (m +2+ n) (hfiber bmglue r))
code-contr r = code-center r , Trunc-elim
  (λ _ → =-preserves-level Trunc-level) (code-coh r)
```

Expanding the definition of `code-center`, it becomes obvious that we have to understand how the transporting function `coe (coerce-path r)` works in the case that `r` is `bmglue q₀₁`. In particular, we would like to show

```
coe-coerce-path-code-bmglue : ∀ {b₁} (q₀₁ : Q a₀ b₁) x
  → coe (coerce-path (bmglue q₀₁)) x
  == Coh.to q₀₀ q₀₁ (bmglue q₀₁)
       (code-bmleft-template-diag (bmglue q₀₁) x)
```

where the function `Coh.to q₀₀ q₀₁ (bmglue q₀₁)` is the forward computational content of the equivalence `Coh.eqv q₀₀ q₀₁ (bmglue q₀₁)` and the mediating function `code-bmleft-template-diag` reconciles the type mismatch.

The mediating function `code-bmleft-template-diag` bridges the gap between the fiber `code-bmleft a₀ idp` (which is `code-bmleft-template a₀ idp idp`) and `code-bmleft-template a₀ r r`:

```
code-bmleft-template-diag : ∀ {p} (r : bmleft a₀ == p)
  → code-bmleft a₀ idp → code-bmleft-template a₀ r r
code-bmleft-template-diag r = Trunc-rec Trunc-level
  λ {(q₀₀' , shift) →
    [ q₀₀' , ! (∙'-assoc (bmglue q₀₀) (! (bmglue q₀₀')) r)
           ∙ ap (_∙' r) shift ∙' ∙'-unit-l r ]}
```

which may be visualized as “extending identifications through `r`”; in other words, it asserts any `q₀₀'` satisfying the equation

$$\begin{tikzcd}
\texttt{bmleft a}_0 \arrow[r, bend left, "\texttt{bmglue q}_{00}"] & \texttt{bmright b}_1 \arrow[l, bend left, "\texttt{! (bmglue q}_{00}\texttt{')}"] & = & \texttt{bmleft a}_0 \arrow[loop right, "\texttt{idp}"]
\end{tikzcd}$$

also satisfies the equation:

$$\begin{tikzcd}
\texttt{bmleft a}_0 \arrow[r, bend left, "\texttt{bmglue q}_{00}"] \arrow[dr, "\texttt{r}"'] & \texttt{bmright b}_1 \arrow[l, bend left, "\texttt{! (bmglue q}_{00}\texttt{')}"] & = & \texttt{bmleft a}_0 \arrow[dr, "\texttt{r}"'] & \\
 & \texttt{p} & & & \texttt{p}
\end{tikzcd}$$

It is useful to show its computational behavior (up to identification) when we plug in `idp` for `r`; unsurprisingly, it becomes an identity function because no coercion is needed.

```
abstract
  code-bmleft-template-diag-idp :
    ∀ x → code-bmleft-template-diag idp x == x
  code-bmleft-template-diag-idp =
    Trunc-elim (λ _ → =-preserves-level Trunc-level)
      λ{(q₁₀ , shift) → ap (λ p → [ q₁₀ , p ]) (ap-idf shift)}
```

where `ap-idf` is the lemma showing that `ap` of an identity function is still an identity function. The reason that we avoided applying identification elimination to `r` in the definition of the mediating function `code-bmleft-template-diag` is again to make possible reduction under non-`idp` values (in particular, `bmglue q₀₁`). The above lemma is marked `abstract` because its computational content is insignificant. This concludes the definition of the mediating function.

Going back to the decomposition of `coe (coerce-path (bmglue q₀₁))`, it is time to reap the fruits of our labors of abstracting over `bmglue q₀₁`! The following is a general lemma which exploits the identification elimination on `r`; it would be painful if we had to consider the non-`idp` cases.

```
abstract
  coe-coerce-path-code-bmglue-template : ∀ {p}
    (r : bmleft a₀ == p)
    (lemma : ∀ r' → code-bmleft-template a₀ r r' ≃ code p r')
    (x : code-bmleft a₀ idp)
    → coe (coerce-path-template r
            (code-bmglue-template (code p) r lemma)) x
    == –> (lemma r) (code-bmleft-template-diag r x)
  coe-coerce-path-code-bmglue-template idp lemma x =
    coe (app= (λ= (ua ∘ lemma)) idp) x
    =⟨ ap (λ p → coe p x) (app=-β (ua ∘ lemma) idp) ⟩
  coe (ua (lemma idp)) x
    =⟨ coe-β (lemma idp) x ⟩
  -> (lemma idp) x
    =⟨ ! (ap (-> (lemma idp))
             (code-bmleft-template-diag-idp x)) ⟩
  -> (lemma idp) (code-bmleft-template-diag idp x)
    =∎
```

where `->` extracts the forward function from an equivalence; and the `… =⟨ … ⟩ … =∎` notation helps organize a long identification proof, with each identification within the brackets `⟨ … ⟩` identifying its adjacent expressions. With this lemma, we can finish the decomposition of `coe (coerce-path (bmglue q₀₁))`:

```
abstract
  coe-coerce-path-code-bmglue : ∀ {b₁} (q₀₁ : Q a₀ b₁) x
    → coe (coerce-path (bmglue q₀₁)) x
    == Coh.to q₀₀ q₀₁ (bmglue q₀₁)
         (code-bmleft-template-diag (bmglue q₀₁) x)
  coe-coerce-path-code-bmglue q₀₁ x =
    coe (coerce-path-template (bmglue q₀₁)
        (apd code (bmglue q₀₁))) x
      =⟨ ap (λ p → coe
              (coerce-path-template (bmglue q₀₁) p) x)
            (Code.glue-β q₀₁) ⟩
    coe (coerce-path-template (bmglue q₀₁)
        (code-bmglue q₀₁)) x
      =⟨ coe-coerce-path-code-bmglue-template (bmglue q₀₁)
           (Coh.eqv q₀₀ q₀₁) x ⟩
    Coh.to q₀₀ q₀₁ (bmglue q₀₁)
           (code-bmleft-template-diag (bmglue q₀₁) x)
      =∎
```

More importantly, the desired identification between `code-center (bmglue q₀₁)` and `[ q₀₁ , idp ]` can be constructed from the decomposition lemma and computational rules (up to identification) derived from lemma 3.1.3 (wedge connectivity).

```
abstract
  code-coh-lemma : ∀ {b₁} (q₀₁ : Q a₀ b₁)
    → code-center (bmglue q₀₁) == [ q₀₁ , idp ]
  code-coh-lemma q₀₁ =
    coe (coerce-path (bmglue q₀₁)) code-center-idp
      =⟨ coe-coerce-path-code-bmglue q₀₁ code-center-idp ⟩
    Coh.to' q₀₀ q₀₁ (bmglue q₀₁)
            (q₀₀ , α₁α₁⁻¹α₂=α₂ (bmglue q₀₀) (bmglue q₀₁))
      =⟨ ap (Coh.To.ext q₀₀ (_ , q₀₀) (_ , q₀₁) (bmglue q₀₁))
            (path-lemma (bmglue q₀₀) (bmglue q₀₁)) ⟩
    Coh.To.ext q₀₀ (_ , q₀₀) (_ , q₀₁) (bmglue q₀₁) (! path)
      =⟨ Coh.To.β-r q₀₀ (_ , q₀₁) (bmglue q₀₁) (! path) ⟩
    [ q₀₁ , path ∙' ! path ]
      =⟨ ap (λ p → [ q₀₁ , p ]) (!-inv'-r path) ⟩
    [ q₀₁ , idp ]
      =∎
    where
      path = Coh.βPair.path
        (Coh.βpair-bmright q₀₀ q₀₁ (bmglue q₀₁))

      -- this is defined to be the path
      -- generated by [code-bmleft-template-diag]
      α₁α₁⁻¹α₂=α₂ : ∀ {p₁ p₂ p₃ : BMPushout}
        (α₁ : p₁ == p₂) (α₂ : p₁ == p₃)
        → α₁ ∙' ! α₁ ∙' α₂ == α₂
      α₁α₁⁻¹α₂=α₂ α₁ α₂ = ! (∙'-assoc α₁ (! α₁) α₂)
                        ∙ ap (_∙' α₂) (!-inv'-r α₁)
                        ∙' ∙'-unit-l α₂

      -- the relation of this path
      -- and the one from CoherenceData
      path-lemma : ∀ {p₁ p₂ p₃ : BMPushout}
        (α₁ : p₁ == p₂) (α₂ : p₁ == p₃)
        → α₁α₁⁻¹α₂=α₂ α₁ α₂ == ! (Coh.α₁=α₂α₂⁻¹α₁ α₂ α₁)
      path-lemma idp idp = idp
```

Now that we finished the most difficult part, it is easy to prove the contractibility as already shown above.

