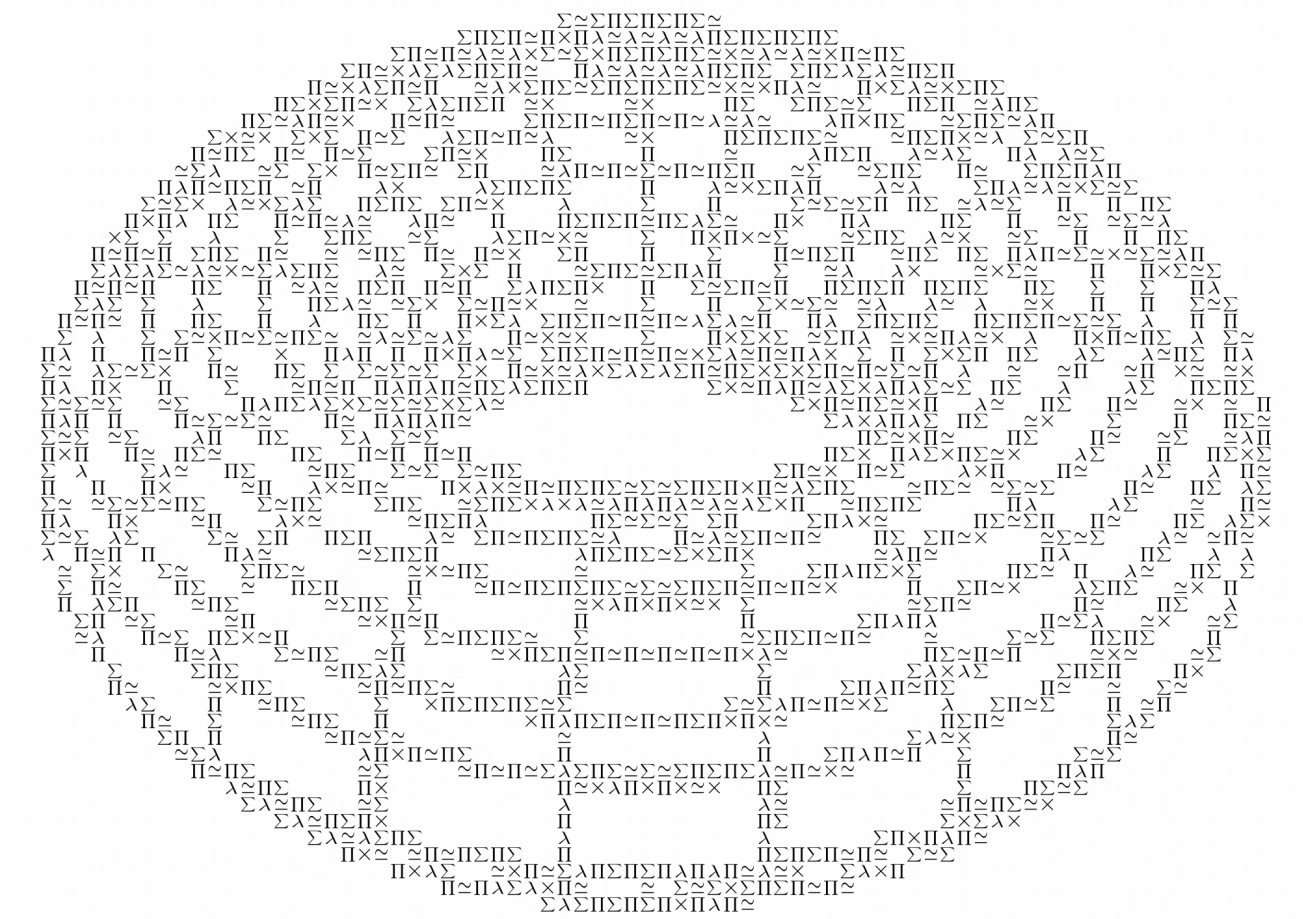
Homotopy Type Theory

Univalent Foundations of Mathematics



The Univalent Foundations Program

Institute for Advanced Study

Homotopy Type Theory

Univalent Foundations of Mathematics

The Univalent Foundations Program

Institute for Advanced Study

"Homotopy Type Theory: Univalent Foundations of Mathematics"

Acknowledgment

Apart from the generous support from the Institute for Advanced Study, some contributors to the book were partially or fully supported by the following agencies and grants:

- Association of Members of the Institute for Advanced Study: a grant to the Institute for Advanced Study
- Agencija za raziskovalno dejavnost Republike Slovenije: P1–0294, N1–0011.
- Air Force Office of Scientific Research: FA9550-11-1-0143, and FA9550-12-1-0370.

  This material is based in part upon work supported by the AFOSR under the above awards. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the AFOSR.

- Engineering and Physical Sciences Research Council: EP/G034109/1, EP/G03298X/1.
- European Union's 7th Framework Programme under grant agreement nr. 243847 (ForMath).
- National Science Foundation: DMS-1001191, CCF-1116703, and DMS-1128155.

  This material is based in part upon work supported by the National Science Foundation under the above awards. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

- The Simonyi Fund: a grant to the Institute for Advanced Study

Preface

About the IAS Special Year on Univalent Foundations of Mathematics

A Special Year on Univalent Foundations was held in 2012-13 at the Institute for Advanced Study, School of Mathematics, organized by Steve Awodey, Thierry Coquand, and Vladimir Voevodsky. The following people were the official participants.

Peter Aczel
Benedikt Ahrens
Thorsten Altenkirch
Steve Awodey
Bruno Barras
Andrej Bauer
Yves Bertot
Marc Bezem
Thierry Coquand
Eric Finster
Daniel Grayson
Hugo Herbelin
André Joyal
Dan Licata
Peter Lumsdaine
Assia Mahboubi
Per Martin-Löf
Sergey Melikhov
Alvaro Pelayo
Andrew Polonsky
Michael Shulman
Matthieu Sozeau
Bas Spitters
Benno van den Berg
Vladimir Voevodsky
Michael Warren
Noam Zeilberger

There were also the following students, whose participation was no less valuable.

Carlo Angiuli
Anthony Bordg
Guillaume Brunerie
Chris Kapulkin
Egbert Rijke
Kristina Sojakova

In addition, there were the following short- and long-term visitors, including student visitors, whose contributions to the Special Year were also essential.

Jeremy Avigad
Cyril Cohen
Robert Constable
Pierre-Louis Curien
Peter Dybjer
Martín Escardó
Kuen-Bang Hou
Nicola Gambino
Richard Garner
Georges Gonthier
Thomas Hales
Robert Harper
Martin Hofmann
Pieter Hofstra
Joachim Kock
Nicolai Kraus
Nuo Li
Zhaohui Luo
Michael Nahas
Erik Palmgren
Emily Riehl
Dana Scott
Philip Scott
Sergei Soloviev

About this book

We did not set out to write a book. The present work has its origins in our collective attempts to develop a new style of "informal type theory" that can be read and understood by a human being, as a complement to a formal proof that can be checked by a machine. Univalent Foundations is closely tied to the idea of a foundation of mathematics that can be implemented in a computer proof assistant. Although such a formalization is not part of this book, much of the material presented here was actually done first in the fully formalized setting inside a proof assistant, and only later "unformalized" to arrive at the presentation you find before you — a remarkable inversion of the usual state of affairs in formalized mathematics.

Each of the above-named individuals contributed something to the Special Year — and so to this book — in the form of ideas, words, or deeds. The spirit of collaboration that prevailed throughout the year was truly extraordinary.

Special thanks are due to the Institute for Advanced Study, without which this book would obviously never have come to be. It proved to be an ideal setting for the creation of this new branch of mathematics: stimulating, congenial, supportive, and profound. May some trace of this unique atmosphere linger in the pages of this book, and in the future development of this new field of study.

The Univalent Foundations Program
Institute for Advanced Study
Princeton, April 2013

Contents

Introduction **1**

I Foundations **15**

1 Type theory **17**
1.1 Type theory versus set theory . . . 17
1.2 Function types . . . 21
1.3 Universes and families . . . 23
1.4 Dependent function types (Π-types) . . . 24
1.5 Product types . . . 25
1.6 Dependent pair types (Σ-types) . . . 28
1.7 Coproduct types . . . 30
1.8 The type of booleans . . . 31
1.9 The natural numbers . . . 33
1.10 Pattern matching and recursion . . . 36
1.11 Propositions as types . . . 37
1.12 Identity types . . . 42
Notes . . . 48
Exercises . . . 50

2 Homotopy type theory **53**
2.1 Types are higher groupoids . . . 56
2.2 Functions are functors . . . 63
2.3 Type families are fibrations . . . 64
2.4 Homotopies and equivalences . . . 67
2.5 The higher groupoid structure of type formers . . . 70
2.6 Cartesian product types . . . 72
2.7 Σ-types . . . 74
2.8 The unit type . . . 76
2.9 Π-types and the function extensionality axiom . . . 77
2.10 Universes and the univalence axiom . . . 79
2.11 Identity type . . . 80

2.12 Coproducts . . . 82
2.13 Natural numbers . . . 85
2.14 Example: equality of structures . . . 87
2.15 Universal properties . . . 89
Notes . . . 91
Exercises . . . 93

3 Sets and logic **95**
3.1 Sets and n-types . . . 95
3.2 Propositions as types? . . . 97
3.3 Mere propositions . . . 99
3.4 Classical vs. intuitionistic logic . . . 101
3.5 Subsets and propositional resizing . . . 102
3.6 The logic of mere propositions . . . 103
3.7 Propositional truncation . . . 104
3.8 The axiom of choice . . . 106
3.9 The principle of unique choice . . . 107
3.10 When are propositions truncated? . . . 109
3.11 Contractibility . . . 111
Notes . . . 113
Exercises . . . 114

4 Equivalences **117**
4.1 Quasi-inverses . . . 118
4.2 Half adjoint equivalences . . . 120
4.3 Bi-invertible maps . . . 124
4.4 Contractible fibers . . . 124
4.5 On the definition of equivalences . . . 126
4.6 Surjections and embeddings . . . 126
4.7 Closure properties of equivalences . . . 127
4.8 The object classifier . . . 129
4.9 Univalence implies function extensionality . . . 132
Notes . . . 134
Exercises . . . 134

5 Induction **135**
5.1 Introduction to inductive types . . . 135
5.2 Uniqueness of inductive types . . . 138
5.3 W-types . . . 140
5.4 Inductive types are initial algebras . . . 143
5.5 Homotopy-inductive types . . . 146
5.6 The general syntax of inductive definitions . . . 150
5.7 Generalizations of inductive types . . . 154
5.8 Identity types and identity systems . . . 156

Notes . . . 160
Exercises . . . 160

6 Higher inductive types **163**
6.1 Introduction . . . 163
6.2 Induction principles and dependent paths . . . 165
6.3 The interval . . . 169
6.4 Circles and spheres . . . 170
6.5 Suspensions . . . 172
6.6 Cell complexes . . . 176
6.7 Hubs and spokes . . . 177
6.8 Pushouts . . . 179
6.9 Truncations . . . 182
6.10 Quotients . . . 185
6.11 Algebra . . . 189
6.12 The flattening lemma . . . 194
6.13 The general syntax of higher inductive definitions . . . 199
Notes . . . 201
Exercises . . . 202

7 Homotopy n-types **203**
7.1 Definition of n-types . . . 203
7.2 Uniqueness of identity proofs and Hedberg's theorem . . . 207
7.3 Truncations . . . 210
7.4 Colimits of n-types . . . 215
7.5 Connectedness . . . 219
7.6 Orthogonal factorization . . . 224
7.7 Modalities . . . 229
Notes . . . 233
Exercises . . . 234

II Mathematics **237**

8 Homotopy theory **239**
8.1 $\pi_1(S^1)$ . . . 243
8.2 Connectedness of suspensions . . . 251
8.3 $\pi_{k\leq n}$ of an n-connected space and $\pi_{k<n}(\mathbb{S}^n)$ . . . 252
8.4 Fiber sequences and the long exact sequence . . . 253
8.5 The Hopf fibration . . . 256
8.6 The Freudenthal suspension theorem . . . 262
8.7 The van Kampen theorem . . . 268
8.8 Whitehead's theorem and Whitehead's principle . . . 277
8.9 A general statement of the encode-decode method . . . 280

8.10 Additional Results . . . 282
Notes . . . 283
Exercises . . . 284

9 Category theory **285**
9.1 Categories and precategories . . . 286
9.2 Functors and transformations . . . 289
9.3 Adjunctions . . . 293
9.4 Equivalences . . . 294
9.5 The Yoneda lemma . . . 299
9.6 Strict categories . . . 303
9.7 †-categories . . . 303
9.8 The Structure identity principle . . . 304
9.9 The Rezk completion . . . 307
Notes . . . 315
Exercises . . . 316

10 Set theory **319**
10.1 The category of sets . . . 319
10.2 Cardinal numbers . . . 328
10.3 Ordinal numbers . . . 331
10.4 Classical well-orderings . . . 336
10.5 The cumulative hierarchy . . . 339
Notes . . . 344
Exercises . . . 344

11 Real numbers **347**
11.1 The field of rational numbers . . . 347
11.2 Dedekind reals . . . 348
11.3 Cauchy reals . . . 355
11.4 Comparison of Cauchy and Dedekind reals . . . 373
11.5 Compactness of the interval . . . 374
11.6 The surreal numbers . . . 380
Notes . . . 391
Exercises . . . 392

Appendix **395**

A Formal type theory **397**
A.1 The first presentation . . . 399
A.2 The second presentation . . . 403
A.3 Homotopy type theory . . . 409
A.4 Basic metatheory . . . 411

Notes 413

Bibliography **414**

Index of symbols **423**

Introduction

Homotopy type theory is a new branch of mathematics that combines aspects of several different fields in a surprising way. It is based on a recently discovered connection between *homotopy theory* and *type theory*. Homotopy theory is an outgrowth of algebraic topology and homological algebra, with relationships to higher category theory; while type theory is a branch of mathematical logic and theoretical computer science. Although the connections between the two are currently the focus of intense investigation, it is increasingly clear that they are just the beginning of a subject that will take more time and more hard work to fully understand. It touches on topics as seemingly distant as the homotopy groups of spheres, the algorithms for type checking, and the definition of weak ∞-groupoids.

Homotopy type theory also brings new ideas into the very foundation of mathematics. On the one hand, there is Voevodsky's subtle and beautiful *univalence axiom*. The univalence axiom implies, in particular, that isomorphic structures can be identified, a principle that mathematicians have been happily using on workdays, despite its incompatibility with the "official" doctrines of conventional foundations. On the other hand, we have *higher inductive types*, which provide direct, logical descriptions of some of the basic spaces and constructions of homotopy theory: spheres, cylinders, truncations, localizations, etc. Both ideas are impossible to capture directly in classical set-theoretic foundations, but when combined in homotopy type theory, they permit an entirely new kind of "logic of homotopy types".

This suggests a new conception of foundations of mathematics, with intrinsic homotopical content, an "invariant" conception of the objects of mathematics — and convenient machine implementations, which can serve as a practical aid to the working mathematician. This is the *Univalent Foundations* program. The present book is intended as a first systematic exposition of the basics of univalent foundations, and a collection of examples of this new style of reasoning — but without requiring the reader to know or learn any formal logic, or to use any computer proof assistant.

We emphasize that homotopy type theory is a young field, and univalent foundations is very much a work in progress. This book should be regarded as a "snapshot" of the state of the field at the time it was written, rather than a polished exposition of an established edifice. As we will discuss briefly later, there are many aspects of homotopy type theory that are not yet fully understood — but as of this writing, its broad outlines seem clear enough. The eventual theory will probably not look exactly like the one described in this book, but it will certainly be *at least* as capable and powerful; given the results presented here, we therefore believe that univalent foundations could eventually replace set theory as the "implicit foundation" for the

unformalized mathematics done by most mathematicians.

Type theory

Type theory was originally invented by Bertrand Russell [Rus08], as a device for blocking the paradoxes in the logical foundations of mathematics that were under investigation at the time. It was later developed as a rigorous formal system in its own right (under the name "λ-calculus") by Alonzo Church [Chu33, Chu40, Chu41]. Although it is not generally regarded as the foundation for classical mathematics, set theory being more customary, type theory still has numerous applications, especially in computer science and the theory of programming languages [Pie02]. Per Martin-Löf [ML98, ML75a, ML82, ML84], among others, developed a "predicative" modification of Church's type system, which is now usually called dependent, constructive, intuitionistic, or simply *Martin-Löf type theory*. This is the basis of the system that we consider here; it was originally intended as a rigorous framework for the formalization of constructive mathematics. In what follows, we will often use "type theory" to refer specifically to this system and similar ones, although type theory as a subject is much broader (see [Som10, KLN04] for the history of type theory).

In type theory, unlike set theory, objects are classified using a primitive notion of *type*, similar to the data-types used in programming languages. These elaborately structured types can be used to express detailed specifications of the objects classified, giving rise to principles of reasoning about these objects. To take a very simple example, the objects of a product type $A \times B$ are known to be of the form (a, b), and so one automatically knows how to construct them and how to decompose them. Similarly, an object of function type $A \to B$ can be acquired from an object of type B parametrized by objects of type A, and can be evaluated at an argument of type A. This rigidly predictable behavior of all objects (as opposed to set theory's more liberal formation principles, allowing inhomogeneous sets) is one aspect of type theory that has led to its extensive use in verifying the correctness of computer programs. The clear reasoning principles associated with the construction of types also form the basis of modern *computer proof assistants*, which are used for formalizing mathematics and verifying the correctness of formalized proofs. We return to this aspect of type theory below.

One problem in understanding type theory from a mathematical point of view, however, has always been that the basic concept of *type* is unlike that of *set* in ways that have been hard to make precise, especially the notion of equality of elements of a type. This difficulty, we believe, has now been largely overcome by the idea of regarding types, not as strange sets (perhaps constructed without using classical logic), but as spaces, viewed from the perspective of homotopy theory.

In homotopy theory one is concerned with spaces and continuous mappings between them, up to homotopy. A *homotopy* between a pair of continuous maps $f\colon X \to Y$ and $g\colon X \to Y$ is a continuous map $H\colon X \times [0,1] \to Y$ satisfying $H(x,0) = f(x)$ and $H(x,1) = g(x)$. The homotopy H may be thought of as a "continuous deformation" of f into g. The spaces X and Y are said to be *homotopy equivalent*, $X \simeq Y$, if there are continuous maps going back and forth, the composites of which are homotopical to the respective identity mappings, i.e., if they are isomorphic "up to homotopy". Homotopy equivalent spaces have the same algebraic invariants (e.g., homology, or the fundamental group), and are said to have the same *homotopy type*.

Homotopy type theory

Homotopy type theory interprets type theory from a homotopical perspective. In homotopy type theory, we regard the types as "spaces" (as studied in homotopy theory) or higher groupoids, and the logical constructions (such as the product $A \times B$) as homotopy-invariant constructions on these spaces. In this way, we are able to manipulate spaces directly without first having to develop point-set topology (or any combinatorial replacement for it, such as the theory of simplicial sets). To briefly explain this perspective, consider first the basic concept of type theory, namely that the *term* a is of *type* A, which is written:

$$a : A.$$

This expression is traditionally thought of as akin to:

"a is an element of the set A."

However, in homotopy type theory we think of it instead as:

"a is a point of the space A."

Similarly, every term $f : A \to B$ in type theory is regarded as a continuous function from the space A to the space B.

We should stress that these "spaces" are treated purely homotopically, not topologically. For instance, there is no notion of "open subset" of a type or of "convergence" of a sequence of elements of a type. We only have "homotopical" notions, such as paths between points and homotopies betwen paths, which also make sense in other models of homotopy theory (such as simplicial sets). Thus, it would be more accurate to say that we treat types as ∞-*groupoids*; this is a name for the "invariant objects" of homotopy theory which can be presented by topological spaces, simplicial sets, or any other model for homotopy theory. However, it is convenient to sometimes use topological words such as "space" and "path", as long as we remember that other topological concepts are not applicable.

(It is tempting to also use the phrase *homotopy type* for these objects, suggesting the dual interpretation of "a type (as in type theory) viewed homotopically" and "a space considered from the point of view of homotopy theory." The latter is a bit different from the classical meaning of "homotopy type" as an *equivalence class* of spaces modulo homotopy equivalence, although it does preserve the meaning of phrases such as "these two spaces have the same homotopy type".)

The idea of interpreting types as structured objects, rather than sets, has a long pedigree, and is known to clarify various mysterious aspects of type theory. For instance, interpreting types as sheaves helps explain the intuitionistic nature of type theoretic logic, while interpreting them as PERs or "domains" helps explain its computational aspects. It also implies that we can use type-theoretic reasoning to study the structured objects, leading to the rich field of categorical logic. The homotopical interpretation fits this same pattern: it clarifies the nature of *identity* (or equality) in type theory, and allows us to use type-theoretic reasoning in the study of homotopy theory.

The key new idea of the homotopy interpretation is that the logical notion of identity $a = b$ of two objects $a, b : A$ of the same type A can be understood as the existence of a path $p : a \rightsquigarrow b$ from

point a to point b in the space A. This also means that two functions $f, g : A \to B$ can be identified if they are homotopic, since a homotopy is just a (continuous) family of paths $p_x : f(x) \rightsquigarrow g(x)$ in B, one for each $x : A$. In type theory, for every type A there is a (formerly somewhat mysterious) type Id_A of identifications of two objects of A; in homotopy type theory, this is just the *path space* A^I of all continuous maps $I \to A$ from the unit interval. In this way, a term $p : \mathsf{Id}_A(a, b)$ represents a path $p : a \rightsquigarrow b$ in A.

The idea of homotopy type theory arose around 2006 in independent work by Awodey and Warren [AW09] and Voevodsky [Voe06], but it was inspired by Hofmann and Streicher's earlier groupoid interpretation [HS98]. Indeed, higher-dimensional category theory (particularly the theory of weak ∞-groupoids) is now known to be intimately connected to homotopy theory, as proposed by Grothendieck and now being studied intensely by mathematicians of both sorts. The original semantic models of Awodey–Warren and Voevodsky use well-known notions and techniques from homotopy theory which are now also in use in higher category theory, such as Quillen model categories and Kan simplicial sets.

Voevodsky recognized that the simplicial interpretation of type theory satisfies a further crucial property, dubbed *univalence*, which had not previously been considered in type theory (although Church's principle of extensionality for propositions turns out to be a very special case of it). Adding univalence to type theory in the form of a new axiom has far-reaching consequences, many of which are natural, simplifying and compelling. The univalence axiom also further strengthens the homotopical view of type theory, since it holds in the simplicial model and other related models, while failing under the view of types as sets.

Univalent foundations

Very briefly, the basic idea of the univalence axiom can be explained as follows. In type theory, one can have a universe $\mathcal{U}$, the terms of which are themselves types, $A : \mathcal{U}$, etc. Those types that are terms of $\mathcal{U}$ are commonly called *small* types. Like any type, $\mathcal{U}$ has an identity type $\mathsf{Id}_{\mathcal{U}}$, which expresses the identity relation $A = B$ between small types. Thinking of types as spaces, $\mathcal{U}$ is a space, the points of which are spaces; to understand its identity type, we must ask, what is a path $p : A \rightsquigarrow B$ between spaces in $\mathcal{U}$? The univalence axiom says that such paths correspond to homotopy equivalences $A \simeq B$, (roughly) as explained above. A bit more precisely, given any (small) types A and B, in addition to the primitive type $\mathsf{Id}_{\mathcal{U}}(A, B)$ of identifications of A with B, there is the defined type $\mathsf{Equiv}(A, B)$ of equivalences from A to B. Since the identity map on any object is an equivalence, there is a canonical map,

$$\mathsf{Id}_{\mathcal{U}}(A, B) \to \mathsf{Equiv}(A, B).$$

The univalence axiom states that this map is itself an equivalence. At the risk of oversimplifying, we can state this succinctly as follows:

Univalence Axiom: $(A = B) \simeq (A \simeq B)$.

In other words, identity is equivalent to equivalence. In particular, one may say that "equivalent types are identical". However, this phrase is somewhat misleading, since it may sound like a sort of "skeletality" condition which *collapses* the notion of equivalence to coincide with identity,

whereas in fact univalence is about *expanding* the notion of identity so as to coincide with the (unchanged) notion of equivalence.

From the homotopical point of view, univalence implies that spaces of the same homotopy type are connected by a path in the universe $\mathcal{U}$, in accord with the intuition of a classifying space for (small) spaces. From the logical point of view, however, it is a radically new idea: it says that isomorphic things can be identified! Mathematicians are of course used to identifying isomorphic structures in practice, but they generally do so by "abuse of notation", or some other informal device, knowing that the objects involved are not "really" identical. But in this new foundational scheme, such structures can be formally identified, in the logical sense that every property or construction involving one also applies to the other. Indeed, the identification is now made explicit, and properties and constructions can be systematically transported along it. Moreover, the different ways in which such identifications may be made themselves form a structure that one can (and should!) take into account.

Thus in sum, for points A and B of the universe $\mathcal{U}$ (i.e., small types), the univalence axiom identifies the following three notions:

- (logical) an identification $p : A = B$ of A and B
- (topological) a path $p : A \rightsquigarrow B$ from A to B in $\mathcal{U}$
- (homotopical) an equivalence $p : A \simeq B$ between A and B.

Higher inductive types

One of the classical advantages of type theory is its simple and effective techniques for working with inductively defined structures. The simplest nontrivial inductively defined structure is the natural numbers, which is inductively generated by zero and the successor function. From this statement one can algorithmically extract the principle of mathematical induction, which characterizes the natural numbers. More general inductive definitions encompass lists and well-founded trees of all sorts, each of which is characterized by a corresponding "induction principle". This includes most data structures used in certain programming languages; hence the usefulness of type theory in formal reasoning about the latter. If conceived in a very general sense, inductive definitions also include examples such as a disjoint union $A + B$, which may be regarded as "inductively" generated by the two injections $A \to A + B$ and $B \to A + B$. The "induction principle" in this case is "proof by case analysis", which characterizes the disjoint union.

In homotopy theory, it is natural to consider also "inductively defined spaces" which are generated not merely by a collection of *points*, but also by collections of *paths* and higher paths. Classically, such spaces are called *CW complexes*. For instance, the circle S^1 is generated by a single point and a single path from that point to itself. Similarly, the 2-sphere S^2 is generated by a single point b and a single two-dimensional path from the constant path at b to itself, while the torus T^2 is generated by a single point, two paths p and q from that point to itself, and a two-dimensional path from $p \cdot q$ to $q \cdot p$.

By using the identification of paths with identities in homotopy type theory, these sort of "inductively defined spaces" can be characterized in type theory by "induction principles", entirely analogously to classical examples such as the natural numbers and the disjoint union. The

resulting *higher inductive types* give a direct "logical" way to reason about familiar spaces such as spheres, which (in combination with univalence) can be used to perform familiar arguments from homotopy theory, such as calculating homotopy groups of spheres, in a purely formal way. The resulting proofs are a marriage of classical homotopy-theoretic ideas with classical type-theoretic ones, yielding new insight into both disciplines.

Moreover, this is only the tip of the iceberg: many abstract constructions from homotopy theory, such as homotopy colimits, suspensions, Postnikov towers, localization, completion, and spectrification, can also be expressed as higher inductive types. Many of these are classically constructed using Quillen's "small object argument", which can be regarded as a finite way of algorithmically describing an infinite CW complex presentation of a space, just as "zero and successor" is a finite algorithmic description of the infinite set of natural numbers. Spaces produced by the small object argument are infamously complicated and difficult to understand; the type-theoretic approach is potentially much simpler, bypassing the need for any explicit construction by giving direct access to the appropriate "induction principle". Thus, the combination of univalence and higher inductive types suggests the possibility of a revolution, of sorts, in the practice of homotopy theory.

Sets in univalent foundations

We have claimed that univalent foundations can eventually serve as a foundation for "all" of mathematics, but so far we have discussed only homotopy theory. Of course, there are many specific examples of the use of type theory without the new homotopy type theory features to formalize mathematics, such as the recent formalization of the Feit-Thompson odd-order theorem in Coq [Geo13].

But the traditional view is that mathematics is founded on set theory, in the sense that all mathematical objects and constructions can be coded into a theory such as Zermelo–Fraenkel set theory. However, it is well-established by now that for most mathematics outside of set theory proper, the intricate hierarchical membership structure of sets in ZF is really unnecessary: a more "structural" theory, such as Lawvere's Elementary Theory of the Category of Sets [Law05], suffices.

In univalent foundations, the basic objects are "homotopy types" rather than sets, but we can *define* a class of types which behave like sets. Homotopically, these can be thought of as spaces in which every connected component is contractible. It is a theorem that the category of such "sets" satisfies Lawvere's axioms (or related ones, depending on the details of the theory; see Theorem 10.1.15). Thus, any sort of mathematics that can be represented in an ETCS-like theory (which, experience suggests, is essentially all of mathematics) can equally well be represented in univalent foundations.

This supports the claim that univalent foundations is at least as good as existing foundations of mathematics. A mathematician working in univalent foundations can build structures out of sets in a familiar way, with more general homotopy types waiting in the foundational background until there is need of them. For this reason, most of the applications in this book have been chosen to be areas where univalent foundations has something *new* to contribute that distinguishes it from existing foundational systems.

Unsurprisingly, homotopy theory and category theory are two of these, but perhaps less obvious is that univalent foundations has something new and interesting to offer even in subjects such as set theory and real analysis. For instance, the univalence axiom allows us to identify isomorphic structures, while higher inductive types allow direct descriptions of objects by their universal properties. Thus we can generally avoid resorting to arbitrarily chosen representatives or transfinite iterative constructions. In fact, even the objects of study in ZF set theory can be characterized, inside the sets of univalent foundations, by such an inductive universal property.

Informal type theory

One difficulty often encountered by the classical mathematician when faced with learning about type theory is that it is usually presented as a fully or partially formalized deductive system. This style, which is very useful for proof-theoretic investigations, is not particularly convenient for use in applied, informal reasoning. Nor is it even familiar to most working mathematicians, even those who might be interested in foundations of mathematics. One objective of the present work is to develop an informal style of doing mathematics in univalent foundations that is at once rigorous and precise, but is also closer to the language and style of presentation of everyday mathematics.

In present-day mathematics, one usually constructs and reasons about mathematical objects in a way that could in principle, one presumes, be formalized in a system of elementary set theory, such as ZFC — at least given enough ingenuity and patience. For the most part, one does not even need to be aware of this possibility, since it largely coincides with the condition that a proof be "fully rigorous" (in the sense that all mathematicians have come to understand intuitively through education and experience). But one does need to learn to be careful about a few aspects of "informal set theory": the use of collections too large or inchoate to be sets; the axiom of choice and its equivalents; even (for undergraduates) the method of proof by contradiction; and so on. Adopting a new foundational system such as homotopy type theory as the *implicit formal basis* of informal reasoning will require adjusting some of one's instincts and practices. The present text is intended to serve as an example of this "new kind of mathematics", which is still informal, but could now in principle be formalized in homotopy type theory, rather than ZFC, again given enough ingenuity and patience.

It is worth emphasizing that, in this new system, such formalization can have real practical benefits. The formal system of type theory is suited to computer systems and has been implemented in existing proof assistants. A proof assistant is a computer program which guides the user in construction of a fully formal proof, only allowing valid steps of reasoning. It also provides some degree of automation, can search libraries for existing theorems, and can even extract numerical algorithms from the resulting (constructive) proofs.

We believe that this aspect of the univalent foundations program distinguishes it from other approaches to foundations, potentially providing a new practical utility for the working mathematician. Indeed, proof assistants based on older type theories have already been used to formalize substantial mathematical proofs, such as the four-color theorem and the Feit–Thompson theorem. Computer implementations of univalent foundations are presently works in progress (like the theory itself). However, even its currently available implementations (which are mostly small modifications to existing proof assistants such as Coq and Agda) have already demon-

strated their worth, not only in the formalization of known proofs, but in the discovery of new ones. Indeed, many of the proofs described in this book were actually *first* done in a fully formalized form in a proof assistant, and are only now being "unformalized" for the first time — a reversal of the usual relation between formal and informal mathematics.

One can imagine a not-too-distant future when it will be possible for mathematicians to verify the correctness of their own papers by working within the system of univalent foundations, formalized in a proof assistant, and that doing so will become as natural as typesetting their own papers in TEX. (Whether this proves to be the publishers' dream or their nightmare remains to be seen.) In principle, this could be equally true for any other foundational system, but we believe it to be more practically attainable using univalent foundations, as witnessed by the present work and its formal counterpart.

Constructivity

One of the most striking differences between classical foundations and type theory is the idea of *proof relevance*, according to which mathematical statements, and even their proofs, become first-class mathematical objects. In type theory, we represent mathematical statements by types, which can be regarded simultaneously as both mathematical constructions and mathematical assertions, a conception also known as *propositions as types*. Accordingly, we can regard a term $a : A$ as both an element of the type A (or in homotopy type theory, a point of the space A), and at the same time, a proof of the proposition A. To take an example, suppose we have sets A and B (discrete spaces), and consider the statement "A is isomorphic to B." In type theory, this can be rendered as:

$$\mathsf{Iso}(A,B) :\equiv \sum_{(f:A\to B)}\sum_{(g:B\to A)}\Big(\big(\prod_{(x:A)} g(f(x)) = x\big) \times \big(\prod_{(y:B)} f(g(y)) = y\big)\Big)$$

Reading the type constructors $\Sigma, \Pi, \times$ here as "there exists", "for all", and "and" respectively yields the usual formulation of "A and B are isomorphic"; on the other hand, reading them as sums and products yields the *type of all isomorphisms* between A and B! To prove that A and B are isomorphic, one constructs a proof $p : \mathsf{Iso}(A,B)$, which is therefore the same as constructing an isomorphism between A and B, i.e., exhibiting a pair of functions f, g together with *proofs* that their composites are the respective identity maps. The latter proofs, in turn, are nothing but homotopies of the appropriate sorts. In this way, *proving a proposition is the same as constructing an element of some particular type.*

In particular, to prove a statement of the form "A and B" is just to prove A and to prove B, i.e., to give an element of the type $A \times B$. And to prove that A implies B is just to find an element of $A \to B$, i.e. a function from A to B (determining a mapping of proofs of A to proofs of B). This "constructive" conception (for more on which, see [Kom32, TvD88a, TvD88b]) is what gives type theory its good computational character. For instance, every proof that something exists carries with it enough information to actually find such an object; and from a proof that "A or B" holds, one can extract either a proof that A holds or one that B holds. Thus, from every proof we can automatically extract an algorithm; this can be very useful in applications to computer programming.

However, this conception of logic does behave in ways that are unfamiliar to most mathematicians. On one hand, a naive translation of the *axiom of choice* yields a statement that we

can simply prove. Essentially, this notion of "there exists" is strong enough to ensure that, by showing that for every $x : A$ there exists a $y : B$ such that $R(x, y)$, we automatically construct a function $f : A \to B$ such that, for all $x : A$, we have $R(x, f(x))$.

On the other hand, this notion of "or" is so strong that a naive translation of the *law of excluded middle* is inconsistent with the univalence axiom. For if we assume "for all A, either A or not A", then since proving "A" means exhibiting an element of it, we would have a uniform way of selecting an element from every nonempty type — a sort of Hilbertian choice operator. However, univalence implies that the element of A selected by such a choice operator must be invariant under all self-equivalences of A, since these are identified with self-identities and every operation must respect identity. But clearly some types have automorphisms with no fixed points, e.g. we can swap the elements of a two-element type.

Thus, the logic of "proposition as types" suggested by traditional type theory is not the "classical" logic familiar to most mathematicians. But it is also different from the logic sometimes called "intuitionistic", which may lack *both* the law of excluded middle and the axiom of choice. For present purposes, it may be called *constructive logic* (but one should be aware that the terms "intuitionistic" and "constructive" are often used differently).

The computational advantages of constructive logic imply that we should not discard it lightly; but for some purposes in classical mathematics, its non-classical character can be problematic. Many mathematicians are, of course, accustomed to rely on the law of excluded middle; while the "axiom of choice" that is available in constructive logic looks superficially similar to its classical namesake, but does not have all of its strong consequences. Fortunately, homotopy type theory gives a finer analysis of this situation, allowing various different kinds of logic to coexist and intermix.

The new insight that makes this possible is that the system of all types, just like spaces in classical homotopy theory, is "stratified" according to the dimensions in which their higher homotopy structure exists or collapses. In particular, Voevodsky has found a purely type-theoretic definition of *homotopy n-types*, corresponding to spaces with no nontrivial homotopy information above dimension n. (The 0-types are the "sets" mentioned previously as satisfying Lawvere's axioms.) Moreover, with higher inductive types, we can universally "truncate" a type into an n-type; in classical homotopy theory this would be its n^{th} Postnikov section.

With these notions in hand, the homotopy (-1)-types, which we call *(mere) propositions*, support a logic that is much more like traditional "intuitionistic" logic. (Classically, every (-1)-type is empty or contractible; we interpret these possibilities as the truth values "false" and "true" respectively.) The "(-1)-truncated axiom of choice" is not automatically true, but is a strong assumption with the same sorts of consequences as its counterpart in classical set theory. Similarly, the "(-1)-truncated law of excluded middle" may be assumed, with many of the same consequences as in classical mathematics. Thus, the homotopical perspective reveals that classical and constructive logic can coexist, as endpoints of a spectrum of different systems, with an infinite number of possibilities in between (the homotopy n-types for $-1 < n < \infty$). We may speak of "LEM_n" and "AC_n", with AC_∞ being provable and LEM_∞ inconsistent with univalence, while AC_{-1} and LEM_{-1} are the versions familiar to classical mathematicians (hence in most cases it is appropriate to assume the subscript (-1) when none is given). Indeed, one can even have useful systems in which only *certain* types satisfy such further "classical" principles, while types

in general remain "constructive."

It is worth emphasizing that univalent foundations does not *require* the use of constructive or intuitionistic logic. Most of classical mathematics which depends on the law of excluded middle and the axiom of choice can be performed in univalent foundations, simply by assuming that these two principles hold (in their proper, (-1)-truncated, form). However, type theory does encourage avoiding these principles when they are unnecessary, for several reasons.

First of all, every mathematician knows that a theorem is more powerful when proven using fewer assumptions, since it applies to more examples. The situation with AC and LEM is no different: type theory admits many interesting "nonstandard" models, such as in sheaf toposes, where classicality principles such as AC and LEM tend to fail. Homotopy type theory admits similar models in higher toposes, such as are studied in [TV02, Rez05, Lur09]. Thus, if we avoid using these principles, the theorems we prove will be valid internally to all such models.

Secondly, one of the additional virtues of type theory is its computable character. In addition to being a foundation for mathematics, type theory is a formal theory of computation, and can be treated as a powerful programming language. From this perspective, the rules of the system cannot be chosen arbitrarily the way set-theoretic axioms can: there must be a harmony between them which allows all proofs to be "executed" as programs. We do not yet fully understand the new principles introduced by homotopy type theory, such as univalence and higher inductive types, from this point of view, but the basic outlines are emerging; see, for example, §§2.5–2.13 and [LH12]. It has been known for a long time, however, that principles such as AC and LEM are fundamentally antithetical to computability, since they assert baldly that certain things exist without giving any way to compute them. Thus, avoiding them is necessary to maintain the character of type theory as a theory of computation.

Fortunately, constructive reasoning is not as hard as it may seem. In some cases, simply by rephrasing some definitions, a theorem can be made constructive and its proof more elegant. Moreover, in univalent foundations this seems to happen more often. For instance:

(i) In set-theoretic foundations, at various points in homotopy theory and category theory one needs the axiom of choice to perform transfinite constructions. But with higher inductive types, we can encode these constructions directly and constructively. In particular, none of the "synthetic" homotopy theory in Chapter 8 requires LEM or AC.

(ii) In set-theoretic foundations, the statement "every fully faithful and essentially surjective functor is an equivalence of categories" is equivalent to the axiom of choice. But with the univalence axiom, it is just *true*; see Chapter 9.

(iii) In set theory, various circumlocutions are required to obtain notions of "cardinal number" and "ordinal number" which canonically represent isomorphism classes of sets and well-ordered sets, respectively — possibly involving the axiom of choice or the axiom of foundation. But with univalence and higher inductive types, we can obtain such representatives directly by truncating the universe; see Chapter 10.

(iv) In set-theoretic foundations, the definition of the real numbers as equivalence classes of Cauchy sequences requires either the law of excluded middle or the axiom of (countable) choice to be well-behaved. But with higher inductive types, we can give a version of this definition which is well-behaved and avoids any choice principles; see Chapter 11.

Of course, these simplifications could as well be taken as evidence that the new methods will not, ultimately, prove to be really constructive. However, we emphasize again that the reader does not have to care, or worry, about constructivity in order to read this book. The point is that in all of the above examples, the version of the theory we give has independent advantages, whether or not LEM and AC are assumed to be available. Constructivity, if attained, will be an added bonus.

Given this discussion of adding new principles such as univalence, higher inductive types, AC, and LEM, one may wonder whether the resulting system remains consistent. (One of the original virtues of type theory, relative to set theory, was that it can be seen to be consistent by proof-theoretic means). As with any foundational system, consistency is a relative question: "consistent with respect to what?" The short answer is that all of the constructions and axioms considered in this book have a model in the category of Kan complexes, due to Voevodsky [KLV12] (see [LS13b] for higher inductive types). Thus, they are known to be formally consistent relative to ZFC (with as many inaccessible cardinals as we need nested univalent universes). Giving a more traditionally type-theoretic account of this consistency is work in progress (see, e.g., [LH12, BCH13]).

We summarize the different points of view of the type-theoretic operations in Table 1.

Type Theory	Logic	Set Theory	Homotopy Type Theory
A	proposition	set	space
$a : A$	proof	element	point
$B(x)$	predicate	family of sets	fibration
$b(x) : B(x)$	conditional proof	family of elements	section
$\mathbf{0}, \mathbf{1}$	$\bot, \top$	$\emptyset, \{\emptyset\}$	$\emptyset, *$
$A + B$	$A \vee B$	disjoint union	coproduct
$A \times B$	$A \wedge B$	set of pairs	product space
$A \to B$	$A \Rightarrow B$	set of functions	function space
$\sum_{(x:A)} B(x)$	$\exists_{x:A} B(x)$	disjoint sum	total space
$\prod_{(x:A)} B(x)$	$\forall_{x:A} B(x)$	product	space of sections
Id_A	x = y	$\{(x, x) \mid x \in A\}$	path space A^I

Table 1: Comparing points of view on type-theoretic operations

Open problems

For those interested in contributing to this new branch of mathematics, it may be encouraging to know that there are many interesting open questions.

Perhaps the most pressing of them is the "constructivity" of the Univalence Axiom, posed by Voevodsky in [Voe12]. The basic system of type theory follows the structure of Gentzen's natural deduction. Logical connectives are defined by their introduction rules, and have elimination rules justified by computation rules. Following this pattern, and using Tait's computability

method, originally designed to analyse Gödel's Dialectica interpretation, one can show the property of *normalization* for type theory. This in turn implies important properties such as decidability of type-checking (a crucial property since type-checking corresponds to proof-checking, and one can argue that we should be able to "recognize a proof when we see one"), and the so-called "canonicity property" that any closed term of the type of natural numbers reduces to a numeral. This last property, and the uniform structure of introduction/elimination for rules, are lost when one extends type theory with an axiom, such as the axiom of function extensionality, or the univalence axiom. Voevodsky has formulated a precise mathematical conjecture connected to this question of canonicity for type theory extended with the axiom of Univalence: given a closed term of the type of natural numbers, is it always possible to find a numeral and a proof that this term is equal to this numeral, where this proof of equality may itself use the univalence axiom? More generally, an important issue is whether it is possible to provide a constructive justification of the univalence axiom. What about if one adds other homotopically motivated constructions, like higher inductive types? These questions remain open at the present time, although methods are currently being developed to try to find answers.

Another basic issue is the difficulty of working with types, such as the natural numbers, that are essentially sets (i.e., discrete spaces), containing only trivial paths. At present, homotopy type theory can really only characterize spaces up to homotopy equivalence, which means that these "discrete spaces" may only be *homotopy equivalent* to discrete spaces. Type-theoretically, this means there are many paths that are equal to reflexivity, but not *judgmentally* equal to it (see §1.1 for the meaning of "judgmentally"). While this homotopy-invariance has advantages, these "meaningless" identity terms do introduce needless complications into arguments and constructions, so it would be convenient to have a systematic way of eliminating or collapsing them.

A more specialized, but no less important, problem is the relation between homotopy type theory and the research on *higher toposes* currently happening at the intersection of higher category theory and homotopy theory. There is a growing conviction among those familiar with both subjects that they are intimately connected. For instance, the notion of a univalent universe should coincide with that of an object classifier, while higher inductive types should be an "elementary" reflection of local presentability. More generally, homotopy type theory should be the "internal language" of $(\infty, 1)$-toposes, just as intuitionistic higher-order logic is the internal language of ordinary 1-toposes. Despite this general consensus, however, details remain to be worked out — in particular, questions of coherence and strictness remain to be addressed — and doing so will undoubtedly lead to further insights into both concepts.

But by far the largest field of work to be done is in the ongoing formalization of everyday mathematics in this new system. Recent successes in formalizing some facts from basic homotopy theory and category theory have been encouraging; some of these are described in Chapters 8 and 9. Obviously, however, much work remains to be done.

How to read this book

This book is divided into two parts. Part I, "Foundations", develops the fundamental concepts of homotopy type theory. This is the mathematical foundation on which the development of specific subjects is built, and which is required for the understanding of the univalent foundations

approach. To a programmer, this is "library code". Since univalent foundations is a new and different kind of mathematics, its basic notions take some getting used to; thus Part I is fairly extensive.

Part II, "Mathematics", consists of four chapters that build on the basic notions of Part I to exhibit some of the new things we can do with univalent foundations in four different areas of mathematics: homotopy theory (Chapter 8), category theory (Chapter 9), set theory (Chapter 10), and real analysis (Chapter 11). The chapters in Part II are more or less independent of each other, although occasionally one will use a lemma proven in another.

A reader who wants to seriously understand univalent foundations, and be able to work in it, will eventually have to read and understand most of Part I. However, a reader who just wants to get a taste of univalent foundations and what it can do may understandably balk at having to work through over 200 pages before getting to the "meat" in Part II. Fortunately, not all of Part I is necessary in order to read the chapters in Part II. Each chapter in Part II begins with a brief overview of its subject, what univalent foundations has to contribute to it, and the necessary background from Part I, so the courageous reader can turn immediately to the appropriate chapter for their favorite subject. For those who want to understand one or more chapters in Part II more deeply than this, but are not ready to read all of Part I, we provide here a brief summary of Part I, with remarks about which parts are necessary for which chapters in Part II.

Chapter 1 is about the basic notions of type theory, prior to any homotopical interpretation. A reader who is familiar with Martin-Löf type theory can quickly skim it to pick up the particulars of the theory we are using. However, readers without experience in type theory will need to read Chapter 1, as there are many subtle differences between type theory and other foundations such as set theory.

Chapter 2 introduces the homotopical viewpoint on type theory, along with the basic notions supporting this view, and describes the homotopical behavior of each component of the type theory from Chapter 1. It also introduces the *univalence axiom* (§2.10) — the first of the two basic innovations of homotopy type theory. Thus, it is quite basic and we encourage everyone to read it, especially §§2.1–2.4.

Chapter 3 describes how we represent logic in homotopy type theory, and its connection to classical logic as well as to constructive and intuitionistic logic. Here we define the law of excluded middle, the axiom of choice, and the axiom of propositional resizing (although, for the most part, we do not need to assume any of these in the rest of the book), as well as the *propositional truncation* which is essential for representing traditional logic. This chapter is essential background for Chapters 10 and 11, less important for Chapter 9, and not so necessary for Chapter 8.

Chapters 4 and 5 study two special topics in detail: equivalences (and related notions) and generalized inductive definitions. While these are important subjects in their own rights and provide a deeper understanding of homotopy type theory, for the most part they are not necessary for Part II. Only a few lemmas from Chapter 4 are used here and there, while the general discussions in §§5.1, 5.6 and 5.7 are helpful for providing the intuition required for Chapter 6. The generalized sorts of inductive definition discussed in §5.7 are also used in a few places in Chapters 10 and 11.

Chapter 6 introduces the second basic innovation of homotopy type theory — *higher induc-*

tive types — with many examples. Higher inductive types are the primary object of study in Chapter 8, and some particular ones play important roles in Chapters 10 and 11. They are not so necessary for Chapter 9, although one example is used in §9.9.

Finally, Chapter 7 discusses homotopy n-types and related notions such as n-connected types. These notions are important for Chapter 8, but not so important in the rest of Part II, although the case $n = -1$ of some of the lemmas are used in §10.1.

This completes Part I. As mentioned above, Part II consists of four largely unrelated chapters, each describing what univalent foundations has to offer to a particular subject.

Of the chapters in Part II, Chapter 8 (Homotopy theory) is perhaps the most radical. Univalent foundations has a very different "synthetic" approach to homotopy theory in which homotopy types are the basic objects (namely, the types) rather than being constructed using topological spaces or some other set-theoretic model. This enables new styles of proof for classical theorems in algebraic topology, of which we present a sampling, from $\pi_1(\mathsf{S}^1) = \mathbb{Z}$ to the Freudenthal suspension theorem.

In Chapter 9 (Category theory), we develop some basic (1-)category theory, adhering to the principle of the univalence axiom that *equality is isomorphism*. This has the pleasant effect of ensuring that all definitions and constructions are automatically invariant under equivalence of categories: indeed, equivalent categories are equal just as equivalent types are equal. (It also has connections to higher category theory and higher topos theory.)

Chapter 10 (Set theory) studies sets in univalent foundations. The category of sets has its usual properties, hence provides a foundation for any mathematics that doesn't need homotopical or higher-categorical structures. We also observe that univalence makes cardinal and ordinal numbers a bit more pleasant, and that higher inductive types yield a cumulative hierarchy satisfying the usual axioms of Zermelo–Fraenkel set theory.

In Chapter 11 (Real numbers), we summarize the construction of Dedekind real numbers, and then observe that higher inductive types allow a definition of Cauchy real numbers that avoids some associated problems in constructive mathematics. Then we sketch a similar approach to Conway's surreal numbers.

Each chapter in this book ends with a Notes section, which collects historical comments, references to the literature, and attributions of results, to the extent possible. We have also included Exercises at the end of each chapter, to assist the reader in gaining familiarity with doing mathematics in univalent foundations.

Finally, recall that this book was written as a massively collaborative effort by a large number of people. We have done our best to achieve consistency in terminology and notation, and to put the mathematics in a linear sequence that flows logically, but it is very likely that some imperfections remain. We ask the reader's forgiveness for any such infelicities, and welcome suggestions for improvement of the next edition.

Part I

Foundations

Chapter 1

Type theory

1.1 Type theory versus set theory

Homotopy type theory is (among other things) a foundational language for mathematics, i.e., an alternative to Zermelo–Fraenkel set theory. However, it behaves differently from set theory in several important ways, and that can take some getting used to. Explaining these differences carefully requires us to be more formal here than we will be in the rest of the book. As stated in the introduction, our goal is to write type theory *informally*; but for a mathematician accustomed to set theory, more precision at the beginning can help avoid some common misconceptions and mistakes.

We note that a set-theoretic foundation has two "layers": the deductive system of first-order logic, and, formulated inside this system, the axioms of a particular theory, such as ZFC. Thus, set theory is not only about sets, but rather about the interplay between sets (the objects of the second layer) and propositions (the objects of the first layer).

By contrast, type theory is its own deductive system: it need not be formulated inside any superstructure, such as first-order logic. Instead of the two basic notions of set theory, sets and propositions, type theory has one basic notion: *types*. Propositions (statements which we can prove, disprove, assume, negate, and so on[1]) are identified with particular types, via the correspondence shown in Table 1 on page 11. Thus, the mathematical activity of *proving a theorem* is identified with a special case of the mathematical activity of *constructing an object*—in this case, an inhabitant of a type that represents a proposition.

This leads us to another difference between type theory and set theory, but to explain it we must say a little about deductive systems in general. Informally, a deductive system is a collection of rules for deriving things called **judgments**. If we think of a deductive system as a formal game, then the judgments are the "positions" in the game which we reach by following the game rules. We can also think of a deductive system as a sort of algebraic theory, in which case the judgments are the elements (like the elements of a group) and the deductive rules are

[1] Confusingly, it is also a common practice (dating back to Euclid) to use the word "proposition" synonomously with "theorem". We will confine ourselves to the logician's usage, according to which a *proposition* is a statement *susceptible to* proof, whereas a *theorem* (or "lemma" or "corollary") is such a statement that *has been* proven. Thus "$0 = 1$" and its negation "$\neg(0 = 1)$" are both propositions, but only the latter is a theorem.

the operations (like the group multiplication). From a logical point of view, the judgments can be considered to be the "external" statements, living in the metatheory, as opposed to the "internal" statements of the theory itself.

In the deductive system of first-order logic (on which set theory is based), there is only one kind of judgment: that a given proposition has a proof. That is, each proposition A gives rise to a judgment "A has a proof", and all judgments are of this form. A rule of first-order logic such as "from A and B infer $A \wedge B$" is actually a rule of "proof construction" which says that given the judgments "A has a proof" and "B has a proof", we may deduce that "$A \wedge B$ has a proof". Note that the judgment "A has a proof" exists at a different level from the *proposition* A itself, which is an internal statement of the theory.

The basic judgment of type theory, analogous to "A has a proof", is written "$a : A$" and pronounced as "the term a has type A", or more loosely "a is an element of A" (or, in homotopy type theory, "a is a point of A"). When A is a type representing a proposition, then a may be called a *witness* to the provability of A, or *evidence* of the truth of A (or even a *proof* of A, but we will try to avoid this confusing terminology). In this case, the judgment $a : A$ is derivable in type theory (for some a) precisely when the analogous judgment "A has a proof" is derivable in first-order logic (modulo differences in the axioms assumed and in the encoding of mathematics, as we will discuss throughout the book).

On the other hand, if the type A is being treated more like a set than like a proposition (although as we will see, the distinction can become blurry), then "$a : A$" may be regarded as analogous to the set-theoretic statement "$a \in A$". However, there is an essential difference in that "$a : A$" is a *judgment* whereas "$a \in A$" is a *proposition*. In particular, when working internally in type theory, we cannot make statements such as "if $a : A$ then it is not the case that $b : B$", nor can we "disprove" the judgment "$a : A$".

A good way to think about this is that in set theory, "membership" is a relation which may or may not hold between two pre-existing objects "a" and "A", while in type theory we cannot talk about an element "a" in isolation: every element *by its very nature* is an element of some type, and that type is (generally speaking) uniquely determined. Thus, when we say informally "let x be a natural number", in set theory this is shorthand for "let x be a thing and assume that $x \in \mathbb{N}$", whereas in type theory "let $x : \mathbb{N}$" is an atomic statement: we cannot introduce a variable without specifying its type.

At first glance, this may seem an uncomfortable restriction, but it is arguably closer to the intuitive mathematical meaning of "let x be a natural number". In practice, it seems that whenever we actually *need* "$a \in A$" to be a proposition rather than a judgment, there is always an ambient set B of which a is known to be an element and A is known to be a subset. This situation is also easy to represent in type theory, by taking a to be an element of the type B, and A to be a predicate on B; see §3.5.

A last difference between type theory and set theory is the treatment of equality. The familiar notion of equality in mathematics is a proposition: e.g. we can disprove an equality or assume an equality as a hypothesis. Since in type theory, propositions are types, this means that equality is a type: for elements $a, b : A$ (that is, both $a : A$ and $b : A$) we have a type "$a =_A b$". (In *homotopy* type theory, of course, this equality proposition can behave in unfamiliar ways: see §1.12 and Chapter 2, and the rest of the book).

However, in type theory there is also a need for an equality *judgment*, existing at the same level as the judgment "$x : A$". This is called **judgmental equality** or **definitional equality**, and we write it as $a \equiv b$ or $a \equiv_A b$. It is helpful to think of this as meaning "equal by definition". For instance, if we define a function $f : \mathbb{N} \to \mathbb{N}$ by the equation $f(x) = x^2$, then the expression $f(3)$ is equal to 3^2 *by definition*. It does not make sense to negate or assume an equality-by-definition; we cannot say "if x is equal to y by definition, then z is not equal to w by definition". Whether or not two expressions are equal by definition is just a matter of expanding out the definitions; in particular, it is algorithmically decidable.

As type theory becomes more complicated, judgmental equality can get more subtle than this, but it is a good intuition to start from. Alternatively, if we regard a deductive system as an algebraic theory, then judgmental equality is simply the equality in that theory, analogous to the equality between elements of a group—the only potential for confusion is that there is *also* an object *inside* the deductive system of type theory (namely the type "$a = b$") which behaves internally as a notion of "equality".

The reason we *want* a judgmental notion of equality is so that it can control the other form of judgment, "$a : A$". For instance, suppose we have given a proof that $3^2 = 9$, i.e. we have derived the judgment $p : (3^2 = 9)$ for some p. Then the same witness p ought to count as a proof that $f(3) = 9$, since $f(3)$ is 3^2 *by definition*. The best way to represent this is with a rule saying that given the judgments $a : A$ and $A \equiv B$, we can derive the judgment $a : B$.

Thus, for us, type theory will be a deductive system based on two forms of judgment:

Judgment	Meaning
$a : A$	"a is an object of type A"
$a \equiv_A b$	"a and b are definitionally equal objects of type A"

When introducing a definitional equality, i.e., defining one thing to be equal to another, we will use the symbol ":≡". Thus, the above definition of the function f would be written as $f(x) :\equiv x^2$.

Because judgments cannot be put together into more complicated statements, the symbols ":" and "≡" bind more loosely than anything else.[2] Thus, for instance, "$p : x = y$" should be parsed as "$p : (x = y)$", which makes sense since "$x = y$" is a type, and not as "$(p : x) = y$", which is senseless since "$p : x$" is a judgment and cannot be equal to anything. Similarly, "$A \equiv x = y$" can only be parsed as "$A \equiv (x = y)$", although in extreme cases such as this, one ought to add parentheses anyway to aid reading comprehension. This is perhaps also an appropriate place to mention that the common mathematical notation "$f : A \to B$", expressing the fact that f is a function from A to B, can be regarded as a typing judgment, since we use "$A \to B$" as notation for the type of functions from A to B (as is standard practice in type theory; see §1.4).

Judgments may depend on assumptions of the form $x : A$, where x is a variable and A is a type. For example, we may construct an object $m + n : \mathbb{N}$ under the assumptions that $m, n : \mathbb{N}$. Another example is that assuming A is a type, $x, y : A$, and $p : x =_A y$, we may construct an element $p^{-1} : y =_A x$. The collection of all such assumptions is called the **context**; from a

[2] In formalized type theory, commas and turnstiles can bind even more loosely. For instance, $x : A, y : B \vdash c : C$ is parsed as $((x : A), (y : B)) \vdash (c : C)$. However, in this book we refrain from such notation until Appendix A.

topological point of view it may be thought of as a "parameter space". In fact, technically the context must be an ordered list of assumptions, since later assumptions may depend on previous ones: the assumption $x : A$ can only be made *after* the assumptions of any variables appearing in the type A.

If the type A in an assumption $x : A$ represents a proposition, then the assumption is a type-theoretic version of a *hypothesis*: we assume that the proposition A holds. When types are regarded as propositions, we may omit the names of their proofs. Thus, in the second example above we may instead say that assuming $x =_A y$, we can prove $y =_A x$. However, since we are doing "proof-relevant" mathematics, we will frequently refer back to proofs as objects. In the example above, for instance, we may want to establish that p^{-1} together with the proofs of transitivity and reflexivity behave like a groupoid; see Chapter 2.

In the rest of this chapter, we attempt to give an informal presentation of type theory, sufficient for the purposes of this book; we will give a more formal account in Appendix A. Aside from some fairly obvious rules (such as the fact that judgmentally equal things can always be substituted for each other), the rules of type theory can be grouped into *type formers*. Each type former consists of a way to construct types (possibly making use of previously constructed types), together with rules for the construction and behavior of elements of that type. In most cases, these rules follow a fairly predictable pattern, but we will not attempt to make this precise.

An important aspect of the type theory presented in this chapter is that it consists entirely of *rules*, without any *axioms*. In the description of deductive systems in terms of judgments, the *rules* are what allow us to conclude one judgment from collection of others, while the *axioms* are the judgments we are given at the outset. If we think of a deductive system as a formal game, then the rules are the rules of the game, while the axioms are the starting position. And if we think of a deductive system as an algebraic theory, then the rules are the operations of the theory, while the axioms are the *generators* for some particular free model of that theory.

In set theory, the only rules are the rules of first-order logic (such as the rule allowing us to deduce "$A \wedge B$ has a proof" from "A has a proof" and "B has a proof"): all the information about the behavior of sets is contained in the axioms. By contrast, in type theory, it is usually the *rules* which contain all the information, with no axioms being necessary. For instance, in §1.5 we will see that there is a rule allowing us to deduce the judgment "$(a, b) : A \times B$" from "$a : A$" and "$b : B$", whereas in set theory the analogous statement would be (a consequence of) the pairing axiom.

The advantage of formulating type theory using only rules is that rules are "procedural". In particular, this property is what makes possible (though it does not automatically ensure) the good computational properties of type theory, such as "canonicity". However, while this style works for traditional type theories, we do not yet understand how to formulate everything we need for *homotopy* type theory in this way. In particular, in §§2.9 and 2.10 and Chapter 6 we will have to augment the rules of type theory presented in this chapter by introducing additional axioms, notably the *univalence axiom*. In this chapter, however, we confine ourselves to a traditional rule-based type theory.

1.2 Function types

Given types A and B, we can construct the type $A \to B$ of functions with domain A and codomain B. Unlike in set theory, functions are not defined as functional relations; rather they are a primitive concept in type theory. We explain the function type by prescribing what we can do with functions, how to construct them and what equalities they induce.

Given a function $f : A \to B$ and an element of the domain $a : A$, we can **apply** the function to obtain an element of the codomain, denoted $f(a) : B$.

But how can we construct elements of $A \to B$? There are two equivalent ways: either by direct definition or by using λ-abstraction. Introducing a function by definition means that we introduce a function by giving it a name — let's say, f — and saying we define $f : A \to B$ by giving an equation

$$f(x) :\equiv b \tag{1.2.1}$$

where x is a variable and b is an expression which may use x. In order for this to be valid, we have to check that $b : B$ assuming $x : A$.

Now we can compute $f(a)$ by replacing the variable x in b with a. As an example, consider the function $f : \mathbb{N} \to \mathbb{N}$ which is defined by $f(x) :\equiv x + x$. (We will define $\mathbb{N}$ and $+$ in §1.9.) Then $f(2)$ is judgmentally equal to $2 + 2$.

If we don't want to introduce a name for the function, we can use **λ-abstraction**. Given an expression $b : B$ which may use $x : A$, as above, we write $\lambda(x : A).\, b$ to indicate the same function defined by (1.2.1). Thus, we have

$$(\lambda(x : A).\, b) : A \to B.$$

For the example in the previous paragraph, we have the typing judgment

$$(\lambda(x : \mathbb{N}).\, x + x) : \mathbb{N} \to \mathbb{N}.$$

We generally omit the type of the variable x and write $\lambda x.\, b$, since the typing $x : A$ is inferrable from the judgment that the function $\lambda x.\, b$ has type $A \to B$. By convention, the "scope" of the variable binding "$\lambda x.$" is the entire rest of the expression, unless delimited with parentheses. Thus, for instance, $\lambda x.\, x + x$ should be parsed as $\lambda x.\, (x + x)$, not as $(\lambda x.\, x) + x$ (which would, in this case, be ill-typed anyway).

Another equivalent notation is

$$(x \mapsto b) : A \to B.$$

We may also sometimes use a blank "$-$" in the expression b in place of a variable, to denote an implicit λ-abstraction. For instance, $g(x, -)$ is another way to write $\lambda y.\, g(x, y)$.

Now a λ-abstraction is a function, so we can apply it to an argument $a : A$. We then have the definitional equality[3]

$$(\lambda x.\, b)(a) \equiv b'$$

where b' is the expression b in which all occurrences of x have been replaced by a. Continuing the above example, we have

$$(\lambda x.\, x + x)(2) \equiv 2 + 2.$$

[3] Use of this equality is often referred to as β-conversion or β-reduction.

Note that from any function $f : A \to B$, we can construct a lambda abstraction function $\lambda x.\, f(x)$. Since this is by definition "the function which applies f to its argument" we consider it to be definitionally equal to f:[4]

$$f \equiv (\lambda x.\, f(x)).$$

The introduction of functions by definitions with explicit parameters can be reduced to simple definitions by using λ-abstraction: i.e., we can read a definition of $f : A \to B$ by

$$f(x) :\equiv b$$

as

$$f :\equiv \lambda x.\, b.$$

When doing calculations involving variables, we have to be careful when replacing a variable with an expression that also involves variables, because we want to preserve the binding structure of expressions. By the *binding structure* we mean the invisible link generated by binders such as λ, Π and Σ (the latter we are going to meet soon) between the place where the variable is introduced and where it is used. As an example, consider $f : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$ defined as

$$f(x) :\equiv \lambda y.\, x + y$$

Now if we have assumed somewhere that $y : \mathbb{N}$, then what is $f(y)$? It would be wrong to just naively replace x by y everywhere in the expression "$\lambda y.\, x + y$" defining $f(x)$, obtaining $\lambda y.\, y + y$, because this means that y gets **captured**. Previously, the substituted y was referring to our assumption, but now it is referring to the parameter of the anonymous function. Hence, this naive substitution would destroy the binding structure, allowing us to perform calculations which are semantically unsound.

But what *is* $f(y)$ in this example? Note that bound (or "dummy") variables such as y in the expression $\lambda y.\, x + y$ have only a local meaning, and can be consistently replaced by any other variable, preserving the binding structure. Indeed, $\lambda y.\, x + y$ is declared to be judgmentally equal[5] to $\lambda z.\, x + z$. It follows that $f(y)$ is judgmentally equal to $\lambda z.\, y + z$, and that answers our question. (Instead of z, any variable distinct from y could have been used, yielding an equal result.)

Of course, this should all be familiar to any mathematician: it is the same phenomenon as the fact that if $f(x) :\equiv \int_1^2 \frac{dt}{x-t}$, then $f(t)$ is not $\int_1^2 \frac{dt}{t-t}$ but rather $\int_1^2 \frac{ds}{t-s}$. A λ-abstraction binds a dummy variable in exactly the same way that an integral does.

We have seen how to define functions in one variable. One way to define functions in several variables would be to use the cartesian product, which will be introduced later; a function with parameters A and B and results in C would be given the type $f : A \times B \to C$. However, there is another choice that avoids using product types, which is called **currying** in functional programming.[6]

The idea of currying is to represent a function of two inputs $a : A$ and $b : B$ as a function which takes *one* input $a : A$ and returns *another function*, which then takes a second input $b : B$ and

[4] Use of this equality is often referred to as **η-conversion** or **η-expansion.**

[5] Use of this equality is called **α-conversion.**

[6] The technique is named after the mathematician Haskell Curry

returns the result. That is, we consider two-variable functions to belong to an iterated function type, $f : A \to (B \to C)$. We may also write this without the parentheses, as $f : A \to B \to C$, with associativity to the right as the default convention. Then given $a : A$ and $b : B$, we can apply f to a and then apply the result to b, obtaining $f(a)(b) : C$. To avoid the proliferation of parentheses, we allow ourselves to write $f(a)(b)$ as $f(a, b)$ even though there are no products involved. Our notation for definitions with explicit parameters extends to this situation: we can define a named function $f : A \to B \to C$ by giving an equation

$$f(x, y) :\equiv c$$

where $c : C$ assuming $x : A$ and $y : B$. Using λ-abstraction this corresponds to

$$f :\equiv \lambda x.\, \lambda y.\, c,$$

which may also be written as

$$f :\equiv x \mapsto y \mapsto c.$$

Currying a function of three or more arguments is a straightforward extension of what we have just described.

1.3 Universes and families

So far, we have been using the expression "A is a type" informally. We are going to make this more precise by introducing **universes**. A universe is a type whose elements are types. As in naive set theory, we might wish for a universe of all types $\mathcal{U}_\infty$ including itself (that is, with $\mathcal{U}_\infty : \mathcal{U}_\infty$). However, as in set theory, this is unsound, i.e. we can deduce from it that every type, including the empty type representing the proposition False, is inhabited. Indeed, using a representation of sets as trees, we can directly encode Russell's paradox [Coq92].

To avoid the paradox we introduce a hierarchy of universes

$$\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \cdots$$

where every universe $\mathcal{U}_i$ is an element of the next universe $\mathcal{U}_{i+1}$. Moreover we assume that our universes are **cumulative**, that is that all the elements of the ith universe are also elements of the $(i+1)^{\text{st}}$ universe, i.e. if $A : \mathcal{U}_i$ then also $A : \mathcal{U}_{i+1}$.

When we say that A is a type, we mean that it inhabits some universe $\mathcal{U}_i$. We usually want to avoid mentioning the level i explicitly, and just assume that levels can be assigned in a consistent way; thus we may write $A : \mathcal{U}$ omitting the level. This way we can even write $\mathcal{U} : \mathcal{U}$, which can be read as $\mathcal{U}_i : \mathcal{U}_{i+1}$, having left the indices implicit. Writing universes in this style is referred to as **typical ambiguity**.

To model a collection of types varying over a given type A, we use functions $B : A \to \mathcal{U}$ whose codomain is a universe. These functions are called **families of types**; they correspond to families of sets as used in set theory. An example of a family is the family of finite sets $\mathsf{Fin} : \mathbb{N} \to \mathcal{U}$, where $\mathsf{Fin}(n)$ is a type with exactly n elements, denoted $0_n, 1_n, \ldots, (n-1)_n$. (We will be able to define this family soon; see Exercise 1.9.)

1.4 Dependent function types (Π-types)

In type theory we often use a more general version of function types, called a Π-type or dependent function type. The elements of a Π-type are functions whose codomain type can vary depending on the element of the domain to which the function is applied. The name "Π-type" is used because this type can also be regarded as the infinite cartesian product over a given type.

Given a type $A : \mathcal{U}$ and a family $B : A \to \mathcal{U}$, we may construct the type of dependent functions $\prod_{(x:A)} B(x) : \mathcal{U}$. There are many alternative notations for this type, such as

$$\prod_{(x:A)} B(x) \qquad \prod_{(x:A)} B(x) \qquad \prod(x : A),\ B(x).$$

If B is constant, then the dependent product type is the ordinary function type:

$$\prod_{(x:A)} B \equiv (A \to B).$$

Indeed, all the constructions of Π-types are generalisations of the corresponding constructions on ordinary function types.

We can introduce dependent functions by explicit definitions: to define $f : \prod_{(x:A)} B(x)$, where f is the name of a dependent function to be defined, we need an expression $b : B(x)$ involving $x : A$, and we write

$$f(x) :\equiv b \qquad \text{for } x : A$$

Alternatively, we can use **λ-abstraction**

$$\lambda x.\, b \ :\ \prod_{x:A} B(x). \tag{1.4.1}$$

The equalities are the same as for the ordinary function type, i.e. given $a : A$ we have $f(a) \equiv b'$ and $(\lambda x.\, b)(a) \equiv b'$, where b' is obtained by replacing all occurrences of x in b by a (avoiding variable capture, as always). Similarly, we have $f \equiv (\lambda x.\, f(x))$ for any $f : \prod_{(x:A)} B(x)$.

An example of a dependent function is $\mathsf{fmax} : \prod_{(n:\mathbb{N})} \mathsf{Fin}(n+1)$ which returns the largest element in a nonempty finite type, $\mathsf{fmax}(n) :\equiv n_{n+1}$. Another important class of dependent function types are functions which are **polymorphic** over a given universe. A polymorphic function is one which takes a type as one of its arguments, and then acts on elements of that type (or other types constructed from it). An example is the polymorphic identity function $\mathsf{id} : \prod_{(A:U)} A \to A$, which we define by $\mathsf{id} :\equiv \lambda(A : \mathcal{U}).\, \lambda(x : A).\, x$, or more succinctly $\mathsf{id}_A :\equiv \lambda x.\, x$.

As for ordinary functions, we use currying to define dependent functions with several arguments. However, in the dependent case the second domain may depend on the first one, and the codomain may depend on both. That is, given $A : U$ and type families $B : A \to U$ and $C : \prod_{(x:A)} B(x) \to \mathcal{U}$, we may construct the type $\prod_{(x:A)} \prod_{(y:B(x))} C(x, y)$ of functions with two arguments. (Like λ-abstractions, Πs automatically scope over the rest of the expression unless delimited; thus $C : \prod_{(x:A)} B(x) \to \mathcal{U}$ means $C : \prod_{(x:A)} (B(x) \to \mathcal{U})$.) Likewise, given $f : \prod_{(x:A)} \prod_{(y:B(x))} C(x, y)$ and arguments $a : A$ and $b : B(a)$, we have $f(a)(b) : C(a, b)$, which, as before, we write as $f(a, b) : C(a, b)$.

1.5 Product types

Given types $A, B : \mathcal{U}$ we introduce the type $A \times B : \mathcal{U}$, which we call their **cartesian product**. We also introduce a nullary product type, called the **unit type** $\mathbf{1} : \mathcal{U}$. We intend the elements of $A \times B$ to be pairs $(a, b) : A \times B$, where $a : A$ and $b : B$, and the only element of $\mathbf{1}$ to be some particular object $\star : \mathbf{1}$. However, unlike in set theory, where we define ordered pairs to be particular sets and then collect them all together into the cartesian product, in type theory, ordered pairs are a primitive concept, as are functions.

There is a general pattern to how one introduces a new primitive concept in type theory, and because products are our second example following this pattern,[7] it is worth emphasizing the general form: To specify a type, we say

(i) how to construct elements of that type. These are called the type's *constructors*. For example, the function type has one constructor, λ-abstraction.

(ii) how to use elements of that type. These are called the type's *eliminators*. For example, the function type has one eliminator, function application.

(iii) equalities relating the constructors and eliminators. For example, for functions, we have the rule that $(\lambda x.\, b)(a)$ is equal to the substitution of a for x in b.

The way to construct pairs is obvious: given $a : A$ and $b : B$, we may form $(a, b) : A \times B$. Similarly, there is a unique way to construct elements of $\mathbf{1}$, namely we have $\star : \mathbf{1}$. However, we do not assert as a rule of type theory that "every element of $A \times B$ is a pair" — it turns out that we can *prove* that using the principles to be introduced below.

Now, how can we *use* pairs, i.e. how can we define functions out of a product type? Let us first consider the definition of a non-dependent function $f : A \times B \to C$. Since we intend the only elements of $A \times B$ to be pairs, in order to define such a function, it should be enough to prescribe the result when f is applied to a pair (a, b). We prescribe these results by providing a function $g : A \to B \to C$ and defining

$$f((a, b)) :\equiv g(a)(b).$$

We avoid writing $g(a, b)$ here, in order to emphasize that g is not a function on a product.

In particular, we can derive the **projection** functions

$$\begin{aligned} \mathsf{pr}_1 &: A \times B \to A \\ \mathsf{pr}_2 &: A \times B \to B \end{aligned}$$

with the defining equations

$$\begin{aligned} \mathsf{pr}_1((a, b)) &:\equiv a \\ \mathsf{pr}_2((a, b)) &:\equiv b \end{aligned}$$

Rather than invoking this principle of function definition every time we want to define a function, an alternative approach is to invoke it once, in a universal case, and then simply apply the

[7] The description of universes above is an exception.

resulting function in all other cases. That is, we may define a function of type

$$\mathsf{rec}_{A\times B} : \prod_{C:\mathcal{U}} (A \to B \to C) \to A \times B \to C \tag{1.5.1}$$

with the defining equation

$$\mathsf{rec}_{A\times B}(C, g, (a, b)) :\equiv g(a)(b).$$

Then instead of defining functions such as pr_1 and pr_2 directly by a defining equation, we could instead define

$$\mathsf{pr}_1 :\equiv \mathsf{rec}_{A\times B}(A, \lambda a.\, \lambda b.\, a)$$
$$\mathsf{pr}_2 :\equiv \mathsf{rec}_{A\times B}(B, \lambda a.\, \lambda b.\, b).$$

We refer to the function $\mathsf{rec}_{A\times B}$ as the **recursor** for product types. We may also speak of the **recursion principle** for cartesian products, meaning the fact that we can define a function $f : A \times B \to C$ as above by giving its value on pairs. The name "recursor" is a bit unfortunate here, since no recursion is taking place. It comes the fact that product types are a degenerate example of a general framework for inductive types, and for types such as the natural numbers, the recursor will actually be recursive.

We leave it as a simple exercise to show that the recursor can be derived from the projections and vice versa.

We also have a recursor for the unit type:

$$\mathsf{rec}_{\mathbf{1}} : \prod_{C:\mathcal{U}} C \to \mathbf{1} \to C$$

with the defining equation

$$\mathsf{rec}_{\mathbf{1}}(C, c, \star) :\equiv c$$

However, this is completely useless, because we could have defined such a function directly by simply ignoring the argument of type $\mathbf{1}$.

To be able to define *dependent* functions over the product type, we have to generalize the recursor. Given $C : A \times B \to \mathcal{U}$, we may define a function $f : \prod_{(x:A\times B)} C(x)$ by providing a function $g : \prod_{(x:A)} \prod_{(y:B)} C((x, y))$ and the defining equation

$$f((x, y)) :\equiv g(x)(y).$$

For example, in this way we can prove that every element of $A \times B$ is a pair; this is called the principle of **surjective pairing**. Specifically, we can construct a function

$$\mathsf{sp} : \prod_{x:A\times B} ((\mathsf{pr}_1(x), \mathsf{pr}_2(x)) =_{A\times B} x).$$

Here we are using the identity type, which we are going to introduce below in §1.12. However, all we need here is to know that there is a reflexivity element $\mathsf{refl}_x : x =_A x$ for any $x : A$. Given this, we can define

$$\mathsf{sp}((a, b)) :\equiv \mathsf{refl}_{(a,b)}$$

This construction works, because in the case that $x :\equiv (a, b)$ we can calculate

$$(\mathsf{pr}_1((a,b)), \mathsf{pr}_2((a,b))) \equiv (a,b)$$

using the defining equations for the projections. Therefore,

$$\mathsf{refl}_{(a,b)} : (\mathsf{pr}_1((a,b)), \mathsf{pr}_2((a,b))) = (a,b)$$

is well-typed, since both sides of the equality are judgmentally equal.

More generally, the ability to define dependent functions in this way means that to prove a property for all elements of a product, it is enough to prove it for its canonical elements, the tuples. When we come to inductive types such as the natural numbers, the analogous property will be the ability to write proofs by induction. Thus, if we do as we did above and apply this principle once in the universal case, we call the resulting function **induction** for product types: given $A, B : \mathcal{U}$ we have

$$\mathsf{ind}_{A\times B} : \prod_{C:A\times B\to\mathcal{U}} \Big(\prod_{(x:A)} \prod_{(y:B)} C((x,y))\Big) \to \prod_{x:A\times B} C(x)$$

with the defining equation

$$\mathsf{ind}_{A\times B}(C, g, (a,b)) :\equiv g(a)(b).$$

Similarly, we may speak of a dependent function defined on pairs being obtained from the **induction principle** of the cartesian product. It is easy to see that the recursor is just the special case of induction in the case that the family C is constant. Because induction describes how to use an element of the product type, induction is also called the **(dependent) eliminator**, and recursion the **non-dependent eliminator**.

Induction for the unit type turns out to be more useful than the recursor:

$$\mathsf{ind}_{\mathbf{1}} : \prod_{C:\mathbf{1}\to\mathcal{U}} C(\star) \to \prod_{x:\mathbf{1}} C(x)$$

with the defining equation

$$\mathsf{ind}_{\mathbf{1}}(C, c, \star) :\equiv c$$

Induction enables us to *prove* that the only inhabitant of the unit type is $\star$, that is we can construct

$$\mathsf{un} : \prod_{x:\mathbf{1}} x = \star$$

by using the defining equations

$$\mathsf{un}(\star) :\equiv \mathsf{refl}_\star$$

or equivalently by using induction:

$$\mathsf{un} :\equiv \mathsf{ind}_{\mathbf{1}}(\lambda x.\, x = \star, \mathsf{refl}_\star).$$

1.6 Dependent pair types (Σ-types)

Just as we generalized function types (§1.2) to dependent function types (§1.4), it is often useful to generalize the product types from §1.5 to allow the type of the second component of a pair to vary depending on the choice of the first component. This is called a Σ-type, because in set theory it corresponds to an indexed sum (in the sense of coproduct or disjoint union) over a given type.