```
  code-coh : ∀ {b₁} (r : bmleft a₀ == bmright b₁)
    (s : hfiber bmglue r) → code-center r == [ s ]
  code-coh ._ (q₀₁ , idp) = code-coh-lemma q₀₁

  code-contr : ∀ {b₁} (r : bmleft a₀ == bmright b₁)
    → is-contr (Trunc (m +2+ n) (hfiber bmglue r))
  code-contr r = code-center r , Trunc-elim
    (λ _ → =-preserves-level Trunc-level) (code-coh r)
```

The final step is to discharge `q₀₀` through the connectivity of `f` (or `g`). In the following code snippet, the lemma `is-connected-is-prop` states that connectivity itself is a

Result	UniTT+HIT	Location of the AGDA entry point	AGDA entry point	#lines
Representations of covering spaces	theorem 3.2.2	`theorems/homotopy/GroupSetsRepresentCovers.agda`	`grpset-equiv-cover`	198
Universal covering spaces	represented by π_1	`theorems/homotopy/GroupSetsRepresentCovers.agda`	`path-set-repr-by-π1`	5 more *(added to the above)*
	lemma 3.2.6	`theorems/homotopy/AnyUniversalCoverIsPathSet.agda`	`theorem`	75
	lemma 3.2.7	`theorems/homotopy/PathSetIsInitalCover.agda`	`Uniqueness.theorem`	35
Blakers–Massey	theorem 3.4.1	`theorems/homotopy/BlakersMassey.agda`	`blakers-massey`	630
Seifert–van Kampen *(the improved version)*	theorem 3.3.4	`theorems/homotopy/VanKampen.agda`	`vankampen`	1168
Cohomology groups	lemma 3.5.5	`theorems/cw/cohomology/ReconstructedCohomologyGroups.agda`	`reconstructed-cohomology-group`	2368 *(for both lemmas)*
	lemma 3.5.6	`theorems/cw/cohomology/ReconstructedCochainsIsoCellularCochains.agda`	`rcc-iso-ccc`	

Table 4.3: Theorem Lookup Table

mere proposition and the lemma `prop-has-level-S` asserts that a mere proposition is of level `S m` for any `m`.

```
blakers-massey : ∀ {a₀ b₀} (r : bmleft a₀ == bmright b₀)
  → has-conn-fibers (m +2+ n) (bmglue {a₀} {b₀})
blakers-massey {a₀} r = Trunc-rec
  (prop-has-level-S is-connected-is-prop)
  (λ{(_ , q₀₀) → code-contr q₀₀ r})
  (fst (f-conn a₀))
```

We are done! The important message of this section is that AGDA proofs and UniTT+HIT proofs work at the same level of abstraction and their lengths are in the same order of magnitude.

4.4 Summary

The latest development is available on GITHUB [29] and the reference version was kept on FIGSHARE [30]. It requires AGDA 2.5.2 or newer to type check. A major part of the code repository (including my entire thesis) can be checked in total about 20 minutes on a reasonably new machine. See table 4.3 for the list of major theorems and lemmas in chapter 3 that have been mechanized in Agda. Note that the naïve version Seifert–van Kampen (theorem 3.3.3) was not mechanized because it was superseded by the improved one (theorem 3.3.4).

Chapter 5

Concluding Remarks

Computer science values efficiency and feasibility, and in the context of mechanization, the main questions are whether we can mechanize proofs and how much time it takes to code and check these mechanized proofs. In this thesis, I have shown many examples of using higher-dimensional types in the mechanization of homotopy theory. While I can never mathematically *prove* my thesis statement that mechanization with higher-dimensional types is easier than without, the volume and the diversity of my contribution within such a short time should constitute compelling evidence.

Although the homotopy-theoretic examples in chapter 3 were presented in UniTT+HIT instead of AGDA code due to typesetting issues, chapter 4 shows that every component in UniTT+HIT can be directly translated into AGDA without any loss of abstraction. Chapter 3 would be largely the same if presented in AGDA code; in fact, section 3.4 was adapted from our paper which uses AGDA syntax. This is because type theory supports formal abstraction that enables high-level reasoning even in mechanized proofs, as pointed out in chapter 1.

That said, by no means do I suggest that the system I used for mechanization is the best one. Currently, many different type theories and their accompanying proof assistants are being actively developed, some listed in section 1.5. I am always looking forward to a new type theory, or even a completely new foundation different from type theory, as long as it serves the purpose well. Indeed, I hope that my writing can help future researchers to construct new systems vastly better than the current ones.

Appendices

Appendix A

How to Typeset This Thesis

Hello, thesis writers! LaTeX and its family are beautifully painful, and thus the philosophy of this thesis[1] forces me to show you what I have learned from my thesis writing. In this era of search engines, I believe it is most useful to give *only* keywords. Also, I will not list obvious resources that you will try to find (such as the package `dtk-logos` or the one for derivation trees).

You should not believe anything written below, unless you have verified the statements against the most recent documentation. There is an intentional bug in this appendix in order to force you to do that.

A.1 Packages You Should Know

First of all, definitely check out these packages (or their successors) *because you may not even know you want them!*

- `mathtools`: this fixes bugs in `amsmath` and provides goodies like `multlined`.
- `microtype`: a compelling reason to stay in pdfLaTeX (thanks to Ryan Kavanagh).
- `mleftright`: I assume you already knew the built-in `\left` and `\right` are bad.
- `letltxmacro`: never do `\let\oldmacro\macro` when redefining LaTeX macros.

A.1.1 Bibliography

Because your thesis is not constrained by outdated journal or proceedings templates, you should consider `biblatex` and Biber instead of `natbib`, BibTeX or other legacy systems. Using new features may break backward compatibility, though.

[1]See the preface on page ix.

A.1.2 Multilingual Support

Consider `polyglossia` or `babel` if your document is multilingual (like this thesis). The package `csquotes` might come in handy.

A.1.3 Compiling Individual Chapters

Try `subfiles` or other similar packages. Just be aware of this possibility.

A.1.4 Cross Referencing

The following three must be loaded *exactly in this order* if you want them all:

1. `varioref`: adding "on page". You may or may not want it.

2. `hyperref`: you want this.

3. `cleveref`: adding "table", "figure" and others. You may or may not want it.

A.1.5 LaTeX Hacking

- `calc`: flexible length expressions.

- `adjustbox`: vertical alignments and various features.

- `etoolbox` and `xpatch`: LaTeX 2_ε hacking.

- `xparse`: programming in the LaTeX3 style instead of the old `\newcommand`.

Remember to `\protect` your code if it is too fragile.

A.1.6 Tables

The ultimate choice for tables consists of:

- `tabu`: a replacement of many, many other similar packages. However, the current version has bugs and is unmaintained. Use at your own risk.

- `booktabs`: Just Beautiful™.

The following are for long tables like table 4.1.

- `longtable`: spanning tables across multiple pages.

- `caption`: adding a caption to anything.

A.1.7 Lists

The package `enumitem` is again a replacement of many other similar packages.

A.1.8 Others

The package `underscore` gives you underscore as underscore in the text mode.

A.2 Unicode

This thesis uses a great varieties of characters scattered across different Unicode blocks, not just Latin alphabets with diacritical marks; thus, the package `inputenc` is far from being sufficient and the real solution is XeLaTeX, LuaLaTeX or their successors. The major price to pay is their immaturity compared to the good old LaTeX 2_ε (or pdfLaTeX); for example,

- it is tricky to have fine-tuned font packages such as `newpxmath` and `newpxtext` peacefully coexist with new mechanisms such as `fontspec` and `polyglossia`; and
- the support of microtypography (or more precisely, the package `microtype`) is limited in XeLaTeX; and
- the package `xy` generates ugly diagonal lines when used in XeLaTeX (and in the meanwhile the package `tikz` is free from this issue).

Good OpenType fonts with the `math` extension and good support of these fonts are essential for beautiful mathematical equations in XeLaTeX and LuaLaTeX, and there is much room for improvement in these areas. This thesis is a proof that XeLaTeX can handle serious typesetting but mathematical symbols demand more subtle tuning.

Appendix B

Buddhism in Martin-Löf Type Theory

It is impossible to formalize Buddhism (as a metaphysics without the rebirth part) because the nature of Buddhism, as many metaphysical theories, lies above the limits of our languages. However, I still want to give readers of my thesis a taste of *emptiness,* an important concept in Buddhism, and show how the Martin-Löf ontological type theory with identification types may approach this.

As diverse as Christianity and other religions, there are many Buddhist schools with distinct practices and teachings. In this appendix, I will loosely follow my understanding of Prāsaṅgika, a subschool of Madhyamaka, especially the works of the well-known Buddhist scholar Candrakīrti [133]. That said, this appendix is aimed at formalizing a particular argument used in Buddhism, *not the actual content of Buddhism.* We may start with the Diamond Sūtra, an influential Buddhist text. The following is an English translation of a paragraph in its chapter 17 [152]:

> Should anyone say, Subhūti, that the Realized One has fully awakened to supreme and perfect awakening, there is no *dharma* whatsoever to which the Realized One has fully awakened as supreme and perfect awakening. In the *dharma* to which the Realized One has fully awakened, there is no truth and no falsehood. Therefore the Realized One preaches "All *dharmas* are Buddha-*dharmas.*" As far as "all *dharmas*" are concerned, Subhūti, all of them are *dharma*-less. That is why they are called "all *dharmas.*"

(The italic marking of *dharmas* is done by me.) This is part of a conversation between the Buddha and Subhūti, one of the distinguished disciples of the Buddha; the Buddha is the sole speaker in this paragraph. Abstracting over *dharmas,* the message is roughly that *a concept P spoken by the Buddha is P-less, and that is why it is called P.* This paradoxical, non-dual figure of speech has been used numerous times in the Diamond Sūtra to help its readers understand the *emptiness of concepts* (including the emptiness of the emptiness itself in the view of Prāsaṅgika). Buddhism in general believes that persistent practice based on this philosophical view can liberate your mind from attachment and thus from all suffering. Nonetheless, let's suffer a little bit more by trying to translate this into the Martin-Löf type theory with identification; there is actually something to say.

For the sake of simplicity, let's ignore all the philosophical incompatibilities and freely reinterpret Gottlob Frege's *concept* in type theory as a family of types P indexed by some type `object` in $\mathcal{U}_i$; that is,

$$\texttt{concept} :\equiv \texttt{object} \to \mathcal{U}_i$$

where an object $o : \texttt{object}$ belongs to a concept P if and only if $P(o)$ is inhabited.[1] The paradoxical nonduality may be formalized as equivalences between all fibers:

$$\begin{aligned} \texttt{is-non-dual} &: \texttt{concept} \to \mathcal{U}_i \\ \texttt{is-non-dual}(P) &:\equiv \prod_{o_1,o_2:\texttt{object}} P(o_1) \simeq P(o_2). \end{aligned}$$

The idea is that a conventional[2] concept P, intuitively, should admit at least one example and one counterexample; for example, you probably want an object of red and another object of no-red before discussing redness. The nonduality then non-affirmingly negates[3] a conventional concept by eliminating the distinction between examples and counterexamples at the ultimate level. In terms of redness, it means ultimately there is no way to separate red objects from no-red objects, and the concept *red* is ultimately empty. In the following, we will see a sufficient and necessary condition of nonduality.

One popular argument for nonduality (that will be formalized in the type theory) is through the *identification of any two objects.* We may start with the philosophical question of *identity* and *change:* When will a banana rot enough that it is no longer a banana? Similarly, how much do I have to disassemble a chair so that it is no longer a chair? Various Buddhist schools hold the view that all conventional objects are interdependent and interconnected, and that any attempt to draw a line between bananas and no-bananas or chairs and no-chairs is misguided at best. Under this belief, if there is a conventional concept as a non-trivial partition of conventional objects, we may continuously move from an example to a counterexample and examine how and where the separation breaks down, and eventually conclude that such a concept is ultimately empty.

The surprise is that we can easily carry out the argument from identification to nonduality in the type theory. The following is a short formal proof of the desired theorem:

$$\lambda\Big(q\colon \prod_{o_1,o_2} o_1 = o_2\Big).\lambda(P{:}\texttt{concept}).\lambda(o_1,o_2{:}\texttt{object}).\texttt{coerce-equiv}\Big(\texttt{ap}_P\Big(q(o_1,o_2)\Big)\Big)$$

$$: \Bigg(\prod_{o_1,o_2:\texttt{object}} o_1 = o_2\Bigg) \to \Bigg(\prod_{P:\texttt{concept}} \texttt{is-non-dual}(P)\Bigg).$$

[1]Truncation is intentionally avoided to simplify the presentation.

[2]The word *conventional* here refers to the conventional-ultimate distinction. There are disagreements about the nature of the *ultimate reality* or the *ultimate truth* between different Buddhist (sub)schools, but in no ways can I address those in this appendix. Therefore, I have to confine myself to conventional concepts. Readers unfamiliar with Buddhism may largely ignore the word *conventional.*

[3]Negating without affirming the opposite.

Moreover, we can show that the inverse is also true! This means the identification of all objects is not only sufficient but also necessary for nonduality, at least in our formulation. Fixing some object o_1, the trick is to choose the identification itself, $\lambda o'_2.o_1 = o'_2$, as the concept. We can see that $P(o_1)$ is inhabited by $\texttt{refl}_{o_1}$ and thus all fibers of P are inhabited by nonduality. This means all objects are identified with o_1.

$$\lambda\Big(n{:}\prod_{P}\prod_{o_1,o_2} P(o_1) = P(o_2)\Big).\lambda(o_1,o_2{:}\texttt{object}).\texttt{fst}\Big(n\big((\lambda o'_2.o_1 = o'_2), o_1, o_2\big)\Big)(\texttt{refl}_{o_1})$$

$$: \Bigg(\prod_{P:\texttt{concept}} \texttt{is-non-dual}(P)\Bigg) \to \Bigg(\prod_{o_1,o_2:\texttt{object}} o_1 = o_2\Bigg).$$

I hope this intellectual exercise is somewhat inspiring, especially after Francis William Lawvere's take on Hegelian philosophy. As usual, all the proofs have been mechanized in Agda. I would like to emphasize again that a large part of Buddhism is about the mental training to attain liberated mind, not these metaphysical discussions; in fact, the attachment to these metaphysical ideas could be a major obstacle to the liberation.

Appendix C

Notes on Linguistic Issues

C.1 Translation in Tai-Min

The abstract and the dedication of this thesis were translated into Tai-Min with the help of my dear friend Phín-tsì Kí. The language *Tai-Min* (Tâibân in Tai-Min), commonly known as *Taiwanese* (Tâiuân'uē in Tai-Min), is one of the most spoken languages in Taiwan. Unfortunately, most native speakers have the misconceptions that there is no writing system for Tai-Min and that Tai-Min can only serve as an inferior, colloquial "dialect". This is mostly due to the decades-long Mandarin-dominating language policy, and as a result I am witnessing the inevitable death of many languages in Taiwan. The story is similar to Frisian in Netherlands and the regional languages in Italy, where most native speakers never learn their writing systems. Though the birth and death of languages are regular part of human history, it is still disturbing to see that many people have lost their access to the rich materials published in their first language less than one century ago.

I hereby provided the translation as a demonstration that there is definitely a full-fledged orthography for Tai-Min; in fact, there are several established standards that were actively in use before the forceful promotion of Mandarin. Moreover, this shows that Tai-Min and many other languages in Taiwan are ready for scientific discussions in the most formal contexts—if their native speakers wanted to.

Remark C.1.1. The writing system used in this thesis is based on the Taiwanese Romanization System [153] but (single) hyphens are either removed or replaced by the apostrophe "'" for the sake of typesetting. Double hyphens are kept due to their special phonetic functionality. Even without the hyphens, a fluent (and literate) speaker should be able to read it without any difficulty.

C.2 Gender-Specific Pronouns

Existing gendered nouns and pronouns in English can be problematic for the people whose perceived gender, biological sex, gender identification and gender expression (including their choice of pronouns) do not perfectly line up under the traditional gender dichotomy. Therefore, I have been actively avoiding these words. As far as I know, the only occurrence of gender-specific pronouns in this thesis is the *he* referring to my advisor Robert Harper in the acknowledgments on page vii; I have personally verified it with my advisor.

Bibliography

[1] Andreas Abel and Thorsten Altenkirch. “A predicative analysis of structural recursion”. In: *Journal of Functional Programming* 12.1 (Jan. 2002), pp. 1–41. DOI: 10.1017/S0956796801004191 (cit. on p. 98).

[2] *Abstract definitions*. URL: http://agda.readthedocs.io/en/latest/language/abstract-definitions.html (visited on 02/15/2017) (cit. on p. 101).