Given a type $A : \mathcal{U}$ and a family $B : A \to \mathcal{U}$, the dependent pair type is written as $\sum_{(x:A)} B(x) : \mathcal{U}$. Alternative notations are

$$\sum_{(x:A)} B(x) \qquad \sum_{(x:A)} B(x) \qquad \sum(x : A),\ B(x).$$

The way to construct elements of this type is by pairing: we have $(a, b) : \sum_{(x:A)} B(x)$ given $a : A$ and $b : B(a)$. If B is constant, then the dependent pair type is the ordinary cartesian product type:

$$(\sum_{x:A} B) \equiv (A \times B).$$

All the constructions on Σ-types arise as straightforward generalisations of the ones for product types, with dependent functions often replacing non-dependent ones.

For instance, the recursion principle says that to define a non-dependent function out of a Σ-type $f : (\sum_{(x:A)} B(x)) \to C$, we provide a function $g : \prod_{(x:A)} B(x) \to C$, and then we can define f via the defining equation

$$f((a, b)) :\equiv g(a)(b)$$

For instance, we can derive the first projection from a Σ-type:

$$\mathsf{pr}_1 \ : \ (\sum_{x:A} B(x)) \to A.$$

by the defining equation

$$\mathsf{pr}_1((a, b)) \ :\equiv \ a.$$

However, since the type of the second component of a pair $(a, b) : \sum_{(x:A)} B(x)$ is $B(a)$, the second projection must be a *dependent* function, whose type involves the first projection function:

$$\mathsf{pr}_2 : \prod_{p:\sum_{(x:A)} B(x)} B(\mathsf{pr}_1(p)).$$

Thus we need the *induction* principle for Σ-types (the "dependent eliminator"). This says that to construct a dependent function out of a Σ-type into a family $C : (\sum_{(x:A)} B(x)) \to \mathcal{U}$, we need a function

$$g : \prod_{(a:A)} \prod_{(b:B(x))} C((a, b)).$$

We can then derive a function

$$f : \prod_{p:\sum_{(x:A)} B(x)} C(p)$$

with defining equation

$$f((a,b)) :\equiv g(a)(b).$$

Applying this with $C(p) :\equiv B(\mathsf{pr}_1(p))$, we can define $\mathsf{pr}_2 : \prod_{(p:\sum_{(x:A)} B(x))} B(\mathsf{pr}_1(p))$ with the obvious equation

$$\mathsf{pr}_2((a,b)) :\equiv b.$$

To convince ourselves that this is correct, we note that $B(\mathsf{pr}_1((a,b))) \equiv B(a)$, using the defining equation for pr_1, and indeed $b : B(a)$.

We can package the recursion and induction principles into the recursor for Σ:

$$\mathsf{rec}_{\sum_{(x:A)} B(x)} : \prod_{(C:\mathcal{U})} \left(\prod_{(x:A)} B(x) \to C\right) \to \left(\sum_{(x:A)} B(x)\right) \to C$$

with the defining equation

$$\mathsf{rec}_{\sum_{(x:A)} B(x)}(C, g, (a,b)) :\equiv g(a)(b)$$

and the corresponding induction operator:

$$\mathsf{ind}_{\sum_{(x:A)} B(x)} : \prod_{(C:(\sum_{(x:A)} B(x)) \to \mathcal{U})} \left(\prod_{(a:A)} \prod_{(b:B(x))} C((a,b))\right) \to \prod_{(p:\sum_{(x:A)} B(x))} C(p)$$

with the defining equation

$$\mathsf{ind}_{\sum_{(x:A)} B(x)}(C, g, (a,b)) :\equiv g(a)(b).$$

As before, the recursor is the special case of induction when the family C is constant.

As a further example, consider the following principle, where A and B are types and $R : A \to B \to \mathcal{U}$.

$$\mathsf{ac} : \left(\prod_{(x:A)} \sum_{(y:B)} R(x,y)\right) \to \left(\sum_{(f:A\to B)} \prod_{(x:A)} R(x, f(x))\right)$$

We may regard R as a "proof-relevant relation" between A and B, with $R(a,b)$ the type of witnesses for relatedness of $a : A$ and $b : B$. Then ac says intuitively that if we have a dependent function g assigning to every $a : A$ a dependent pair (b,r) where $b : B$ and $r : R(a,b)$, then we have a function $f : A \to B$ and a dependent function assigning to every $a : A$ a witness that $R(a, f(a))$. Our intuition tells us that we can just split up the values of g into their components. Indeed, using the projections we have just defined, we can define:

$$\mathsf{ac}(g) :\equiv \left(\lambda x.\, \mathsf{pr}_1(g(x)),\, \lambda x.\, \mathsf{pr}_2(g(x))\right).$$

To verify that this is well-typed, note that if $g : \prod_{(x:A)} \sum_{(y:B)} R(x,y)$, we have

$$\begin{aligned} \lambda x.\, \mathsf{pr}_1(g(x)) &: A \to B \\ \lambda x.\, \mathsf{pr}_2(g(x)) &: \textstyle\prod_{(x:A)} R(a, \mathsf{pr}_1(g(x))). \end{aligned}$$

Moreover, the type $\prod_{(x:A)} R(a, \mathsf{pr}_1(g(x)))$ is the result of substituting the function $\lambda x.\, \mathsf{pr}_1(g(x))$ for f in the family being summed over in the codomain of ac:

$$\prod_{(x:A)} R(x, \mathsf{pr}_1(f(x))) \equiv \Big(\lambda f.\, \prod_{(x:A)} R(x, f(x))\Big)\big(\lambda x.\, \mathsf{pr}_1(g(x))\big).$$

Thus, we have

$$\Big(\lambda x.\, \mathsf{pr}_1(g(x)),\ \lambda x.\, \mathsf{pr}_2(g(x))\Big) : \sum_{(f:A\to B)} \prod_{(x:A)} R(x, f(x))$$

as required.

If we read Π as "for all" and Σ as "there exists", then the type of the function ac expresses: *if for all $x : A$ there is a $y : B$ such that $R(x, y)$, then there is a function $f : A \to B$ such that for all $x : A$ we have $R(x, f(x))$*. Since this sounds like a version of the axiom of choice, the function ac has traditionally been called the **type-theoretic axiom of choice**. However, no choice is actually involved, since the choices have already been given to us in the premise: all we have to do is take it apart into two functions: one representing the choice and the other its correctness. In §3.8 we will give another formulation of an "axiom of choice" which is closer to the usual one.

1.7 Coproduct types

Given $A, B : \mathcal{U}$, we introduce their **coproduct** type $A + B : \mathcal{U}$. This corresponds to the *disjoint union* in set theory, and we may also use that name for it. In type theory, as was the case with functions and products, the coproduct must be a fundamental construction, since there is no previously given notion of "union of types". We also introduce a nullary version: the **empty type $\mathbf{0}$** : $\mathcal{U}$.

There are two ways to construct elements of $A + B$, either as $\mathsf{inl}(a) : A + B$ for $a : A$, or as $\mathsf{inr}(b) : A + B$ for $b : B$. There are no ways to construct elements of the empty type.

To construct a non-dependent function $f : A + B \to C$, we need functions $g_0 : A \to C$ and $g_1 : B \to C$. Then f is defined via the defining equations

$$\begin{aligned} f(\mathsf{inl}(a)) &:\equiv g_0(a) \\ f(\mathsf{inl}(b)) &:\equiv g_1(b). \end{aligned}$$

That is, the function f is defined by cases. As before, we can derive the recursor:

$$\mathsf{rec}_{A+B} : \prod_{(C:\mathcal{U})} (A \to C) \to (B \to C) \to A + B \to C$$

with the defining equations

$$\begin{aligned} \mathsf{rec}_{A+B}(C, g_0, g_1, \mathsf{inl}(a)) &:\equiv g_0(a) \\ \mathsf{rec}_{A+B}(C, g_0, g_1, \mathsf{inr}(b)) &:\equiv g_1(b) \end{aligned}$$

We can always construct a function $f : \mathbf{0} \to C$ without having to give any defining equations, because there are no elements of $\mathbf{0}$ on which to define f. Thus, the recursor for $\mathbf{0}$ is

$$\mathsf{rec}_{\mathbf{0}} : \prod_{(C:\mathcal{U})} \mathbf{0} \to C,$$

which constructs the canonical function from the empty type to any other type. Logically, it corresponds to the principle *ex falso quod libet*.

To construct a dependent function $f : \prod_{(x:A+B)} C(x)$ out of a coproduct, we assume as given the family $C : (A + B) \to \mathcal{U}$, and require

$$\begin{aligned} g_0 &: \prod_{a:A} C(\mathsf{inl}(a)) \\ g_1 &: \prod_{b:B} C(\mathsf{inr}(b)). \end{aligned}$$

This yields f with the defining equations:

$$\begin{aligned} f(\mathsf{inl}(a)) &:\equiv g_0(a) \\ f(\mathsf{inl}(b)) &:\equiv g_1(b). \end{aligned}$$

We package this scheme into the induction principle for coproducts:

$$\mathsf{ind}_{A+B} : \prod_{(C:(A+B)\to\mathcal{U})} \left(\prod_{(a:A)} C(\mathsf{inl}(a))\right) \to \left(\prod_{(b:B)} C(\mathsf{inr}(b))\right) \to \prod_{(x:A+B)} C(x)$$

As before, the recursor arises in the case that the family C is constant.

The induction principle for the empty type

$$\mathsf{ind}_{\mathbf{0}} : \prod_{(C:\mathbf{0}\to\mathcal{U})} \prod_{(z:\mathbf{0})} C(z)$$

gives us a way to define a trivial dependent function out of the empty type.

We define **negation** of a type A to be

$$\neg A :\equiv A \to \mathbf{0}.$$

To construct an element of $\neg A$ is the same thing as to construct a function $A \to \mathbf{0}$, which is done by assuming $x : A$ and deriving an element of $\mathbf{0}$.

1.8 The type of booleans

The type of booleans $\mathbf{2} : \mathcal{U}$ is intended to have exactly two elements $1_{\mathbf{2}}, 0_{\mathbf{2}} : \mathbf{2}$. It is clear that we could construct this type out of coproduct and unit types as $\mathbf{1} + \mathbf{1}$. However, since it is used frequently, we give the explicit rules here. Indeed, we are going to observe that we can also go the other way and derive binary coproducts from Σ-types and $\mathbf{2}$.

To derive a function $f : \mathbf{2} \to C$ we need $c_0, c_1 : C$ and add the defining equations

$$\begin{aligned} f(0_{\mathbf{2}}) &:\equiv c_0 \\ f(1_{\mathbf{2}}) &:\equiv c_1 \end{aligned}$$

The recursor corresponds to the if-then-else construct in functional programming:

$$\mathsf{rec}_{\mathbf{2}} : \prod_{C:\mathcal{U}} C \to C \to \mathbf{2} \to C$$

with the defining equations

$$\begin{aligned} \mathsf{rec}_{\mathbf{2}}(C, c_0, c_1, 0_{\mathbf{2}}) &:\equiv c_0 \\ \mathsf{rec}_{\mathbf{2}}(C, c_0, c_1, 1_{\mathbf{2}}) &:\equiv c_1 \end{aligned}$$

Given a family $C : \mathbf{2} \to \mathcal{U}$, to derive a dependent function $f : \prod_{(x:\mathbf{2})} C(x)$ we need $c_0 : C(0_{\mathbf{2}})$ and $c_1 : C(1_{\mathbf{2}})$, in which case we can give the defining equations

$$\begin{aligned} f(0_{\mathbf{2}}) &:\equiv c_0 \\ f(1_{\mathbf{2}}) &:\equiv c_1. \end{aligned}$$

We package this up into the induction principle

$$\mathsf{ind}_{\mathbf{2}} : \prod_{(C:\mathbf{2}\to\mathcal{U})} C(0_{\mathbf{2}}) \to C(1_{\mathbf{2}}) \to \textstyle\prod_{(x:\mathbf{2})} C(x)$$

with the defining equations

$$\begin{aligned} \mathsf{ind}_{\mathbf{2}}(C, c_0, c_1, 0_{\mathbf{2}}) &:\equiv c_0 \\ \mathsf{ind}_{\mathbf{2}}(C, c_0, c_1, 1_{\mathbf{2}}) &:\equiv c_1 \end{aligned}$$

We have remarked that Σ-types can be regarded as analogous to indexed disjoint unions, while coproducts are binary disjoint unions. It is natural to expect that a binary disjoint union $A + B$ could be constructed as an indexed one over the two-element type $\mathbf{2}$. For this we need a type family $P : \mathbf{2} \to \mathcal{U}$ such that $P(0_{\mathbf{2}}) \equiv A$ and $P(1_{\mathbf{2}}) \equiv B$. Indeed, we can obtain such a family precisely by the recursion principle for $\mathbf{2}$. (The ability to define *type families* by induction and recursion, using the fact that the universe $\mathcal{U}$ is itself a type, is one of the most subtle and important aspects of type theory.) Thus, we could have defined

$$A + B :\equiv \sum_{x:\mathbf{2}} \mathsf{rec}_{\mathbf{2}}(\mathcal{U}, A, B, x)$$

with

$$\begin{aligned} \mathsf{inl}(a) &:\equiv (0_{\mathbf{2}}, a) \\ \mathsf{inr}(b) &:\equiv (1_{\mathbf{2}}, b). \end{aligned}$$

We leave it as an exercise to derive the induction principle of a coproduct type from this definition. (See also Exercise 1.5 and §5.2.)

We can apply the same idea to products and Π-types: we could have defined

$$A \times B :\equiv \prod_{x:\mathbf{2}} \mathsf{rec}_{\mathbf{2}}(\mathcal{U}, A, B, x)$$

Pairs could then be constructed using induction for $\mathbf{2}$:

$$(a, b) :\equiv \mathsf{ind}_{\mathbf{2}}(\mathsf{rec}_{\mathbf{2}}(\mathcal{U}, A, B), a, b)$$

while the projections are straightforward applications

$$\begin{aligned} \mathsf{pr}_1(p) &:\equiv p(1_{\mathbf{2}}) \\ \mathsf{pr}_2(p) &:\equiv p(0_{\mathbf{2}}). \end{aligned}$$

The derivation of the induction principle for binary products defined in this way is a bit more involved, and requires function extensionality, which we will introduce in §2.9. Moreover, we do not get the same judgmental equalities; see Exercise 1.6. This is a recurrent issue when encoding one type as another; we will return to it in §5.5.

We may occasionally refer to the elements $1_{\mathbf{2}}$ and $0_{\mathbf{2}}$ of $\mathbf{2}$ as "true" and "false" respectively. However, note that unlike in classical mathematics, we do not use elements of $\mathbf{2}$ as truth values or as propositions. (Instead we identify propositions with types; see §1.11.) In particular, the type $A \to \mathbf{2}$ is not generally the power set of A; it represents only the "decidable" subsets of A.

1.9 The natural numbers

The rules we have introduced so far do not allow us to construct any infinite types. The simplest infinite type we can think of (and which is also, of course, extremely useful) is the type $\mathbb{N} : \mathcal{U}$ of natural numbers. The elements of $\mathbb{N}$ are constructed using $0 : \mathbb{N}$ and the successor operation $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$. When denoting natural numbers, we adopt the usual decimal notation $1 :\equiv \mathsf{succ}(0)$, $2 :\equiv \mathsf{succ}(1)$, $3 :\equiv \mathsf{succ}(2)$, ...

The essential property of the natural numbers is that we can define functions by recursion and perform proofs by induction — where now the words "recursion" and "induction" have a more familiar meaning. To construct a non-dependent function $f : \mathbb{N} \to C$ out of the natural numbers by recursion, it is enough to provide a starting point $c_0 : C$ and a "next step" function $c_s : \mathbb{N} \to C \to C$. This gives rise to f with the defining equations

$$\begin{aligned} f(0) &:\equiv c_0 \\ f(\mathsf{succ}(n)) &:\equiv c_s(n, f(n)). \end{aligned}$$

We say that f is defined by **primitive recursion**.

As an example, we look at how to define a function on natural numbers which doubles its argument. In this case we have $C :\equiv \mathbb{N}$. We first need to supply the value of $\mathsf{double}(0)$, which is easy: we put $c_0 :\equiv 0$. Next, to compute the value of $\mathsf{double}(\mathsf{succ}(n))$ for a natural number n, we first compute the value of $\mathsf{double}(n)$ and then perform the successor operation twice. This is captured by the recurrence $c_s(n, y) :\equiv \mathsf{succ}(\mathsf{succ}(y))$. Note that the second argument y of c_s stands for the result of the *recursive call* $\mathsf{double}(n)$.

Defining $\mathsf{double} : \mathbb{N} \to \mathbb{N}$ by primitive recursion in this way, therefore, we obtain the defining equations:

$$\begin{aligned} \mathsf{double}(0) &:\equiv 0 \\ \mathsf{double}(\mathsf{succ}(n)) &:\equiv \mathsf{succ}(\mathsf{succ}(\mathsf{double}(n))). \end{aligned}$$

This indeed has the correct computational behavior: for example, we have

$$\begin{aligned}
\mathsf{double}(2) &\equiv \mathsf{double}(\mathsf{succ}(\mathsf{succ}(0))) \\
&\equiv c_s(\mathsf{succ}(0), \mathsf{double}(\mathsf{succ}(0))) \\
&\equiv \mathsf{succ}(\mathsf{succ}(\mathsf{double}(\mathsf{succ}(0)))) \\
&\equiv \mathsf{succ}(\mathsf{succ}(c_s(0, \mathsf{double}(0)))) \\
&\equiv \mathsf{succ}(\mathsf{succ}(\mathsf{succ}(\mathsf{succ}(\mathsf{double}(0))))) \\
&\equiv \mathsf{succ}(\mathsf{succ}(\mathsf{succ}(\mathsf{succ}(c_0)))) \\
&\equiv \mathsf{succ}(\mathsf{succ}(\mathsf{succ}(\mathsf{succ}(0)))) \\
&\equiv 4.
\end{aligned}$$

We can define multi-variable functions by primitive recursion as well, by currying and allowing C to be a function type. For example, we define addition $\mathsf{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ with $C :\equiv \mathbb{N} \to \mathbb{N}$ and the following data:

$$\begin{aligned}
\mathsf{add}_0 &: \mathbb{N} \to \mathbb{N} \\
\mathsf{add}_0(n) &:\equiv n \\
\mathsf{add}_s &: \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}) \\
\mathsf{add}_s(n, g, m) &:\equiv \mathsf{succ}(g(m))
\end{aligned}$$

We thus obtain $\mathsf{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ satisfying the definitional equalities

$$\begin{aligned}
\mathsf{add}(0, n) &\equiv n \\
\mathsf{add}(\mathsf{succ}(m), n) &\equiv \mathsf{succ}(\mathsf{add}(m, n))
\end{aligned}$$

As usual, we write $\mathsf{add}(m, n)$ as $m + n$. The reader is invited to verify that $2 + 2 \equiv 4$.

As in previous cases, we can package the principle of primitive recursion into a recursor:

$$\mathsf{rec}_{\mathbb{N}} : \prod_{(C:\mathcal{U})} C \to (\mathbb{N} \to C \to C) \to \mathbb{N} \to C$$

with the defining equations

$$\begin{aligned}
\mathsf{rec}_{\mathbb{N}}(C, c_0, c_s, 0) &:\equiv c_0 \\
\mathsf{rec}_{\mathbb{N}}(C, c_0, c_s, \mathsf{succ}(n)) &:\equiv c_s(n, \mathsf{rec}_{\mathbb{N}}(C, c_0, c_s, n))
\end{aligned}$$

Using $\mathsf{rec}_{\mathbb{N}}$ we can present double and add as follows:

$$\mathsf{double} :\equiv \mathsf{rec}_{\mathbb{N}}\big(\mathbb{N},\ 0,\ \lambda n.\, \lambda y.\, \mathsf{succ}(\mathsf{succ}(y))\big) \tag{1.9.1}$$

$$\mathsf{add} :\equiv \mathsf{rec}_{\mathbb{N}}\big(\mathbb{N} \to \mathbb{N},\ \lambda n.\, n,\ \lambda n.\, \lambda g.\, \lambda m.\, \mathsf{succ}(g(m))\big) \tag{1.9.2}$$

Of course, all functions definable only using the primitive recursion principle will be *computable*. (The presence of higher function types does, however, mean we can define more than the usual

primitive recursive functions; see e.g. Exercise 1.10.) This is appropriate in constructive mathematics; in §§3.4 and 3.8 we will see how to augment type theory so that we can define more general mathematical functions.

We now follow the same approach as for other types, generalizing primitive recursion to dependent functions to obtain an *induction principle*. Thus, assume as given a family $C : \mathbb{N} \to \mathcal{U}$, an element $c_0 : C(0)$, and a function $c_s : \prod_{(n:\mathbb{N})} C(n) \to C(\mathsf{succ}(n))$; then we can construct $f : \prod_{(n:\mathbb{N})} C(n)$ with the defining equations:

$$\begin{aligned} f(0) &:\equiv c_0 \\ f(\mathsf{succ}(n)) &:\equiv c_s(n, f(n)) \end{aligned}$$

We can also package this into a single function

$$\mathsf{ind}_{\mathbb{N}} : \prod_{(C:\mathbb{N}\to\mathcal{U})} C(0) \to \left(\textstyle\prod_{(n:\mathbb{N})} C(n) \to C(\mathsf{succ}(n))\right) \to \textstyle\prod_{(n:\mathbb{N})} C(n)$$

with the defining equations

$$\begin{aligned} \mathsf{ind}_{\mathbb{N}}(C, c_0, c_s, 0) &:\equiv c_0 \\ \mathsf{ind}_{\mathbb{N}}(C, c_0, c_s, \mathsf{succ}(n)) &:\equiv c_s(n, \mathsf{ind}_{\mathbb{N}}(C, C_0, c_s, n)) \end{aligned}$$

Here we finally see the connection to the classical notion of proof by induction. Recall that in type theory we represent propositions by types, and proving a proposition by inhabiting the corresponding type. In particular, a *property* of natural numbers is represented by a family of types $P : \mathbb{N} \to \mathcal{U}$. From this point of view, the above induction principle says that if we can prove $P(0)$, and if for any n we can prove $P(\mathsf{succ}(n))$ assuming $P(n)$, then we have $P(n)$ for all n. This is, of course, exactly the usual principle of proof by induction on natural numbers.

As an example consider how we might represent an explicit proof that $+$ is associative. (We will not actually write out proofs in this style, but it serves as a useful example for understanding how induction is represented formally in type theory.) To derive

$$\mathsf{ass} : \prod_{i,j,k:\mathbb{N}} i + (j + k) = (i + j) + k,$$

it is sufficient to supply

$$\mathsf{ass}_0 : \prod_{j,k:\mathbb{N}} 0 + (j + k) = (0 + j) + k$$

and

$$\mathsf{ass}_s : \prod_{i:\mathbb{N}} \left(\prod_{j,k:\mathbb{N}} i + (j + k) = (i + j) + k \right) \to \prod_{j,k:\mathbb{N}} \mathsf{succ}(i) + (j + k) = (\mathsf{succ}(i) + j) + k.$$

To derive ass_0, recall that $0 + n \equiv n$, and hence $0 + (j + k) \equiv j + k \equiv (0 + j) + k$. Hence we can just set

$$ass_0(j, k) :\equiv \mathsf{refl}_{j+k}$$

For ass_s, recall that the definition of $+$ gives $\mathsf{succ}(m) + n \equiv \mathsf{succ}(m+n)$, and hence

$$\begin{array}{rcll} \mathsf{succ}(i) + (j+k) & \equiv & \mathsf{succ}(i + (j+k)) & \text{and} \\ (\mathsf{succ}(i) + j) + k & \equiv & \mathsf{succ}((i+j)+k). & \end{array}$$

Thus, the output type of ass_s is equivalently $\mathsf{succ}(i + (j+k)) = \mathsf{succ}((i+j)+k)$. But its input (the "inductive hypothesis") yields $i + (j+k) = (i+j) + k$, so it suffices to invoke the fact that if two natural numbers are equal, then so are their successors. (We will prove this obvious fact formally from the induction principle of identity types in Lemma 2.2.1.) We call this latter fact $\mathsf{ap}_{\mathsf{succ}} : (m =_{\mathbb{N}} n) \to (\mathsf{succ}(m) =_{\mathbb{N}} \mathsf{succ}(n))$, so we can define

$$\mathsf{ass}_s(i,h,j,k) :\equiv \mathsf{ap}_{\mathsf{succ}}(h(j,k)).$$

Putting these together with $\mathsf{ind}_{\mathbb{N}}$, we obtain a proof of associativity.

1.10 Pattern matching and recursion

The natural numbers introduce an additional subtlety over the types considered up until now. In the case of coproducts, for instance, we could define a function $f : A + B \to C$ either with the recursor:

$$f :\equiv \mathsf{rec}_{A+B}(C, g_0, g_1)$$

or by giving the defining equations:

$$\begin{aligned} f(\mathsf{inl}(a)) &:\equiv g_0(a) \\ f(\mathsf{inr}(b)) &:\equiv g_1(b). \end{aligned}$$

To go from the former expression of f to the latter, we simply use the computation rules for the recursor. Conversely, given any defining equations

$$\begin{aligned} f(\mathsf{inl}(a)) &:\equiv c_0 \\ f(\mathsf{inr}(b)) &:\equiv c_1 \end{aligned}$$

where c_0 and c_1 are expressions that may involve the variables a and b, we can express these equations equivalently in terms of the recursor by using λ-abstraction:

$$f :\equiv \mathsf{rec}_{A+B}(C, \lambda a.\, c_0, \lambda b.\, c_1).$$

In the case of the natural numbers, however, the "defining equations" of a function such as double:

$$\mathsf{double}(0) :\equiv 0 \tag{1.10.1}$$

$$\mathsf{double}(\mathsf{succ}(n)) :\equiv \mathsf{succ}(\mathsf{succ}(\mathsf{double}(n))) \tag{1.10.2}$$

involve *the function* double *itself* on the right-hand side. However, we would still like to be able to give these equations, rather than (1.9.1), as the definition of double, since they are much more

convenient and readable. The solution is to read the expression "$\mathsf{double}(n)$" on the right-hand side of (1.10.2) as standing in for the result of the recursive call, which in a definition of the form $\mathsf{double} :\equiv \mathsf{rec}_{\mathbb{N}}(\mathbb{N}, c_0, c_s)$ would be the second argument of c_s.

More generally, if we have a "definition" of a function $f : \mathbb{N} \to C$ such as

$$f(0) :\equiv c_0$$
$$f(\mathsf{succ}(n)) :\equiv c_s$$

where c_0 is an expression of type C, and c_s is an expression of type C which may involve the variable n and also the symbol "$f(n)$", we may translate it to a definition

$$f :\equiv \mathsf{rec}_{\mathbb{N}}(C,\ c_0,\ \lambda n.\, \lambda r.\, c_s')$$

where c_s' is obtained from c_s by replacing all occurrences of "$f(n)$" by the new variable r.

This style of defining functions by recursion (or, more generally, dependent functions by induction) is so convenient that we frequently adopt it. It is called definition by **pattern matching**. Of course, it is very similar to how a computer programmer may define a recursive function with a body that literally contains recursive calls to itself. However, unlike the programmer, we are restricted in what sort of recursive calls we can make: in order for such a definition to be re-expressible using the recursion principle, the function f being defined can only appear in the body of $f(\mathsf{succ}(n))$ as part of the composite symbol "$f(n)$". Otherwise, we could write nonsense functions such as

$$f(0) :\equiv 0$$
$$f(\mathsf{succ}(n)) :\equiv f(\mathsf{succ}(\mathsf{succ}(n))).$$

If a programmer wrote such a function, it would simply call itself forever on any positive input, going into an infinite loop and never returning a value. In mathematics, however, to be worthy of the name, a *function* must always associate a unique output value to every input value, so this would be unacceptable.

This point will be even more important when we introduce more complicated inductive types in Chapters 5, 6 and 11. Whenever we introduce a new kind of inductive definition, we always begin by deriving its induction principle. Only then do we introduce an appropriate sort of "pattern matching" which can be justified as a shorthand for the induction principle.

1.11 Propositions as types

As mentioned in the introduction, to show that a proposition is true in type theory corresponds to exhibiting an element of the type corresponding to that proposition. We regard the elements of this type as *evidence* or *witnesses* that the proposition is true. (They are sometimes even called *proofs*, but this terminology can be misleading, so we generally avoid it.) In general, however, we will not construct witnesses explicitly; instead we present the proofs in ordinary mathematical prose, in such a way that *in principle* they could be translated into an element of a type. This is no different from reasoning in classical set theory, where we don't expect to see an explicit derivation using the rules of predicate logic and the axioms of set theory.

However, the type-theoretic perspective on proofs is nevertheless different in important ways. The basic principle of the logic of type theory is that a proposition is not merely true or false, but rather can be seen as classifying all possible witnesses of its truth. Under this conception, proofs are not just the means by which mathematics is communicated, but rather are mathematical objects in their own right, on a par with more familiar objects such as numbers, mappings, groups, and so on. Thus, since types classify the available mathematical objects and govern how they interact, propositions are nothing but special types — namely, types that classify proofs.

The basic observation which makes this identification feasible is that we have the following natural correspondence between *logical* operations on propositions, expressed in English, and *type-theoretic* operations on their corresponding types of witnesses.

English	Type Theory
True	$\mathbf{1}$
False	$\mathbf{0}$
A and B	$A \times B$
A or B	$A + B$
If A then B	$A \to B$
A if and only if B	$(A \to B) \times (B \to A)$
Not A	$A \to \mathbf{0}$

The point of the correspondence is that in each case, the rules for constructing and using elements of the type on the right correspond to the rules for reasoning about the proposition on the left. For instance, the basic way to prove a statement of the form "A and B" is to prove A and also prove B, while the basic way to construct an element of $A \times B$ is as a pair (a, b), where a is an element (or witness) of A and b is an element (or witness) of B. Similarly, the basic way to prove an implication "if A then B" is to assume A and prove B, while the basic way to construct an element of $A \to B$ is to give an expression which denotes an element (witness) of B which may involve an unspecified variable element (witness) of type A.

This gives us a way to translate propositions, and proofs, of them, written in English into types, and elements of them, in type theory. For example, suppose we want to prove the tautology

$$\textit{"If not } A \textit{ and not } B\textit{, then not } (A \textit{ or } B)\textit{"}. \tag{1.11.1}$$

An ordinary English proof of this fact might go as follows.

> Suppose not A and not B, and also suppose A or B; we will derive a contradiction. There are two cases. If A holds, then since not A, we have a contradiction. Similarly, if B holds, then since not B, we also have a contradiction. Thus we have a contradiction in either case, so not (A or B).

Now, the type corresponding to our tautology (1.11.1), according to the rules given above, is

$$(A \to \mathbf{0}) \times (B \to \mathbf{0}) \to (A + B \to \mathbf{0}) \tag{1.11.2}$$

so we should be able to translate the above proof into an element of this type.

As an example of how such a translation works, let us describe how a mathematician reading the above English proof might simultaneously construct, in his or her head, an element of (1.11.2). The introductory phrase "Suppose not A and not B" translates into defining a function, with an implicit application of the recursion principle for the cartesian product in its domain $(A \to \mathbf{0}) \times (B \to \mathbf{0})$. This introduces unnamed variables (hypotheses) x and y of types $A \to \mathbf{0}$ and $B \to \mathbf{0}$. At this point our partial definition of an element of (1.11.2) can be written as

$$f((x,y)) :\equiv \ \Box \ : A + B \to \mathbf{0}$$

with a "hole" $\Box$ of type $A + B \to \mathbf{0}$ indicating what remains to be done. (We could equivalently write $f :\equiv \mathsf{rec}_{(A\to\mathbf{0})\times(B\to\mathbf{0})}(A + B \to \mathbf{0}, \lambda x.\, \lambda y.\, \Box)$, using the recursor instead of pattern matching.) The next phrase "also suppose A or B; we will derive a contradiction" indicates filling this hole by a function definition, introducing another unnamed hypothesis $z : A + B$, leading to the proof state:

$$f((x,y))(z) :\equiv \ \Box \ : \mathbf{0}$$

Now saying "there are two cases" indicates a case split, i.e. an application of the recursion principle for the coproduct $A + B$. If we write this using the recursor, it would be

$$f((x,y))(z) :\equiv \mathsf{rec}_{A+B}(\mathbf{0}, \lambda a.\, \Box, \lambda b.\, \Box, z)$$

while if we write it using pattern matching, it would be

$$\begin{aligned} f((x,y))(\mathsf{inl}(a)) &:\equiv \ \Box \ : \mathbf{0} \\ f((x,y))(\mathsf{inr}(b)) &:\equiv \ \Box \ : \mathbf{0} \end{aligned}$$

Note that in both cases we now have two "holes" of type $\mathbf{0}$ to fill in, corresponding to the two cases where we have to derive a contradiction. Finally, the conclusion of a contradiction from $a : A$ and $x : A \to \mathbf{0}$ is simply application of the function x to a, and similarly in the other case; thus our eventual definition is

$$\begin{aligned} f((x,y))(\mathsf{inl}(a)) &:\equiv x(a) \\ f((x,y))(\mathsf{inr}(b)) &:\equiv y(b). \end{aligned}$$

As an exercise, you should verify the converse tautology *"If not (A or B), then (not A) and (not B)"* by exhibiting an element of

$$((A + B) \to \mathbf{0}) \to (A \to \mathbf{0}) \times (B \to \mathbf{0}),$$

for any types A and B, using the rules we have just introduced.

However, not all classical tautologies hold under this interpretation. For example, the rule *"If not (A and B), then (not A) or (not B)"* is not valid: we cannot, in general, construct an element of the corresponding type

$$((A \times B) \to \mathbf{0}) \to (A \to \mathbf{0}) + (B \to \mathbf{0}).$$

This reflects the fact that the "natural" logic of type theory is *constructive*, as discussed in the introduction. Formally, this means it does not include certain classical principles, such as the law of excluded middle (LEM) or proof by contradiction[8], and others which depend on them.

Philosophically, constructive logic is so-called because it confines itself to constructions that can be carried out *effectively*, which is to say those with a computational meaning. Without being too precise, this means there is some sort of algorithm specifying, step-by-step, how to build an object (and, as a special case, how to see that a theorem is true). This requires omission of LEM, since there is no *effective* procedure for deciding whether a proposition is true or false.

The constructivity of type-theoretic logic means it has an intrinsic computational meaning, which is of interest to computer scientists. It also means that type theory provides *axiomatic freedom*. For example, while by default there is no construction witnessing LEM, the logic is still compatible with the existence of one (see §3.4). Thus, because type theory does not *deny* LEM, we may consistently add it as an assumption, and work conventionally without restriction. In this respect, type theory enriches, rather than constrains, conventional mathematical practice.

We encourage the reader who is unfamiliar with constructive logic to work through some more examples as a means of getting familiar with it. See Exercises 1.12 and 1.13 for some suggestions.

So far we have discussed only propositional logic. Now we consider *predicate* logic, where in addition to logical connectives like "and" and "or" we have quantifiers "there exists" and "for all". In this case, types play a dual role: they serve as propositions and also as types in the conventional sense, i.e., domains we quantify over. A predicate over a type A is represented as a family $P : A \to \mathcal{U}$, assigning to every element $a : A$ a type $P(a)$ corresponding to the proposition that P holds for a. We now extend the above translation with an explanation of the quantifiers:

English	Type Theory
For all $x : A$, $P(x)$ holds	$\prod_{(x:A)} P(x)$
There exists $x : A$ such that $P(x)$	$\sum_{(x:A)} P(x)$

As before, we can show that tautologies of (constructive) predicate logic translate into inhabited types. For example, *If for all $x : A$, $P(x)$ and $Q(x)$ then (for all $x : A$, $P(x)$) and (for all $x : A$, $Q(x)$)* translates to

$$(\textstyle\prod_{(x:A)} P(x) \times Q(x)) \to (\prod_{(x:A)} P(x)) \times (\prod_{(x:A)} Q(x)).$$

An informal proof of this tautology might go as follows:

> Suppose for all x, $P(x)$ and $Q(x)$. First, we suppose given x and prove $P(x)$. By assumption, we have $P(x)$ and $Q(x)$, and hence we have $P(x)$. Second, we suppose given x and prove $Q(x)$. Again by assumption, we have $P(x)$ and $Q(x)$, and hence we have $Q(x)$.

[8] Note that the pattern "assume A and derive a contradiction" is perfectly valid in constructive logic as a way (in fact, *the* way) to prove "not A"; indeed, we used it in our example above. The form of "proof by contradiction" which is disallowed is "assume not A and derive a contradiction" as a way of proving A.

The first sentence begins defining an implication as a function, by introducing a witness for its hypothesis:

$$f(p) :\equiv \ \square \ : (\textstyle\prod_{(x:A)} P(x)) \times (\prod_{(x:A)} Q(x))$$

At this point there is an implicit use of the pairing constructor to produce an element of a product type, which is somewhat signposted in this example by the words "first" and "second":

$$f(p) :\equiv \Big(\ \square \ : \textstyle\prod_{(x:A)} P(x) \ , \ \square \ : \prod_{(x:A)} Q(x) \ \Big).$$

The phrase "we suppose given x and prove $P(x)$" now indicates defining a *dependent* function in the usual way, introducing a variable for its input. Since this is inside a pairing constructor, it is natural to write it as a λ-abstraction:

$$f(p) :\equiv \Big(\ \lambda x. \ (\square \ : P(x)) \ , \ \square \ : \textstyle\prod_{(x:A)} Q(x) \ \Big).$$

Now "we have $P(x)$ and $Q(x)$" invokes the hypothesis, obtaining $p(x) : P(x) \times Q(x)$, and "hence we have $P(x)$" implicitly applies the appropriate projection:

$$f(p) :\equiv \Big(\ \lambda x. \, \mathsf{pr}_1(p(x)) \ , \ \square \ : \textstyle\prod_{(x:A)} Q(x) \ \Big).$$

The next two sentences fill the other hole in the obvious way:

$$f(p) :\equiv \Big(\ \lambda x. \, \mathsf{pr}_1(p(x)) \ , \ \lambda x. \, \mathsf{pr}_2(p(x)) \ \Big).$$

Of course, the English proofs we have been using as examples are much more verbose than those that mathematicians usually use in practice; they are more like the sort of language one uses in an "introduction to proofs" class. The practicing mathematician has learned to fill in the gaps, so in practice we can omit plenty of details; and we will generally do so. The criterion of validity for proofs, however, is always that they can be translated back into the construction of an element of the corresponding type.

As a more concrete example, consider how to define inequalities of natural numbers. One natural definition is that $n \leq m$ if there exists a $k : \mathbb{N}$ such that $n + k = m$. (This uses again the identity types that we will introduce in the next section, but we will not need very much about them.) Under the propositions-as-types translation, this would yield:

$$(n \leq m) :\equiv \sum_{k:\mathbb{N}} (n + k = m).$$

The reader is invited to prove the familiar properties of $\leq$ from this definition. For strict inequality, there are a couple of natural choices, such as

$$(n < m) :\equiv \sum_{k:\mathbb{N}} (n + \mathsf{succ}(k) = m)$$

or

$$(n < m) :\equiv (n \leq m) \times \neg(n = m).$$

The former is more natural in constructive mathematics, but in this case it is actually equivalent to the latter, since $\mathbb{N}$ has "decidable equality" (see Theorem 7.2.7).

Note that we can use the universes in type theory to represent "higher order logic" — that is, we can quantify over all propositions or over all predicates. For example, we can represent the proposition *for all properties* $P : A \to \mathcal{U}$, *if* $P(a)$ *then* $P(b)$ as

$$\prod_{P:A\to\mathcal{U}} P(a) \to P(b)$$

where $A : \mathcal{U}$ and $a, b : A$. However, *a priori* this proposition lives in a different, higher, universe than the propositions we are quantifying over; that is

$$\Big(\prod_{P:A\to\mathcal{U}_i} P(a) \to P(b)\Big) : \mathcal{U}_{i+1}.$$

We will return to this question in §3.5.