[3] Benedikt Ahrens, Krzysztof Kapulkin and Michael Shulman. “Univalent categories and the Rezk completion”. In: *Mathematical Structures in Computer Science* 25.5 (Jan. 2015), pp. 1010–1039. DOI: 10.1017/S0960129514000486 (cit. on p. 41).

[4] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra and Fredrik Nordvall Forsberg. *Quotient inductive-inductive types*. 2016. arXiv: 1612.02346v1 [cs.LO] (cit. on p. 28).

[5] Thorsten Altenkirch, Paolo Capriotti and Nicolai Kraus. “Extending Homotopy Type Theory with Strict Equality”. In: *25th EACSL Annual Conference on Computer Science Logic (CSL)*. (Marseille, France, Aug. 29–Sept. 1, 2016). Ed. by Jean-Marc Talbot and Laurent Regnier. Vol. 62. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 21:1–21:17. ISBN: 978-3-95977-022-4. DOI: 10.4230/LIPIcs.CSL.2016.21 (cit. on p. 8).

[6] Thorsten Altenkirch and Ambrus Kaposi. *A syntax for cubical type theory*. URL: http://www.cs.nott.ac.uk/~psztxa/publ/ctt.pdf (visited on 03/18/2017) (cit. on p. 9).

[7] Mirian Andrés, Laureano Lambán, Julio Rubio and José Luis Ruiz-Reina. “Formalizing simplicial topology in ACL2”. In: *Proceedings of the Seventh International Workshop on the ACL2 Theorem Prover and its Applications*. Vol. 2007. 2007, pp. 34–39. URL: http://www.cs.uwyo.edu/~ruben/acl2-07/uploads/Main/007.pdf. Linked to the conference website. (Cit. on pp. 6, 89).

[8] Mathieu Anel, Georg Biedermann, Eric Finster and André Joyal. “The Generalized Blakers–Massey Theorem”. 2016. In preparation (cit. on pp. 2, 32, 89).

[9] Carlo Angiuli, Robert Harper and Todd Wilson. “Computational Higher-dimensional Type Theory”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. (Paris, France). New York, NY, USA: ACM, 2017, pp. 680–693. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009861 (cit. on p. 9).

[10] Carlo Angiuli, Edward Morehouse, Daniel R. Licata and Robert Harper. “Homotopical patch theory”. In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. (Gothenburg, Sweden, Sept. 1–3, 2014), pp. 243–256. ISBN: 978-1-4503-2873-9. DOI: 10.1145/2628136.2628158 (cit. on pp. 5, 6, 8, 10).

[11] Michael Francis Atiyah. “Bordism and Cobordism”. In: *Mathematical Proceedings of the Cambridge Philosophical Society* 57 (2 1961), pp. 200–208. ISSN: 0305-0041. DOI: 10.1017/S0305004100035064 (cit. on p. 80).

[12] Michael Francis Atiyah. “K-Theory Past and Present”. In: 2000. arXiv: 0012213v1 [math.KT] (cit. on p. 80).

[13] Steve Awodey. *Type theory and homotopy*. 2010. arXiv: 1010.1810v1 [math.CT].

[14] Steve Awodey, Nicola Gambino and Kristina Sojakova. “Inductive Types in Homotopy Type Theory”. In: *2012 27th Annual IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, June 2012, pp. 95–104. DOI: 10.1109/LICS.2012.21 (cit. on p. 25).

[15] Steve Awodey, Álvaro Pelayo and Michael A. Warren. “Voevovkdsky’s univalence axiom in homotopy type theory”. In: *Notices of the American Mathematical Society* 60.9 (2013), pp. 1164–1167. arXiv: 1302.4731v1 [math.HO].

[16] Steve Awodey and Michael A. Warren. “Homotopy theoretic models of identity types”. In: *Mathematical Proceedings of the Cambridge Philosophical Society* 146.1 (Jan. 2009), pp. 45–55. ISSN: 0305-0041. DOI: 10.1017/S0305004108001783 (cit. on pp. 6, 11).

[17] Jesús María Aransay Azofra. “Razonamiento mecanizado en álgebra homológica”. PhD thesis. Universidad de La Rioja, 2006. URL: https://dialnet.unirioja.es/servlet/tesis?codigo=403 (cit. on pp. 6, 89).

[18] Roland Backhouse, Paul Chisholm, Grant Malcolm and Erik Saaman. “Do-it-yourself type theory”. In: *Formal Aspects of Computing* 1.1 (1989), pp. 19–84. ISSN: 1433-299X. DOI: 10.1007/BF01887198 (cit. on p. 22).

[19] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Mike Shulman, Matthieu Sozeau and Bas Spitters. *The HoTT Library: A formalization of homotopy type theory in Coq*. 2016. arXiv: 1610.04591v2 [cs.LO] (cit. on pp. 6, 31).

[20] Benno van den Berg and Richard Garner. “Types are weak ω-groupoids”. In: *Proceedings of the London Mathematical Society* 102.2 (2011), pp. 370–394. ISSN: 0024-6115. DOI: 10.1112/plms/pdq026 (cit. on pp. 6, 16).

[21] Benno van den Berg and Richard Garner. “Topological and simplicial models of identity types”. In: *ACM Transactions on Computational Logic* 13.1 (2012), 3:1–3:44. ISSN: 1529-3785. DOI: `10.1145/2071368.2071371` (cit. on pp. 6, 11).

[22] Jean-Philippe Bernardy, Thierry Coquand and Guilhem Moulin. “A Presheaf Model of Parametric Type Theory”. In: *Electronic Notes in Theoretical Computer Science* 319 (2015), pp. 67–82. ISSN: 1571-0661. DOI: `10.1016/j.entcs.2015.12.006`.

[23] Marc Bezem, Thierry Coquand and Simon Huber. “A Model of Type Theory in Cubical Sets”. In: *19th International Conference on Types for Proofs and Programs.* (2013). Vol. 26. Leibniz International Proceedings in Informatics. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 107–128. ISBN: 978-3-939897-72-9. DOI: `10.4230/LIPIcs.TYPES.2013.107` (cit. on pp. 9, 91).

[24] Mark Bickford. *A formalization of Thierry Coquand's cubical type theory*. URL: `http://www.nuprl.org/wip/Mathematics/cubical!type!theory/index.html` (visited on 12/04/2016) (cit. on p. 9).

[25] Lars Birkedal, Aleš Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters and Andrea Vezzosi. “Guarded Cubical Type Theory: Path Equality for Guarded Recursion”. In: *25th EACSL Annual Conference on Computer Science Logic (CSL).* (Marseille, France, Aug. 29–Sept. 1, 2016). Ed. by Jean-Marc Talbot and Laurent Regnier. Vol. 62. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 23:1–23:17. ISBN: 978-3-95977-022-4. DOI: `10.4230/LIPIcs.CSL.2016.23` (cit. on p. 9).

[26] Ronald Brown. *Higher Dimensional Group Theory*. URL: `http://groupoids.org.uk/hdaweb2.html` (visited on 03/19/2017) (cit. on p. 89).

[27] Guillaume Brunerie. “On the homotopy groups of spheres in homotopy type theory”. PhD thesis. Université Nice Sophia Antipolis, 2016. arXiv: `1606.05916v1 [math.AT]` (cit. on p. 31).

[28] Guillaume Brunerie. *Implementation of higher inductive types in HoTT-Agda*. URL: `https://github.com/HoTT/HoTT-Agda/blob/master/core/lib/types/HIT_README.txt` (visited on 12/18/2016) (cit. on p. 100).

[29] Guillaume Brunerie, Kuen-Bang Hou (Favonia), Evan Cavallo, Jesper Cockx, Christian Sattler, Chris Jeris, Michael Shulman *et al. Homotopy Type Theory in Agda*. URL: `https://github.com/HoTT/HoTT-Agda` (visited on 12/02/2016) (cit. on pp. 6, 31, 101, 113).

[30] Guillaume Brunerie, Kuen-Bang Hou (Favonia), Evan Cavallo, Jesper Cockx, Christian Sattler, Chris Jeris, Michael Shulman *et al. Homotopy Type Theory in Agda*. DOI: `10.6084/m9.figshare.4763251` (cit. on p. 113).

[31] Ulrik Buchholtz. *Cellular complexes in Lean*. URL: `https://github.com/leanprover/lean2/blob/master/hott/homotopy/cellcomplex.hlean` (visited on 03/20/2017) (cit. on p. 74).

[32] Ulrik Buchholtz and Egbert Rijke. *The Cayley-Dickson Construction in Homotopy Type Theory*. 2016. arXiv: `1610.01134v1 [math.AT]` (cit. on p. 31).