We have descibed here a "proof-relevant" translation of propositions, where the proofs of disjunctions and existential statements carry some information. For instance, if we have an inhabitant of $A + B$, regarded as a witness of "A or B", then we know whether it came from A or from B. Similarly, if we have an inhabitant of $\sum_{(x:A)} P(x)$, regarded as a witness of "there exists $x : A$ such that $P(x)$", then we know what the element x is (it is the first component of the dependent pair). In §3.7 we will introduce a modification to this logic that is sometimes appropriate, in which this additional information is discarded.

1.12 Identity types

While the previous constructions can be seen as generalisations of standard set theoretic constructions, our way of handling identity seems to be a particular feature of type theory. According to the propositions-as-types conception, the *proposition* that two elements of the same type $a, b : A$ are equal must correspond to some *type*. Since this proposition depends on what a and b are, these *equality types* or *identity types* must be type families dependent on two copies of A.

We may write the family as $\mathsf{Id}_A : A \to A \to \mathcal{U}$, so that $\mathsf{Id}_A(a, b)$ is the type representing the proposition of equality between a and b. Once we are familiar with propositions-as-types, however, it is convenient to also use the standard equality symbol for this; thus "$a = b$" will also be a notation for the *type* $\mathsf{Id}_A(a, b)$ corresponding to the proposition that a equals b. For clarity, we may also write "$a =_A b$" to specify the type A.

Just as we remarked in §1.11 that the propositions-as-types versions of "or" and "there exists" can include more information than just the fact that the proposition is true, nothing prevents an element of the type $a = b$ from also including more information. Indeed, this is the cornerstone of the homotopical interpretation, where we regard witnesses of $a = b$ as *paths* or *equivalences* between a and b in the space A. Just as there can be more than one path between two points of a space, there can be more than one witness that two objects are equal. Put differently, we may regard $a = b$ as the type of *identifications* of a and b, and there may be many different ways in which a and b can be identified. We will return to the interpretation in Chapter 2; for now we focus on the basic rules for the identity type.

Given a type $A : \mathcal{U}$ and two elements $a, b : A$, we can form the type $a =_A b : \mathcal{U}$ in the same universe. The basic way to construct an element of $a = b$ is to know that a and b are the same. Thus, we have a dependent function

$$\mathsf{refl} : \prod_{a:A} (a =_A a)$$

called **reflexivity**, which says that every element of A is equal to itself (in a specified way).

In particular, this means that if a and b are *judgmentally* equal, $a \equiv b$, then we also have an element $\mathsf{refl}_a : a =_A b$. This is well-typed because $a \equiv b$ means that also the type $a =_A b$ is judgmentally equal to $a = a$, which is the type of refl_a.

The induction principle for the identity types is one of the most subtle parts of type theory, and crucial to the homotopy interpretation. We begin by considering an important consequence of it, the principle that "equals may be substituted for equals," as expressed by the following:

Indiscernability of identicals: For every family

$$C : A \to \mathcal{U}$$

there is a function

$$f : \prod_{(x,y:A)} \prod_{(p:x=_A y)} C(x) \to C(y)$$

such that

$$f(x, x, \mathsf{refl}_x) :\equiv \mathsf{id}_{C(x)}.$$

This says that every family of types C respects equality, in the sense that applying C to *equal* elements of A also results in a function between the resulting types. The displayed equality states that the function associated to reflexivity is the identity function (and we shall see that, in general, the function $f(x, y, p) : C(x) \to C(y)$ is always an equivalence of types).

Indiscernability of identicals can be regarded as a recursion principle for the identity type, analogous to those given for booleans and natural numbers above. It gives a mapping property of $x =_A y$ with respect to certain other reflexive, binary relations on A, namely those of the form $C(x) \to C(y)$. We could also formulate a more general recursion principle with respect to reflexive relations of the more general form $C(x, y)$. However, in order to fully characterize the identity type, we must generalize it to an induction principle, which not only considers maps out of $x =_A y$ but also families over it. Put differently, we consider not only allowing equals to be substituted for equals, but also taking into account the evidence p for the equality.

1.12.1 Path induction

The induction principle for the identity type is called **path induction** in view of the homotopical interpretation to be explained in the introduction to Chapter 2. It can be seen as stating that the family of identity types is freely generated by the elements of the form $\mathsf{refl}_x : x = x$.

Path induction: Given a family

$$C : \prod_{(x,y:A)} \prod_{(p:x=_A y)} \mathcal{U}$$

and a function

$$c : \prod_{x:A} C(x, x, \mathsf{refl}_x),$$

there is a function

$$f : \prod_{(x,y:A)} \prod_{(p:x=_A y)} C(x, y, p)$$

such that

$$f(x, x, \mathsf{refl}_x) :\equiv c(x).$$

To understand this principle, consider first the simpler case when C does not depend on p. Then we have $C : A \to A \to \mathcal{U}$, which we may regard as a predicate depending on two elements of A. We are interested in knowing when the proposition $C(x, y)$ holds for some pair of elements $x, y : A$. In this case, the hypothesis of path induction says that we know $C(x, x)$ holds for all $x : A$, i.e. that if we evaluate C at the pair x, x, we get a true proposition — so C is a reflexive relation. The conclusion then tells us that $C(x, y)$ holds whenever $x = y$. This is exactly the more general recursion principle for reflexive relations mentioned above.

The general, inductive form of the rule allows C to also depend on the witness $p : x = y$ to the identity between x and y. In the premise, we not only replace x, y by x, x, but also simultaneously replace p by reflexivity: to prove a property for all elements x, y and paths $p : x = y$ between them, it suffices to consider all the cases where the elements are x, x and the path is $\mathsf{refl}_x : x = x$. If we were viewing types just as sets, it would be unclear what this buys us, but since there may be many different identifications $p : x = y$ between x and y, it makes sense to keep track of them in considering families over the type $x =_A y$. In Chapter 2 we will see that this is very important to the homotopy interpretation.

If we package up path induction into a single function, it takes the form:

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

with the equality

$$\mathsf{ind}_{=_A}(C, c, x, x, \mathsf{refl}_x) :\equiv c(x)$$

The function $\mathsf{ind}_{=_A}$ is traditionally called J. We leave it as an easy exercise to show that indiscernability of identicals follows from path induction.

Given a proof $p : a = b$, path induction requires you to replace *both* a and b with the same unknown element x; thus in order to define an element of a family C, for all pairs of elements of A, it suffices to define it on the diagonal. In some proofs, however, it is simpler to make use of an equation $p : a = b$ by replacing all occurrences of b with a (or vice versa), because it is sometimes easier to do the remainder of the proof for the specific element a mentioned in the equality than for a general unknown x. This motivates a second induction principle for identity types, which says that the family of types $a =_A x$ is generated by the element $\mathsf{refl}_a : a = a$. As we show below, this second principle is equivalent to the first; it is just sometimes a more convenient formulation.

Based path induction: Fix an element $a : A$, and suppose given a family

$$C : \prod_{(x:A)} \prod_{(p:a=_A x)} \mathcal{U}$$

and an element

$$c : C(a, \mathsf{refl}_a).$$

Then we obtain a function

$$f : \prod_{(x:A)} \prod_{(p:a=x)} C(x, p)$$

such that

$$f(a, \mathsf{refl}_a) :\equiv c.$$

Here, $C(x, p)$ is a family of types, where x is an element of A and p is an element of the identity type $a =_A x$, for fixed a in A. The based path induction principle says that to define an element of this family for all x and p, it suffices to consider just the case where x is a and p is $\mathsf{refl}_a : a = a$.

Packaged as a function, based path induction becomes:

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

with the equality

$$\mathsf{ind}'_{=_A}(a, C, c, a, \mathsf{refl}_a) :\equiv c.$$

Below, we show that path induction and based path induction are equivalent. Because of this, we will sometimes be sloppy and also refer to based path induction simply as "path induction," relying on the reader to infer which principle is meant from the form of the proof.

Remark 1.12.1. Intuitively, the induction principle for the natural numbers expresses the fact that the only natural numbers are 0 and $\mathsf{succ}(n)$, so if we prove a property for these cases, then we have proved it for all natural numbers. Applying this same reading to path induction, we might loosely say that path induction expresses the fact that the only path is refl, so if we prove a property for reflexivity, then we have proved it for all paths. However, this reading is quite confusing in the context of the homotopy interpretation of paths, where there may be many different ways in which two elements a and b can be identified, and therefore many different elements of the identity type! How can there be many different paths, but at the same time we have an induction principle asserting that the only path is reflexivity?

The key observation is that it is not the identity *type* that is inductively defined, but the identity *family*. In particular, path induction says that the *family* of types $(x =_A y)$, as x, y vary over all elements of A, is inductively defined by the elements of the form refl_x. This means that to give an element of any other family $C(x, y, p)$ dependent on a *generic* element (x, y, p) of the identity family, it suffices to consider the cases of the form (x, x, refl_x). In the homotopy interpretation, this says that the type of triples (x, y, p), where x and y are the endpoints of the path p (in other words, the Σ-type $\sum_{(x,y:A)}(x = y)$), is inductively generated by the constant loops at each point x. In homotopy theory, the space corresponding to $\sum_{(x,y:A)}(x = y)$ is the

free path space — the space of paths in A whose endpoints may vary — and it is in fact the case that any point of this space is homotopic to the constant loop at some point, since we can simply retract one of its endpoints along the given path.

Similarly, based path induction says that for a fixed $a : A$, the *family* of types $(a =_A y)$, as y varies over all elements of A, is inductively defined by the element refl_a. Thus, to give an element of any other family $C(y, p)$ dependent on a generic element (y, p) of this family, it suffices to consider the case (a, refl_a). Homotopically, this expresses the fact that the space of paths starting at some chosen point (the *based path space* at that point, which type-theoretically is $\sum_{(y:A)}(a = y)$) is contractible to the constant loop on the chosen point. Note that according to propositions-as-types, the type $\sum_{(y:A)}(a = y)$ can be regarded as "the type of all elements of A which are equal to a", a type-theoretic version of the "singleton subset" $\{a\}$.

Neither of these two principles provides a way to give an element of a family $C(p)$ where p has *two fixed endpoints* a and b. In particular, for a family $C(p : a =_A a)$ dependent on a loop, we *cannot* apply path induction and consider only the case for $C(\mathsf{refl}_a)$, and consequently, we cannot prove that all loops are reflexivity. Thus, inductively defining the identity family does not prohibit non-reflexivity paths in specific instances of the identity type. In other words, a path $p : x = x$ may not be equal to reflexivity as an element of $(x = x)$, but the pair (x, p) will nevertheless be equal to the pair (x, refl_x) as elements of $\sum_{(y:A)}(x = y)$.

1.12.2 Equivalence of path induction and based path induction

The two induction principles for the identity type introduced above are equivalent. It is easy to see that path induction follows from based path induction principle. Indeed, let us assume the premises of path induction:

$$\begin{aligned} C &: \prod_{(x,y:A)} \prod_{(p:x=_A y)} \mathcal{U} \\ c &: \prod_{x:A} C(x, x, \mathsf{refl}_x). \end{aligned}$$

Now, given an element $x : A$, we can instantiate both of the above, obtaining

$$\begin{aligned} C' &: \prod_{(y:A)} \prod_{(p:x=_A y)} \mathcal{U} \\ C' &:\equiv C(x) \\ c' &: C'(x, \mathsf{refl}_x) \\ c' &:\equiv c(x). \end{aligned}$$

Clearly, C' and c' match the premises of based path induction and hence we can construct

$$g : \prod_{(y:A)} \prod_{(p:x=y)} C'(y, p)$$

with the defining equality

$$g(x, \mathsf{refl}_x) :\equiv c'.$$

Now we observe that g's codomain is equal to $C(x, y, p)$. Thus, discharging our assumption $x : A$, we can derive a function

$$f : \prod_{(x,y:A)} \prod_{(p:x=_A y)} C(x, y, p)$$

with the required judgmental equality $f(x, x, \mathsf{refl}_x) \equiv g(x, \mathsf{refl}_x) :\equiv c' :\equiv c(x)$.

Another proof of this fact is to observe that any such f can be obtained as an instance of $\mathsf{ind}'_{=_A}$ so it suffices to define $\mathsf{ind}'_{=_A}$ in terms of $\mathsf{ind}_{=_A}$ as

$$\mathsf{ind}'_{=_A}(C, c, x, y, p) :\equiv \mathsf{ind}_{=_A}(x, C(x), c(x), y, p).$$

The other direction is a bit trickier; it is not clear how we can use a particular instance of path induction to derive a particular instance of based path induction. What we can do instead is to construct one instance of path induction which shows all possible instantiations of based path induction at once. Define

$$\begin{aligned} D &: \prod_{(x,y:A)} \prod_{(p:x=_A y)} \mathcal{U} \\ D(x, y, p) &:\equiv \prod_{C:\prod_{(z:A)} \prod_{(p:x=_A z)} \mathcal{U}} C(x, \mathsf{refl}_x) \to C(y, p). \end{aligned}$$

Then we can construct the function

$$\begin{aligned} d &: \prod_{x:A} D(x, x, \mathsf{refl}_x) \\ d &:\equiv \lambda x.\, \lambda C.\, \lambda(c : C(x, \mathsf{refl}_x)).\, c \end{aligned}$$

and hence using path induction obtain

$$f : \prod_{(x,y:A)} \prod_{(p:x=_A y)} D(x, y, p)$$

with $f(x, x, \mathsf{refl}_x) :\equiv d(x)$. Unfolding the definition of D, we can expand the type of f:

$$f : \prod_{(x,y:A)} \prod_{(p:x=_A y)} \prod_{(C:\prod_{(z:A)} \prod_{(p:x=_A z)} \mathcal{U})} C(x, \mathsf{refl}_x) \to C(y, p).$$

Now given $x : A$ and $p : a =_A x$, we can derive the conclusion of based path induction:

$$f(a, x, p, C, c) : C(x, p)$$

Notice that we also obtain the correct definitional equality.

Another proof is to observe that any use of based path induction is an instance of $\mathsf{ind}_{=_A}$ and to define

$$\begin{aligned} &\mathsf{ind}_{=_A}(a, C, c, x, p) :\equiv \\ &\qquad \mathsf{ind}'_{=_A}\big(\lambda x, y.\, \lambda p.\, \textstyle\prod_{(C:\prod_{(z:A)} \prod_{(p:x=_A z)} \mathcal{U})} C(x, \mathsf{refl}_x) \to C(y, p), \lambda C.\, \lambda x.\, x, a, x, p, C, c\big) \end{aligned} \tag{1.12.2}$$

Note that the construction given above uses universes. That is, if we want to model $\mathsf{ind}_{=_A}$ with $C : \prod_{(x:A)} \prod_{(p:a=_A x)} \mathcal{U}_i$, we need to use $\mathsf{ind}'_{=_A}$ with

$$D : \prod_{(x,y:A)} \prod_{(p:x=_A y)} \mathcal{U}_{i+1}$$

since D quantifies over all C of the given type. While this is compatible with our definition of universes, it is also possible to derive $\mathsf{ind}_{=_A}$ without using universes: we can show that $\mathsf{ind}'_{=_A}$ entails Lemmas 2.3.1 and 3.11.8, and that these two principles imply $\mathsf{ind}_{=_A}$ directly. We leave the details to the reader as Exercise 1.7.

We can use either of the foregoing formulations of identity types to establish that equality is an equivalence relation, that every function preserves equality and that every family respects equality. We leave the details to the next chapter, where this will be derived and explained in the context of homotopy type theory.

1.12.3 Inequality

Let us also say something about **inequality**, which is negation of equality:[9]

$$(x \neq_A y) :\equiv \neg(x =_A y).$$

Just like negation, inequality plays a less important role here than it does in classical mathematics. For example, we cannot prove that two things are equal by proving that they are not unequal: that would be an application the classical law of double negation, see §3.4. Sometimes it is useful to phrase inequality in a positive way. For example, in Theorem 11.2.4 we shall prove that a real number x has an inverse if, and only if, its distance from 0 is positive, which is a stronger requirement than $x \neq 0$.

Notes

The type theory presented here is a version of Martin-Löf's intuitionistic type theory [ML75a, ML98, ML75b, ML82, ML84], which itself is based on and influenced by the foundational work of Brouwer [Bee85], Heyting [Hey66], Scott [Sco70], de Bruijn [dB73], Howard [How80], Tait [Tai67, Tai68], and Lawvere [Law06]. Three principal variants of Martin-Löf's type theory underlie the NuPRL [CAB+86], COQ [Coq12], and AGDA [Nor07] computer implementations of type theory. The theory given here differs from these formulations in a number of respects, some of which are critical to the homotopy interpretation, while others are technical conveniences or involve concepts that have not yet been studied in the homotopical setting.

Most significantly, the type theory described here is derived from the *intensional* version of Martin-Löf's type theory [ML75b], rather than the *extensional* version [ML82]. Whereas the extensional theory makes no distinction between judgmental and propositional equality, the intensional theory regards judgmental equality as purely definitional, and admits a much broader, proof-relevant interpretation of the identity type that is central to the homotopy interpretation.

[9] Note that this is negation of the *propositional* identity type. Of course, it makes no sense to negate judgmental equality $\equiv$, because judgments are not subject to logical operations.

From the homotopical perspective, extensional type theory confines itself to homotopically discrete sets (see §3.1), whereas the intensional theory admits types with higher-dimensional structure. The NuPRL system [CAB+86] is extensional, whereas both COQ [Coq12] and AGDA [Nor07] are intensional. Among intensional type theories, there are a number of variants that differ in the structure of identity proofs. The most liberal interpretation, on which we rely here, admits a *proof-relevant* interpretation of equality, whereas more restricted variants impose restrictions such as *uniqueness of identity proofs (UIP)* [Str93], stating that any two proofs of equality are judgmentally equal, and *Axiom K* [Str93], stating that the only proof of equality is reflexivity (up to judgmental equality). These additional requirements may be selectively imposed in the Coq and AGDA systems.

Another point of variation among intensional theories is the strength of judgmental equality, particularly as regards objects of function type. Here we include the equation $f \equiv \lambda x.\, f(x)$, which is called *$\eta$-conversion*, as a principle of judgmental equality. This principle is used, for example, in §4.9, to show that univalence implies propositional function extensionality. Other forms of "η equivalences" are sometimes considered for other types: for instance, the η-rule for cartesian products would be a judgmental version of surjective pairing, $u \equiv (\mathsf{pr}_1(u), \mathsf{pr}_2(u))$. This and the corresponding version for dependent pairs would be reasonable choices (which we did not make), but we cannot include all such rules because the corresponding "η-rule" for identity types would trivialize all the higher homotopical structure. So we are *forced* to leave it out, and the question then becomes where to draw the line. With regards to inductive types, we discuss these points further in §5.5.

It is important for our purposes that (propositional) equality of functions is taken to be *extensional*, as expressed by Axiom 2.9.3. This decision is significant for our purposes, because it specifies that equality of functions is as expected in mathematics. Although we include Axiom 2.9.3 as an axiom, it may be derived from the univalence axiom and η-equivalence (see §4.9), as well as from the existence of an interval type (see Lemma 6.3.2).

Regarding inductive types such as products, Σ-types, coproducts, natural numbers, and so on (see Chapter 5), there are additional choices regarding how to precisely formulate induction and recursion. Formally, one may describe type theory by taking either *pattern matching* or *induction principles* as basic and deriving the other; see Appendix A. However, pattern matching in general is not yet well understood from the homotopical perspective (in particular, "nested" or "deep" pattern-matching is difficult to make general sense of for higher inductive types). Moreover, it can be dangerous unless sufficient care is taken: for instance, the form of pattern-matching implemented by default in AGDA allows proving Axiom K. For these reasons, we have chosen to regard the induction principle as the basic property of an inductive definition, with pattern matching justified in terms of induction.

In general, a kind of type in type theory (such as function types, product types, identity types, etc.) is associated with four kinds of rules: *formation rules*, which specify how to form types of this sort; *introduction rules* or *constructors*, which specify how to construct elements of such types; *elimination rules* or *eliminators*, which specify how to use elements of such types to construct elements of other types; and *computation rules*, which specify what happens when the introduction and elimination rules are combined. The computation rules are also called *reduction rules* or *conversion rules*. For inductive types such as products, Σs, coproducts, and so on, the introduction

rules are the constructors (like pairing or injections), and the elimination rule is the induction principle, for which we have already mentioned the alternative terminology "eliminator". For function types and Π-types, the introduction rule is λ-abstraction, while the elimination rule is application of a function to an argument.

Unlike the type theory of COQ, we do not include a primitive type of propositions. Instead, as discussed in §1.11, we embrace the *propositions-as-types (PAT)* principle, identifying propositions with types. This was suggested originally by de Bruijn [dB73], Howard [How80], Tait [Tai68], and Martin-Löf [ML98]. (Our decision is explained more fully in §§3.2 and 3.3.)

We do, however, include a full cumulative hierarchy of universes, so that the type formation and equality judgments become instances of the membership and equality judgments for a universe. As a convenience, we regard objects of a universe as types, rather than as codes for types; in the terminology of [ML84], this means we use "Russell-style universes" rather than "Tarski-style universes". An alternative would be to use Tarski-style universes, with an explicit coercion function required to make an element $A : \mathcal{U}$ of a universe into a type $\mathsf{El}(A)$, and just say that the coercion is omitted when working informally.

We also treat the universe hierarchy as cumulative, in that every type in $\mathcal{U}_i$ is also in $\mathcal{U}_j$ for each $j \geq i$. There are different ways to implement cumulativity formally: the simplest is just to include a rule that if $A : \mathcal{U}_i$ then $A : \mathcal{U}_j$. However, this has the annoying consequence that for a type family $B : A \to \mathcal{U}_i$ we cannot conclude $B : A \to \mathcal{U}_j$, although we can conclude $\lambda a.\, B(a) : A \to \mathcal{U}_j$. A more sophisticated approach that solves this problem is to introduce a judgmental subtyping relation $<:$ generated by $\mathcal{U}_i <: \mathcal{U}_j$, but this makes the type theory more complicated to study. Another alternative would be to include an explicit coercion function $\uparrow : \mathcal{U}_i \to \mathcal{U}_j$, which could be omitted when working informally.

The path induction principle for identity types was formulated by Martin-Löf [ML98]. The based path induction principle for identity types arose in the NuPRL type theory [CAB+86] to express the concept of "pointwise functionality" for hypothetical judgments (Section 8.1 of [CAB+86].) The same principle was later formulated by Paulin-Mohring [PM93] as an alternative to Martin-Löf's rule. Their equivalence was proved by [**?**].

Exercises

Exercise 1.1. Given functions $f : A \to B$ and $g : B \to C$, define their **composite** $g \circ f : A \to C$. Show that we have $h \circ (g \circ f) \equiv (h \circ g) \circ f$.

Exercise 1.2. Derive the non-dependent eliminator for products $\mathsf{rec}_{A \times B}$ using only the projections and verify that the definitional equalities are valid. Do the same for Σ-types.

Exercise 1.3. Derive the dependent eliminator for products $\mathsf{ind}_{A \times B}$ using only the projections and sp and verify that the definitional equalities are valid. Generalize sp to Σ-types and do the same for Σ-types. *(This requires concepts from Chapter 2.)*

Exercise 1.4. Assuming as given only the *iterator* for natural numbers

$$\mathsf{iter} : \prod_{C:\mathcal{U}} C \to (C \to C) \to \mathbb{N} \to C$$

with the defining equations

$$\begin{aligned} \mathsf{iter}(C, c_0, c_s, 0) &:\equiv c_0 \\ \mathsf{iter}(C, c_0, c_s, \mathsf{succ}(n)) &:\equiv c_s(\mathsf{iter}(C, C_0, c_s, n) \end{aligned}$$

derive the recursor $\mathsf{rec}_{\mathbb{N}}$.

Exercise 1.5. Show that if we define $A + B :\equiv \sum_{(x:\mathbf{2})} \mathsf{rec}_{\mathbf{2}}(\mathcal{U}, A, B, x)$, then we can give a definition of ind_{A+B} for which the definitional equalities stated in §1.7 hold.

Exercise 1.6. Show that if we define $A \times B :\equiv \prod_{(x:\mathbf{2})} \mathsf{rec}_{\mathbf{2}}(\mathcal{U}, A, B, x)$, then we can give a definition of $\mathsf{ind}_{A\times B}$ for which the definitional equalities stated in §1.5 hold propositionally (i.e. using equality types).

Exercise 1.7. Give an alternative derivation of $\mathsf{ind}_{=_A}$ from $\mathsf{ind}'_{=_A}$ which avoids the use of universes. *(This requires concepts from later chapters.)*

Exercise 1.8. Define multiplication and exponentiation using $\mathsf{rec}_{\mathbb{N}}$. Formally verify that $(\mathbb{N}, +, 0, \times, 1)$ is a semiring using only $\mathsf{ind}_{\mathbb{N}}$.

Exercise 1.9. Define the type family $\mathsf{Fin} : \mathbb{N} \to \mathcal{U}$ mentioned at the end of §1.3, and the dependent function $\mathsf{fmax} : \prod_{(n:\mathbb{N})} \mathsf{Fin}(n+1)$ mentioned in §1.4.

Exercise 1.10. Show that the Ackermann function $\mathsf{ack} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ is definable using only $\mathsf{rec}_{\mathbb{N}}$ satisfying the following equations:

$$\begin{aligned} \mathsf{ack}(0, n) &\equiv \mathsf{succ}(n) \\ \mathsf{ack}(\mathsf{succ}(m), 0) &\equiv 1 \\ \mathsf{ack}(\mathsf{succ}(m), \mathsf{succ}(n)) &\equiv \mathsf{ack}(m, \mathsf{ack}(\mathsf{succ}(m), n)) \end{aligned}$$

Exercise 1.11. Show that for any type A, we have $\neg\neg\neg A \to \neg A$.

Exercise 1.12. Using the propositions as types interpretation, formally derive the following tautologies.

(i) If A, then (if B then A).
(ii) If A, then not (not A).
(iii) If (not A or not B), then not (A and B).

Exercise 1.13. Using propositions-as-types, derive the double negation of the principle of excluded middle, i.e. prove *not (not (P or not P))*.

Exercise 1.14. Why do the induction principles for identity types not allow us to construct a function $f : \prod_{(x:A)} \prod_{(p:x=x)} (p = \mathsf{refl}_x)$ with the defining equation

$$f(x, \mathsf{refl}_x) :\equiv \mathsf{refl}_{\mathsf{refl}_x} \quad ?$$

Exercise 1.15. Show that indiscernability of identicals follows from path induction.

Chapter 2

Homotopy type theory

The central new idea in homotopy type theory is that types can be regarded as spaces in homotopy theory, or higher-dimensional groupoids in category theory.

We begin with a brief summary of the connection between homotopy theory and higher-dimensional category theory. In classical homotopy theory, a space X is a set of points equipped with a topology, and a path between points x and y is represented by a continuous map $p : [0,1] \to X$, where $p(0) = x$ and $p(1) = y$. This function can be thought of as giving a point in X at each "moment in time". For many purposes, strict equality of paths (meaning, pointwise equal functions) is too fine a notion. For example, one can define operations of path concatenation (if p is a path from x to y and q is a path from y to z, then the concatenation $p \cdot q$ is a path from x to z) and inverses (p^{-1} is a path from y to x). However, there are natural equations between these operations that do not hold for strict equality: for example, the path $p \cdot p^{-1}$ (which walks from x to y, and then back along the same route, as time goes from 0 to 1) is not strictly equal to the identity path (which stays still at x at all times).

The remedy is to consider a coarser notion of equality of paths called *homotopy*. A homotopy between a pair of continuous maps $f\colon X_1 \to X_2$ and $g\colon X_1 \to X_2$ is a continuous map $H\colon X_1 \times [0,1] \to X_2$ satisfying $H(x,0) = f(x)$ and $H(x,1) = g(x)$. In the specific case of paths p and q, a homotopy is a continuous map $H(t,x) : [0,1] \times [0,1] \to X$: it's the image in X of a square that fills in the space between p and q, which can be thought of as a "continuous deformation" between p and q, or a 2-dimensional *path between paths*. For example, because $p \cdot p^{-1}$ walks out and back along the same route, you know that you can continuously shrink $p \cdot p^{-1}$ down to the identity path—it won't, for example, get snagged around a hole in the space. Homotopy is an equivalence relation, and operations such as concatenation, inverses, etc., respect it. Moreover, the homotopy equivalence classes of loops at some point x_0 (where two loops p and q are equated when there is a *based* homotopy between them, which is a homotopy H as above that additionally satisfies $H(0,t) = H(1,t) = x_0$ for all t) form a group called the *fundamental group*. This group is an *algebraic invariant* of a space, which can be used to investigate whether two spaces are *homotopy equivalent* (there are continuous maps back and forth whose composites are homotopic to the identity), because equivalent spaces have isomorphic fundamental groups.

Because homotopies are themselves a kind of 2-dimensional path, there is a natural notion of 3-dimensional *homotopy between homotopies*, and then *homotopy between homotopies between ho-*

motopies, and so on. This infinite tower of points, path, homotopies, homotopies between homotopies, ..., equipped with algebraic operations such as the fundamental group, is an instance of an algebraic structure called a (weak) ∞-*groupoid*. An ∞-groupoid consists of a collection of objects, and then a collection of *morphisms* between objects, and then *morphisms between morphisms*, and so on, equipped with some complex algebraic structure. Morphisms at each level have identity, composition, and inverse operations, which are weak in the sense that they satisfy the groupoid laws (associativity of composition, identity is a unit for composition, inverses cancel) only up to morphisms at the next level, and this weakness gives rise to further structure. For example, because associativity of composition of morphisms $p \cdot (q \cdot r) = (p \cdot q) \cdot r$ is itself a higher-dimensional morphism, one needs an additional operation relating various proofs of associativity: the various ways to reassociate $p \cdot (q \cdot (r \cdot s))$ into $((p \cdot q) \cdot r) \cdot s$ give rise to Mac Lane's pentagon. Weakness also creates non-trivial interactions between levels.

Every topological space X has a *fundamental* ∞-*groupoid* whose k-morphisms are the k-dimensional paths in X. The weakness of the ∞-groupoid corresponds directly to the fact that paths form a group only up to homotopy, with the $k+1$-paths serving as the homotopies between the k-paths. Moreover, the view of a space as an ∞-groupoid preserves enough aspects of the space to do homotopy theory: the fundamental ∞-groupoid construction is adjoint to the geometric realization of an ∞-groupoid as a space, and this adjunction preserves homotopy theory (this is called the *homotopy hypothesis/theorem*, because whether it is a hypothesis or theorem depends on how you define ∞-groupoid). For example, you can easily define the fundamental group of an ∞-groupoid, and if you calculate the fundamental group of the fundamental ∞-groupoid of a space, and it will agree with the classical definition of fundamental group of that space. Because of this correspondence, homotopy theory and higher-dimensional category theory are intimately related.

Now, in homotopy type theory each type can be seen to have the structure of an ∞-groupoid. Recall that for any type A, and any $x, y : A$, we have a identity type $x =_A y$, also written $\mathsf{Id}_A(x, y)$ or just $x = y$. Logically, we may think of elements of $x = y$ as evidence that x and y are equal, or as identifications of x with y. Furthermore, type theory (unlike, say, first-order logic) allows us to consider such elements of $x =_A y$ also as individuals which may be the subjects of further propositions. Therefore, we can *iterate* the identity type: we can form the type $p =_{(x=_A y)} q$ of identifications between identifications p, q, and the type $r =_{(p=_{(x=_A y)} q)} s$, and so on. The structure of this tower of identity types corresponds precisely to that of the continuous paths and (higher) homotopies between them in a space, or an ∞-groupoid. Under the interpretation of $p, q : x =_A y$ as paths from x to y, an element $r : p =_{(x=_A y)} q$ can be thought of as a homotopy, or a morphism between morphisms. Similarly, $r =_{(p=_{(x=_A y)} q)} s$ is the type of 3-dimensional paths between two 2-dimensional paths, and so on. If the type A is "set-like", such as $\mathbb{N}$, these iterated identity types will be uninteresting (see §3.1), but in the general case they can model non-trivial homotopy types.

An important difference between homotopy type theory and classical homotopy theory is that homotopy type theory provides a *synthetic* description of spaces, in the following sense. Synthetic geometry is geometry in the style of Euclid: one starts from some basic notions (points and lines), constructions (a line connecting any two points), and axioms (all right angles are equal), and deduces consequences logically. This is in contrast with analytic geometry, where no-

tions such as points and lines are represented concretely using cartesian coordinates in $\mathbb{R}^n$—lines are sets of points—and the basic constructions and axioms are derived from this representation. While classical homotopy theory is analytic (spaces and lines are made of points), homotopy type theory is synthetic: points, paths, paths between paths are basic, indivisible, primitive notions.

Moreover, one of the amazing things about homotopy type theory is that all of the basic constructions and axioms—all of the higher groupoid structure—-arises automatically from the induction principle for identity types, as stated in §1.12. Recall that this says that if

- for every $x, y : A$ and every $p : x =_A y$ we have a type $D(x, y, p)$, and
- for every $a : A$ we have an element $d(a) : D(a, a, \mathsf{refl}_a)$,

then

- there exists an element $J_{D,d}(x, y, p) : D(x, y, p)$ for *every* two elements $x, y : A$ and $p : x =_A y$, such that $J_{D,d}(a, a, \mathsf{refl}_a) \equiv d(a)$.

In other words, given dependent functions

$$D : \prod_{(x,y:A)} \prod_{(p:x=y)} \mathcal{U}$$
$$d : \prod_{a:A} D(a, a, \mathsf{refl}_a)$$

there is a dependent function

$$J_{D,d} : \prod_{(x,y:A)} \prod_{(p:x=y)} D(x, y, p)$$

such that

$$J_{D,d}(a, a, \mathsf{refl}_a) \equiv d(a) \tag{2.0.1}$$

for every $a : A$. The notation J is traditional for this function, but we will not use it very much. Usually, every time we apply this induction rule we will either not care about the specific function being defined, or we will immediately give it a different name.

Informally, the induction principle for identity types says that if we want to construct an object (or prove a statement) which depends on an inhabitant $p : x =_A y$ of an identity type, then it suffices to perform the construction (or the proof) in the special case when x and y are the same (judgmentally) and p is the reflexivity element $\mathsf{refl}_x : x = x$ (judgmentally). When writing informally, we may express this with a phrase such as "by induction, it suffices to assume…". This reduction to the "reflexivity case" is analogous to the reduction to the "base case" and "inductive step" in an ordinary proof by induction on the natural numbers, and also to the "left case" and "right case" in a proof by case analysis on a disjoint union or disjunction.

The "conversion rule" (2.0.1) is less familiar in the context of proof by induction on natural numbers, but there is an analogous notion in the related concept of definition by recursion. If a sequence $(a_n)_{n\in\mathbb{N}}$ is defined by giving a_0 and specifying a_{n+1} in terms of a_n, then in fact the 0^{th} term of the resulting sequence *is* the given one, and the given recurrence relation relating a_{n+1} to a_n holds for the resulting sequence. (This may seem so obvious as to not be worth saying, but if we view a definition by recursion as an algorithm for calculating values of a sequence, then

it is precisely the process of executing that algorithm.) The rule (2.0.1) is analogous: it says that if we define an object $f(p)$ for all $p : x = y$ by specifying what the value should be when p is $\mathsf{refl}_x : x = x$, then the value we specified is in fact the value of $f(\mathsf{refl}_x)$.

This induction principle gives each type the structure of an ∞-groupoid, and each function between two types the structure of an ∞-functor between two such groupoids. This is interesting from a mathematical point view, because it gives a new way to work with ∞-groupoids. It is interesting from a type-theoretic point view, because it reveals new operations that are associated with each type and function. In the remainder of this chapter, we begin to explain this structure.

2.1 Types are higher groupoids

We now derive from this induction principle the beginnings of the structure of a higher groupoid. We begin with symmetry of equality, which, in topological language, means that "paths can be reversed".

Lemma 2.1.1. *For every type A and every $x, y : A$ there is a function*

$$(x = y) \to (y = x)$$

denoted $p \mapsto p^{-1}$, such that ${\mathsf{refl}_x}^{-1} \equiv \mathsf{refl}_x$ for each $x : A$.

First proof. Let $D : \prod_{(x,y:A)} \prod_{(p:x=y)} \mathcal{U}$ be the type family defined by $D(x, y, p) :\equiv (y = x)$. In other words, D is a function assigning to any $x, y : A$ and $p : x = y$ a type, namely the type $y = x$. Then we have an element

$$d :\equiv \lambda x.\, \mathsf{refl}_x : \prod_{x:A} D(x, x, \mathsf{refl}_x).$$

Thus, the eliminator J for identity types gives us an element $J_{D,d}(x, y, p) : (y = x)$ for each $p : (x = y)$. We can now define the desired function $(-)^{-1}$ to be $\lambda p.\, J_{D,d}(x, y, p)$, i.e. we set $p^{-1} :\equiv J_{D,d}(x, y, p)$. The conversion rule (2.0.1) gives ${\mathsf{refl}_x}^{-1} \equiv \mathsf{refl}_x$, as required. □

We have written out this proof in a very formal style, which may be helpful while the induction rule on identity types is unfamiliar. However, eventually we prefer to use more natural language, such as in the following equivalent proof.

Second proof. We want to construct, for each $x, y : A$ and $p : x = y$, an element $p^{-1} : y = x$. By induction, it suffices to do this in the case when y is x and p is refl_x. But in this case, the type $x = y$ of p and the type $y = x$ in which we are trying to construct p^{-1} are both simply $x = x$. Thus, in the "reflexivity case", we can define ${\mathsf{refl}_x}^{-1}$ to be simply refl_x. The general case then follows by the induction principle, and the conversion rule ${\mathsf{refl}_x}^{-1} \equiv \mathsf{refl}_x$ is precisely the proof in the reflexivity case that we gave. □

We will write out the next few proofs in both styles, to help the reader become accustomed to the latter one. Next we prove the transitivity of equality, or equivalently we "concatenate paths".

Lemma 2.1.2. *For every type A and every $x, y, z : A$ there is a function*

$$(x = y) \to \big((y = z) \to (x = z)\big)$$

written $(p, q) \mapsto p \cdot q$, such that $\mathsf{refl}_x \cdot \mathsf{refl}_x \equiv \mathsf{refl}_x$ for any $x : A$.

First proof. Let $D : \prod_{(x,y:A)} \prod_{(p:x=y)} \mathcal{U}$ be the type family

$$D(x, y, p) :\equiv \prod_{(z:A)} \prod_{(q:y=z)} (x = z).$$

Note that $D(x, x, \mathsf{refl}_x) \equiv \prod_{(z:A)} \prod_{(q:x=z)} (x = z)$. Thus, in order to apply the induction principle for identity types to this D, we need a function of type

$$\prod_{x:A} D(x, x, \mathsf{refl}_x) \tag{2.1.3}$$

which is to say, of type

$$\prod_{(x,z:A)} \prod_{(q:x=z)} (x = z).$$

Now let $E : \prod_{(x,z:A)} \prod_{(q:x=z)} \mathcal{U}$ be the type family $E(x, z, q) :\equiv (x = z)$. Note that $E(x, x, \mathsf{refl}_x) \equiv (x = x)$. Thus, we have the function

$$e(x) :\equiv \mathsf{refl}_x : E(x, x, \mathsf{refl}_x).$$

By the induction principle for identity types applied to E, we obtain a function

$$d(x, z, q) : \prod_{(x,z:A)} \prod_{(q:x=z)} E(x, z, q).$$

But $E(x, z, q) \equiv (x = z)$, so this is (2.1.3). Thus, we can use this function d and apply the induction principle for identity types to D, to obtain our desired function of type

$$\prod_{(x,y,z:A)} \prod_{(q:y=z)} \prod_{(p:x=y)} (x = z).$$

The conversion rules for the two induction principles give us $\mathsf{refl}_x \cdot \mathsf{refl}_x \equiv \mathsf{refl}_x$ for any $x : A$. □

Second proof. We want to construct, for every $x, y, z : A$ and every $p : x = y$ and $q : y = z$, an element of $x = z$. By induction on p, it suffices to assume that y is x and p is refl_x. In this case, the type $y = z$ of q is $x = z$. Now by induction on q, it suffices to assume also that z is x and q is refl_x. But in this case, $x = z$ is $x = x$, and we have $\mathsf{refl}_x : (x = x)$. □

The reader may well feel that we have given an overly convoluted proof of this lemma. In fact, we could stop after the induction on p, since at that point what we want to produce is an equality $x = z$, and we already have such an equality, namely q. Why do we go on to do another induction on q?

The answer is that, as described in the introduction, we are doing *proof-relevant* mathematics. When we prove a lemma, we are defining an inhabitant of some type, and it can matter what

specific element we defined in the course of the proof, not merely the type that that element inhabits (that is, the *statement* of the lemma). Lemma 2.1.2 has three obvious proofs: we could do induction over p, induction over q, or induction over both of them. If we proved it three different ways, we would have three different elements of the same type. It's not hard to show that these three elements are equal (see Exercise 2.1), but as they are not *definitionally* equal, there can still be reasons to prefer one over another.

In the case of Lemma 2.1.2, the difference hinges on the computation rule. If we proved the lemma using a single induction over p, then we would end up with a computation rule of the form $\mathsf{refl}_y \cdot q \equiv q$. If we proved it with a single induction over q, we would have instead $p \cdot \mathsf{refl}_x \equiv p$, while proving it with a double induction (as we did) gives only $\mathsf{refl}_x \cdot \mathsf{refl}_x \equiv \mathsf{refl}_x$.

The asymmetrical computation rules can sometimes be convenient when doing formalized mathematics, as they allow the computer to simplify more things automatically. However, in informal mathematics, and arguably even in the formalized case, it can be confusing to have a concatenation operation which behaves asymmetrically and to have to remember which side is the "special" one. Treating both sides symmetrically makes for more robust proofs; this is why we have given the proof that we did. (However, this is admittedly a stylistic choice.)

The table below summarizes the "equality" and "homotopical" points of view on what we have done so far.

Equality	Homotopy
reflexivity	constant path
symmetry	inversion of paths
transitivity	concatenation of paths

Because of proof-relevance, we can't stop after proving "symmetry" and "transitivity" of equality: we need to know that these *operations* on equalities are well-behaved. (This issue is invisible in set theory, where symmetry and transitivity are mere *properties* of equality, rather than structure on paths.) From the homotopy-theoretic point of view, concatenation and inversion are just the "first level" of higher groupoid structure — we also need coherence laws on these operations, and analogous operations at higher dimensions. For instance, we need to know that concatenation is *associative*, and that inversion provides *inverses* with respect to concatenation.

Lemma 2.1.4. *Suppose $A : \mathcal{U}$, that $x, y, z, w : A$ and that $p : x = y$ and $q : y = z$ and $r : z = w$. We have the following:*

(i) $p = p \cdot \mathsf{refl}_y$ *and* $p = \mathsf{refl}_x \cdot p$.
(ii) $p^{-1} \cdot p = \mathsf{refl}_y$ *and* $p \cdot p^{-1} = \mathsf{refl}_x$.
(iii) $(p^{-1})^{-1} = p$.
(iv) $p \cdot (q \cdot r) = (p \cdot q) \cdot r$.

Note, in particular, that (i)–(iv) are themselves propositional equalities, living in the identity types *of* identity types, such as $p =_{x=y} q$ for $p, q : x = y$. Topologically, they are *paths of paths*, i.e. homotopies. It is a familiar fact in topology that when we concatenate a path p with the reversed path p^{-1}, we don't literally obtain a constant path (which corresponds to the equality refl in type theory) — instead we have a homotopy, or higher path, from $p \cdot p^{-1}$ to the constant path.

Proof of Lemma 2.1.4. All the proofs use the induction principle for equalities.

(i) *First proof:* let $D : \prod_{(x,y:A)} \prod_{(p:x=y)} \mathcal{U}$ be the type family given by

$$D(x, y, p) :\equiv (p = p \cdot \mathsf{refl}_y).$$

Then $D(x, x, \mathsf{refl}_x)$ is $\mathsf{refl}_x = \mathsf{refl}_x \cdot \mathsf{refl}_x$. Since $\mathsf{refl}_x \cdot \mathsf{refl}_x \equiv \mathsf{refl}_x$, it follows that $D(x, x, \mathsf{refl}_x) \equiv (\mathsf{refl}_x = \mathsf{refl}_x)$. Thus, there is a function

$$d :\equiv \lambda x.\, \mathsf{refl}_{\mathsf{refl}_x} : \prod_{x:A} D(x, x, \mathsf{refl}_x).$$

Now the induction principle for identity types gives an element $J(D, d, p) : (p = p \cdot \mathsf{refl}_y)$ for each $p : x = y$. The other equality is proven similarly.

Second proof: by induction on p, it suffices to assume that y is x and that p is refl_x. But in this case, we have $\mathsf{refl}_x \cdot \mathsf{refl}_x \equiv \mathsf{refl}_x$.

(ii) *First proof:* let $D : \prod_{(x,y:A)} \prod_{(p:x=y)} \mathcal{U}$ be the type family given by

$$D(x, y, p) :\equiv (p^{-1} \cdot p = \mathsf{refl}_y).$$

Then $D(x, x, \mathsf{refl}_x)$ is ${\mathsf{refl}_x}^{-1} \cdot \mathsf{refl}_x = \mathsf{refl}_x$. Since ${\mathsf{refl}_x}^{-1} \equiv \mathsf{refl}_x$ and $\mathsf{refl}_x \cdot \mathsf{refl}_x \equiv \mathsf{refl}_x$, we get that $D(x, x, \mathsf{refl}_x) \equiv (\mathsf{refl}_x = \mathsf{refl}_x)$. Hence we find the function

$$d :\equiv \lambda x.\, \mathsf{refl}_{\mathsf{refl}_x} : \prod_{x:A} D(x, x, \mathsf{refl}_x).$$

Now path induction gives an element $J(D, d, p) : p^{-1} \cdot p = \mathsf{refl}_y$ for each $p : x = y$ in A. The other equality is similar.

Second proof By induction, it suffices to assume p is refl_x. But in this case, we have $p^{-1} \cdot p \equiv {\mathsf{refl}_x}^{-1} \cdot \mathsf{refl}_x \equiv \mathsf{refl}_x$.

(iii) *First proof:* let $D : \prod_{(x,y:A)} \prod_{(p:x=y)} \mathcal{U}$ be the type family given by

$$D(x, y, p) :\equiv ({p^{-1}}^{-1} = p).$$

Then $D(x, x, \mathsf{refl}_x)$ is the type $({\mathsf{refl}_x}^{-1^{-1}} = \mathsf{refl}_x)$. But since ${\mathsf{refl}_x}^{-1} \equiv \mathsf{refl}_x$ for each $x : A$, we have ${\mathsf{refl}_x}^{-1^{-1}} \equiv {\mathsf{refl}_x}^{-1} \equiv \mathsf{refl}_x$, and thus $D(x, x, \mathsf{refl}_x) \equiv (\mathsf{refl}_x = \mathsf{refl}_x)$. Hence we find the function

$$d :\equiv \lambda x.\, \mathsf{refl}_{\mathsf{refl}_x} : \prod_{x:A} D(x, x, \mathsf{refl}_x).$$

Now path induction gives an element $J(D, d, p) : {p^{-1}}^{-1} = p$ for each $p : x = y$.

Second proof: by induction, it suffices to assume p is refl_x. But in this case, we have ${p^{-1}}^{-1} \equiv {\mathsf{refl}_x}^{-1^{-1}} \equiv \mathsf{refl}_x$.

(iv) *First proof:* let $D_1 : \prod_{(x,y:A)} \prod_{(p:x=y)} \mathcal{U}$ be the type family given by

$$D_1(x,y,p) :\equiv \prod_{(z,w:A)} \prod_{(q:y=z)} \prod_{(r:z=w)} \big(p \cdot (q \cdot r) = (p \cdot q) \cdot r\big).$$

Then $D_1(x,x,\mathsf{refl}_x)$ is

$$\prod_{(z,w:A)} \prod_{(q:x=z)} \prod_{(r:z=w)} \big(\mathsf{refl}_x \cdot (q \cdot r) = (\mathsf{refl}_x \cdot q) \cdot r\big).$$

To construct an element of this type, let $D_2 : \prod_{(x,z:A)} \prod_{(q:x=z)} \mathcal{U}$ be the type family

$$D_2(x,z,q) :\equiv \prod_{(w:A)} \prod_{(r:z=w)} \big(\mathsf{refl}_x \cdot (q \cdot r) = (\mathsf{refl}_x \cdot q) \cdot r\big).$$

Then $D_2(x,x,\mathsf{refl}_x)$ is

$$\prod_{(w:A)} \prod_{(r:x=w)} \big(\mathsf{refl}_x \cdot (\mathsf{refl}_x \cdot r) = (\mathsf{refl}_x \cdot \mathsf{refl}_x) \cdot r\big).$$

To construct an element of *this* type, let $D_3 : \prod_{(x,w:A)} \prod_{(r:x=w)} \mathcal{U}$ be the type family

$$D_3(x,w,r) :\equiv \big(\mathsf{refl}_x \cdot (\mathsf{refl}_x \cdot r) = (\mathsf{refl}_x \cdot \mathsf{refl}_x) \cdot r\big).$$

Then $D_3(x,x,\mathsf{refl}_x)$ is

$$\big(\mathsf{refl}_x \cdot (\mathsf{refl}_x \cdot \mathsf{refl}_x) = (\mathsf{refl}_x \cdot \mathsf{refl}_x) \cdot \mathsf{refl}_x\big)$$

which is definitionally equal to the type $(\mathsf{refl}_x = \mathsf{refl}_x)$, and is therefore inhabited by $\mathsf{refl}_{\mathsf{refl}_x}$. Applying the identity elimination rule three times, therefore, we obtain an element of the overall desired type.

Second proof: by induction, it suffices to assume p, q, and r are all refl_x. But in this case, we have

$$\begin{aligned} p \cdot (q \cdot r) &\equiv \mathsf{refl}_x \cdot (\mathsf{refl}_x \cdot \mathsf{refl}_x) \\ &\equiv \mathsf{refl}_x \\ &\equiv (\mathsf{refl}_x \cdot \mathsf{refl}_x) \cdot \mathsf{refl}_x \\ &\equiv (p \cdot q) \cdot r. \end{aligned}$$

Thus, we have $\mathsf{refl}_{\mathsf{refl}_x}$ inhabiting this type. □

Remark 2.1.5. There are other ways to define all of these higher paths. For instance, in Lemma 2.1.4(iv) we might do induction only over one or two paths rather than all three. Each possibility will produce a *definitionally* different proof, but they will all be equal to each other. Such an equality between any two particular proofs can, again, be proven by induction, reducing all the paths in question to reflexivities and then observing that both proofs reduce themselves to reflexivities.

We are still not really done with the higher groupoid structure: the paths (i)–(iv) must also satisfy their own higher coherence laws, which are themselves higher paths, and so on "all the way up to infinity" (this can be made precise using e.g. the notion of a globular operad). However, for most purposes it is unnecessary to make the whole infinite-dimensional structure explicit. One of the nice things about homotopy type theory is that all of this structure can be **proven** starting from only the inductive property of identity types, so we can make explicit as much or as little of it as we need. In particular, in this book we will not need any of the complicated combinatorics involved in making precise notions such as "coherent structure at all higher levels".

One particularly important case of iterated identity types (path types) is when the start and end points are the same. In set theory, the propositon $a = a$ is entirely uninteresting, but in homotopy theory paths from a point to itself are called *loops* and carry lots of interesting higher structure. Thus, given a type A with a point $a : A$, we define its **loop space** $\Omega(A, a)$ to be the type $a =_A a$. We may sometimes write simply ΩA if the point a is understood from context.

Since any two elements of ΩA are paths with the same start and end points, they can be concatenated; thus we have an operation $\Omega A \times \Omega A \to \Omega A$. More generally, the higher groupoid structure of A gives ΩA the analogous structure of a "higher group".

It can also be useful to consider the loop space *of* the loop space of A, which is the space of 2-dimensional loops on the identity loop at a. This is written $\Omega^2(A, a)$ and represented in type theory by the type $\mathsf{refl}_a =_{(a=_A a)} \mathsf{refl}_a$. While $\Omega^2(A, a)$, as a loop space, is again a "higher group", it now also has some additional structure resulting from the fact that its elements are 2-dimensional loops between 1-dimensional loops.

Theorem 2.1.6 (Eckmann-Hilton)**.** *The composition operation on the second loop space*

$$\Omega^2(A) \times \Omega^2(A) \to \Omega^2(A)$$

is commutative: $\alpha \cdot \beta = \beta \cdot \alpha$, for any $\alpha, \beta : \Omega^2(A)$.

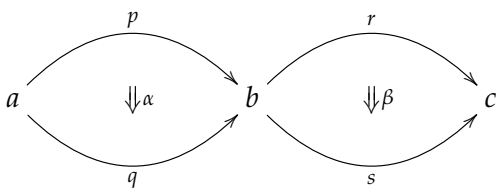
Proof. First, observe that the composition of 1-loops $\Omega A \times \Omega A \to \Omega A$ induces an operation

$$\star : \Omega^2(A) \times \Omega^2(A) \to \Omega^2(A)$$

as follows: consider elements $a, b, c : A$ and 1- and 2-paths,

$$\begin{aligned} p &: a = b, & r &: b = c \\ q &: a = b, & s &: b = c \\ \alpha &: p = q, & \beta &: r = s \end{aligned}$$

as depicted in the following diagram.



Composing the upper and lower 1-paths, respectively, we get two paths $p \cdot r,\ q \cdot s : a = c$, and there is then a "horizontal composition"

$$\alpha \star \beta : p \cdot r = q \cdot s$$

between them, defined as follows: first let $\alpha \cdot_r r : p \cdot r = q \cdot r$ be determined by path induction on r, then let $q \cdot_\ell \beta : q \cdot r = q \cdot s$ be given by induction on q. Since these paths are composable in the type of 2-paths, we can define the **horizontal composition** by:

$$\alpha \star \beta \ :\equiv\ (\alpha \cdot_r r) \cdot (q \cdot_\ell \beta)\,.$$

Now suppose that $a \equiv b \equiv c$, so that all the above 1-paths are in $\Omega(A, a)$, and assume moreover that $q \equiv \mathsf{refl}_a \equiv r$, so that α and β become composable. In that case, we therefore have

$$\alpha \star \beta \ =\ (\alpha \cdot_r \mathsf{refl}_a) \cdot (\mathsf{refl}_a \cdot_\ell \beta) = \alpha \cdot \beta\,.$$

On the other hand, we can define another horizontal composition analogously by

$$\alpha \star' \beta \ :\equiv\ (p \cdot_\ell \beta) \cdot (\alpha \cdot_r s)\,.$$

and setting $p \equiv \mathsf{refl}_a \equiv s$ we learn that in that case

$$\alpha \star' \beta \ =\ (\mathsf{refl}_a \cdot_\ell \beta) \cdot (\alpha \cdot_r \mathsf{refl}_a) = \beta \cdot \alpha\,.$$

But, in general, the two ways of defining horizontal composition agree, $\alpha \star \beta = \alpha \star' \beta$, as we can see by induction on α and β. Thus when $p \equiv q \equiv \mathsf{refl}_a \equiv r \equiv s$ we have

$$\alpha \cdot \beta = \alpha \star \beta = \alpha \star' \beta = \beta \cdot \alpha\,. \qquad \square$$

The foregoing fact, which is known as the *Eckmann-Hilton argument*, comes from classical homotopy theory, and indeed it is used in Chapter 8 below to show that the higher homotopy groups of a type are always abelian groups.

As this example suggests, the algebra of higher path types is much more intricate than just the groupoid-like structure at each level; the levels interact to give many further operations and laws, as in the study of iterated loop spaces in homotopy theory. Indeed, as in classical homotopy theory, we can make the following general definitions:

Definition 2.1.7. A **pointed type** (A, a) is a type $A : \mathcal{U}$ together with a point $a : A$. We write $\mathcal{U}_\bullet :\equiv \sum_{(A:\mathcal{U})} A$ for the type of pointed types in the universe $\mathcal{U}$.

Definition 2.1.8. Given a pointed type (A, a), we define the **loop space** of (A, a) to be the following pointed type:

$$\Omega(A, a) = ((a =_A a), \mathsf{refl}_A(a)).$$

For $n : \mathbb{N}$, the **n-fold iterated loop space** of a pointed type (A, a) is defined recursively by:

$$\Omega^0(A, a) = (A, a)$$
$$\Omega^{n+1}(A, a) = \Omega^n(\Omega(A, a)).$$

We will return to iterated loop spaces in Chapters 6 to 8.

2.2 Functions are functors

Now we wish to establish that functions $f : A \to B$ behave functorially on paths. In traditional type theory, this is equivalently the statement that functions respect equality. Topologically, this corresponds to saying that every function is "continuous", i.e. preserves paths.

Lemma 2.2.1. *Suppose that $f : A \to B$ is a function. Then for any $x, y : A$ there is an operation*

$$\mathsf{ap}_f : (x =_A y) \to (f(x) =_B f(y)).$$

Moreover, for each $x : A$ we have $\mathsf{ap}_f(\mathsf{refl}_x) \equiv \mathsf{refl}_{f(x)}$.

The notation ap_f can be read either as the application of f to a path, or as the action on paths of f.

First proof. Let $D : \prod_{(x,y:A)} \prod_{(p:x=y)} \mathcal{U}$ be the type family defined by

$$D(x, y, p) :\equiv (f(x) = f(y)).$$

Then we have

$$d :\equiv \lambda x.\, \mathsf{refl}_{f(x)} : \prod_{x:A} D(x, x, \mathsf{refl}_x).$$

Applying J, we obtain $\mathsf{ap}_f : \prod_{(x,y:A)} \prod_{(p:x=y)} (f(x) = g(x))$. The conversion rule implies $\mathsf{ap}_f(\mathsf{refl}_x) \equiv \mathsf{refl}_{f(x)}$ for each $x : A$. □

Second proof. By induction, it suffices to assume p is refl_x. In this case, we may define $\mathsf{ap}_f(p) :\equiv \mathsf{refl}_{f(x)} : f(x) \equiv f(x)$. □

We will often write $\mathsf{ap}_f(p)$ as simply $f(p)$. This is strictly speaking ambiguous, but generally no confusion arises. It matches the common convention in category theory of using the same symbol for the application of a functor to objects and to morphisms.

We note that ap behaves functorially, in all the ways that one might expect.

Lemma 2.2.2. *For functions $f : A \to B$ and $g : B \to C$ and paths $p : x =_A y$ and $q : y =_A z$, we have:*

(i) $\mathsf{ap}_f(p \cdot q) = \mathsf{ap}_f(p) \cdot \mathsf{ap}_f(q)$.

(ii) $\mathsf{ap}_f(p^{-1}) = \mathsf{ap}_f(p)^{-1}$.

(iii) $\mathsf{ap}_g(\mathsf{ap}_f(p)) = \mathsf{ap}_{g \circ f}(p)$.

(iv) $\mathsf{ap}_{\mathsf{id}_A}(p) = p$.

Proof. Left to the reader. □

2.3 Type families are fibrations

Since *dependently typed* functions are essential in type theory, we will also need a version of Lemma 2.2.1 for these. However, this is not quite so simple to state, because if $f : \prod_{(x:A)} B(x)$ and $p : x = y$, then $f(x) : B(x)$ and $f(y) : B(y)$ are elements of distinct types, so that *a priori* we cannot even ask whether they are equal. The missing ingredient is that p itself gives us a way to relate the types $B(x)$ and $B(y)$.

Lemma 2.3.1 (Transport)**.** *Suppose that P is a type family over A and that $p : x =_A y$. Then there is a function $p_* : P(x) \to P(y)$.*

First proof. Let $D : \prod_{(x,y:A)} \prod_{(p:x=y)} \mathcal{U}$ be the type family defined by

$$D(x, y, p) :\equiv P(x) \to P(y).$$

Then we have the function

$$d :\equiv \lambda x.\, \mathsf{id}_{P(x)} : \prod_{x:A} D(x, x, \mathsf{refl}_x),$$

so that the induction principle gives us $J_{D,d}(x, y, p) : P(x) \to P(y)$ for $p : x = y$, which we define to be p_*. □

Second proof. By induction, it suffices to assume p is refl_x. But in this case, we can take $(\mathsf{refl}_x)_* : P(x) \to P(x)$ to be the identity function. □

Sometimes, it is necessary to notate the type family P in which the transport operation happens. In this case, we may write

$$\mathsf{transport}^P(p, \_) : P(x) \to P(y).$$

Recall that a type family P over a type A can be seen as a property of elements of A, which holds at x in A if $P(x)$ is inhabited. Then the transportation lemma says that P respects equality, in the sense that if x is equal to y, then $P(x)$ holds if and only if $P(y)$ holds. In fact, we will see later on that if $x = y$ then actually $P(x)$ and $P(y)$ are *equivalent*.

Topologically, the transportation lemma can be viewed as a "path lifting" operation in a fibration. We think of a type family $P : A \to \mathcal{U}$ as a fibration with base A and total space $\sum_{(x:A)} P(x)$, with $P(x)$ being the fiber over x. The defining property of a fibration is that given a path $p : x = y$ in the base space A and a point $u : P(x)$ in the fiber over x, we may lift the path p to a path in the total space starting at u. The point $p_*(u)$ can be thought of as the other endpoint of this lifted path. We can also define the path itself in type theory:

Lemma 2.3.2 (Path lifting property)**.** *Let $P : A \to \mathcal{U}$ be a type family over A and assume we have $u : P(x)$ for some $x : A$. Then for any $p : x = y$, we have*

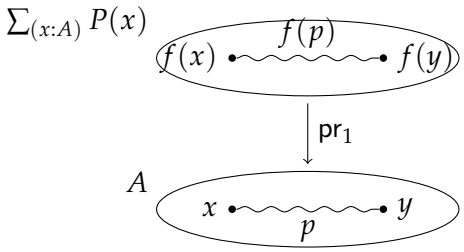
$$\mathsf{lift}(u, p) : (x, u) = (y, p_*(u))$$

in $\sum_{(x:A)} P(x)$.

Proof. Left to the reader. We will prove a more general theorem in §2.7. □

Remark 2.3.3. Although we may think of a type family $P : A \to \mathcal{U}$ as like a fibration, it is generally not a good idea to say things like "the fibration $P : A \to \mathcal{U}$", since this sounds like we are talking about a fibration with base $\mathcal{U}$ and total space A. To repeat, when a type family $P : A \to \mathcal{U}$ is regarded as a fibration, the base is A and the total space is $\sum_{(x:A)} P(x)$.

Now we can prove the dependent version of Lemma 2.2.1. The topological intuition is that given $f : \prod_{(x:A)} P(x)$ and a path $p : x =_A y$, we ought to be able to apply f to p and obtain a path in the total space of P which "lies over" p, as shown below.



We *can* obtain such a thing from Lemma 2.2.1. Given $f : \prod_{(x:A)} P(x)$, we can define a non-dependent function $f' : A \to \sum_{(x:A)} P(x)$ by setting $f'(x) :\equiv (x, f(x))$, and then consider $f'(p) : f'(x) = f'(y)$. However, it is not obvious from the type of such a path that it lies over a specific path in A (in this case, p), which is sometimes important.

The solution is to use the transport lemma. Since there is a canonical path from $u : P(x)$ to $p_*(u) : P(y)$ which (at least intuitively) lies over p, any path from u to $v : P(y)$ lying over p should factor through this path, essentially uniquely, by a path from $p_*(u)$ to v lying entirely in the fiber $P(y)$. Thus, up to equivalence, it makes sense to define "a path from u to v lying over $p : x = y$" to mean a path $p_*(u) = v$ in $P(y)$. And, indeed, we can show that dependent functions produce such paths.

Lemma 2.3.4 (Dependent map)**.** *Suppose $f : \prod_{(x:A)} P(x)$; then we have a map*

$$\mathsf{apd}_f : \prod_{p:x=y} \big(p_*(f(x)) =_{P(y)} f(y)\big).$$

First proof. Let $D : \prod_{(x,y:A)} \prod_{(p:x=y)} \mathcal{U}$ be the type family defined by

$$D(x, y, p) :\equiv p_*(f(x)) = f(y).$$

Then $D(x, x, \mathsf{refl}_x)$ is $(\mathsf{refl}_x)_*(f(x)) = f(x)$. But since $(\mathsf{refl}_x)_*(f(x)) \equiv f(x)$, we get that $D(x, x, \mathsf{refl}_x) \equiv (f(x) = f(x))$. Thus, we find the function

$$d :\equiv \lambda x.\, \mathsf{refl}_{f(x)} : \prod_{x:A} D(x, x, \mathsf{refl}_x)$$

and now J gives us $\mathsf{apd}_f(p) : p_*(f(x)) = f(y)$ for each $p : x = y$. □

Second proof. By induction, it suffices to assume p is refl_x. But in this case, the desired equation is $(\mathsf{refl}_x)_*(f(x)) \equiv f(x)$, which holds judgmentally. □

We will refer generally to paths which "lie over other paths" in this sense as *dependent paths*. They will play an increasingly important role starting in Chapter 6. In §2.5 we will see that for a few particular kinds of type families, there are equivalent ways to represent the notion of dependent paths that are sometimes more convenient.

Now recall from §1.4 that a non-dependently typed function $f : A \to B$ is just the special case of a dependently typed function $f : \prod_{(x:A)} P(x)$ when P is a constant type family, $P(x) :\equiv B$. In this case, apd_f and ap_f are closely related, because of the following lemma:

Lemma 2.3.5. *If $P : A \to \mathcal{U}$ is defined by $P(x) :\equiv B$ for a fixed $B : \mathcal{U}$, then for any $x, y : A$ and $p : x = y$ and $b : B$ we have a path*

$$\mathsf{transportconst}^B_p(b) : \mathsf{transport}^P(p, b) = b.$$

First proof. Fix a $b : B$, and let $D : \prod_{(x,y:A)} \prod_{(p:x=y)} \mathcal{U}$ be the type family defined by

$$D(x, y, p) :\equiv (\mathsf{transport}^P(p, b) = b).$$

Then $D(x, x, \mathsf{refl}_x)$ is $(\mathsf{transport}^P(\mathsf{refl}_x, b) = b)$, which is judgmentally equal to $(b = b)$ by the computation rule for transporting. Thus, we have the function

$$d :\equiv \lambda x.\, \mathsf{refl}_b : \prod_{x:A} D(x, x, \mathsf{refl}_x).$$

Now path induction gives us an element of $\prod_{(x,y:A)} \prod_{(p:x=y)} (\mathsf{transport}^P(p, b) = b)$, as desired. □

Second proof. By induction, it suffices to assume y is x and p is refl_x. But $\mathsf{transport}^P(\mathsf{refl}_x, b) \equiv b$, so in this case what we have to prove is $b = b$, and we have refl_b for this. □

Thus, by concatenating with $\mathsf{transportconst}^B_p(b)$, for any $x, y : A$ and $p : x = y$ and $f : A \to B$ we obtain functions

$$\big(f(x) = f(y)\big) \to \big(p_*(f(x)) = f(y)\big) \qquad \text{and} \tag{2.3.6}$$

$$\big(p_*(f(x)) = f(y)\big) \to \big(f(x) = f(y)\big). \tag{2.3.7}$$

In fact, these functions are inverse equivalences (in the sense to be introduced in §2.4), and they relate $\mathsf{ap}_f(p)$ to $\mathsf{apd}_f(p)$.

Lemma 2.3.8. *For $f : A \to B$ and $p : x =_A y$, we have*

$$\mathsf{apd}_f(p) = \mathsf{transportconst}^B_p(f(x)) \cdot \mathsf{ap}_f(p)$$

First proof. Let $D : \prod_{(x,y:A)} \prod_{(p:x=y)} \mathcal{U}$ be the type family defined by

$$D(x, y, p) :\equiv \big(\mathsf{apd}_f(p) = \mathsf{transportconst}^B_p(f(x)) \cdot \mathsf{ap}_f(p)\big).$$

Thus, we have

$$D(x, x, \mathsf{refl}_x) \equiv \big(\mathsf{apd}_f(\mathsf{refl}_x) = \mathsf{transportconst}^B_{\mathsf{refl}_x}(f(x)) \cdot \mathsf{ap}_f(\mathsf{refl}_x)\big).$$

But by definition, all three paths appearing in this type are $\mathsf{refl}_{f(x)}$, so we have

$$\mathsf{refl}_{\mathsf{refl}_{f(x)}} : D(x, x, \mathsf{refl}_x).$$

Thus, path induction gives us an element of $\prod_{(x,y:A)} \prod_{(p:x=y)} D(x, y, p)$, which is what we wanted. □

Second proof. By induction, it suffices to assume y is x and p is refl_x. In this case, what we have to prove is $\mathsf{refl}_{f(x)} = \mathsf{refl}_{f(x)} \cdot \mathsf{refl}_{f(x)}$, which is true judgmentally. □

Because the types of apd_f and ap_f are different, it is often clearer to use different notations for them.

At this point, we hope the reader is starting to get a feel for proofs by induction on identity types. From now on we stop giving both styles of proofs, allowing ourselves to use whatever is most clear and convenient (and often the second, more concise one). Here are a few other useful lemmas about transport; we leave it to the reader to give the proofs (in either style).

Lemma 2.3.9. *Given $P : A \to \mathcal{U}$ with $p : x =_A y$ and $q : y =_A z$ while $u : P(x)$, we have*

$$q_*(p_*(u)) = (p \cdot q)_*(u).$$

Lemma 2.3.10. *For a function $f : A \to B$ and a type family $P : B \to \mathcal{U}$, and any $p : x =_A y$ and $u : P(f(x))$, we have*

$$\mathsf{transport}^{P \circ f}(p, u) = \mathsf{transport}^{P}(\mathsf{ap}_f(p), u).$$

Lemma 2.3.11. *For $P, Q : A \to \mathcal{U}$ and a family of functions $f : \prod_{(x:A)} P(x) \to Q(x)$, and any $p : x =_A y$ and $u : P(x)$, we have*

$$\mathsf{transport}^{Q}(p, f_x(u)) = f_y(\mathsf{transport}^{P}(p, u)).$$

2.4 Homotopies and equivalences

So far, we have seen how the identity type $x =_A y$ can be regarded as a type of *identifications*, *paths*, or *equivalences* between two elements x and y of a type A. Now we investigate the appropriate notions of "identification" or "sameness" between *functions* and between *types*. In §2.5, we will see that homotopy type theory allows us to identify these with instances of the identity type, but before we can do that we need to understand them in their own right.

Traditionally, we regard two functions as the same if they take equal values on all inputs. Under the propositions-as-types interpretation, this suggests that two functions f and g (perhaps dependently typed) should be the same if the type $\prod_{(x:A)}(f(x) = g(x))$ is inhabited. Under the homotopical interpretation, this dependent function type consists of *continuous* paths or *functorial* equivalences, and thus may be regarded as the type of *homotopies* or of *natural isomorphisms*. We will adopt the topological terminology for this.

Definition 2.4.1. Let $f, g : \prod_{(x:A)} P(x)$ be two sections of a type family $P : A \to \mathcal{U}$. A **homotopy** from f to g is a dependent function of type

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

Note that a homotopy is not the same as an identification $(f = g)$. In §2.9 we will show that homotopies and identifications are nevertheless "equivalent".

The following proofs are left to the reader.

Lemma 2.4.2. *Homotopy is an equivalence relation on each function type $A \to B$. That is, we have elements of the types*

$$\prod_{f:A\to B} (f \sim f)$$
$$\prod_{f,g:A\to B} (f \sim g) \to (g \sim f)$$
$$\prod_{f,g,h:A\to B} (f \sim g) \to (g \sim h) \to (f \sim h).$$

Lemma 2.4.3. *Composition is associative and unital up to homotopy. That is:*

(i) If $f : A \to B$ then $f \circ \mathsf{id}_A \sim f \sim \mathsf{id}_B \circ f$.

(ii) If $f : A \to B, g : B \to C$ and $h : C \to D$ then $h \circ (g \circ f) \sim (h \circ g) \circ f$.

The first level of the continuity/naturality of homotopies can be expressed as follows:

Lemma 2.4.4. *Suppose $H : f \sim g$ is a homotopy between functions $f, g : A \to B$ and let $p : x =_A y$. Then we have*

$$H(x) \cdot g(p) = f(p) \cdot H(y).$$

We may also draw this as a commutative diagram:

$$\begin{array}{ccc} f(x) & \xrightarrow{f(p)} & f(y) \\ {\scriptstyle H(x)}\downarrow & & \downarrow{\scriptstyle H(y)} \\ g(x) & \xrightarrow[g(p)]{} & g(y) \end{array}$$

Proof. By induction, we may assume p is refl_x. Since ap_f and ap_g compute on reflexivity, in this case what we must show is

$$H(x) \cdot \mathsf{refl}_{g(x)} = \mathsf{refl}_{f(x)} \cdot H(x).$$

But this follows since both sides are equal to $H(x)$. □

Corollary 2.4.5. *Let $H : f \sim \mathsf{id}_A$ be a homotopy, with $f : A \to A$. Then for any $x : A$ we have*

$$H(f(x)) = f(H(x)).$$

The above path will be denoted by $\mathsf{swap}_{H,f}(x)$.

Proof. By naturality of H, the following diagram commutes:

$$\begin{array}{ccc} ffx & \xrightarrow{f(Hx)} & fx \\ {\scriptstyle H(fx)}\downarrow & & \downarrow{\scriptstyle Hx} \\ fx & \xrightarrow[Hx]{} & x \end{array}$$

Canceling $H(x)$, we see that $H(f(x)) = f(H(x))$ as desired. □

Moving on to types, from a traditional perspective one may say that a function $f : A \to B$ is an *isomorphism* if there is a function $g : B \to A$ such that both composites $f \circ g$ and $g \circ f$ are pointwise equal to the identity, i.e. such that $f \circ g \sim \mathsf{id}_B$ and $g \circ f \sim \mathsf{id}_A$. A homotopical perspective suggests that this should be called a *homotopy equivalence*, and from a categorical one, it should be called an *equivalence of (higher) groupoids*. However, when doing proof-relevant mathematics, the corresponding type

$$\sum_{g:B\to A} \big((f \circ g \sim \mathsf{id}_B) \times (g \circ f \sim \mathsf{id}_A)\big) \tag{2.4.6}$$

is poorly behaved. For instance, for a single function $f : A \to B$ there may be multiple unequal inhabitants of (2.4.6). (This is closely related to the observation in higher category theory that often one needs to consider *adjoint* equivalences rather than plain equivalences.) For this reason, we give (2.4.6) the following historically accurate, but slightly derogatory-sounding name instead.

Definition 2.4.7. For a function $f : A \to B$, a **quasi-inverse** of f is a triple (g, α, β) consisting of a function $g : B \to A$ and homotopies $\alpha : f \circ g \sim \mathsf{id}_B$ and $\beta : g \circ f \sim \mathsf{id}_A$.

Thus, (2.4.6) is **the type of quasi-inverses of** f; we may denote it by $\mathsf{qinv}(f)$.

Example 2.4.8. The identity function $\mathsf{id}_A : A \to A$ has a quasi-inverse given by id_A itself, together with homotopies defined by $\alpha(y) :\equiv \mathsf{refl}_y$ and $\beta(x) :\equiv \mathsf{refl}_x$.

Example 2.4.9. For any $p : x =_A y$ and $z : A$, the functions

$$(p \cdot -) : (y =_A z) \to (x =_A z) \qquad \text{and}$$
$$(- \cdot p) : (z =_A x) \to (z =_A y)$$

have quasi-inverses given by $(p^{-1} \cdot -)$ and $(- \cdot p^{-1})$, respectively.

Example 2.4.10. For any $p : x =_A y$ and $P : A \to \mathcal{U}$, the function

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

has a quasi-inverse given by $\mathsf{transport}^P(p^{-1}, -)$.

In general, we will only use the word *isomorphism* (and similar words such as *bijection*) in the special case when the types A and B "behave like sets" (see §3.1). In this case, the type (2.4.6) is unproblematic. We will reserve the word *equivalence* for an improved notion with the following properties:

(i) For each $f : A \to B$ there is a function $\mathsf{qinv}(f) \to \mathsf{isequiv}(f)$.
(ii) Similarly, for each f we have $\mathsf{isequiv}(f) \to \mathsf{qinv}(f)$; thus the two are "logically equivalent".
(iii) For any two inhabitants $e_1, e_2 : \mathsf{isequiv}(f)$ we have $e_1 = e_2$.

In Chapter 4 we will see that there are many different definitions of $\mathsf{isequiv}(f)$ which satisfy these three properties, but that all of them are equivalent. For now, to convince the reader that

such things exist, we mention only the easiest such definition (though it is not the one we will eventually settle on in Chapter 4):

$$\mathsf{isequiv}(f) :\equiv \Big(\sum_{g:B\to A} (f\circ g \sim \mathsf{id}_B)\Big) \times \Big(\sum_{h:B\to A} (h\circ f \sim \mathsf{id}_A)\Big) \tag{2.4.11}$$

We can show (i) and (ii) for this definition now. A function $\mathsf{qinv}(f) \to \mathsf{isequiv}(f)$ is easy to define by taking (g, α, β) to (g, α, g, β). In the other direction, given (g, α, h, β), let γ be the composite homotopy

$$g \overset{\beta}{\sim} h\circ f\circ g \overset{\alpha}{\sim} h$$

and let $\beta' : g\circ f \sim \mathsf{id}_A$ be obtained from γ and β. Then $(g, \alpha, \beta') : \mathsf{qinv}(f)$.

Property (iii) for this definition is not too hard to prove either, but it requires identifying the identity types of cartesian products and dependent pair types, which we will discuss in §2.5. Thus, we postpone it as well. At this point, the main thing to take away is that there is a well-behaved type which we can pronounce as "f is an equivalence", and that we can prove f to be an equivalence by exhibiting a quasi-inverse to it. In practice, this is the most common approach.

In accord with the proof-relevant philosophy, *an equivalence* from A to B is defined to be a function $f : A \to B$ together with an inhabitant of $\mathsf{isequiv}(f)$, i.e. a proof that it is an equivalence. We write $(A \simeq B)$ for the type of equivalences from A to B, i.e. the type

$$(A \simeq B) :\equiv \sum_{f:A\to B} \mathsf{isequiv}(f).$$

Property (iii) above will ensure that if two equivalences are equal as functions (that is, the underlying elements of $A \to B$ are equal), then they are also equal as equivalences (see §2.7).

We conclude by observing:

Lemma 2.4.12. *Type equivalence is an equivalence relation on $\mathcal{U}$. More specifically:*

(i) For any A, the identity function id_A is an equivalence; hence $A \simeq A$.

(ii) For any $f : A \simeq B$, we have an equivalence $f^{-1} : B \simeq A$.

(iii) For any $f : A \simeq B$ and $g : B \simeq C$, we have $g\circ f : A \simeq C$.

Proof. The identity function is clearly its own quasi-inverse; hence it is an equivalence.

If $f : A \to B$ is an equivalence, then it has a quasi-inverse, say $f^{-1} : B \to A$. Then f is also a quasi-inverse of f^{-1}, so f^{-1} is an equivalence $B \to A$.

Finally, given $f : A \simeq B$ and $g : B \simeq C$ with quasi-inverses f^{-1} and g^{-1}, say, then for any $a : A$ we have $f^{-1}g^{-1}gfa = f^{-1}fa = a$, and for any $c : C$ we have $gff^{-1}g^{-1}c = gg^{-1}c = c$. Thus $f^{-1}\circ g^{-1}$ is a quasi-inverse to $g\circ f$, hence the latter is an equivalence. □

2.5 The higher groupoid structure of type formers

In Chapter 1, we introduced many ways to form new types: cartesian products, disjoint unions, dependent products, dependent sums, etc. In §§2.1–2.3, we saw that *all* types in homotopy type

theory behave like spaces or higher groupoids. Our goal in the rest of the chapter is to make explicit how this higher structure behaves in the case of the particular types defined in Chapter 1.

It turns out that for many types A, the equality types $x =_A y$ can be characterized, up to equivalence, in terms of whatever data was used to construct A. For example, if A is a cartesian product $B \times C$, and $x \equiv (b, c)$ and $y \equiv (b', c')$, then we have an equivalence

$$\Big((b, c) = (b', c')\Big) \simeq \Big((b = b') \times (c = c')\Big). \tag{2.5.1}$$

In more traditional language, two ordered pairs are equal just when their components are equal (but the equivalence (2.5.1) says rather more than this). The higher structure of the identity types can also be expressed in terms of these equivalences; for instance, concatenating two equalities between pairs corresponds to pairwise concatenation.