[33] N. Burgoyne and Paul Fong. "The Schur multipliers of the Mathieu groups". In: *Nagoya Mathematical Journal* 27 (1966), pp. 733–745. ISSN: 0027-7630. URL: `http://www.ams.org/mathscinet-getitem?mr=197542` (cit. on p. 1).

[34] N. Burgoyne and Paul Fong. "A correction to: "The Schur multipliers of the Mathieu groups"". In: *Nagoya Mathematical Journal* 31 (1968), pp. 297–304. ISSN: 0027-7630. URL: `http://www.ams.org/mathscinet-getitem?mr=219626` (cit. on p. 1).

[35] Evan Cavallo. "Synthetic Cohomology in Homotopy Type Theory". MA thesis. 2015. URL: `http://www.cs.cmu.edu/~ecavallo/works/thesis.pdf` (visited on 12/12/2016) (cit. on pp. 31, 79).

[36] Yirng-An Chen, Edmund Clarke, Pei-Hsin Ho, Yatin Hoskote, Timothy Kam, Manpreet Khaira, John O'Leary and Xudong Zhao. "Verification of all circuits in a floating-point unit using word-level model checking". In: *Formal Methods in Computer-Aided Design*. (Palo Alto, CA, USA, Nov. 6–8, 1996). Vol. 1166. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, Nov. 1996, pp. 19–33. ISBN: 978-3-540-61937-6. DOI: `10.1007/BFb0031797` (cit. on p. 1).

[37] Jesper Cockx and Andreas Abel. *Sprinkles of extensionality for your vanilla type theory*. URL: `https://lirias.kuleuven.be/handle/123456789/544219` (visited on 12/18/2016) (cit. on p. 100).

[38] Jesper Cockx, Dominique Devriese and Frank Piessens. "Pattern matching without K". In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. (Gothenburg, Sweden, Sept. 1–3, 2014), pp. 257–268. ISBN: 978-1-4503-2873-9. DOI: `10.1145/2628136.2628139` (cit. on p. 98).

[39] Cyril Cohen, Thierry Coquand, Simon Huber and Anders Mörtberg. "Cubical type theory: a constructive interpretation of the univalence axiom". In: *21th International Conference on Types for Proofs and Programs*. Leibniz International Proceedings in Informatics. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016. arXiv: `1611.02108v1 [cs.LO]`. Forthcoming (cit. on p. 9).

[40] Cyril Cohen, Thierry Coquand, Simon Huber and Anders Mörtberg. *Implementation of Univalence in Cubical Sets*. URL: `https://github.com/simhu/cubical` (visited on 08/30/2015). Coded in HASKELL (cit. on p. 91).

[41] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki and S. F. Smith. *Implementing Mathematics with The Nuprl Proof Development System*. CreateSpace Independent Publishing Platform, 2012. ISBN: 978-1468059106 (cit. on p. 7).

[42] Thierry Coquand and Gérard Huet. "The calculus of constructions". In: *Information and Computation* 76.2 (1988), pp. 95–120. ISSN: 0890-5401. DOI: 10.1016/0890-5401(88)90005-3 (cit. on p. 91).

[43] Thierry Coquand and Christine Paulin. "Inductively Defined Types". In: *Proceedings of the International Conference on Computer Logic (COLOG)*. (Tallinn, USSR, Dec. 12–16, 1998). London, UK: Springer-Verlag, 1990, pp. 50–66. ISBN: 978-3-540-46963-6. DOI: 10.1007/3-540-52335-9_47 (cit. on p. 22).

[44] Thierry Coquand and Arnaud Spiwack. "Towards Constructive Homological Algebra in Type Theory". In: *Towards Mechanized Mathematical Assistants*. Mathematical Knowledge Management (MKM). (Hagenberg, Austria, June 27–30, 2007). Ed. by Manuel Kauers, Manfred Kerber, Robert Miner and Wolfgang Windsteiger. Berlin, Heidelberg: Springer, 2007, pp. 40–54. ISBN: 978-3-540-73086-6. DOI: 10.1007/978-3-540-73086-6_4 (cit. on pp. 6, 89).

[45] *Cubical Type Theory*. URL: https://github.com/mortberg/cubicaltt (visited on 11/29/2016) (cit. on p. 9).

[46] Floris van Doorn. *Constructing the Propositional Truncation using Non-recursive HITs*. 2015. arXiv: 1512.02274v1 [math.LO] (cit. on pp. 42, 88).

[47] Peter Dybjer. "Inductive families". In: *Formal Aspects of Computing* 6.4 (1994), pp. 440–465. ISSN: 1433-299X. DOI: 10.1007/BF01211308 (cit. on p. 22).

[48] Peter Dybjer. "Representing inductively defined sets by wellorderings in Martin-Löf's type theory". In: *Theoretical Computer Science* 176.1 (1997), pp. 329–335. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(96)00145-4 (cit. on p. 25).

[49] Peter Dybjer. "A general formulation of simultaneous inductive-recursive definitions in type theory". In: *The Journal of Symbolic Logic* 65.02 (2000), pp. 525–549. ISSN: 0022-4812. DOI: 10.2307/2586554 (cit. on pp. 22, 98).

[50] Samuel Eilenberg and Norman E. Steenrod. "Axiomatic Approach to Homology Theory". In: *Proceedings of the National Academy of Sciences of the United States of America* 31.4 (1945), pp. 117–120. ISSN: 0027-8424. URL: http://www.pnas.org/content/31/4/117.full.pdf (visited on 08/31/2015) (cit. on p. 79).

[51] Nicola Gambino and Richard Garner. "The identity type weak factorisation system". In: *Theoretical Computer Science* 409.1 (2008), pp. 94–109. DOI: 10.1016/j.tcs.2008.08.030 (cit. on p. 6).

[52] Richard Garner. "Two-dimensional Models of Type Theory". In: *Mathematical. Structures in Computer Science* 19.4 (Aug. 2009), pp. 687–736. ISSN: 0960-1295. DOI: 10.1017/S0960129509007646.

[53] Gaëtan Gilbert. "Formalising Real Numbers in Homotopy Type Theory". In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP)*. (Paris, France, 2017). New York, NY, USA: ACM, 2017, pp. 112–124. ISBN: 978-1-4503-4705-1. DOI: 10.1145/3018610.3018614.

[54] Jean-Yves Girard. “Interprétation Fonctionnelle et élimination des Coupures de l’Arithmétique d’Ordre Supérieur”. PhD thesis. Université Paris 7, 1972 (cit. on p. 12).

[55] Healfdene Goguen, Conor McBride and James McKinna. “Eliminating Dependent Pattern Matching”. In: *Algebra, Meaning, and Computation*. Essays dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday. Vol. 4060. Lecture Notes in Computer Science. Springer, 2006, pp. 521–540. DOI: 10.1007/11780274_27 (cit. on p. 98).

[56] Georges Gonthier. “A computer-checked proof of the four colour theorem”. 2005. URL: http://research.microsoft.com/en-us/um/people/gonthier/4colproof.pdf (cit. on p. 2).

[57] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O’ Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi and Laurent Théry. “A Machine-Checked Proof of the Odd Order Theorem”. In: *Interactive Theorem Proving*. Vol. 7998. Lecture Notes in Computer Science. Springer, 2013, pp. 163–179. ISBN: 978-3-642-39633-5. DOI: 10.1007/978-3-642-39634-2_14 (cit. on p. 2).

[58] Daniel R. Grayson. *An attempt at making a prototype proof checker for experimental purposes*. URL: https://github.com/DanGrayson/checker (visited on 11/29/2016) (cit. on p. 8).

[59] Alexander Grothendieck. “Pursuing Stacks”. 1983. URL: https://thescrivener.github.io/PursuingStacks/ (visited on 08/30/2015). Linked to a LaTeX version of the original typescript. Pre-published (cit. on p. 9).

[60] *Guarded Cubical Type Theory*. URL: https://github.com/hansbugge/cubicaltt/tree/gcubical (visited on 12/07/2016) (cit. on p. 9).

[61] Thomas Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu and Roland Zumkeller. *A formal proof of the Kepler conjecture*. 2015. arXiv: 1501.02155v1 [math.MG] (cit. on p. 2).

[62] Robert Harper and Robert Pollack. “Type checking with universes”. In: *Theoretical Computer Science* 89.1 (1991), pp. 107–136. ISSN: 0304-3975. DOI: 10.1016/0304-3975(90)90108-T (cit. on p. 98).

[63] John Harrison. “Floating point verification in HOL light: the exponential function”. In: *International Conference on Algebraic Methodology and Software Technology (AMAST)*. (Sydney, Australia, Dec. 13–17, 1997). Ed. by Michael Johnson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 246–260. ISBN: 978-3-540-69661-2. DOI: 10.1007/BFb0000475 (cit. on p. 1).