Similarly, when a type family $P : A \to \mathcal{U}$ is built up fiberwise using the type forming rules from Chapter 1, the operation $\mathsf{transport}^P(p, -)$ can be characterized, up to homotopy, in terms of the corresponding operations on the data that went into P. For instance, if $P(x) \equiv B(x) \times C(x)$, then we have

$$\mathsf{transport}^P(p, (b, c)) = \Big(\mathsf{transport}^B(p, b), \mathsf{transport}^C(p, c)\Big).$$

Finally, the type forming rules are also functorial, and if a function f is built from this functoriality, then the operations ap_f and apd_f can be computed based on the corresponding ones on the data going into f. For instance, if $g : B \to B'$ and $h : C \to C'$ and we define $f : B \times C \to B' \times C'$ by $f(b, c) :\equiv (g(b), h(c))$, then modulo the equivalence (2.5.1), we can identify ap_f with "$(\mathsf{ap}_g, \mathsf{ap}_h)$".

The rest of this chapter will be devoted to stating and proving theorems of this sort for all the basic type forming rules, with one section for each basic type former. It is here that we encounter most clearly the defects of currently available type theories. As will become clear in later chapters, it would be most convienient and intuitive if these characterizations of identity types, transport, and so on were *judgmental* equalities. However, in the theory presented in Chapter 1, the identity types are defined simultaneously for all types by their induction principle, so we cannot "redefine" them to be different things at different types. Thus, the characterizations for particular types to be discussed in this chapter are, for the most part, *theorems* which we have to discover and prove.

Actually, the type theory of Chapter 1 is insufficient to prove the desired theorems for two of the type formers: Π-types and universes. For this reason, we are forced to introduce axioms into our type theory, in order to make those "theorems" true. Type-theoretically, an *axiom* (c.f. §1.1) is an "atomic" element that is declared to inhabit some specified type, without there being any rules governing its behavior other than those pertaining to the type it inhabits.

The axiom for Π-types (§2.9) is familiar to type theorists: it is called *function extensionality*, and states (roughly) that if two functions are homotopic in the sense of §2.4, then they are equal. The axiom for universes, however (§2.10), is a new contribution of homotopy type theory due to Voevodsky: it is called the *univalence axiom*, and states (roughly) that if two types are equivalent in the sense of §2.4, then they are equal. We have already remarked on this axiom in the introduction; it will play a very important role in this book.[1]

[1] We have chosen to introduce these principles as axioms, but there are potentially other ways to formulate a type theory in which they hold. See the Notes to this chapter.

It is important to note, though, that not *all* identity types can be "defined" by induction over the construction of types. Counterexamples include most nontrivial higher inductive types (see Chapters 6 and 8). For instance, calculating the identity types of the types $\mathbb{S}^n$ (see §6.4) is equivalent to calculating the higher homotopy groups of spheres, a deep and important field of research in algebraic topology.

2.6 Cartesian product types

Given types A and B, consider the cartesian product type $A \times B$. For any elements $x, y : A \times B$ and a path $p : x =_{A\times B} y$, by functoriality we can extract paths $\mathsf{pr}_1(p) : \mathsf{pr}_1(x) =_A \mathsf{pr}_1(y)$ and $\mathsf{pr}_2(p) : \mathsf{pr}_2(x) =_B \mathsf{pr}_2(y)$. Thus, we have a function

$$(x =_{A\times B} y) \;\to\; (\mathsf{pr}_1(x) =_A \mathsf{pr}_1(y)) \times (\mathsf{pr}_2(x) =_B \mathsf{pr}_2(y)). \tag{2.6.1}$$

Theorem 2.6.2. *For any x and y, the function (2.6.1) is an equivalence.*

Read logically, this says that two pairs are equal if they are equal componentwise. Read category-theoretically, this says that the morphisms in a product groupoid are pairs of morphisms. Read homotopy-theoretically, this says that the paths in a product space are pairs of paths.

Proof. We need a function in the other direction:

$$(\mathsf{pr}_1(x) =_A \mathsf{pr}_1(y)) \times (\mathsf{pr}_2(x) =_B \mathsf{pr}_2(y)) \;\to\; (x =_{A\times B} y). \tag{2.6.3}$$

By the induction rule for cartesian products, we may assume that x and y are both pairs, i.e. $x \equiv (a, b)$ and $y \equiv (a', b')$ for some $a, a' : A$ and $b, b' : B$. In this case, what we want is a function

$$(a =_A a') \times (b =_B b') \;\to\; \big((a, b) =_{A\times B} (a', b')\big).$$

Now by induction for the cartesian product in its domain, we may assume given $p : a = a'$ and $q : b = b'$. And by two path inductions, we may assume that $a \equiv a'$ and $b \equiv b'$ and both p and q are reflexivity. But in this case, we have $(a, b) \equiv (a', b')$ and so we can take the output to also be reflexivity.

It remains to prove that (2.6.3) is quasi-inverse to (2.6.1). This is a simple sequence of inductions, but they have to be done in the right order.

In one direction, let us start with $r : x =_{A\times B} y$. We first do a path induction on r in order to assume that $x \equiv y$ and r is reflexivity. In this case, since $\mathsf{ap}_{\mathsf{pr}_1}$ and $\mathsf{ap}_{\mathsf{pr}_2}$ are defined by path induction, (2.6.1) takes $r \equiv \mathsf{refl}_x$ to the pair $(\mathsf{refl}_{\mathsf{pr}_1 x}, \mathsf{refl}_{\mathsf{pr}_2 x})$. Now by induction on x, we may assume $x \equiv (a, b)$, so that this is $(\mathsf{refl}_a, \mathsf{refl}_b)$. Thus, (2.6.3) takes it by definition to $\mathsf{refl}_{(a,b)}$, which (under our current assumptions) is r.

In the other direction, if we start with $s : (\mathsf{pr}_1(x) =_A \mathsf{pr}_1(y)) \times (\mathsf{pr}_2(x) =_B \mathsf{pr}_2(y))$, then we first do induction on x and y to assume that they are pairs (a, b) and (a', b'), and then induction on $s : (a =_A a') \times (b =_B b')$ to reduce it to a pair (p, q) where $p : a = a'$ and $q : b = b'$. Now by induction on p and q, we may assume they are reflexivities refl_a and refl_b, in which case (2.6.3) yields $\mathsf{refl}_{(a,b)}$ and then (2.6.1) returns us to $(\mathsf{refl}_a, \mathsf{refl}_b) \equiv (p, q) \equiv s$. □

In particular, we have shown that (2.6.1) has an inverse (2.6.3), which we may denote by

$$\mathsf{pair}^{=} : (\mathsf{pr}_1(x) = \mathsf{pr}_1(y)) \times (\mathsf{pr}_1(x) = \mathsf{pr}_1(y)) \;\to\; (x = y)$$

Note that a special case of this yields the η-equivalence rule for products: $z = (\mathsf{pr}_1(z), \mathsf{pr}_2(z))$.

It can be helpful to view $\mathsf{pair}^{=}$ as a *constructor* or *introduction rule* for $x = y$, analogous to the "pairing" constructor of $A \times B$ itself, which introduces the pair (a, b) given $a : A$ and $b : B$. From this perspective, the two components of (2.6.1):

$$\mathsf{ap}_{\mathsf{pr}_1} : (x = y) \to (\mathsf{pr}_1(x) = \mathsf{pr}_1(y))$$
$$\mathsf{ap}_{\mathsf{pr}_2} : (x = y) \to (\mathsf{pr}_2(x) = \mathsf{pr}_2(y))$$

are *elimination* rules. Similarly, the two homotopies which witness (2.6.3) as quasi-inverse to (2.6.1) consist respectively, of *computation* or *β-reduction* rules:

$$\mathsf{ap}_{\mathsf{pr}_1}(\mathsf{pair}^{=}(p, q)) = p \qquad \text{for } p : \mathsf{pr}_1 x = \mathsf{pr}_1 y$$
$$\mathsf{ap}_{\mathsf{pr}_2}(\mathsf{pair}^{=}(p, q)) = q \qquad \text{for } q : \mathsf{pr}_2 x = \mathsf{pr}_2 y$$

and an *η-equivalence* rule:

$$r = \mathsf{pair}^{=}(\mathsf{ap}_{\mathsf{pr}_1}(r), \mathsf{ap}_{\mathsf{pr}_2}(r)) \qquad \text{for } r : x =_{A\times B} y$$

We can also characterize the reflexivity, inverses, and composition of paths in $A \times B$ componentwise:

$$\begin{aligned} \mathsf{refl}_{(z:A\times B)} &= \mathsf{pair}^{=}(\mathsf{refl}_{\mathsf{pr}_1 z}, \mathsf{refl}_{\mathsf{pr}_2 z}) \\ p^{-1} &= \mathsf{pair}^{=}\big(\mathsf{ap}_{\mathsf{pr}_1}(p)^{-1}, \mathsf{ap}_{\mathsf{pr}_2}(p)^{-1}\big) \\ p \cdot q &= \mathsf{pair}^{=}\big(\mathsf{ap}_{\mathsf{pr}_1}(p) \cdot \mathsf{ap}_{\mathsf{pr}_1}(q), \mathsf{ap}_{\mathsf{pr}_2}(p) \cdot \mathsf{ap}_{\mathsf{pr}_2}(q)\big) \end{aligned}$$

The same is true for the rest of the higher groupoid structure considered in §2.1. All of these equations can be derived by using path induction on the given paths and then returning reflexivity.

We now consider transport in a pointwise product of type families. Given type families $A, B : Z \to \mathcal{U}$, we abusively write $A \times B : Z \to \mathcal{U}$ for the type family defined by $(A \times B)(z) :\equiv A(z) \times B(z)$. Now given $p : z =_Z w$ and $x : A(z) \times B(z)$, we can transport x along p to obtain an element of $A(w) \times B(w)$.

Theorem 2.6.4. *In the above situation, we have*

$$\mathsf{transport}^{A\times B}(p, x) =_{A(y)\times B(y)} (\mathsf{transport}^{A}(p, \mathsf{pr}_1 x), \mathsf{transport}^{B}(p, \mathsf{pr}_2 x))$$

Proof. By path induction, we may assume p is reflexivity, in which case we have

$$\mathsf{transport}^{A\times B}(p, x) \equiv x$$
$$\mathsf{transport}^{A}(p, \mathsf{pr}_1 x) \equiv \mathsf{pr}_1 x$$
$$\mathsf{transport}^{A}(p, \mathsf{pr}_2 x) \equiv \mathsf{pr}_2 x.$$

Thus, it remains to show $x = (\mathsf{pr}_1 x, \mathsf{pr}_2 x)$. But this is η-equivalence, which as we remarked above follows from Theorem 2.6.2. □

Finally, we consider the functoriality of ap under cartesian products. Suppose given types A, B, A', B' and functions $g : A \to A'$ and $h : B \to B'$; then we can define a function $f : A \times B \to A' \times B'$ by $f(x) :\equiv (g(\mathsf{pr}_1 x), h(\mathsf{pr}_2 x))$.

Theorem 2.6.5. *In the above situation, given $x, y : A \times B$ and $p : \mathsf{pr}_1 x = \mathsf{pr}_1 y$ and $q : \mathsf{pr}_2 x = \mathsf{pr}_2 y$, we have*

$$f(\mathsf{pair}^{=}(p, q)) =_{(f(x)=f(y))} \mathsf{pair}^{=}(g(p), h(q)).$$

Proof. Note first that the above equation is well-typed. On the one hand, since $\mathsf{pair}^{=}(p, q) : x = y$ we have $f(\mathsf{pair}^{=}(p, q)) : f(x) = f(y)$. On the other hand, since $\mathsf{pr}_1(f(x)) \equiv g(\mathsf{pr}_1 x)$ and $\mathsf{pr}_2(f(x)) \equiv h(\mathsf{pr}_2 x)$, we also have $\mathsf{pair}^{=}(g(p), h(q)) : f(x) = f(y)$.

Now, by induction, we may assume $x \equiv (a, b)$ and $y \equiv (a', b')$, in which case we have $p : a = a'$ and $q : b = b'$. Thus, by path induction, we may assume p and q are reflexivity, in which case the desired equation holds judgmentally. □

2.7 Σ-types

Let A be a type and $B : A \to \mathcal{U}$ a type family. Recall that the Σ-type, or dependent pair type, $\sum_{(x:A)} B(x)$ is a generalization of the cartesian product type. Thus, we expect its higher groupoid structure to also be a generalization of the previous section. In particular, its paths should be pairs of paths, but it takes a little thought to give the correct types of these paths.

Suppose that we have a path $p : w = w'$ in $\sum_{(x:A)} P(x)$. Then we get $\mathsf{pr}_1(p) : \mathsf{pr}_1(w) = \mathsf{pr}_1(w')$. However, we cannot directly ask whether $\mathsf{pr}_2(w)$ is identical to $\mathsf{pr}_2(w')$ since they don't have to be in the same type. But we can transport $\mathsf{pr}_2(w)$ along the path $\mathsf{pr}_1(p)$, and this does give us an element of the same type as $\mathsf{pr}_2(w')$. By path induction, we do in fact obtain a path $\mathsf{pr}_1(p)_*(\mathsf{pr}_2(w)) = \mathsf{pr}_2(w')$.

Recall from the discussion preceeding Lemma 2.3.4 that $\mathsf{pr}_1(p)_*(\mathsf{pr}_2(w)) = \mathsf{pr}_2(w')$ can be regarded as the type of paths from $\mathsf{pr}_2(w)$ to $\mathsf{pr}_2(w')$ which lie over the path $\mathsf{pr}_1(p)$ in A. Thus, we are saying that a path $w = w'$ in the total space determines (and is determined by) a path $p : \mathsf{pr}_1(w) = \mathsf{pr}_1(w')$ in A together with a path from $\mathsf{pr}_2(w)$ to $\mathsf{pr}_2(w')$ lying over p, which seems sensible.

Remark 2.7.1. Note that if we have $x : A$ and $u, v : P(x)$ such that $(x, u) = (x, v)$, it does not follow that $u = v$. All we can conclude is that there exists $p : x = x$ such that $p_*(u) = v$. This is a well-known source of confusion for newcomers to type theory, but it makes sense from a topological viewpoint: the existence of a path $(x, u) = (x, v)$ in the total space of a fibration betweeen two points that happen to lie in the same fiber does not imply the existence of a path $u = v$ lying entirely *within* that fiber.

The next theorem states that we can also reverse this process. Since it is a direct generalization of Theorem 2.6.2, we will be more concise.

Theorem 2.7.2. *Suppose that $P : A \to \mathcal{U}$ is a type family over a type A and let $w, w' : \sum_{(x:A)} P(x)$. Then there is an equivalence*

$$(w = w') \simeq \sum_{(p:\mathsf{pr}_1(w)=\mathsf{pr}_1(w'))} p_*(\mathsf{pr}_2(w)) = \mathsf{pr}_2(w').$$

Proof. We define for any $w, w' : \sum_{(x:A)} P(x)$, a function

$$f : (w = w') \to \sum_{(p:\mathsf{pr}_1(w)=\mathsf{pr}_1(w'))} p_*(\mathsf{pr}_2(w)) = \mathsf{pr}_2(w')$$

by path induction, with

$$f(w, w, \mathsf{refl}_w) :\equiv (\mathsf{refl}_{\mathsf{pr}_1(w)}, \mathsf{refl}_{\mathsf{pr}_2(w)}).$$

We want to show that f is an equivalence.

In the reverse direction, we define

$$g : \prod_{(w,w':\sum_{(x:A)} P(x))} \left(\sum_{(p:\mathsf{pr}_1(w)=\mathsf{pr}_1(w'))} p_*(\mathsf{pr}_2(w)) = \mathsf{pr}_2(w') \right) \to (w = w')$$

by first inducting on w and w', which splits them into (w_1, w_2) and (w_1', w_2') respectively, so it suffices to show

$$\left(\sum_{(p:w_1=w_1')} p_*(w_2) = w_2' \right) \to ((w_1, w_2) = (w_1', w_2'))$$

Next, given a pair $\sum_{(p:w_1=w_1')} p_*(w_2) = w_2'$, we can use Σ-induction to get $p : w_1 = w_1'$ and $q : p_*(w_2) = w_2'$. Inducting on p, we have $q : \mathsf{refl}_*(w_2) = w_2'$, and it suffices to show $(w_1, w_2) = (w_1, w_2')$. But $\mathsf{refl}_*(w_2) \equiv w_2$, so inducting on q reduces to the goal to $(w_1, w_2) = (w_1, w_2)$, which we can prove with $\mathsf{refl}_{(w_1,w_2)}$.

Next we show that $f \circ g$ is the identity for all w, w' and r, where r has type

$$\sum_{(p:\mathsf{pr}_1(w)=\mathsf{pr}_1(w'))} (p_*(\mathsf{pr}_2(w)) = \mathsf{pr}_2(w')).$$

First, we break apart the pairs w, w', and r by pair induction, as in the definition of g, and then use two path inductions to reduce both components of r to refl. Then it suffices to show that $f(g(\mathsf{refl}, \mathsf{refl})) = \mathsf{refl}$, which is true by definition.

Similarly, to show that $g \circ f$ is the identity for all w, w', and $p : w = w'$, we can do path-induction p, and then induction to split w, at which point it suffices to show that $g(f(\mathsf{refl}_{(w_1,w_2)})) = \mathsf{refl}_{(w_1,w_2)}$, which is true by definition.

Thus, f has a quasi-inverse, and is therefore an equivalence. □

As we did in the case of cartesian products, we have η-equivalence as a special case.

Corollary 2.7.3. *For $z : \sum_{(x:A)} P(x)$, we have $z = (\mathsf{pr}_1(z), \mathsf{pr}_2(z))$.*

Proof. We have $\mathsf{refl}_{\mathsf{pr}_1(z)} : \mathsf{pr}_1(z) = \mathsf{pr}_1(\mathsf{pr}_1(z), \mathsf{pr}_2(z))$, so by Theorem 2.7.2 it will suffice to exhibit a path $(\mathsf{refl}_{\mathsf{pr}_1(z)})_*(\mathsf{pr}_2(z)) = \mathsf{pr}_2(\mathsf{pr}_1(z), \mathsf{pr}_2(z))$. But both sides are judgmentally equal to $\mathsf{pr}_2(z)$. □

Like with binary cartesian products, we can think of the backward direction of Theorem 2.7.2 as an introduction form ($\mathsf{pair}^=$), the forward direction as elimination forms ($\mathsf{ap}_{\mathsf{pr}_1}$ and $\mathsf{ap}_{\mathsf{pr}_2}$), and the equivalence as giving β and η rules for these.

Note that the lifted path $\mathsf{lift}(u, p)$ of $p : x = y$ at $u : P(x)$ defined in Lemma 2.3.2 may be identified with the special case of the introduction form

$$\mathsf{pair}^{=}(p, \mathsf{refl}_{p_*(u)}) : (x, u) = (y, p_*(u)).$$

This appears in the statement of action of transport on Σ-types, which is also a generalization of the action for binary cartesian products:

Theorem 2.7.4. *Suppose we have type families $P : A \to \mathcal{U}$ and $Q : \left(\sum_{(x:A)} P(x)\right) \to \mathcal{U}$. Then we can construct the type family over A defined by*

$$x \mapsto \sum_{u:P(x)} Q(x, u).$$

For any path $p : x = y$ and any $(u, z) : \sum_{(u:P(x))} Q(x, u)$ we have

$$p_*(u, z) = \left(p_*(u),\ \mathsf{pair}^{=}(p, \mathsf{refl}_{p_*(u)})_*(z)\right).$$

Proof. Immediate by path induction. □

We leave it to the reader to state and prove a generalization of Theorem 2.6.5 (see Exercise 2.6), and to characterize the reflexivity, inverses, and composition of Σ-types componentwise.

2.8 The unit type

Trivial cases are sometimes important, so we mention briefly the case of the unit type $\mathbf{1}$.

Theorem 2.8.1. *For any $x, y : \mathbf{1}$, we have $(x = y) \simeq \mathbf{1}$.*

Proof. A function $(x = y) \to \mathbf{1}$ is easy to define by sending everything to $\star$. Conversely, for any $x, y : \mathbf{1}$ we may assume by induction that $x \equiv \star \equiv y$. In this case we have $\mathsf{refl}_\star : x = y$, yielding a constant function $\mathbf{1} \to (x = y)$.

To show that these are inverses, consider first an element $u : \mathbf{1}$. We may assume that $u \equiv \star$, but this is also the result of the composite $\mathbf{1} \to (x = y) \to \mathbf{1}$.

On the other hand, suppose given $p : x = y$. By path induction, we may assume $x \equiv y$ and p is refl_x. We may then assume that x is $\star$, in which case the composite $(x = y) \to \mathbf{1} \to (x = y)$ takes p to refl_x, i.e. to p. □

In particular, any two elements of $\mathbf{1}$ are equal. We leave it to the reader to formulate this equivalence in terms of introduction, elimination, β, and η rules. The transport lemma for $\mathbf{1}$ is simply the transport lemma for constant type families.

2.9 Π-types and the function extensionality axiom

Given a type A and a type family $B : A \to \mathcal{U}$, consider the dependent function type $\prod_{(x:A)} B(x)$. We expect the type $f = g$ of paths from f to g in $\prod_{(x:A)} B(x)$ to be equivalent to the type of pointwise paths:

$$(f = g) \simeq \left(\prod_{x:A} (f(x) =_{B(x)} g(x))\right) \tag{2.9.1}$$

From a traditional perspective, this would say that two functions which are equal at each point are equal as functions. From a topological perspective, it would say that a path in a function space is the same as a continuous homotopy. And from a categorical perspective, it would say that an isomorphism in a functor category is a natural family of isomorphisms.

Unlike the case in the previous sections, however, the basic type theory presented in Chapter 1 is insufficient to prove (2.9.1). All we can say is that there is a certain function

$$\mathsf{happly} : (f = g) \to \prod_{x:A} (f(x) =_{B(x)} g(x)) \tag{2.9.2}$$

which is easily defined by path induction. For the moment, therefore, we will assume:

Axiom 2.9.3 (Function extensionality)**.** *For any A, B, f, and g, the function* (2.9.2) *is an equivalence.*

We will see in later chapters that this axiom follows both from univalence (see §§2.10 and 4.9) and from an interval type (see §6.3).

In particular, Axiom 2.9.3 implies that (2.9.2) has a quasi-inverse

$$\mathsf{funext} : \left(\prod_{x:A} (f(x) = g(x))\right) \to (f = g).$$

This function is also referred to as "function extensionality". As we did with $\mathsf{pair}^=$ in §2.6, we can regard funext as an *introduction rule* for the type $f = g$. From this point of view, happly is the *elimination rule*, while the homotopies witnessing funext as quasi-inverse to happly become a β-reduction rule

$$\mathsf{happly}(\mathsf{funext}(h), x) = h(x) \qquad \text{for } h : \prod_{x:A} (f(x) = g(x))$$

and an η-equivalence rule:

$$p = \mathsf{funext}(x \mapsto \mathsf{happly}(p, x)) \qquad \text{for } p : (f = g)$$

We can also compute the identity, inverses, and composition in Π-types; they are simply given by pointwise operations.

$$\begin{aligned}
\mathsf{refl}_f &= \mathsf{funext}(x \mapsto \mathsf{refl}_{f(x)}) \\
\alpha^{-1} &= \mathsf{funext}(x \mapsto \mathsf{happly}(\alpha)(x)^{-1}) \\
\alpha \cdot \beta &= \mathsf{funext}(x \mapsto \mathsf{happly}(\alpha)(x) \cdot \mathsf{happly}(\beta)(x))
\end{aligned}$$

Since the non-dependent function type $A \to B$ is a special case of the dependent function type $\prod_{(x:A)} B(x)$ when B is independent of x, everything we have said above applies in non-dependent cases as well. The rules for transport, however, are somewhat simpler in the non-dependent case. Given a type X, a path $p : x_1 =_X x_2$, type families $A, B : X \to \mathcal{U}$, and a function $f : A(x_1) \to B(x_1)$, we have

$$\mathsf{transport}^{A\to B}(p, f) = \Big(x \mapsto \mathsf{transport}^{B}(p, f(\mathsf{transport}^{A}(p^{-1}, x)))\Big) \tag{2.9.4}$$

where $A \to B$ denotes abusively the type family $X \to \mathcal{U}$ defined by

$$(A \to B)(x) :\equiv (A(x) \to B(x)).$$

In other words, when we transport a function $f : A(x_1) \to B(x_1)$ along a path $p : x_1 = x_2$, we obtain the function $A(x_2) \to B(x_2)$ which transports its argument backwards along p (in the type family A), applies f, and then transports the result forwards along p (in the type family B). This can be proven easily by path induction.

Transporting dependent functions is similar, but more complicated. Suppose given X and p as before, type families $A : X \to \mathcal{U}$ and $B : \prod_{(x:X)}(A(x) \to \mathcal{U})$, and also a dependent function $f : \prod_{(a:A(x_1))} B(x_1, a)$. Then for $p : x_1 =_A x_2$ and $a : A(x_2)$, we have

$$\mathsf{transport}^{\Pi_A(B)}(p, f)(a) = \mathsf{transport}^{\hat{B}}\Big((\mathsf{pair}^{=}(p^{-1}, \mathsf{refl}_{p^{-1}{}_*(a)}))^{-1}, f(\mathsf{transport}^{A}(p^{-1}, a))\Big)$$

where $\Pi_A(B)$ and $\hat{B}$ denote respectively the type families

$$\begin{array}{rcll} \Pi_A(B) & :\equiv & \big(x \mapsto \prod_{(a:A(x))} B(x, a)\big) & : \quad X \to \mathcal{U} \\ \hat{B} & :\equiv & \big(w \mapsto B(\mathsf{pr}_1 w, \mathsf{pr}_2 w)\big) & : \quad \big(\sum_{(x:X)} A(x)\big) \to \mathcal{U}. \end{array} \tag{2.9.5}$$

If these formulas look a bit intimidating, don't worry about the details. The basic idea is just the same as for the non-dependent function type: we transport the argument backwards, apply the function, and then transport the result forwards again.

Now recall that for a general type family $P : X \to \mathcal{U}$, in §2.2 we defined the type of *dependent paths* over $p : x =_X y$ from $u : P(x)$ to $v : P(y)$ to be $p_*(u) =_{P(y)} v$. When P is a family of function types, there is an equivalent way to represent this which is often more convenient.

Lemma 2.9.6. *Given type families $A, B : X \to \mathcal{U}$ and $p : x =_X y$, and also $f : A(x) \to B(x)$ and $g : A(y) \to B(y)$, we have an equivalence*

$$\big(p_*(f) = g\big) \simeq \prod_{a:A(x)} \big(p_*(f(a)) = g(p_*(a))\big).$$

Moreover, if $q : p_*(f) = g$ corresponds under this equivalence to $\hat{q}$, then for $a : A(x)$, the path

$$\mathsf{happly}(q, p_*(a)) : (p_*(f))(p_*(a)) = g(p_*(a))$$

is equal to the composite

$$\begin{aligned} (p_*(f))(p_*(a)) &= p_*\Big(f(p^{-1}{}_*(p_*(a)))\Big) && \text{by (2.9.4)} \\ &= p_*(f(a)) \\ &= g(p_*(a)) && \text{by } \hat{q}. \end{aligned}$$

Proof. By path induction, we may assume p is reflexivity, in which case the desired equivalence reduces to function extensionality. The second statement then follows by the computation rule for function extensionality. □

As usual, the case of dependent functions is similar, but more complicated.

Lemma 2.9.7. *Given type families $A : X \to \mathcal{U}$ and $B : \prod_{(x:X)} A(x) \to \mathcal{U}$ and $p : x =_X y$, and also $f : \prod_{(a:A(x))} B(x,a)$ and $g : \prod_{(a:A(y))} B(y,a)$, we have an equivalence*

$$(p_*(f) = g) \;\simeq\; \Big(\prod_{a:A(x)} \mathsf{transport}^{\hat{B}}(\mathsf{pair}^{=}(p, \mathsf{refl}_{p_*(a)}), f(a)) = g(p_*(a)) \Big)$$

with $\hat{B}$ as in (2.9.5).

We leave it to the reader to prove this and to formulate a suitable computation rule.

2.10 Universes and the univalence axiom

Given two types A and B, we may consider them as elements of some universe type $\mathcal{U}$, and thereby form the identity type $A =_{\mathcal{U}} B$. As mentioned in the introduction, *univalence* is the identification of $A =_{\mathcal{U}} B$ with the type $(A \simeq B)$ of equivalences from A to B, which we described in §2.4. We perform this identification by way of the following canonical function.

Lemma 2.10.1. *For types $A, B : \mathcal{U}$, there is a certain function,*

$$\mathsf{idtoeqv} : (A =_{\mathcal{U}} B) \to (A \simeq B), \tag{2.10.2}$$

defined in the proof.

Proof. We could construct this directly by induction on equality, but the following description is more convenient. Note that the identity function $\mathsf{id}_{\mathcal{U}} : \mathcal{U} \to \mathcal{U}$ may be regarded as a type family indexed by the universe $\mathcal{U}$; it assigns to each type $X : \mathcal{U}$ the type X itself. (When regarded as a fibration, its total space is the type $\sum_{(A:\mathcal{U})} A$ of "pointed types"; see also §4.8.) Thus, given a path $p : A =_{\mathcal{U}} B$, we have a transport function $p_* : A \to B$. We claim that p_* is an equivalence. But by induction, it suffices to assume that p is refl_A, in which case $p_* \equiv \mathsf{id}_A$, which is an equivalence by Example 2.4.8. Thus, we can define $\mathsf{idtoeqv}(p)$ to be p_* (together with the above proof that it is an equivalence). □

We would like to say that idtoeqv is an equivalence. However, as with happly for function types, the type theory described in Chapter 1 is insufficient to guarantee this. Thus, as we did for function extensionality, we formulate this property as an axiom: Voevodsky's *univalence axiom*.

Axiom 2.10.3 (Univalence)**.** *For any $A, B : \mathcal{U}$, the function* (2.10.2) *is an equivalence,*

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

Remark 2.10.4. It is important for the univalence axiom that we defined $A \simeq B$ using a "good" version of isequiv as described in §2.4, rather than (say) as $\sum_{(f:A\to B)} \mathsf{qinv}(f)$.

In particular, univalence means that *equivalent types may be identified*. As we did in previous sections, it is useful to break this equivalence into:

- An introduction rule for $(A =_{\mathcal{U}} B)$:
$$\mathsf{ua} : (A \simeq B) \to (A =_{\mathcal{U}} B)$$
- The elimination rule, which is idtoeqv:
$$\mathsf{idtoeqv} \equiv \mathsf{transport}^{X \mapsto X} : (A =_{\mathcal{U}} B) \to (A \simeq B)$$
- β-reduction:
$$\mathsf{transport}^{X \mapsto X}(\mathsf{ua}(f), x) = f(x)$$
- η-equivalence: For any $p : A = B$
$$p = \mathsf{ua}(\mathsf{transport}^{X \mapsto X}(p))$$

We can also identify the reflexivity, concatenation, and inverses of equalities in the universe with the corresponding operations on equivalences:

$$\begin{aligned} \mathsf{refl}_A &= \mathsf{ua}(\mathsf{id}_A) \\ \mathsf{ua}(f) \cdot \mathsf{ua}(g) &= \mathsf{ua}(g \circ f) \\ \mathsf{ua}(f)^{-1} &= \mathsf{ua}(f^{-1}) \end{aligned}$$

The first of these follows because $\mathsf{id}_A = \mathsf{idtoeqv}(\mathsf{refl}_A)$ by definition of idtoeqv, and ua is the inverse of idtoeqv. For the second, if we define $p :\equiv \mathsf{ua}(f)$ and $q :\equiv \mathsf{ua}(g)$, then we have

$$\mathsf{ua}(g \circ f) = \mathsf{ua}(\mathsf{idtoeqv}(q) \circ \mathsf{idtoeqv}(p)) = \mathsf{ua}(\mathsf{idtoeqv}(p \cdot q)) = p \cdot q$$

using Lemma 2.3.9 and the definition of idtoeqv. The third is similar.

The following observation, which is a special case of Lemma 2.3.10, is often useful when applying the univalence axiom.

Lemma 2.10.5. *For any type family $B : A \to \mathcal{U}$ and $x, y : A$ with a path $p : x = y$ and $u : B(x)$, we have*

$$\begin{aligned} \mathsf{transport}^{B}(p, u) &= \mathsf{transport}^{X \mapsto X}(\mathsf{ap}_B(p), u) \\ &= \mathsf{idtoeqv}(\mathsf{ap}_B(p))(u). \end{aligned}$$

2.11 Identity type

Just as the type $a =_A a'$ is characterized up to isomorphism, with a separate "definition" for each A, there is no simple characterization of the type $p =_{a =_A a'} q$ of paths between paths $p, q : a =_A a'$. However, our other general classes of theorems do extend to identity types, such as the fact that they respect equivalence.

Theorem 2.11.1. *If $f : A \to B$ is an equivalence, then for all $a, a' : A$, so is*

$$\mathsf{ap}_f : (a =_A a') \to (f(a) =_B f(a')).$$

Proof. Let f^{-1} be a quasi-inverse of f, with homotopies $\alpha : \prod_{(b:B)}(f(f^{-1}(b)) = b)$ and $\beta : \prod_{(a:A)}(f^{-1}(f(a)) = a)$. The quasi-inverse of ap_f is, essentially, $\mathsf{ap}_{f^{-1}}$. However, the type of $\mathsf{ap}_{f^{-1}}$ is

$$\mathsf{ap}_{f^{-1}} : (f(a) = f(a')) \to (f^{-1}(f(a)) = f^{-1}(f(a'))).$$

Thus, in order to obtain an element of $a =_A a'$ we must concatenate with the paths $\beta(a)^{-1}$ and $\beta(a')$ on either side. To show that this gives a quasi-inverse of ap_f, on one hand we must show that for any $p : a = a'$ we have

$$\beta(a)^{-1} \cdot \mathsf{ap}_{f^{-1}}(\mathsf{ap}_f(p)) \cdot \beta(a') = p.$$

This follows from the functoriality of ap on function composition (Lemma 2.2.2(iii)) and the naturality of homotopies (Lemma 2.4.4). On the other hand, we must show that for any $q : f(a) = f(a')$ we have

$$\mathsf{ap}_f\big(\beta(a)^{-1} \cdot \mathsf{ap}_{f^{-1}}(q) \cdot \beta(a')\big) = q.$$

This follows in the same way, using also the functoriality of ap on path-concatenation and inverses (Lemma 2.2.2(i) and (ii)). □

Thus, if for some type A we have a full characterization of $a =_A a'$, the type $p =_{a=_A a'} q$ is determined as well. For example:

- Paths $p = q$, where $p, q : w =_{A \times B} w'$, are equivalent to pairs of paths

$$\mathsf{ap}_{\mathsf{pr}_1} p =_{\mathsf{pr}_1 w =_A \mathsf{pr}_1 w'} \mathsf{ap}_{\mathsf{pr}_1} q \quad \text{and} \quad \mathsf{ap}_{\mathsf{pr}_2} p =_{\mathsf{pr}_2 w =_B \mathsf{pr}_2 w'} \mathsf{ap}_{\mathsf{pr}_2} q.$$

- Paths $p = q$, where $p, q : f =_{\prod_{(x:A)} B(x)} g$, are equivalent to homotopies

$$\prod_{x:A} (\mathsf{happly}(p)(x) =_{B(x)} \mathsf{happly}(q)(x)).$$

Next we consider transport in families of paths, i.e. transport in $C : A \to \mathcal{U}$ where each $C(x)$ is an identity type. The simplest case is when $C(x)$ is a type of paths in A itself, perhaps with one endpoint fixed.

Lemma 2.11.2. *For any A and $a : A$, with $p : x_1 = x_2$, we have*

$$\begin{aligned}
&\mathsf{transport}^{x \mapsto (a = x)}(p, q) = q \cdot p && \text{for } q : a = x_1 \\
&\mathsf{transport}^{x \mapsto (x = a)}(p, q) = p^{-1} \cdot q && \text{for } q : x_1 = a \\
&\mathsf{transport}^{x \mapsto (x = x)}(p, q) = p^{-1} \cdot q \cdot p && \text{for } q : x_1 = x_1.
\end{aligned}$$

Proof. Path induction on p, followed by the unit laws for composition. □

In other words, transporting with $x \mapsto c = x$ is post-composition, and transporting with $x \mapsto x = c$ is contravariant pre-composition. These may be familiar as the functorial actions of the covariant and contravariant hom-functors $\hom(c, -)$ and $\hom(-, c)$ in category theory.

Combining Lemmas 2.3.10 and 2.11.2, we obtain a more general form:

Theorem 2.11.3. *For $f, g : A \to B$, with $p : a =_A a'$ and $q : f(a) =_B g(a)$, we have*

$$\mathsf{transport}^{x \mapsto f(x) =_B g(x)}(p, q) =_{f(a') =_B g(a')} (\mathsf{ap}_f p)^{-1} \cdot q \cdot \mathsf{ap}_g p.$$

Because $\mathsf{ap}_{(x \mapsto x)}$ is the identity function and $\mathsf{ap}_{(x \mapsto c)}$ (where c is a constant) is refl_c, Lemma 2.11.2 is a special case. A yet more general version is when B can be a family of types indexed on A:

Theorem 2.11.4. *Let $B : A \to \mathcal{U}$ and $f, g : \prod_{(x:A)} B(x)$, with $p : a =_A a'$ and $q : f(a) =_{B(a)} g(a)$. Then we have*

$$\mathsf{transport}^{x \mapsto f(x) =_{B(x)} g(x)}(p, q) = (\mathsf{ap}_f p)^{-1} \cdot \mathsf{apd}_{(\mathsf{transport}^A p)}(q) \cdot \mathsf{ap}_g p$$

Finally, as in §2.9, for families of identity types there is another equivalent characterization of dependent paths.

Theorem 2.11.5. *For $p : a =_A a'$ with $q : a = a$ and $r : a' = a'$, we have*

$$\left(\mathsf{transport}^{x \mapsto (x = x)}(p, q) = r\right) \simeq \left(q \cdot p = p \cdot r\right)$$

Proof. Path induction on p, followed by the fact that composing with the unit equalities $q \cdot 1 = q$ and $r = 1 \cdot r$ is an equivalence. □

There are more general equivalences involving the application of functions, akin to Theorems 2.11.3 and 2.11.4.

2.12 Coproducts

So far, most of the type formers we have considered have been what are called *negative*. Intuitively, this means that their elements are determined by their behavior under the elimination rules: a (dependent) pair is determined by its projections, and a (dependent) function is determined by its values. The identity types of negative types can almost always be characterized straightforwardly, along with all of their higher structure, as we have done in §§2.6–2.9. The universe is not exactly a negative type, but its identity types behave similarly: we have a straightforward characterization (univalence) and a description of the higher structure. Identity types themselves, of course, are a special case.

We now consider our first example of a *positive* type former. A positive type is one which is "presented" by certain constructors, with the universal property of a presentation being expressed by its elimination rule. (Categorically speaking, a positive type has a "mapping out" universal property, while a negative type has a "mapping in" universal property.) Because computing with presentations is, in general, an uncomputable problem, for positive types we cannot

always expect a straightforward characterization of the identity type. However, in many particular cases, a characterization or partial characterization does exist, and can be obtained by the general method that we introduce with this example.

(Technically, our chosen presentation of cartesian products and Σ-types is also positive. However, because these types also admit a negative presentation which differs only slightly, their identity types have a direct characterization that does not require the method to be described here.)

Consider the coproduct type $A + B$, which is "presented" by the injections $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$. Intuitively, we expect that $A + B$ contains exact copies of A and B disjointly, so that we should have

$$(\mathsf{inl}(a_1) = \mathsf{inl}(a_2)) \simeq (a_1 = a_2) \tag{2.12.1}$$
$$(\mathsf{inr}(b_1) = \mathsf{inr}(b_2)) \simeq (b_1 = b_2) \tag{2.12.2}$$
$$(\mathsf{inl}(a) = \mathsf{inr}(b)) \simeq \mathbf{0} \tag{2.12.3}$$

We prove this as follows. Fix an element $a_0 : A$; we will characterize the type family

$$(x \mapsto (\mathsf{inl}(a_0) = x)) : A + B \to \mathcal{U}. \tag{2.12.4}$$

A similar argument would characterize the analogous family $x \mapsto (x = \mathsf{inr}(b_0))$ for any $b_0 : B$. Together, these characterizations imply (2.12.1)–(2.12.3).

In order to characterize (2.12.4), we will define a type family $\mathsf{code} : A + B \to \mathcal{U}$ and show that $\prod_{(x:A+B)}((\mathsf{inl}(a) = x) \simeq \mathsf{code}(x))$. Since we want to conclude (2.12.1) from this, we should have $\mathsf{code}(\mathsf{inl}(a)) = (a_0 = a)$, and since we also want to conclude (2.12.3), we should have $\mathsf{code}(\mathsf{inr}(b)) = \mathbf{0}$. The essential insight is that we can use the elimination property of $A + B$ to *define* $\mathsf{code} : A + B \to \mathcal{U}$ by these two equations:

$$\mathsf{code}(\mathsf{inl}(a)) :\equiv (a_0 = a)$$
$$\mathsf{code}(\mathsf{inr}(b)) :\equiv \mathbf{0}$$

This is a very simple example of a proof technique that is used quite a bit in formalizing homotopy theory in homotopy type theory; see e.g. §8.1. We can now show:

Theorem 2.12.5. *For all $x : A + B$ we have $(\mathsf{inl}(a) = x) \simeq \mathsf{code}(x)$.*

Proof. The key to the following proof is that we do it for all points x together, enabling us to use the elimination principle for the coproduct. We first define a function

$$\mathsf{encode} : \prod_{(x:A+B)} \prod_{(p:\mathsf{inl}(a_0)=x)} \mathsf{code}(x)$$

by transporting reflexivity along p:

$$\mathsf{encode}(x, p) :\equiv \mathsf{transport}^{\mathsf{code}}(p, \mathsf{refl}_{a_0}).$$

Note that $\mathsf{refl}_{a_0} : \mathsf{code}(\mathsf{inl}(a_0))$, since $\mathsf{code}(\mathsf{inl}(a_0)) \equiv (a_0 = a_0)$ by definition of code. Next, we define a function

$$\mathsf{decode} : \prod_{(x:A+B)} \prod_{(c:\mathsf{code}(x))} (\mathsf{inl}(a_0) = x)$$

To define $\mathsf{decode}(x, c)$, we may first use the elimination principle of $A + B$ to divide into cases based on whether x is of the form $\mathsf{inl}(a)$ or the form $\mathsf{inr}(b)$.

In the first case, where $x \equiv \mathsf{inl}(a)$, then $\mathsf{code}(x) \equiv (a_0 = a)$, so that c is an identification between a_0 and a. Thus, $\mathsf{ap}_{\mathsf{inl}}(c) : (\mathsf{inl}(a_0) = \mathsf{inl}(a))$ so we can define this to be $\mathsf{decode}(\mathsf{inl}(a), c)$.

In the second case, where $x \equiv \mathsf{inr}(b)$, then $\mathsf{code}(x) \equiv \mathbf{0}$, so that c inhabits the empty type. Thus, the elimination rule of $\mathbf{0}$ yields a value for $\mathsf{decode}(\mathsf{inr}(b), c)$.

This completes the definition of decode; we now show that $\mathsf{encode}(x, -)$ and $\mathsf{decode}(x, -)$ are quasi-inverses for all x. On the one hand, suppose given $x : A + B$ and $p : \mathsf{inl}(a_0) = x$; we want to show $\mathsf{decode}(x, \mathsf{encode}(x, p)) = p$. But now by (based) path induction, it suffices to consider $x \equiv \mathsf{inl}(a_0)$ and $p \equiv \mathsf{refl}_{\mathsf{inl}(a_0)}$:

$$\begin{aligned}
\mathsf{decode}(x, \mathsf{encode}(x, p)) &\equiv \mathsf{decode}(\mathsf{inl}(a_0), \mathsf{encode}(\mathsf{inl}(a_0), \mathsf{refl}_{\mathsf{inl}(a_0)})) \\
&\equiv \mathsf{decode}(\mathsf{inl}(a_0), \mathsf{transport}^{\mathsf{code}}(\mathsf{refl}_{\mathsf{inl}(a_0)}, \mathsf{refl}_{a_0})) \\
&\equiv \mathsf{decode}(\mathsf{inl}(a_0), \mathsf{refl}_{a_0}) \\
&\equiv \mathsf{inl}(\mathsf{refl}_{a_0}) \\
&\equiv \mathsf{refl}_{\mathsf{inl}(a_0)} \\
&\equiv p.
\end{aligned}$$

On the other hand, let $x : A + B$ and $c : \mathsf{code}(x)$; we want to show $\mathsf{encode}(x, \mathsf{decode}(x, c)) = c$. We may again divide into cases based on x. If $x \equiv \mathsf{inl}(a)$, then $c : a_0 = a$ and $\mathsf{decode}(x, c) \equiv \mathsf{ap}_{\mathsf{inl}}(c)$, so that

$$\begin{aligned}
\mathsf{encode}(x, \mathsf{decode}(x, c)) &\equiv \mathsf{transport}^{\mathsf{code}}(\mathsf{ap}_{\mathsf{inl}}(c), \mathsf{refl}_{a_0}) \\
&= \mathsf{transport}^{a \mapsto (a_0 = a)}(c, \mathsf{refl}_{a_0}) && \text{(by Lemma 2.3.10)} \\
&= \mathsf{refl}_{a_0} \cdot c && \text{(by Lemma 2.11.2)} \\
&= c. && \qquad \square
\end{aligned}$$

Of course, there is a corresponding theorem if we fix $b_0 : B$ instead of $a_0 : A$.

For any $a : A$ and $b : B$ there are functions

$$\mathsf{encode}(a, -) : (\mathsf{inl}(a_0) = \mathsf{inl}(a)) \to (a_0 = a)$$

and

$$\mathsf{encode}(b, -) : (\mathsf{inl}(a_0) = \mathsf{inr}(b)) \to \mathbf{0}.$$

The second one states "$\mathsf{inl}(a_0)$ is not equal to $\mathsf{inr}(b)$", i.e. the images of inl and inr are disjoint. The traditional reading of the first one, where identity types are viewed as propositions, the first one is just injectivity of inl. The homotopical version gives more information: the types $\mathsf{inl}(a_0) = \mathsf{inl}(a)$ and $a_0 = a$ are actually equivalent, as are $\mathsf{inr}(b_0) = \mathsf{inr}(b)$ and $b_0 = b$.

Remark 2.12.6. In particular, since the two-element type $\mathbf{2}$ is equivalent to $\mathbf{1} + \mathbf{1}$, we have $1_{\mathbf{2}} \neq 0_{\mathbf{2}}$.

This proof illustrates a general method for describing path spaces, which we will use often. To characterize a path space, the first step is to define a comparison fibration code that provides a more explicit description of the paths. There are several different methods for proving that such

a comparison fibration is equivalent to the paths (we show a few different proofs of the same result in §8.1). The one we have used here is called the **encode-decode method**: the key idea is to define decode generally for all instances of the fibration $(\prod_{(x:A+B)} \prod_{(c:\mathsf{code}(x))} (\mathsf{inl}(a_0) = x))$, so that path induction can be used to analyze the composite $\mathsf{decode}(x, \mathsf{encode}(x, p))$.

As usual, we can also characterize the action of transport in coproduct types. Given a type X, a path $p : x_1 =_X x_2$, and type families $A, B : X \to \mathcal{U}$, we have

$$\begin{aligned} \mathsf{transport}^{A+B}(p, \mathsf{inl}(a)) &= \mathsf{inl}(\mathsf{transport}^{A}(p, a)), \\ \mathsf{transport}^{A+B}(p, \mathsf{inr}(b)) &= \mathsf{inr}(\mathsf{transport}^{B}(p, b)), \end{aligned}$$

where as usual, $A + B$ in the supscript denotes abusively the type family $x \mapsto A(x) + B(x)$. The proof is an easy path induction.

2.13 Natural numbers

We use the encode-decode method to characterize the path space of the the natural numbers, which are also a positive type. In this case the codes for identities are a type family

$$\mathsf{code} : \mathbb{N} \to \mathbb{N} \to \mathcal{U},$$

defined by double recursion over $\mathbb{N}$ as follows:

$$\begin{aligned} \mathsf{code}(0, 0) &\equiv \mathbf{1} \\ \mathsf{code}(\mathsf{succ}(m), 0) &\equiv \mathbf{0} \\ \mathsf{code}(0, \mathsf{succ}(n)) &\equiv \mathbf{0} \\ \mathsf{code}(\mathsf{succ}(m), \mathsf{succ}(n)) &\equiv \mathsf{code}(m, n). \end{aligned}$$

We also define by recursion a dependent function $r : \prod_{(n:\mathbb{N})} \mathsf{code}(n, n)$, with

$$\begin{aligned} r(0) &\equiv \star \\ r(\mathsf{succ}(n)) &\equiv r(n). \end{aligned}$$

Theorem 2.13.1. *For all $m, n : \mathbb{N}$ we have $(m = n) \simeq \mathsf{code}(m, n)$.*

Proof. We define

$$\mathsf{encode} : \prod_{m,n:\mathbb{N}} (m = n) \to \mathsf{code}(m, n)$$

by transporting, $\mathsf{encode}(m, n, p) :\equiv \mathsf{transport}^{\mathsf{code}(m,-)}(p, r(m))$. And we define

$$\mathsf{decode} : \prod_{m,n:\mathbb{N}} \mathsf{code}(m, n) \to (m = n)$$

by double induction on m, n. When m and n are both 0, we need a function $\mathbf{1} \to (0 = 0)$, which we define to send everything to refl_0. When m is a successor and n is 0 or vice versa, the domain

$\mathsf{code}(m,n)$ is $\mathbf{0}$, so the eliminator for $\mathbf{0}$ suffices. And when both are successors, we can define $\mathsf{decode}(\mathsf{succ}(m),\mathsf{succ}(n))$ to be the composite

$$\mathsf{code}(\mathsf{succ}(m),\mathsf{succ}(n)) \equiv \mathsf{code}(m,n) \xrightarrow{\mathsf{decode}(m,n)} (m = n) \xrightarrow{\mathsf{ap}_{\mathsf{succ}}} (\mathsf{succ}(m) = \mathsf{succ}(n)).$$

Next we show that $\mathsf{encode}(m,n)$ and $\mathsf{decode}(m,n)$ are quasi-inverses for all m,n.

On one hand, if we start with $p : m = n$, then by induction on p it suffices to show

$$\mathsf{decode}(n,n,\mathsf{encode}(n,n,\mathsf{refl}_n)) = \mathsf{refl}_n.$$

But $\mathsf{encode}(n,n,\mathsf{refl}_n) \equiv r(n)$, so it suffices to show that $\mathsf{decode}(n,n,r(n)) = \mathsf{refl}_n$. We can prove this by induction on n. If $n \equiv 0$, then $\mathsf{decode}(0,0,r(0)) = \mathsf{refl}_0$ by definition of decode. And in the case of a successor, by the inductive hypothesis we have $\mathsf{decode}(n,n,r(n)) = \mathsf{refl}_n$, so it suffices to observe that $\mathsf{ap}_{\mathsf{succ}}(\mathsf{refl}_n) \equiv \mathsf{refl}_{\mathsf{succ}(n)}$.

On the other hand, if we start with $c : \mathsf{code}(m,n)$, then we proceed by double induction on m and n. If both are 0, then $\mathsf{decode}(0,0,c) \equiv \mathsf{refl}_0$, while $\mathsf{encode}(0,0,\mathsf{refl}_0) \equiv r(0) \equiv \star$. Thus, it suffices to recall from §2.8 that every inhabitant of $\mathbf{1}$ is equal to $\star$. If m is 0 but n is a successor, or vice versa, then $c : \mathbf{0}$, so we are done. And in the case of two successors, we have

$$\begin{aligned}
&\mathsf{encode}(\mathsf{succ}(m),\mathsf{succ}(n),\mathsf{decode}(\mathsf{succ}(m),\mathsf{succ}(n),c)) \\
&\qquad = \mathsf{encode}(\mathsf{succ}(m),\mathsf{succ}(n),\mathsf{ap}_{\mathsf{succ}}(\mathsf{decode}(m,n,c))) \\
&\qquad = \mathsf{transport}^{\mathsf{code}(\mathsf{succ}(m),-)}(\mathsf{ap}_{\mathsf{succ}}(\mathsf{decode}(m,n,c)), r(\mathsf{succ}(m))) \\
&\qquad = \mathsf{transport}^{\mathsf{code}(\mathsf{succ}(m),\mathsf{succ}(-))}(\mathsf{decode}(m,n,c), r(\mathsf{succ}(m))) \\
&\qquad = \mathsf{transport}^{\mathsf{code}(m,-)}(\mathsf{decode}(m,n,c), r(m)) \\
&\qquad = \mathsf{encode}(m,n,\mathsf{decode}(m,n,c)) \\
&\qquad = c
\end{aligned}$$

using the inductive hypothesis. □

In particular, we have

$$\mathsf{encode}(\mathsf{succ}(m),0) : (\mathsf{succ}(m) = 0) \to \mathbf{0} \tag{2.13.2}$$

which shows that "0 is not the successor of any natural number". We also have the composite

$$(\mathsf{succ}(m) = \mathsf{succ}(n)) \xrightarrow{\mathsf{encode}} \mathsf{code}(\mathsf{succ}(m),\mathsf{succ}(n)) \equiv \mathsf{code}(m,n) \xrightarrow{\mathsf{decode}} (m = n) \tag{2.13.3}$$

which shows that the function succ is injective.

We will study more general positive types in Chapters 5 and 6. In Chapter 8, we will see that the same technique used here to characterize the identity types of coproducts and $\mathbb{N}$ can also be used to calculate homotopy groups of spheres.

2.14 Example: equality of structures

We now consider one example to illustrate the interaction between the groupoid structure on a type and the type formers. In the introduction we remarked that one of the advantages of univalence is that two isomorphic things are interchangeable, in the sense that every property or construction involving one also applies to the other. Common "abuses of notation" become formally true. Univalence itself says that equivalent types are equal, and therefore interchangable, which includes e.g. the common practice of identifying bijective sets. Moreover, when we define other mathematical objects as sets, or even general types, equipped with structure or properties, we can derive the correct notion of equality for them from univalence. We illustrate this point with a significant example in Chapter 9, where we define the basic notions of category theory in such a way that equality of categories is equivalence, equality of functors is natural isomorphism, etc. See in particular §9.8. In this section, we describe a very simple example, coming from algebra.

For simplicity, we use *semigroups* as our example, where a semigroup is a type equipped with an associative "multiplication" operation. The same ideas apply to other algebraic structures, such as monoids, groups, and rings.

Definition 2.14.1. Given a type A, the type $\mathsf{SemigroupStr}(A)$ of **semigroup structures** with carrier A is defined by

$$\mathsf{SemigroupStr}(A) :\equiv \sum_{(m:A\to A\to A)} \prod_{(x,y,z:A)} m(x,m(y,z)) = m(m(x,y),z).$$

A **semigroup** is a type together with such a structure:

$$\mathsf{Semigroup} :\equiv \sum_{A:\mathcal{U}} \mathsf{SemigroupStr}(A)$$

In the next two sections, we describe two ways in which univalence makes it easier to work with such semigroups.

2.14.1 Lifting equivalences

When working loosely, one might say that a bijection between sets A and B "obviously" induces an isomorphism between semigroup structures on A and semigroup structures on B. With univalence, this is indeed obvious, because given an equivalence between types A and B, we can automatically derive a semigroup structure on B from one on A, and moreover show that this derivation is an equivalence of semigroup structures. The reason is that SemigroupStr(-) is a family of types, and therefore has an action on paths between types given by transport:

$$\mathsf{transport}^{X\mapsto \mathsf{SemigroupStr}(X)}(\mathsf{ua}(e)) : \mathsf{SemigroupStr}(A) \to \mathsf{SemigroupStr}(B)$$

Moreover, this map is an equivalence, because $\mathsf{transport}^C\alpha$ is always an equivalence with inverse $\mathsf{transport}^C(\alpha^{-1})$ (by Lemma 2.3.9 and Lemma 2.1.4).

While the univalence axiom ensures that this map exists, we need to use facts about transport defined in the preceeding sections to calculate what it actually does. Let (m, a) be a semigroup structure on A, and we investigate the induced semigroup structure on B given by

$$\mathsf{transport}^{X \mapsto \mathsf{SemigroupStr}(X)}(\mathsf{ua}(e), (m, a))$$

First, because SemigroupStr(X) is defined to be a Σ-type, by Theorem 2.7.4,

$$\begin{aligned}&\mathsf{transport}^{X \mapsto \mathsf{SemigroupStr}(X)}(\mathsf{ua}(e), (m, a)) \\ =&(\mathsf{transport}^{X \mapsto (X \to X \to X)}(\mathsf{ua}(e), m), \mathsf{transport}^{(X,m) \mapsto \mathsf{Assoc}(X,m)}(\mathsf{pair}^{=}(\mathsf{ua}(e), \mathsf{refl}))a)\end{aligned} \tag{2.14.2}$$

where $\mathsf{Assoc}(X, m)$ is the type $\prod_{(x,y,z:X)} m(x, m(y, z)) = m(m(x, y), z)$. That is, the induced semigroup structure consists of an induced multiplication operation on B

$$\begin{aligned}&m' : B \to B \to B \\ &m'(b_1, b_2) :\equiv \mathsf{transport}^{X \mapsto (X \to X \to X)}(\mathsf{ua}(e), m)(b_1, b_2)\end{aligned}$$

together with an induced proof that m' is associative. By function extensionality, it suffices to investigate the behavior of m' when applied to arguments $b_1, b_2 : B$. By applying (2.9.4) twice, we have that $m'(b_1, b_2)$ is equal to

$$\mathsf{transport}^{X \mapsto X}(\mathsf{ua}(e), m(\mathsf{transport}^{X \mapsto X}(\mathsf{ua}(e)^{-1}, b_1), \mathsf{transport}^{X \mapsto X}(\mathsf{ua}(e)^{-1}, b_2)))$$

Then, by Axiom 2.10.3, this is equal to

$$e(m(e^{-1}(b_1), e^{-1}(b_2)))$$

Thus, given two elements of B, the induced multiplication m' sends them to A using the equivalence e, multiplies them in A, and then brings the result back to B by e, just as one would expect.

Moreover, though we do not show the proof, one can calculate that the induced proof that m' is associative (the second component of the pair in (2.14.2)) is equal to a function sending $b_1, b_2, b_3 : B$ to a path given by the following steps:

$$\begin{aligned}m'(b_1, m'(b_2, b_3)) &= e(m(e^{-1}(m'(b_1, b_2)), e^{-1}(b_3))) \\ &= e(m(e^{-1}(e(m(e^{-1}(b_1), e^{-1}(b_2)))), e^{-1}(b_3))) \\ &= e(m(m(e^{-1}(b_1), e^{-1}(b_2)), e^{-1}(b_3))) \\ &= e(m(e^{-1}(b_1), m(e^{-1}(b_2), e^{-1}(b_3)))) \\ &= e(m(e^{-1}(b_1), e^{-1}(e(m(e^{-1}(b_2), e^{-1}(b_3)))))) \\ &= e(m(e^{-1}(b_1), e^{-1}(m'(b_2, b_3)))) \\ &= m'(b_1, m'(b_2, b_3)).\end{aligned} \tag{2.14.3}$$

These steps use the proof a that m is associative and the inverse laws for e. From an algebra perspective, it may seem strange to investigate the identity of a proof that an operation is associative, but this makes sense if we think of A and B as general spaces, with non-trivial homotopies between paths. In Chapter 3, we will introduce the notion of a *set*, which is a type with only trivial homotopies, and if we consider semigroup structures on sets, then any two such associativity proofs are automatically equal.

2.14.2 Equality of semigroups

Using the equations for path spaces discussed in the previous sections, we can investigate when two semigroups are equal. Given semigroups (A, m, a) and (B, m', a'), by Theorem 2.7.2, the type of paths $(A, m, a) =_{\mathsf{Semigroup}} (B, m', a')$ is equal to the type of pairs

$$p_1 : A =_{\mathcal{U}} B \qquad \text{and} \qquad p_2 : \mathsf{transport}^{X \mapsto \mathsf{SemigroupStr}(X)}(p_1, (m, a))(m', a').$$

By univalence, p_1 is $\mathsf{ua}(e)$ for some equivalence e. By Theorem 2.7.2, function extensionality, and the above analysis of $\mathsf{transport}^{X \mapsto \mathsf{SemigroupStr}(X)}$, p_2 is equivalent to a pair of proofs, the first of which shows that

$$\prod_{y_1, y_2 : B} e(m(e^{-1}(y_1), e^{-1}(y_2))) = m'(y_1, y_2) \tag{2.14.4}$$

and the second of which shows that a' is equal to the induced associativity proof constructed from a in (2.14.3). But by cancellation of inverses (2.14.4) is equivalent to

$$\prod_{x_1, x_2 : A} e(m(x_1, x_2)) = m'(e(x_2), e(x_2))$$

This says that e commutes with the binary operation, in the sense that it takes multiplcation in A (m) to multiplication in B (m'). A similar rearrangement is possible for the equation relating a and a'. Thus, an equality of semigroups consists exactly of an equivalence on the carrier types that commutes with the semigroup structure.

For general types, the proof of associativity is thought of as part of the structure of a semigroup. However, if we restrict to set-like types (again, see Chapter 3), associativity becomes a mere property, so the equation relating a and a' is trivially true. Moreover, in this case, an equivalence between sets is exactly a bijection. Thus, we have arrived at a standard definition of a *semigroup isomorphism*: a bijection on the carrier sets that preserves the multiplication operation. It is also possible to use the category-theoretic definition of isomorphism, by defining a *semigroup homomorphism* to be a map that preserves the multiplcation, and arrive at the conclusion that equality of semigroups is the same as two mutually inverse homomorphisms; but we will not show the details here.

The conclusion is that, thanks to univalence, semigroups are equal precisely when they are isomorphic as algebraic structures. The conclusion applies more generally: in homotopy type theory, all constructions of mathematical structures automatically respect isomorphisms, without any tedious proofs or abuse of notation.

2.15 Universal properties

By combining the path computation rules described in the preceding sections, we can show that various type forming operations satisfy the expected universal properties. For instance, given types X, A, B, we have a function

$$(X \to A \times B) \;\to\; (X \to A) \times (X \to B) \tag{2.15.1}$$

defined by $f \mapsto (\mathsf{pr}_1 \circ f, \mathsf{pr}_2 \circ f)$.

Theorem 2.15.2. (2.15.1) *is an equivalence.*

Proof. We define a quasi-inverse to send (g, h) to the function $x \mapsto (g(x), h(x))$. (Technically, we have used the induction principle for the cartesian product $(X \to A) \times (X \to B)$, to reduce to the case of a pair.)

Now given $f : X \to A \times B$, the round-trip composite yields the function

$$x \mapsto (\mathsf{pr}_1(f(x)), \mathsf{pr}_2(f(x))). \tag{2.15.3}$$

By Theorem 2.6.2, for any $x : X$ we have $(\mathsf{pr}_1(f(x)), \mathsf{pr}_2(f(x))) = f(x)$. Thus, by function extensionality, the function (2.15.3) is equal to f.

On the other hand, given (g, h), the round-trip composite yields the pair $(x \mapsto g(x), x \mapsto h(x))$. By function extensionaility, the two components of this are equal to g and h respectively, so by Theorem 2.6.2, the pair is equal to (g, h). □

In fact, we also have a dependently typed version of this universal property. Suppose given a type X and type families $A, B : X \to \mathcal{U}$. Then we have a function

$$\Big(\prod_{x:X} (A(x) \times B(x))\Big) \to \Big(\prod_{x:X} A(x)\Big) \times \Big(\prod_{x:X} B(x)\Big) \tag{2.15.4}$$

defined as before by $f \mapsto (\mathsf{pr}_1 \circ f, \mathsf{pr}_2 \circ f)$.

Theorem 2.15.5. (2.15.4) *is an equivalence.*

Proof. Left to the reader. □

Just as Σ-types are a generalization of cartesian products, they satisfy a generalized version of this universal property. Jumping right to the dependently typed version, suppose we have a type X and type families $A : X \to \mathcal{U}$ and $P : \prod_{(x:X)} A(x) \to \mathcal{U}$. Then we have a function

$$\Big(\prod_{(x:X)} \sum_{(a:A(x))} P(x, a)\Big) \to \Big(\sum_{(g:\prod_{(x:X)} A(x))} \prod_{(x:X)} P(x, g(x))\Big) \tag{2.15.6}$$

Note that if we have $P(x, a) :\equiv B(x)$ for some $B : X \to \mathcal{U}$, then (2.15.6) reduces to (2.15.4).

Theorem 2.15.7. (2.15.6) *is an equivalence.*

Proof. As before, we define a quasi-inverse to send (g, h) to the function $x \mapsto (g(x), h(x))$. Now given $f : \prod_{(x:X)} \sum_{(a:A(x))} P(x, a)$, the round-trip composite yields the function

$$x \mapsto (\mathsf{pr}_1(f(x)), \mathsf{pr}_2(f(x))). \tag{2.15.8}$$

Now for any $x : X$, by Corollary 2.7.3 (η-equivalence for Σ-types) we have

$$(\mathsf{pr}_1(f(x)), \mathsf{pr}_2(f(x))) = f(x).$$

Thus, by function extensionality, (2.15.8) is equal to f.

On the other hand, given (g, h), the round-trip composite yields the pair $(x \mapsto g(x), x \mapsto h(x))$. But $x \mapsto g(x)$ and $x \mapsto h(x)$ are judgmentally equal to g and h, respectively, and hence this pair of functions is also equal to (g, h). □

This is noteworthy because the propositions-as-types interpretation of (2.15.6) is "the axiom of choice". If we read Σ as "there exists" and Π (sometimes) as "for all", we can pronounce:

- $\prod_{(x:X)} \sum_{(a:A(x))} P(x,a)$ as "for all $x : X$ there exists an $a : A(x)$ such that $P(x,a)$", and
- $\sum_{(g:\prod_{(x:X)} A(x))} \prod_{(x:X)} P(x,g(x))$ as "there exists a choice function $g : \prod_{(x:X)} A(x)$ such that for all $x : X$ we have $P(x,g(x))$".

Thus, Theorem 2.15.7 says that not only is the axiom of choice "true", it hypotheses are equivalent to its conclusion. (On the other hand, the classical mathematician may find that (2.15.6) does not carry the usual meaning of the axiom of choice, since we have already specified the values of g, and there are no choices left to be made. We will return to this point in §3.8.)

The above universal property for pair types is for "mapping in", which is familiar from the category-theoretic notion of products. However, pair types also have a universal property for "mapping out", which may look less familiar. In the case of cartesian products, the non-dependent version simply expresses the cartesian closedness adjunction:

$$\big((A \times B) \to C\big) \;\simeq\; \big(A \to (B \to C)\big).$$

The dependent version of this is formulated for a type family $C : A \times B \to \mathcal{U}$:

$$\Big(\prod_{w:A\times B} C(w)\Big) \;\simeq\; \Big(\prod_{(x:A)} \prod_{(y:B)} C(x,y)\Big).$$

Here the left-to-right function is simply the induction principle for $A \times B$, while the right-to-left is evaluation at a pair. We leave it to the reader to prove that these are quasi-inverses. There is also a version for Σ-types:

$$\Big(\prod_{w:\sum_{(x:A)} B(x)} C(w)\Big) \;\simeq\; \Big(\prod_{(x:A)} \prod_{(y:B(x))} C(x,y)\Big) \tag{2.15.9}$$

Again, the left-to-right function is the induction principle.

Some other induction principles are also part of universal properties of this sort. For instance, path induction is the right-to-left direction of an equivalence as follows:

$$\Big(\prod_{(x:A)} \prod_{(p:a=x)} B(x,p)\Big) \;\simeq\; B(a, \mathsf{refl}_a) \tag{2.15.10}$$

for any $a : A$ and type family $B : \prod_{(x:A)}(a = x) \to \mathcal{U}$. However, inductive types with recursion, such as the natural numbers, have more complicated universal properties; see Chapter 5.

Notes

The definition of identity types and the elimination rule J are due to Martin-Löf [ML98]. As mentioned in the notes to Chapter 1, our identity types are those that belong to *intensional* type theory, by contrast with those of *extensional* type theory which have an additional "reflection rule" saying that if $p : x = y$, then in fact $x \equiv y$. This reflection rule implies that all the higher

groupoid structure collapses (see Exercise 2.10), so for nontrivial homotopy we must use the intensional version. One may argue, however, that homotopy type theory is more "extensional" than traditional extensional type theory, because of the function extensionality and univalence rules.

The proofs of symmetry (inversion) and transitivity (concatenation) for equalities are well-known in type theory. The fact that these make each type into a 1-groupoid (up to homotopy) is also folklore, and was exploited in [HS98] to give the first "homotopy" style semantics for type theory.

The actual homotopical interpretation, with identity types as path spaces, and dependent types as fibrations, is due to [AW09], who used the formalism of Quillen model categories. An interpretation in (strict) ∞-groupoids was also given in the thesis [War08]. For a construction of *all* the higher operations and coherences of an ∞-groupoid in type theory, see [Lum10] and [vdBG11].

Operations such as $\mathsf{transport}^P(p, -)$ and ap_f, and one good notion of equivalence, were first studied extensively in type theory by Voevodsky, using the proof assistant Coq. Subsequent researchers have found many other equivalent definitions of equivalence, which are compared in Chapter 4.

The "computational" interpretation of identity types, transport, and so on described in §2.5 has been emphasized by [LH12]. They also described a "1-truncated" type theory (see Chapter 7) in which these rules really are computation steps (that is, definitional equalities which a computer can "evaluate"). The possibility of extending this to the full untruncated theory is a subject of current research.

The naive form of function extensionality which says that "if two functions are pointwise equal, then they are equal" is a common axiom in type theory, going all the way back to [WR27]. Some stronger forms of function extensionality were considered in [Gar09]. The version we have used, which identifies the identity types of function types up to equivalence, was first studied by Voevodsky, who also proved that it is implied by the naive version (and by univalence; see §4.9).

The univalence axiom is also due to Voevodsky. It was originally motivated by semantic considerations; see [KLV12].

In the type theory we are using in this book, function extensionality and univalence have to be assumed as axioms, i.e. elements asserted to belong to some type but not constructed according to the rules for that type. While serviceable, this has a few drawbacks. For instance, type theory is formally better-behaved if we can base it entirely on rules rather than asserting axioms. It is also sometimes inconvenient that the theorems of §§2.6–2.13 are only propositional equalities (paths) or equivalences, since then we must explicitly mention whenever we pass back and forth across them.

One direction of current research in homotopy type theory is to describe a type system in which these rules are *judgmental* equalities, solving both of these problems at once. So far this has only been done in some simple cases, although preliminary results such as [LH12] are promising. There are also other potential ways to introduce univalence and function extensionality into a type theory, such as having a sufficiently powerful notion of "higher quotients" or "higher inductive-recursive types".

The simple conclusions in §§2.12–2.13 such as "coproduct injections are injective and disjoint" are well-known in type theory, and the construction of the function encode is the usual way to prove them. The more refined approach we have described, which characterizes the entire identity type of a positive type (up to equivalence), is a more recent development; see e.g. [LS13a].

The type-theoretic axiom of choice (2.15.6) was noticed in William Howard's original paper [How80] on the propositions-as-types correspondence, and was studied further by Martin-Löf with the introduction of his dependent type theory. It is mentioned as a "distributivity law" in Bourbaki's set theory [Bou68].

Exercises

Exercise 2.1. Show that the three obvious proofs of Lemma 2.1.2 are pairwise equal.

Exercise 2.2. Show that the three equalities of proofs constructed in the previous exercise form a commutative triangle. In other words, if the three definitions of concatenation are denoted by $(p \cdot_1 q)$, $(p \cdot_2 q)$, and $(p \cdot_3 q)$, then the concatenated equality

$$(p \cdot_1 q) = (p \cdot_2 q) = (p \cdot_3 q)$$

is equal to the equality $(p \cdot_1 q) = (p \cdot_3 q)$.

Exercise 2.3. Give a fourth, different, proof of Lemma 2.1.2, and prove that it is equal to the others.

Exercise 2.4. Prove that the functions (2.3.6) and (2.3.7) are inverse equivalences.

Exercise 2.5. Prove that if $p : x = y$, then the function $(p \cdot -) : (y = z) \to (x = z)$ is an equivalence.

Exercise 2.6. State and prove a generalization of Theorem 2.6.5 from cartesian products to Σ-types.

Exercise 2.7. State and prove an analogue of Theorem 2.6.5 for coproducts.

Exercise 2.8. Prove that coproducts have the expected universal property:

$$(A + B \to X) \simeq (A \to X) \times (B \to X)$$

Can you generalize this to an equivalence involving dependent functions?

Exercise 2.9. Show that $(\mathbf{2} \simeq \mathbf{2}) \simeq \mathbf{2}$.

Exercise 2.10. Suppose we add to type theory the *equality reflection rule* which says that if there is an element $p : x = y$, then in fact $x \equiv y$. Prove that for any $p : x = x$ we have $p \equiv \mathsf{refl}_x$, and hence every type is a set in the sense of §3.1.

Chapter 3

Sets and logic

Type theory, formal or informal, is a collection of rules for manipulating types and their elements. But when writing mathematics informally in natural language, we generally use familiar words, particularly logical connectives such as "and" and "or", and logical quantifiers such as "for all" and "there exists". In contrast to set theory, type theory offers us more than one way to regard these English phrases as operations on types. This potential ambiguity needs to be resolved, by setting out local or global conventions, by introducing new annotations to informal mathematics, or both. This requires some getting used to, but is offset by the fact that because type theory permits this finer analysis of logic, we can represent mathematics more faithfully, with fewer "abuses of language" than in set-theoretic foundations. In this chapter we will explain the issues involved, and justify the choices we have made.

3.1 Sets and n-types

In order to explain the connection between the logic of type theory and the logic of set theory, it is helpful to have a notion of *set* in type theory. While types in general behave like spaces or higher groupoids, there is a subclass of them that behave more like the sets in a traditional set-theoretic system. Categorically, we may consider *discrete* groupoids, which are determined by a set of objects and only identity morphisms as higher morphisms; while topologically, we may consider spaces having the discrete topology. More generally, we may consider groupoids or spaces that are *equivalent* to ones of this sort; since everything we do in type theory is up to homotopy, we can't expect to tell the difference.

Intuitively, we would expect a type to "be a set" in this sense if it has no higher homotopical information: any two parallel paths are equal (up to homotopy), and similarly for parallel higher paths at all dimensions. Fortunately, because everything in homotopy type theory is automatically functorial/continuous, it turns out to be sufficient to ask this at the bottom level.

Definition 3.1.1. A type A is a **set** if for all $x, y : A$ and all $p, q : x = y$, we have $p = q$.

More precisely, the proposition $\mathsf{isSet}(A)$ is defined to be the type

$$\mathsf{isSet}(A) :\equiv \prod_{(x,y:A)} \prod_{(p,q:x=y)} (p = q).$$

As mentioned in §1.1, the sets in homotopy type theory are not like the sets in ZF set theory, in that there is no global "membership predicate" $\in$. They are more like the sets used in structural mathematics and in category theory, whose elements are "abstract points" to which we give structure with functions and relations. This is all we need in order to use them as a foundational system for most set-based mathematics; we will see some examples in Chapter 10.

Which types are sets? In Chapter 7 we will study a more general form of this question in depth, but for now we can observe some easy examples.

Example 3.1.2. The type $\mathbf{1}$ is a set. For by Theorem 2.8.1, for any $x, y : \mathbf{1}$ the type $(x = y)$ is equivalent to $\mathbf{1}$. Since any two elements of $\mathbf{1}$ are equal, this implies that any two elements of $x = y$ are equal.

Example 3.1.3. The type $\mathbf{0}$ is a set, for given any $x, y : \mathbf{0}$ we may deduce anything we like by contradiction.

Example 3.1.4. The type $\mathbb{N}$ of natural numbers is also a set. This follows from Theorem 2.13.1, since all equality types $x =_{\mathbb{N}} y$ are equivalent to either $\mathbf{1}$ or $\mathbf{0}$, and any two inhabitants of $\mathbf{1}$ or $\mathbf{0}$ are equal. We will see another proof of this fact in Chapter 7.

Most of the type forming operations we have considered so far also preserve sets.

Example 3.1.5. If A and B are sets, then so is $A \times B$. For given $x, y : A \times B$ and $p, q : x = y$, by Theorem 2.6.2 we have $p = \mathsf{pair}^{=}(\mathsf{ap}_{\mathsf{pr}_1}(p), \mathsf{ap}_{\mathsf{pr}_2}(p))$ and $q = \mathsf{pair}^{=}(\mathsf{ap}_{\mathsf{pr}_1}(q), \mathsf{ap}_{\mathsf{pr}_2}(q))$. But $\mathsf{ap}_{\mathsf{pr}_1}(p) = \mathsf{ap}_{\mathsf{pr}_1}(q)$ since A is a set, and $\mathsf{ap}_{\mathsf{pr}_2}(p) = \mathsf{ap}_{\mathsf{pr}_2}(q)$ since B is a set; hence $p = q$.

Similarly, if A is a set and $B : A \to \mathcal{U}$ is such that each $B(x)$ is a set, then $\sum_{(x:A)} B(x)$ is a set.

Example 3.1.6. If A is *any* type and $B : A \to \mathcal{U}$ is such that each $B(x)$ is a set, then the type $\prod_{(x:A)} B(x)$ is a set. For suppose $f, g : \prod_{(x:A)} B(x)$ and $p, q : f = g$. By function extensionality, we have $p = \mathsf{funext}(x \mapsto \mathsf{happly}(p, x))$ and likewise $q = \mathsf{funext}(x \mapsto \mathsf{happly}(q, x))$. But for any $x : A$, we have $\mathsf{happly}(p, x) : f(x) = g(x)$ and also $\mathsf{happly}(q, x) : f(x) = g(x)$, so since $B(x)$ is a set we have $\mathsf{happly}(p, x) = \mathsf{happly}(q, x)$. Now using function extensionality again, the dependent functions $(x \mapsto \mathsf{happly}(p, x))$ and $(x \mapsto \mathsf{happly}(q, x))$ are equal, and hence (applying $\mathsf{ap}_{\mathsf{funext}}$) so are p and q.

For more examples, see Exercises 3.2 and 3.3. For a more systematic investigation of the subsystem (category) of all sets in homotopy type theory, see Chapter 10.

Sets are just the first rung on a ladder of what are called *homotopy n-types*. The next rung consists of 1-*types*, which are analogous to 1-groupoids in category theory. The defining property of a set (which we may also call a 0-*type*) is that it has no non-trivial paths. Similarly, the defining property of a 1-type is that it has no non-trivial paths between paths:

Definition 3.1.7. A type A is a **1-type** if for all $x, y : A$ and $p, q : x = y$ and $r, s : p = q$, we have $r = s$.

Similary, we can define 2-types, 3-types, and so on. We will define the general notion of n-type inductively in Chapter 7, and study the relationships between them.

However, for now it is useful to have two facts in mind. First, the levels are upward-closed: if A is an n-type then A is an $(n+1)$-type. For example:

Lemma 3.1.8. *If A is a set (that is, $\mathsf{isSet}(A)$ is inhabited), then A is a 1-type.*

Proof. Suppose $f : \mathsf{isSet}(A)$; then for any $x, y : A$ and $p, q : x = y$ we have $f(x, y, p, q) : p = q$. Fix x, y, and p, and define $g : \prod_{(q:x=y)}(p = q)$ by $g(q) :\equiv f(x, y, p, q)$. Then for any $r : q = q'$, we have $\mathsf{apd}_g(r) : r_*(g(q)) = g(q')$. By Lemma 2.11.2, therefore, we have $g(q) \cdot r = g(q')$.