[64] John Harrison. “Formal verification of floating point trigonometric functions”. In: *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. (Austin, TX, USA, Nov. 1–3, 2000). Ed. by Warren A. Hunt and Steven D. Johnson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 254–270. ISBN: 978-3-540-40922-9. DOI: 10.1007/3-540-40922-X_14 (cit. on p. 1).

[65] Allen Hatcher. *Algebraic Topology*. Cambridge, UK: Cambridge University Press, 2002. ISBN: 978-0-521-79540-1. URL: http://www.math.cornell.edu/~hatcher/AT/ATpage.html. Page numbers are based on the online free version (cit. on p. 50).

[66] Michael Hedberg. “A coherence theorem for Martin-Löf’s type theory”. In: *Journal of Functional Programming* 8.04 (1998), pp. 413–436. ISSN: 0956-7968. DOI: 10.1017/S0956796898003153 (cit. on p. 19).

[67] Martin Hofmann and Thomas Streicher. “The groupoid interpretation of type theory”. In: *Twenty-five years of constructive type theory*. (Venice, 1995). Vol. 36. Oxford Logic Guides. Oxford, UK: Oxford University Press, 1998, pp. 83–111. ISBN: 978-0-19-850127-5. Google Books: pLnKggT_In4C (cit. on pp. 7, 18, 19).

[68] Pieter Hofstra and Michael A. Warren. *Combinatorial realizability models of type theory*. 2012. arXiv: 1205.5527v1 [math.LO].

[69] Kuen-Bang Hou (Favonia). “Covering spaces in homotopy type theory”. Submitted (cit. on pp. 31, 33, 49).

[70] Kuen-Bang Hou (Favonia). *Experiment prenex (ML-style) universe polymorphism with typical ambiguity and implicit cumulativity*. URL: https://github.com/agda/agda/issues/2043 (visited on 06/14/2016) (cit. on p. 98).

[71] Kuen-Bang Hou (Favonia), Eric Finster, Daniel R. Licata and Peter LeFanu Lumsdaine. “A Mechanization of the Blakers–Massey Connectivity Theorem in Homotopy Type Theory”. In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. (New York, NY, USA, July 5–8, 2016). New York, NY, USA: ACM, 2016, pp. 565–574. ISBN: 978-1-4503-4391-6. DOI: 10.1145/2933575.2934545 (cit. on pp. 31, 65).

[72] Kuen-Bang Hou (Favonia) and Michael Shulman. “The Seifert-van Kampen Theorem in Homotopy Type Theory”. In: *25th EACSL Annual Conference on Computer Science Logic (CSL)*. (Marseille, France, Aug. 29–Sept. 1, 2016). Ed. by Jean-Marc Talbot and Laurent Regnier. Vol. 62. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer In-

formatik, 2016, 22:1–22:16. ISBN: 978-3-95977-022-4. DOI: 10.4230/LIPIcs.CSL.2016.22 (cit. on pp. 31, 56, 57).

[73] Antonius J. C. Hurkens. “A simplification of Girard’s paradox”. In: *Typed Lambda Calculi and Applications*. Vol. 902. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1995, pp. 266–278. ISBN: 978-3-540-59048-4. DOI: 10.1007/BFb0014058 (cit. on p. 12).

[74] Valery Isaev. *Homotopy Type Theory with an interval type*. URL: https://valis.github.io/doc.pdf (visited on 03/17/2017) (cit. on p. 9).

[75] *ISO 9:1995. Information and documentation—Transliteration of Cyrillic characters into Latin characters—Slavic and non-Slavic languages*. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=3589 (visited on 01/08/2017) (cit. on p. 18).

[76] Zvonimir Janko. “A new finite simple group of order 86 · 775 · 571 · 046 · 077 · 562 · 880 which possesses M_{24} and the full covering group of M_{22} as subgroups”. In: *Journal of Algebra* 42.2 (1976), pp. 564–596. ISSN: 0021-8693. DOI: 10.1016/0021-8693(76)90115-0 (cit. on p. 1).

[77] Wolfram Kahl. *Slow typechecking of single one-line definition*. Aug. 12, 2015. URL: https://github.com/agda/agda/issues/1625 (cit. on p. 101).

[78] Krzysztof Kapulkin and Peter LeFanu Lumsdaine. *The simplicial model of univalent foundations*. After Voevodsky. 2012. arXiv: 1211.2851v4 [math.LO] (cit. on pp. 6, 11, 18).

[79] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch and Simon Winwood. “seL4: Formal Verification of an OS Kernel”. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*. (Big Sky, Montana, USA, Oct. 11–14, 2009). New York, NY, USA: ACM, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596 (cit. on p. 1).

[80] Nicolai Kraus. *The General Universal Property of the Propositional Truncation*. 2014. arXiv: 1411.2682v3 [math.LO] (cit. on p. 48).

[81] Nicolai Kraus. “Constructions with Non-Recursive Higher Inductive Types”. In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. (New York, NY, USA, July 5–8, 2016). New York, NY, USA: ACM, 2016, pp. 595–604. ISBN: 978-1-4503-4391-6. DOI: 10.1145/2933575.2933586 (cit. on pp. 42, 88).

[82] Nicolai Kraus, Martín Hötzel Escardó, Thierry Coquand and Thorsten Altenkirch. “Generalizations of Hedberg’s Theorem”. In: *International Conference on Typed Lambda Calculi and Applications (TLCA)*. (Eindhoven, The Netherlands, June 26–28, 2013). 2013, pp. 173–188. DOI: 10.1007/978-3-642-38946-7_14 (cit. on p. 48).

[83] Nicolai Kraus, Martín Hötzel Escardó, Thierry Coquand and Thorsten Altenkirch. *Notions of Anonymous Existence in Martin-Löf Type Theory*. 2016. arXiv: `1610.03346v1 [cs.LO]` (cit. on p. 48).

[84] Chin Soon Lee, Neil D. Jones and Amir M. Ben-Amram. "The Size-change Principle for Program Termination". In: *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. (London, UK, 2001). New York, NY, USA: ACM, 2001, pp. 81–92. ISBN: 978-1-58113-336-3. DOI: `10.1145/360204.360210` (cit. on p. 98).

[85] Xavier Leroy. *The CompCert C Verified Compiler*. Documentation and user' s manual. 2012. URL: `https://hal.inria.fr/hal-01091802/` (cit. on p. 1).

[86] Daniel R. Licata. *Running Circles Around (In) Your Proof Assistant; or, Quotients that Compute*. Apr. 23, 2011. URL: `https://homotopytypetheory.org/2011/04/23/running-circles-around-in-your-proof-assistant/` (visited on 08/30/2015) (cit. on p. 99).

[87] Daniel R. Licata *et al.* URL: `https://github.com/dlicata335/hott-agda/` (visited on 08/29/2015). Coded in AGDA (cit. on p. 6).

[88] Daniel R. Licata and Guillaume Brunerie. "$\pi_n(S^n)$ in Homotopy Type Theory". In: *Certified Programs and Proofs (CPP)*. (Melbourne, VIC, Australia, Dec. 11–13, 2013). Ed. by Georges Gonthier and Michael Norrish. Cham: Springer International Publishing, 2013, pp. 1–16. ISBN: 978-3-319-03545-1. DOI: `10.1007/978-3-319-03545-1_1` (cit. on pp. 31, 66).

[89] Daniel R. Licata and Guillaume Brunerie. "A Cubical Approach to Synthetic Homotopy Theory". In: *Proceedings of the 2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. (Kyoto, Japan, July 6–10, 2015). Washington, DC, USA: IEEE Computer Society, 2015, pp. 92–103. ISBN: 978-1-4799-8875-4. DOI: `10.1109/LICS.2015.19` (cit. on pp. 25, 26, 31).

[90] Daniel R. Licata and Guillaume Brunerie. *A Cubical Type Theory*. URL: `http://dlicata.web.wesleyan.edu/pubs/lb14cubical/lb14cubes-oxford.pdf` (visited on 11/29/2016) (cit. on p. 9).

[91] Daniel R. Licata and Eric Finster. "Eilenberg-MacLane Spaces in Homotopy Type Theory". In: *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. (Vienna, Austria, July 14–18, 2014). New York, NY, USA: ACM, 2014, 66:1–66:9. ISBN: 978-1-4503-2886-9. DOI: `10.1145/2603088.2603153` (cit. on pp. 31, 66).

[92] Daniel R. Licata and Robert Harper. "Canonicity for 2-dimensional Type Theory". In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. (Philadelphia, PA, USA, Jan. 22–28, 2012). New York, NY, USA: ACM, 2012, pp. 337–348. ISBN: 978-1-4503-1083-3. DOI: 10.1145/2103656.2103697 (cit. on p. 8).

[93] Daniel R. Licata and Michael Shulman. "Calculating the Fundamental Group of the Circle in Homotopy Type Theory". In: *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. (New Orleans, USA, June 25–28, 2013). Washington, DC, USA: IEEE Computer Society, 2013, pp. 223–232. ISBN: 978-0-7695-5020-6. DOI: 10.1109/LICS.2013.28 (cit. on pp. 29, 31, 54).

[94] Peter LeFanu Lumsdaine. "Weak ω-categories from intensional type theory". In: *Typed Lambda Calculi and Applications*. Vol. 5608. Lecture Notes in Computer Science. Springer, 2009, pp. 172–187. DOI: 10.1007/978-3-642-02273-9_14 (cit. on pp. 6, 16).

[95] Peter LeFanu Lumsdaine. *Higher Inductive Types: a tour of the menagerie*. Apr. 24, 2011. URL: https://homotopytypetheory.org/2011/04/24/higher-inductive-types-a-tour-of-the-menagerie/ (visited on 08/30/2015) (cit. on p. 11).

[96] Peter LeFanu Lumsdaine and Michael Shulman. "Semantics of higher inductive types". URL: https://ncatlab.org/homotopytypetheory/files/hit-semantics.pdf (visited on 12/04/2016). In preparation (cit. on p. 28).

[97] Zhaohui Luo and Healfdene Goguen. "Inductive data types: Well-ordering types revisited". In: *Logical Environments*. Cambridge, UK: Cambridge University Press, 1993, pp. 198–218. ISBN: 978-0-521-43312-9. URL: http://www.lfcs.inf.ed.ac.uk/reports/92/ECS-LFCS-92-209/ (cit. on p. 25).

[98] Donald MacKenzie. *Mechanizing proof: computing, risk, and trust*. MIT Press, 2004. ISBN: 978-0-262-63295-9 (cit. on p. 2).

[99] Per Martin-Löf. "Hauptsatz for the intuitionistic theory of iterated inductive definitions". In: *Proceedings of the 2nd Scandinavian logic symposium*. Ed. by Jan Erik Fenstad. Vol. 63. Studies in Logic and the Foundations of Mathematics. North-Holland Publishing Company, 1971, pp. 179–216. DOI: 10.1016/S0049-237X(08)70847-4 (cit. on p. 22).

[100] Per Martin-Löf. "An Intuitionistic Theory of Types: Predicative Part". In: *Logic Colloquium '73, Proceedings of the Logic Colloquium*. Vol. 80. Studies in Logic and the Foundations of Mathematics. Elsevier, 1975, pp. 73–118. DOI: 10.1016/S0049-237X(08)71945-1 (cit. on pp. 7, 12).

[101] Per Martin-Löf. “Constructive mathematics and computer programming”. In: *Logic, Methodology and Philosophy of Science VI, Proceedings of the Sixth International Congress of Logic, Methodology and Philosophy of Science*. Ed. by L. Jonathan Cohen, Jerzy Łoś, Helmut Pfeiffer and Klaus-Peter Podewski. Vol. 104. Studies in Logic and the Foundations of Mathematics. North-Holland Publishing Company, 1982, pp. 153–175. DOI: `10.1016/S0049-237X(09)70189-2` (cit. on p. 7).

[102] Per Martin-Löf. *Intuitionistic type theory*. Notes by Giovanni Sambin. Napoli, Italy: Bibliopolis, 1984. ISBN: 978-88-7088-105-9 (cit. on pp. 7, 22).

[103] Émile Mathieu. “Mémoire sur l'étude des fonctions de plusieurs quantités, sur la maniere de les former et sur les substitutions qui les laissent invariables”. In: *Journal de mathématiques pures et appliquées* 6 (1861), pp. 241–323 (cit. on p. 1).

[104] Émile Mathieu. “Sur la fonction cinq fois transitive de 24 quantités.” In: *Journal de mathématiques pures et appliquées* (1873), pp. 25–46 (cit. on p. 1).

[105] Conor McBride. “Dependently Typed Functional Programs and their Proofs”. PhD thesis. University of Edinburgh, 1999. URL: `https://www.era.lib.ed.ac.uk/bitstream/id/600/ECS-LFCS-00-419.pdf` (cit. on p. 98).

[106] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn and Jakob von Raumer. “The Lean Theorem Prover (System Description)”. In: *Automated Deduction - CADE-25*. Vol. 9195. Lecture Notes in Computer Science. Switzerland: Springer International Publishing, 2015, pp. 378–388. ISBN: 978-3-319-21401-6. DOI: `10.1007/978-3-319-21401-6_26` (cit. on pp. 6, 31, 51, 91).

[107] Amnon Neeman. “A counterexample to a 1961 “theorem” in homological algebra”. In: *Inventiones mathematicae* 148.2 (2002), pp. 397–420. ISSN: 1432-1297. DOI: `10.1007/s002220100197` (cit. on p. 1).

[108] Fredrik Nordvall Forsberg. “Inductive-inductive definitions”. PhD thesis. Swansea University, 2013. URL: `http://cs.swan.ac.uk/~csfnf/thesis/thesis.pdf` (cit. on pp. 22, 98).

[109] Ulf Norell. “Towards a practical programming language based on dependent type theory”. PhD thesis. Chalmers University of Technology, 2007. URL: `http://www.cse.chalmers.se/~ulfn/papers/thesis.html` (cit. on pp. 31, 91).

[110] Ian Orton and Andrew M. Pitts. “Axioms for Modelling Cubical Type Theory in a Topos”. In: *25th EACSL Annual Conference on Computer Science Logic (CSL)*. (Marseille, France, Aug. 29–Sept. 1, 2016). Ed. by Jean-Marc Talbot and Laurent Regnier. Vol. 62. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 24:1–24:19. ISBN: 978-3-95977-022-4. DOI: `10.4230/LIPIcs.CSL.2016.24` (cit. on p. 9).

[111] Christine Paulin-Mohring. “Inductive definitions in the system Coq rules and properties”. In: *Typed Lambda Calculi and Applications. International Conference on Typed Lambda Calculi and Applications (TLCA)*. (Utrech, The Netherlands, Mar. 16–18, 1993). Ed. by Marc Bezem and Jan Friso Groote. Berlin, Heidelberg: Springer, 1993, pp. 328–345. ISBN: 978-3-540-47586-6. DOI: `10.1007/BFb0037116` (cit. on pp. 17, 22).

[112] Álvaro Pelayo and Michael A. Warren. “Homotopy type theory and Voevodsky’ s univalent foundations”. In: *Bulletin of the American Mathematical Society* 51.4 (2014), pp. 597–648. ISSN: 0273-0979. DOI: `10.1090/S0273-0979-2014-01456-9` (cit. on p. 9).

[113] Michael A. Warren Peter LeFanu Lumsdaine. *The local universes model: an overlooked coherence construction for dependent type theories*. 2014. arXiv: `1411.1736v2 [math.LO]`.

[114] Frank Pfenning and Carsten Schürmann. “Algorithms for Equality and Unification in the Presence of Notational Definitions”. In: *Selected Papers from the International Workshop on Types for Proofs and Programs (TYPES)*. (1998). London, UK: Springer-Verlag, 1999, pp. 179–193. ISBN: 978-3-540-66537-3. URL: `http://dl.acm.org/citation.cfm?id=646538.696024` (cit. on p. 101).

[115] Jakob von Raumer. “Formalization of Non-Abelian Topology for Homotopy Type Theory”. MA thesis. Karlsruhe Institute of Technology, 2015. URL: `http://von-raumer.de/msc-thesis.pdf` (cit. on pp. 31, 89).

[116] Jason Reed. *Proof Irrelevance and Strict Definitions in a Logical Framework*. URL: `http://reports-archive.adm.cs.cmu.edu/anon/2002/CMU-CS-02-153.pdf` (visited on 12/19/2016) (cit. on p. 101).

[117] Charles Rezk. *Proof of the Blakers-Massey theorem*. 2015. URL: `http://www.math.uiuc.edu/~rezk/freudenthal-and-blakers-massey.pdf` (visited on 03/20/2017) (cit. on pp. 2, 32, 89).

[118] Egbert Rijke. “Homotopy Type Theory”. MA thesis. Utrecht University, 2012. URL: `http://hottheory.files.wordpress.com/2012/08/hott2.pdf` (visited on 12/07/2016) (cit. on p. 31).

[119] Egbert Rijke. *The join construction*. 2017. arXiv: `1701.07538v1 [math.CT]` (cit. on pp. 42, 88).

[120] Egbert Rijke and Bas Spitters. “Sets in homotopy type theory”. In: *Mathematical Structures in Computer Science* 25.5 (June 2015), pp. 1172–1202. DOI: `10.1017/S0960129514000553`.

[121] Jan-Erik Roos. “Sur les foncteurs dérivés de projlim. Applications.(French)”. In: *Comptes Rendus de l’Académie des Sciences* 252 (1961), pp. 3702–3704. URL: `http://urn.kb.se/resolve?urn=urn%3Anbn%3Ase%3Asu%3Adiva-13344` (cit. on p. 1).

[122] Jan-Erik Roos. "Derived functors of inverse limits revisited". In: *Journal of the London Mathematical Society* 73.1 (Feb. 2006), pp. 65–83. DOI: `10.1112/S0024610705022416` (cit. on p. 1).

[123] Julio Rubio and Francis Sergeraert. *Constructive Homological Algebra and Applications*. 2012. arXiv: `1208.3816v3 [math.KT]` (cit. on pp. 6, 89).

[124] David M. Russinoff. "A Mechanically Checked Proof of IEEE Compliance of the Floating Point Multiplication, Division and Square Root Algorithms of the AMD-K7™ Processor". In: *LMS Journal of Computation and Mathematics* 1 (1998), pp. 148–200. DOI: `10.1112/S1461157000000176` (cit. on p. 1).

[125] Urs Schreiber and Michael Shulman. "Quantum Gauge Field Theory in Cohesive Homotopy Type Theory". In: *Proceedings 9th Workshop on Quantum Physics and Logic*. 2014, pp. 109–126. arXiv: `1408.0054v1 [math-ph]` (cit. on p. 8).

[126] Michael Shulman. *Cohomology*. July 24, 2013. URL: `http://homotopytypetheory.org/2013/07/24/cohomology/` (visited on 08/29/2015) (cit. on pp. 79, 80).

[127] Michael Shulman. *Univalence for inverse EI diagrams*. 2015. arXiv: `1508.02410v2 [math.AT]`.

[128] Michael Shulman. *Brouwer's fixed-point theorem in real-cohesive homotopy type theory*. 2016. arXiv: `1509.07584v2 [math.CT]` (cit. on p. 8).

[129] Kristina Sojakova. "Higher Inductive Types as Homotopy-Initial Algebras". PhD thesis. Carnegie Mellon University, 2016. URL: `https://reports-archive.adm.cs.cmu.edu/anon/2016/CMU-CS-16-125.pdf` (cit. on pp. 25, 28).

[130] Matthieu Sozeau and Nicolas Tabareau. "Universe Polymorphism in Coq". In: *Interactive Theorem Proving: 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014. Proceedings.* (Vienna, Austria, July 14–17, 2014). Ed. by Gerwin Klein and Ruben Gamboa. Cham: Springer International Publishing, 2014, pp. 499–514. ISBN: 978-3-319-08970-6. DOI: `10.1007/978-3-319-08970-6_32` (cit. on p. 98).

[131] Jonathan Sterling, Danny Gratzer, David Christiansen, Darin Morrison, Eugene Akentyev and James Wilcox. *RedPRL – the People's Refinement Logic*. URL: `http://redprl.org/` (visited on 11/09/2016) (cit. on p. 9).

[132] Thomas Streicher. "Investigations Into Intentional Type Theory". 1993. URL: `http://www.mathematik.tu-darmstadt.de/~streicher/HabilStreicher.pdf`. Scanned version (cit. on p. 18).

[133] Sonam Thakchoe. "The Theory of Two Truths in India". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Spring 2017. Metaphysics Research Lab, Stanford University, 2017. URL: `https://plato.stanford.edu/entries/twotruths-india/` (visited on 01/27/2017) (cit. on p. 123).

[134] *The Coq Proof Assistant*. URL: `https://coq.inria.fr/` (cit. on pp. 31, 91).

[135] *The HoTT library*. URL: `https://github.com/HoTT/HoTT` (visited on 08/29/2015). Coded in Coq (cit. on pp. 6, 31).

[136] *The Kenzo program*. URL: `https://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/` (visited on 10/24/2015) (cit. on p. 6).

[137] *Univalent Foundations of Mathematics*. 2012. URL: `https://www.math.ias.edu/sp/univalent` (visited on 08/30/2015) (cit. on p. 2).

[138] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations for Mathematics*. Git commit hash g662cdd8. Institute for Advanced Study, 2013. URL: `http://homotopytypetheory.org/book` (cit. on pp. 6–8, 11, 16, 19, 21, 28, 29, 31, 33, 39, 41, 46, 51, 80).

[139] Vladimir Voevodsky. *A very short note on homotopy λ-calculus*. Sept. 2006. URL: `http://www.math.ias.edu/vladimir/files/2006_09_Hlambda.pdf` (visited on 08/30/2015) (cit. on pp. 11, 18).

[140] Vladimir Voevodsky. "Notes on type systems". 2009. URL: `http://www.math.ias.edu/vladimir/files/expressions_current.pdf` (visited on 08/30/2015). Pre-published (cit. on pp. 11, 18).

[141] Vladimir Voevodsky. *A simple type system with two identity types*. Feb. 25, 2013. URL: `https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/HTS.pdf` (visited on 11/29/2016) (cit. on pp. 8, 18).

[142] Vladimir Voevodsky. "Foundations of mathematics—their past, present and future". 2014. URL: `http://www.multimedia.ethz.ch/speakers/bernays/2014` (cit. on p. 9).

[143] Vladimir Voevodsky. "The Origins and Motivations of Univalent Foundations". In: *The Institute Letter* (Summer 2014), pp. 8–9. URL: `https://www.ias.edu/ias-letter/voevodsky-origins` (cit. on p. 10).

[144] Vladimir Voevodsky. *Univalent Foundations—new type theoretic foundations of mathematics*. Apr. 22, 2014. URL: `https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/2014_04_22_slides.pdf` (cit. on p. 18).

[145] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson *et al.* UniMath*: Univalent Mathematics*. URL: `https://github.com/UniMath/UniMath` (visited on 08/30/2015). Coded in Coq (cit. on p. 9).

[146] Michael A. Warren. "Homotopy theoretic aspects of constructive type theory". PhD thesis. Carnegie Mellon University, 2008. URL: `http://mawarren.net/papers/phd.pdf` (cit. on pp. 6, 11).

[147] Michael A. Warren. "The Strict ω-Groupoid Interpretation of Type Theory". In: *Models, Logics, and Higher-Dimensional Categories: A Tribute to the Work of Mihály Makkai*. Vol. 53. CRM Proceedings and Lecture Notes. American Mathematical Society and Centre de Recherches Mathématiques, 2011. ISBN: 978-0-8218-7281-9. Google Books: `-PZpEXuvvm4C`.

[148] Brent Abraham Yorgey. "Combinatorial Species and Finite Sets". PhD thesis. University of Pennsylvania, 2014. URL: `https://github.com/byorgey/thesis/blob/master/Yorgey-thesis-final-2014-11-17.pdf`.

[149] Gaoyong Zhang. "Intersection Bodies and the Busemann-Petty Inequalities in $\mathbb{R}^4$". In: *Annals of Mathematics* 140.2 (1994), pp. 331–346. ISSN: 0003-486X. DOI: `10.2307/2118603` (cit. on p. 1).

[150] Gaoyong Zhang. "A Positive Solution to the Busemann-Petty Problem in $\mathbb{R}^4$". In: *Annals of Mathematics* 149.2 (1999), pp. 535–543. ISSN: 0003-486X. DOI: `10.2307/120974` (cit. on p. 1).

[151] *ГОСТ 7.79-2000. Система стандартов по информации, библиотечному и издательскому делу. Правила транслитерации кирилловского письма латинским алфавитом.* URL: `http://gost.ruscable.ru/cgi-bin/catalog/catalog.cgi?i=6464` (дата обр. 08.01.2017) (цит. на с. 18).

[152] वज्रच्छेदिका प्रज्ञापारमिता. URL: `https://www2.hf.uio.no/polyglotta/index.php?page=volume&vid=22#permlink` (visited on 01/21/2017) (cit. on p. 123).

[153] 中華民國教育部. 臺灣閩南語羅馬字拼音方案使用手冊. URL: `http://language.moe.gov.tw/001/Upload/FileUpload/3677-15601/Documents/tshiutsheh.pdf` (visited on 02/15/2017) (cit. on p. 127).