In particular, suppose given x, y, p, q and $r, s : p = q$, as in Definition 3.1.7, and define g as above. Then $g(p) \cdot r = g(q)$ and also $g(p) \cdot s = g(q)$, hence by cancellation $r = s$. □

Second, this stratification of types by level is not degenerate, in the sense that not all types are sets:

Example 3.1.9. The universe $\mathcal{U}$ is not a set. To prove this, it suffices to exhibit a type A and a path $p : A = A$ which is not equal to refl_A. Take $A = \mathbf{2}$, and let $f : A \to A$ be defined by $f(1_\mathbf{2}) :\equiv 0_\mathbf{2}$ and $f(0_\mathbf{2}) :\equiv 1_\mathbf{2}$. Then $f(f(x)) = x$ for all x (by an easy case analysis), so f is an equivalence. Hence, by univalence, f gives rise to a path $p : A = A$.

If p were equal to refl_A, then (again by univalence) f would equal the identity function of A. But this would imply that $1_\mathbf{2} = 0_\mathbf{2}$, contradicting Remark 2.12.6.

In Chapters 6 and 8 we will show that for any n, there are types which are not n-types.

Note that A is a 1-type exactly when for any $x, y : A$, the identity type $x =_A y$ is a set. (Thus, Lemma 3.1.8 could equivalently be read as saying that the identity types of a set are also sets.) This will be the basis of the inductive definition of n-types we will give in Chapter 7.

We can also extend this characterization "downwards" from sets. That is, a type A is a set just when for any $x, y : A$, any two elements of $x =_A y$ are equal. Since sets are equivalently 0-types, it is natural to call a type a (-1)-*type* if it has this latter property (any two elements of it are equal). Such types may be regarded as *propositions in a narrow sense*, and their study is just what is usually called "logic"; it will occupy us for the rest of this chapter.

3.2 Propositions as types?

Until now, we have been following the straightforward "propositions as types" philosophy described in §1.11, according to which English phrases such as "there exists an $x : A$ such that $P(x)$" are interpreted by corresponding types such as $\sum_{(x:A)} P(x)$, with the proof of a statement being regarded as judging some specific element to inhabit that type. However, we have also seen some ways in which the "logic" resulting from this reading seems unfamiliar to a classical mathematician. For instance, in Theorem 2.15.7 we saw that the statement

$$\text{"If for all } x : X \text{ there exists an } a : A(x) \text{ such that } P(x, a), \text{ then there exists a function } g : \prod_{(x:A)} A(x) \text{ such that for all } x : X \text{ we have } P(x, g(x))\text{,"} \tag{3.2.1}$$

which looks like the classical *axiom of choice*, is always true under this reading. This is a noteworthy, and often useful, feature of the propositions-as-types logic, but it also illustrates how significantly it differs from the classical interpretation of logic, under which the axiom of choice is not a logical truth, but an additional "axiom".

On the other hand, we can now also show that corresponding statements looking like the classical *law of double negation* and *law of excluded middle* are incompatible with the univalence axiom.

Theorem 3.2.2. *It is not the case that for all $A : \mathcal{U}$ we have $\neg(\neg A) \to A$.*

Proof. Recall that $\neg A \equiv (A \to \mathbf{0})$. We also read "it is not the case that ..." as the operator $\neg$. Thus, in order to prove this statement, it suffices to assume given some $f : \prod_{(A:\mathcal{U})}(\neg\neg A \to A)$ and construct an element of $\mathbf{0}$.

The idea of the following proof is to observe that f, like any function in type theory, is "continuous". By univalence, this implies that f is *natural* with respect to equivalences of types. From this, and a fixed-point-free autoequivalence, we will be able to extract a contradiction.

Let $e : \mathbf{2} \simeq \mathbf{2}$ be the equivalence defined by $e(0_{\mathbf{2}}) :\equiv 1_{\mathbf{2}}$ and $e(1_{\mathbf{2}}) :\equiv 0_{\mathbf{2}}$, as in Example 3.1.9. Let $p : \mathbf{2} = \mathbf{2}$ be the path corresponding to e by univalence, i.e. $p :\equiv \mathsf{ua}(e)$. Then we have $f(\mathbf{2}) : \neg\neg\mathbf{2} \to \mathbf{2}$ and

$$\mathsf{apd}_f(p) : \mathsf{transport}^{A\mapsto(\neg\neg A\to A)}(p, f(\mathbf{2})) = f(\mathbf{2}).$$

Hence, for any $u : \neg\neg\mathbf{2}$, we have

$$\mathsf{happly}(\mathsf{apd}_f(p), u) : \mathsf{transport}^{A\mapsto(\neg\neg A\to A)}(p, f(\mathbf{2}))(u) = f(\mathbf{2})(u).$$

Now by (2.9.4), transporting $f(\mathbf{2}) : \neg\neg\mathbf{2} \to \mathbf{2}$ along p in the type family $A \mapsto (\neg\neg A \to A)$ is equal to the function which transports its argument along p^{-1} in the type family $A \mapsto \neg\neg A$, applies $f(\mathbf{2})$, then transports the result along p in the type family $A \mapsto A$:

$$\mathsf{transport}^{A\mapsto(\neg\neg A\to A)}(p, f(\mathbf{2}))(u) = \mathsf{transport}^{A\mapsto A}(p, f(\mathbf{2})(\mathsf{transport}^{A\mapsto\neg\neg A}(p^{-1}, u)))$$

However, any two points $u, v : \neg\neg\mathbf{2}$ are equal by function extensionality, since for any $x : \neg\mathbf{2}$ we have $u(x) : \mathbf{0}$ and thus we can derive any conclusion, in particular $u(x) = v(x)$. Thus, we have $\mathsf{transport}^{A\mapsto\neg\neg A}(p^{-1}, u) = u$, and so from $\mathsf{happly}(\mathsf{apd}_f(p), u)$ we obtain an equality

$$\mathsf{transport}^{A\mapsto A}(p, f(\mathbf{2})(u)) = f(\mathbf{2})(u).$$

Finally, as discussed in §2.10, transporting in the type family $A \mapsto A$ along the path $p \equiv \mathsf{ua}(e)$ is equivalent to applying the equivalence e; thus we have

$$e(f(\mathbf{2})(u)) = f(\mathbf{2})(u). \tag{3.2.3}$$

However, we can also prove that

$$\prod_{x:\mathbf{2}} \neg(e(x) = x). \tag{3.2.4}$$

This follows from a case analysis on x: both cases are immediate from the definition of e and the injectivity of inl and inr which we proved in §2.12. Thus, applying (3.2.4) to $f(\mathbf{2})(u)$ and (3.2.3), we obtain an element of $\mathbf{0}$. □

Remark 3.2.5. In particular, this implies that there can be no Hilbert-style "choice operator" which selects an element of every nonempty type. The point is that no such operator can be *natural*, and under the univalence axiom, all functions acting on types must be natural with respect to equivalences.

Remark 3.2.6. It is, however, still the case that $\neg\neg\neg A \to \neg A$ for any A; see Exercise 1.11.

Corollary 3.2.7. *It is not the case that for all $A : \mathcal{U}$ we have $A + (\neg A)$.*

Proof. Suppose we had $g : \prod_{(A:\mathcal{U})}(A + (\neg A))$. We will show that then $\prod_{(A:\mathcal{U})}(\neg\neg A \to A)$, so that we can apply Theorem 3.2.2. Thus, suppose $A : \mathcal{U}$ and $u : \neg\neg A$; we want to construct an element of A.

Now $g(A) : A + (\neg A)$, so by case analysis, we may assume either $g(A) \equiv \mathsf{inl}(a)$ for some $a : A$, or $g(A) \equiv \mathsf{inr}(w)$ for some $w : \neg A$. In the first case, we have $a : A$, while in the second case we have $u(w) : \mathbf{0}$ and so we can obtain anything we wish (such as A). Thus, in both cases we have an element of A, as desired. □

Thus, if we want to assume the univalence axiom (which, of course, we do) and still leave ourselves the option of classical reasoning (which is also desirable), we cannot use the unmodified propositions-as-types principle to interpret *all* informal mathematical statements into type theory, since then the law of excluded middle would be false. However, neither do we want to discard propositions-as-types entirely, because of its many good properties (such as simplicity, constructivity, and computability). We now discuss a modification of propositions-as-types which resolves these problems; in §3.10 we will return to the question of which logic to use when.

3.3 Mere propositions

We have seen that the propositions-as-types logic has both good and bad properties. Both have a common cause: when types are viewed as propositions, they can contain more information than mere truth or falsity, and all "logical" constructions on them must respect this additional information. This suggests that we could obtain a more conventional logic by restricting attention to types that do *not* contain any more information than a truth value, and only regarding these as logical propositions.

Such a type A will be "true" if it is inhabited, and "false" if its inhabitation yields a contradiction (i.e. if $\neg A \equiv (A \to \mathbf{0})$ is inhabited). What we want to avoid, in order to obtain a more traditional sort of logic, is treating as logical propositions those types for which giving an element of them gives more information than simply knowing that the type is inhabited. For instance, if we are given an element of $\mathbf{2}$, then we receive more information than the mere fact that $\mathbf{2}$ contains some element. Indeed, we receive exactly *one bit* more information: we know *which* element of $\mathbf{2}$ we were given. By contrast, if we are given an element of $\mathbf{1}$, then we receive no more information than the mere fact that $\mathbf{1}$ contains an element, since any two elements of $\mathbf{1}$ are equal to each other. This suggests the following definition.

Definition 3.3.1. A type P is a **mere proposition** if for all $x, y : P$ we have $x = y$.

Note that since we are still doing mathematics *in* type theory, this is a definition *in* type theory, which means it is a type — or, rather, a type family. Specifically, for any $P : \mathcal{U}$, the type $\mathsf{isProp}(P)$ is defined to be

$$\mathsf{isProp}(P) :\equiv \prod_{x,y:P} (x = y).$$

Thus, to assert that "P is a mere proposition" means to exhibit an inhabitant of $\mathsf{isProp}(P)$, which is a dependent function connecting any two elements of P by a path. The continuity/naturality

of this function implies that not only are any two elements of P equal, but P contains no higher homotopy either.

Lemma 3.3.2. *If P is a mere proposition and $x_0 : P$, then $P \simeq \mathbf{1}$.*

Proof. Define $f : P \to \mathbf{1}$ by $f(x) :\equiv \star$, and $g : \mathbf{1} \to P$ by $g(u) :\equiv x_0$. The claim follows from the next lemma, and the observation that $\mathbf{1}$ is a mere proposition by Theorem 2.8.1. □

Lemma 3.3.3. *If P and Q are mere propositions such that $P \to Q$ and $Q \to P$, then $P \simeq Q$.*

Proof. Suppose given $f : P \to Q$ and $g : Q \to P$. Then for any $x : P$, we have $g(f(x)) = x$ since P is a mere proposition. Similarly, for any $y : Q$ we have $f(g(y)) = y$ since Q is a mere proposition; thus f and g are quasi-inverses. □

In homotopy theory, a space that is homotopy equivalent to $\mathbf{1}$ is said to be *contractible*. Thus, any mere proposition which is inhabited is contractible (see also §3.11). On the other hand, the uninhabited type $\mathbf{0}$ is also (vacuously) a mere proposition. In classical mathematics, at least, these are the only two possibilities.

Mere propositions are also called *subterminal objects* (if thinking categorically), *subsingletons* (if thinking set-theoretically), or *h-propositions*. The discussion in §3.1 suggests we should also call them (-1)*-types*; we will return to this in Chapter 7. The adjective "mere" emphasizes that although any type may be regarded as a proposition (which we prove by giving an inhabitant of it), a type that is a mere proposition cannot usefully be regarded as any *more* than a proposition: there is no additional information contained in a witness of its truth.

Note that a type A is a set if and only if for all $x, y : A$, the identity type $x =_A y$ is a mere proposition. On the other hand, by copying and simplifying the proof of Lemma 3.1.8, we have:

Lemma 3.3.4. *Every mere proposition is a set.*

Proof. Suppose $f : \mathsf{isProp}(A)$; thus for all $x, y : A$ we have $f(x, y) : x = y$. Fix $x : A$ and define $g(y) :\equiv f(x, y)$. Then for any $y, z : A$ and $p : y = z$ we have $\mathsf{apd}_g(p) : p_*(g(y)) = g(z)$. Hence by Lemma 2.11.2, we have $g(y) \cdot p = g(z)$, which is to say that $p = g(y)^{-1} \cdot g(z)$. Thus, for any $p, q : x = y$, we have $p = g(x)^{-1} \cdot g(y) = q$. □

In particular, this implies:

Lemma 3.3.5. *For any type A, the types $\mathsf{isProp}(A)$ and $\mathsf{isSet}(A)$ are mere propositions.*

Proof. Suppose $f, g : \mathsf{isProp}(A)$. By function extensionality, to show $f = g$ it suffices to show $f(x, y) = g(x, y)$ for any $x, y : A$. But $f(x, y)$ and $g(x, y)$ are both paths in A, and hence are equal because, by either f or g, we have that A is a proposition, and hence by Lemma 3.3.4 is a set. Similarly, suppose $f, g : \mathsf{isSet}(A)$, which is to say that for all $a, b : A$, $f(a, b) : a = b$ and $g(a, b) : a = b$. But by then since A is a set (by either f or g), it follows that $f(a, b) = g(a, b)$, and hence $f = g$ by function extensionality. □

We have seen one other example so far: condition (iii) in §2.4 asserts that for any function f, the type $\mathsf{isequiv}(f)$ should be a mere proposition.

3.4 Classical vs. intuitionistic logic

With the notion of mere proposition in hand, we can now give the proper formulation of the **law of excluded middle** in homotopy type theory:

$$\mathsf{LEM} :\equiv \prod_{A:\mathcal{U}} \Big(\mathsf{isProp}(A) \to (A + \neg A)\Big). \tag{3.4.1}$$

Similarly, the **law of double negation** is

$$\mathsf{DN} :\equiv \prod_{A:\mathcal{U}} \Big(\mathsf{isProp}(A) \to (\neg\neg A \to A)\Big). \tag{3.4.2}$$

These formulations avoid the paradoxes of Theorem 3.2.2 and Corollary 3.2.7, since $\mathbf{2}$ is not a mere proposition. In order to distinguish these from the more general propositions-as-types formulations, we rename the latter:

$$\begin{aligned} \mathsf{LEM}_\infty &:\equiv \prod_{A:\mathcal{U}} (A + \neg A) \\ \mathsf{DN}_\infty &:\equiv \prod_{A:\mathcal{U}} (\neg\neg A \to A). \end{aligned}$$

For emphasis, the proper versions (3.4.1) and (3.4.2) may be denoted LEM_{-1} and DN_{-1}; see also Exercise 7.5.

Although LEM and DN are not consequences of the basic type theory described in Chapter 1, they may be consistently assumed as axioms (unlike their ∞ counterparts). For instance, we will assume them in §10.4. (The two are also easily seen to be equivalent to each other; see Exercise 3.17.)

However, it can be surprising how far we can get without using such axioms. Quite often, a simple reformulation of a definition or theorem enables us to avoid invoking excluded middle or double negation. While this takes a little getting used to sometimes, it is often worth the hassle, resulting in more elegant and more general proofs. We discussed some of the benefits of this in the introduction.

For instance, in classical mathematics, double negations are frequently used unnecessarily. A very simple example is the common assumption that a set A is "nonempty", which literally means it is *not* the case that A contains *no* elements. Almost always what is really meant is the positive assertion that A *does* contain at least one element, and by removing the double negation we make the statement less dependent on LEM. Thus we say that a type A is **inhabited** to mean that we assert A itself as a proposition (i.e. we construct an element of A, usually unnamed).

Similarly, it is not uncommon in classical mathematics to find unnecessary proofs by contradiction. Of course, proof by contradiction proceeds by way of the law of double negation: we assume $\neg A$ and derive a contradiction, thereby deducing $\neg\neg A$, and thus by DN we obtain A. However, often the derivation of a contradiction from $\neg A$ can be rephrased slightly so as to yield a direct proof of A, avoiding the need for DN.

It is also important to note that if the goal is to prove a *negation*, then "proof by contradiction" does not involve DN. In fact, since $\neg A$ is by definition the type $A \to \mathbf{0}$, by definition to prove $\neg A$

is to prove a contradiction ($\mathbf{0}$) under the assumption of A. Similarly, the law of double negation does hold for negated propositions: $\neg\neg\neg A \to \neg A$. With practice, one learns to distinguish more carefully between negated and un-negated propositions and to notice when LEM and DN are being used and when they are not.

Thus, contrary to how it may appear on the surface, doing mathematics "constructively" does not usually involve giving up important theorems, but rather finding the best way to state the definitions so as to make the important theorems constructively provable. That is, we may freely use the LEM when first investigating a subject, but once that subject is better understood, we can hope to refine its definitions and proofs so as to avoid that axiom. This sort of observation is even more pronounced in *homotopy* type theory, where the powerful tools of univalence and higher inductive types allow us to constructively attack many problems that traditionally would require classical reasoning. We will see several examples of this in Part II.

3.5 Subsets and propositional resizing

As another example of the usefulness of mere propositions, we discuss subsets (and more generally subtypes). Suppose $P : A \to \mathcal{U}$ is a type family, with each type $P(x)$ regarded as a proposition. Then P itself is a *predicate* on A, or a *property* of elements of A.

In set theory, whenever we have a predicate on P on a set A, we may form the subset $\{\, x \in A \mid P(x) \,\}$. In type theory, the obvious analogue is the Σ-type $\sum_{(x:A)} P(x)$. An inhabitant of $\sum_{(x:A)} P(x)$ is, of course, a pair (x, p) where $x : A$ and p is a proof of $P(x)$. However, for general P, an element $a : A$ might give rise to more than one distinct element of $\sum_{(x:A)} P(x)$, if the proposition $P(a)$ has more than one distinct proof. This is counter to the usual intuition of a *subset*. But if P is a *mere* proposition, then this cannot happen.

Lemma 3.5.1. *Suppose $P : A \to \mathcal{U}$ is a type family such that $P(x)$ is a mere proposition for all $x : A$. If $u, v : \sum_{(x:A)} P(x)$ are such that $\mathsf{pr}_1(u) = \mathsf{pr}_1(v)$, then $u = v$.*

Proof. Suppose $p : \mathsf{pr}_1(u) = \mathsf{pr}_1(v)$. By Theorem 2.7.2, to show $u = v$ it suffices to show $p_*(\mathsf{pr}_2(u)) = \mathsf{pr}_2(v)$. But $p_*(\mathsf{pr}_2(u))$ and $\mathsf{pr}_2(v)$ are both elements of $P(\mathsf{pr}_1(v))$, which is a mere proposition; hence they are equal. □

For instance, recall that in §2.4 we defined

$$(A \simeq B) :\equiv \sum_{f:A\to B} \mathsf{isequiv}(f),$$

where each type $\mathsf{isequiv}(f)$ was supposed to be a mere proposition. It follows that if two equivalences have equal underlying functions, then they are equal as equivalences.

Henceforth, if $P : A \to \mathcal{U}$ is a family of mere propositions, we may write

$$\{\, x : A \mid P(x) \,\}$$

as an alternative notation for $\sum_{(x:A)} P(x)$. If A is a set, we call $\{\, x : A \mid P(x) \,\}$ a **subset** of A; for general A we might call it a **subtype**.

As another example, we may define the "subuniverses" of sets and of mere propositions in a universe $\mathcal{U}$:

$$\mathsf{Set}_{\mathcal{U}} :\equiv \{\, A : \mathcal{U} \mid \mathsf{isSet}(A) \,\}$$
$$\mathsf{Prop}_{\mathcal{U}} :\equiv \{\, A : \mathcal{U} \mid \mathsf{isProp}(A) \,\}.$$

An element of $\mathsf{Set}_{\mathcal{U}}$ is a type $A : \mathcal{U}$ together with evidence $s : \mathsf{isSet}(A)$, and similarly for $\mathsf{Prop}_{\mathcal{U}}$. Lemma 3.5.1 implies that $(A, s) =_{\mathsf{Set}_{\mathcal{U}}} (B, t)$ is equivalent to $A =_{\mathcal{U}} B$ (and hence to $A \simeq B$). Thus, we will frequently abuse notation and write simply $A : \mathsf{Set}_{\mathcal{U}}$ instead of $(A, s) : \mathsf{Set}_{\mathcal{U}}$. We may also drop the subscript $\mathcal{U}$ if there is no need to specify the universe in question.

Recall that for any two universes $\mathcal{U}_i$ and $\mathcal{U}_{i+1}$, if $A : \mathcal{U}_i$ then also $A : \mathcal{U}_{i+1}$. Thus, for any $(A, s) : \mathsf{Set}_{\mathcal{U}_i}$ we also have $(A, s) : \mathsf{Set}_{\mathcal{U}_{i+1}}$, and similarly for $\mathsf{Prop}_{\mathcal{U}_i}$, giving natural maps

$$\mathsf{Set}_{\mathcal{U}_i} \to \mathsf{Set}_{\mathcal{U}_{i+1}} \tag{3.5.2}$$
$$\mathsf{Prop}_{\mathcal{U}_i} \to \mathsf{Prop}_{\mathcal{U}_{i+1}}. \tag{3.5.3}$$

The map (3.5.2) cannot be an equivalence, since then we could reproduce the paradoxes of Cantorian set theory. However, although (3.5.3) is not automatically an equivalence in the type theory we have presented so far, it is consistent to suppose that it is. That is, we may consider adding to type theory the following axiom.

Axiom 3.5.4 (Propositional resizing)**.** *The map* $\mathsf{Prop}_{\mathcal{U}_i} \to \mathsf{Prop}_{\mathcal{U}_{i+1}}$ *is an equivalence.*

We refer to this axiom as **propositional resizing**, since it means that any mere proposition in the universe $\mathcal{U}_{i+1}$ can be "resized" to an equivalent one in the smaller universe $\mathcal{U}_i$. It follows automatically if $\mathcal{U}_{i+1}$ satisfies LEM (see Exercise 3.9). We will not assume this axiom in general, although in some places we will use it as an explicit hypothesis. It is a form of *impredicativity* for mere propositions, and by avoiding its use, the type theory is said to remain *predicative*.

In practice, what we want most frequently is a slightly different statement: that a universe $\mathcal{U}$ under consideration contains a type which "classifies all mere propositions". In other words, we want a type $\Omega : \mathcal{U}$ together with a Ω-indexed family of mere propositions, which contains every mere proposition up to equivalence. This statement follows from propositional resizing as stated above if $\mathcal{U}$ is not the smallest universe $\mathcal{U}_0$, since then we can define $\Omega :\equiv \mathsf{Prop}_{\mathcal{U}_0}$.

One use for impredicativity is to define powersets. It is natural to define the powerset of a set A to be $A \to \mathsf{Prop}_{\mathcal{U}}$; but in the absence of impredicativity, this definition depends (even up to equivalence) on the choice of the universe $\mathcal{U}$. But with propositional resizing, we can define the powerset to be $A \to \Omega$, which is then independent of $\mathcal{U}$. See also §10.1.4.

3.6 The logic of mere propositions

We mentioned in §1.1 that in contrast to type theory, which has only one basic notion (types), set-theoretic foundations have two basic notions: sets and propositions. Thus, a classical mathematician is accustomed to manipulating these two kinds of objects separately.

It is possible to recover a similar dichotomy in type theory, with the role of the set-theoretic propositions being played by the types (and type families) that are *mere* propositions. In many

cases, the logical connectives and quantifiers can be represented in this logic by simply restricting the corresponding type-former to the mere propositions. Of course, this requires knowing that the type-former in question preserves mere propositions.

Example 3.6.1. If A and B are mere propositions, so is $A \times B$. This is easy to show using the characterization of paths in products, just like Example 3.1.5 but simpler. Thus, the connective "and" preserves mere propositions.

Example 3.6.2. If A is any type and $B : A \to \mathcal{U}$ is such that for all $x : A$, the type $B(x)$ is a mere proposition, then $\prod_{(x:A)} B(x)$ is a mere proposition. The proof is just like Example 3.1.6 but simpler: given $f, g : \prod_{(x:A)} B(x)$, for any $x : A$ we have $f(x) = g(x)$ since $B(x)$ is a mere proposition. But then by function extensionality, we have $f = g$.

In particular, if B is a mere proposition, then so is $A \to B$ regardless of what A is. In even more particular, since $\mathbf{0}$ is a mere proposition, so is $\neg A \equiv (A \to \mathbf{0})$. Thus, the connectives "implies" and "not" preserve mere propositions, as does the quantifier "for all".

On the other hand, some type formers do not preserve mere propositions. Even if A and B are mere propositions, $A + B$ will not in general be. For instance, $\mathbf{1}$ is a mere proposition, but $\mathbf{2} = \mathbf{1} + \mathbf{1}$ is not. Logically speaking, $A + B$ is a "purely constructive" sort of "or": a witness of it contains the additional information of *which* disjunct is true. Sometimes this is very useful, but if we want a more classical sort of "or" that preserves mere propositions, we need a way to "truncate" this type into a mere proposition by forgetting this additional information.

The same issue arises with the Σ-type $\sum_{(x:A)} P(x)$. This is a purely constructive interpretation of "there exists an $x : A$ such that $P(x)$" which remembers the witness x, and hence is not generally a mere proposition even if each type $P(x)$ is. (Recall that we observed in §3.5 that $\sum_{(x:A)} P(x)$ can also be regarded as "the subset of those $x : A$ such that $P(x)$".)

3.7 Propositional truncation

The *propositional truncation*, also called the (-1)*-truncation*, *bracket type*, or *squash type*, is an additional type former which "truncates" or "squashes" a type down to a mere proposition, forgetting all information contained in inhabitants of that type other than their existence.

More precisely, for any type A, there is a type $\|A\|$. It has two constructors:

- For any $a : A$ we have $|a| : \|A\|$.
- For any $x, y : \|A\|$, we have $x = y$.

The first constructor means that if A is inhabited, so is $\|A\|$. The second ensures that $\|A\|$ is a mere proposition; usually we leave the witness of this fact nameless.

The induction principle of $\|A\|$ says that:

- If B is a mere proposition and we have $f : A \to B$, then there is an induced $g : \|A\| \to B$ such that $g(|a|) \equiv f(a)$ for all $a : A$.

In other words, any mere proposition which follows from (the inhabitedness of) A already follows from $\|A\|$. Thus, $\|A\|$, as a mere proposition, contains no more information than the inhabitedness of A.

In Exercises 3.13 and 3.14 and §6.9 we will describe some ways to construct $\|A\|$ in terms of more general things. For now, we simply assume it as an additional rule alongside those of Chapter 1.

With the propositional truncation, we can extend the "logic of mere propositions" to cover disjunction and the existential quantifier. Specifically, $\|A + B\|$ is a mere propositional version of "A or B", which does not "remember" the information of which disjunct is true.

The induction principle of truncation implies that we can still do a case analysis on $\|A + B\|$ *when attempting to prove a mere proposition*. That is, suppose we have an assumption $u : \|A + B\|$ and we are trying to prove a mere proposition Q. In other words, we are trying to define an element of $\|A + B\| \to Q$. Since Q is a mere proposition, by the induction principle for propositional truncation, it suffices to construct a function $A + B \to Q$. But now we can use case analysis on $A + B$.

Similarly, for a type family $P : A \to \mathcal{U}$, we can consider $\left\|\sum_{(x:A)} P(x)\right\|$, which is a mere propositional version of "there exists an $x : A$ such that $P(x)$". As for disjunction, by combining the induction principles of truncation and Σ-types, if we have an assumption of type $\left\|\sum_{(x:A)} P(x)\right\|$, we may introduce new assumptions $x : A$ and $y : P(x)$ *when attempting to prove a mere proposition*. In other words, if we know that there exists some $x : A$ such that $P(x)$, but we don't have a particular such x in hand, then we are free to make use of such an x as long as we aren't trying to construct anything which might depend on the particular value of x. Requiring the codomain to be a mere proposition expresses this independence of the result on the witness, since all possible inhabitants of such a type must be equal.

For the purposes of set-level mathematics in Chapters 10 and 11, where we deal mostly with sets and mere propositions, it is convenient to use the traditional logical notations to refer only to "propositionally truncated logic".

Definition 3.7.1. We define **traditional logical notation** using truncation as follows, where P and Q denote mere propositions (or families thereof):

$$\begin{aligned}
\top &:\equiv \mathbf{1} \\
\bot &:\equiv \mathbf{0} \\
P \wedge Q &:\equiv P \times Q \\
P \Rightarrow Q &:\equiv P \to Q \\
P \Leftrightarrow Q &:\equiv P = Q \\
\neg P &:\equiv P \to \mathbf{0} \\
P \vee Q &:\equiv \|P + Q\| \\
\forall (x : A).\, P(x) &:\equiv \prod_{x:A} P(x) \\
\exists (x : A).\, P(x) &:\equiv \left\|\sum_{x:A} P(x)\right\|
\end{aligned}$$

The notations $\wedge$ and $\vee$ are also used in homotopy theory for the smash product and the wedge of pointed spaces, which we will introduce in Chapter 6. This technically creates a potential for conflict, but no confusion will generally arise.

3.8 The axiom of choice

We can now properly formulate the axiom of choice in homotopy type theory. Assume a type X and type families $A : X \to \mathcal{U}$ and $P : \prod_{(x:X)} A(x) \to \mathcal{U}$, and moreover that

- X is a set,
- $A(x)$ is a set for all $x : X$, and
- $P(x, a)$ is a mere proposition for all $x : X$ and $a : A(x)$.

The **axiom of choice** AC asserts that under these assumptions,

$$\Big(\prod_{x:X} \Big\| \sum_{a:A(x)} P(x,a) \Big\|\Big) \to \Big\| \sum_{(g:\prod_{(x:X)} A(x))} \prod_{(x:X)} P(x, g(x)) \Big\| \tag{3.8.1}$$

Of course, this is a direct translation of (3.2.1) where we read "there exists $x : A$ such that $B(x)$" as $\left\|\sum_{(x:A)} B(x)\right\|$, so we could have written the statement in the familiar logical notation as

$$\Big(\forall(x : X).\, \exists(a : A(x)).\, P(x,a)\Big) \Rightarrow \Big(\exists(g : \textstyle\prod_{(x:X)} A(x)).\, \forall(x : X).\, P(x, g(x))\Big).$$

In particular, note that the propositional truncation appears twice. The truncation in the domain means we assume that for every x there exists some $a : A(x)$ such that $P(x, a)$, but that these values are not chosen or specified in any known way. The truncation in the codomain means we conclude that there exists some function g, but this function is not determined or specified in any known way.

In fact, because of Theorem 2.15.7, this axiom can also be expressed in a simpler form.

Lemma 3.8.2. *The axiom of choice* (3.8.1) *is equivalent to the statement that for any set X and any $Y : X \to \mathcal{U}$ such that each $Y(x)$ is a set, we have*

$$\Big(\prod_{x:X} \Big\| Y(x) \Big\|\Big) \to \Big\| \prod_{x:X} Y(x) \Big\|. \tag{3.8.3}$$

This corresponds to a well-known equivalent form of the classical axiom of choice, namely "the cartesian product of a family of nonempty sets is nonempty."

Proof. By Theorem 2.15.7, the codomain of (3.8.1) is equivalent to

$$\Big\| \prod_{(x:X)} \sum_{(a:A(x))} P(x,a) \Big\|.$$

Thus, (3.8.1) is equivalent to the instance of (3.8.3) where $Y(x) :\equiv \sum_{(a:A(x))} P(x, a)$. Conversely, (3.8.3) is equivalent to the instance of (3.8.1) where $A(x) :\equiv Y(x)$ and $P(x, a) :\equiv \mathbf{1}$. Thus, the two are logically equivalent. Since both are mere propositions, by Lemma 3.3.3 they are equivalent types. □

As with LEM and DN, the equivalent forms (3.8.1) and (3.8.3) are not a consequence of our basic type theory, but they may consistently be assumed as axioms.

Remark 3.8.4. It is easy to show that the right side of (3.8.3) always implies the left. Since both are mere propositions, by Lemma 3.3.3 the axiom of choice is also equivalent to asking for an equivalence

$$\left(\prod_{x:X} \left\| Y(x) \right\|\right) \simeq \left\| \prod_{x:X} Y(x) \right\|$$

This illustrates a common pitfall: although dependent function types preserve mere propositions (Example 3.6.2), they do not commute with truncation: $\left\| \prod_{(x:A)} P(x) \right\|$ is not generally equivalent to $\prod_{(x:A)} \|P(x)\|$. The axiom of choice, if we assume it, says that this is true *for sets*; as we will see below, it fails in general.

The restriction in the axiom of choice to types that are sets can be relaxed to a certain extent. For instance, we may allow A and P in (3.8.1), or Y in (3.8.3), to be arbitrary type families; this results in a seemingly stronger statement that is equally consistent. We may also replace the propositional truncation by the more general n-truncations to be considered in Chapter 7, obtaining a spectrum of axioms AC_n interpolating between (3.8.1), which we call simply AC (or AC_{-1} for emphasis), and Theorem 2.15.7, which we shall call AC_∞. See also Exercises 7.6 and 7.7. However, observe that we cannot relax the requirement that X be a set.

Lemma 3.8.5. *There exists a type X and a family $Y : X \to \mathcal{U}$ such that each $Y(x)$ is a set, but such that* (3.8.3) *is false.*

Proof. Define $X :\equiv \sum_{(A:\mathcal{U})} \|\mathbf{2} = A\|$, and let $x_0 :\equiv (\mathbf{2}, |\mathsf{refl}_{\mathbf{2}}|) : X$. Then by the identification of paths in Σ-types, the fact that $\|A = \mathbf{2}\|$ is a mere proposition, and univalence, we have $((A, p) =_X (B, p)) \simeq (A \simeq B)$. In particular, $(x_0 =_X x_0) \simeq (\mathbf{2} \simeq \mathbf{2})$, so as in Example 3.1.9, X is not a set. But if we define $Y(x) :\equiv (x_0 = x)$, then each $Y(x)$ is a set.

Now by definition, for any $(A, p) : X$ we have $\|\mathbf{2} = A\|$, and hence $\|x_0 = (A, p)\|$. Thus, we have $\prod_{(x:X)} \|Y(x)\|$. If (3.8.3) held for this X and Y, then we would also have $\left\| \prod_{(x:X)} Y(x) \right\|$. Since we are trying to derive a contradiction ($\mathbf{0}$), which is a mere proposition, we may assume $\prod_{(x:X)} Y(x)$, i.e. that $\prod_{(x:X)} (x_0 = x)$. But this implies X is a mere proposition, and hence a set, which is a contradiction. □

3.9 The principle of unique choice

The following observation is trivial, but very useful.

Lemma 3.9.1. *If P is a mere proposition, then $P \simeq \|P\|$.*

Proof. Of course, we have $P \to \|P\|$ by definition. And since P is a mere proposition, the universal property of $\|P\|$ applied to $\mathsf{id}_P : P \to P$ yields $\|P\| \to P$. These functions are quasi-inverses by Lemma 3.3.3. □

Among its important consequences is the following.

Corollary 3.9.2 (The principle of unique choice)**.** *Suppose a type family $P : A \to \mathcal{U}$ such that*

(i) For each x, the type $P(x)$ is a mere proposition, and

(ii) For each x we have $\|P(x)\|$.

Then we have $\prod_{(x:A)} P(x)$.

Proof. Immediate from the two assumptions and the previous lemma. □

The corollary also encapsulates a very useful technique of reasoning. Namely, suppose we know that $\|A\|$, and we want to use this to construct an element of some other type B. We would like to use an element of A in our construction of an element of B, but this is allowed only if B is a mere proposition, so that we can apply the induction principle for the propositional truncation $\|A\|$; the most we could hope to do in general is to show $\|B\|$. Instead, we can extend B with additional data which characterizes *uniquely* the object we wish to construct. Specifically, we define a predicate $Q : B \to \mathcal{U}$ such that $\sum_{(x:B)} Q(x)$ is a mere proposition. Then from an element of A we construct an element $b : B$ such that $Q(b)$, hence from $\|A\|$ we can construct $\left\|\sum_{(x:B)} Q(x)\right\|$, and because $\left\|\sum_{(x:B)} Q(x)\right\|$ is equivalent to $\sum_{(x:B)} Q(x)$ an element of B may be projected from it. We provide an example below.

A similar issue arises in set-theoretic mathematics, although it manifests slightly differently. If we are trying to define a function $f : A \to B$, and depending on an element $a : A$ we are able to prove mere existence of some $b : B$, we are not done yet because we need to actually pinpoint an element of B, not just prove its existence. One option is of course to refine the argument to unique existence of $b : B$, as we did in type theory. But in set theory the problem can often be avoided more simply by an application of the axiom of choice, which picks the required elements for us. In homotopy type theory, however, quite apart from any desire to avoid choice, the available forms of choice are simply less applicable, since they require that the domain of choice be a *set*. Thus, if A is not a set (such as perhaps a universe $\mathcal{U}$), there is no consistent form of choice that will allow us to simply pick an element of B for each $a : A$ to use in defining $f(a)$.

Theorem 3.9.3. *Suppose $P : \mathbb{N} \to \mathcal{U}$ is such that each $P(n)$ is a mere proposition, and that $\prod_{(n:\mathbb{N})}(P(n) + \neg P(n))$ (such a predicate is called* decidable*). Then*

$$\left\|\sum_{n:\mathbb{N}} P(n)\right\| \to \sum_{n:\mathbb{N}} P(n).$$

Sketch of proof. The hypotheses imply that

$$\Big(\sum_{n:\mathbb{N}} P(n)\Big) \to \sum_{n:\mathbb{N}} \Big(P(n) \times \prod_{m:\mathbb{N}} \big((m < n) \to \neg P(m)\big)\Big).$$

In words, given n such that $P(n)$, we can find the least such n: we test every $m < n$ in turn, using decidability to do a case analysis, until we find the first one that satisfies $P(m)$. However, the right-hand side of the above implication is a mere proposition: if both n and n' are least numbers satisfying P then they must be equal. Therefore, we also have

$$\left\|\sum_{n:\mathbb{N}} P(n)\right\| \to \sum_{n:\mathbb{N}} \Big(P(n) \times \prod_{m:\mathbb{N}} \big((m < n) \to \neg P(m)\big)\Big)$$

from which the claim follows. □

3.10 When are propositions truncated?

At first glance, it may seem that the truncated versions of $+$ and Σ are actually closer to the informal mathematical meaning of "or" and "there exists" than the untruncated ones. Certainly, they are closer to the *precise* meaning of "or" and "there exists" in the first-order logic which underlies formal set theory, since the latter makes no attempt to remember any witnesses to the truth of propositions. However, it may come as a surprise to realize that the practice of *informal* mathematics is often more accurately described by the untruncated forms.

For example, consider a statement like "every prime number is either 2 or odd." The working mathematician feels no compunction about using this fact not only to prove *theorems* about prime numbers, but also to perform *constructions* on prime numbers, perhaps doing one thing in the case of 2 and another in the case of an odd prime. The end result of the construction is not merely the truth of some statement, but a piece of data which may depend on the parity of the prime number. Thus, from a type-theoretic perspective, such a construction is naturally phrased using the induction principle for the coproduct type "$(p = 2) + (p \text{ is odd})$", not its propositional truncation.

Admittedly, this is not an ideal example, since "$p = 2$" and "p is odd" are mutually exclusive, so that $(p = 2) + (p \text{ is odd})$ is in fact already a mere proposition and hence equivalent to its truncation (see Exercise 3.6). More compelling examples come from the existential quantifier. It is not uncommon to prove a theorem of the form "there exists an x such that ..." and then refer later on to "the x constructed in Theorem Y" (note the definite article). Moreover, when deriving further properties of this x, one may use phrases such as "by the construction of x in the proof of Theorem Y".

A very common example is "A is isomorphic to B", which strictly speaking means only that there exists *some* isomorphism between A and B. But almost invariably, when proving such a statement, one exhibits a specific isomorphism or proves that some previously known map is an isomorphism, and it often matters later on what particular isomorphism was given.

Set-theoretically trained mathematicians often feel a twinge of guilt at such "abuses of language". We may attempt to apologize for them, expunge them from final drafts, or weasel out of them with vague words like "canonical". The problem is exacerbated by the fact that in formalized set theory, there is technically no way to "construct" objects at all — we can only prove that an object with certain properties exists. Untruncated logic in type theory thus captures some common practices of informal mathematics that the set theoretic reconstruction obscures. (This is similar to how the univalence axiom validates the common, but formally unjustified, practice of identifying isomorphic objects.)

On the other hand, sometimes truncated logic is essential. We have seen this in the statements of LEM and AC; some other examples will appear later on in the book. Thus, we are faced with the problem: when writing informal type theory, what should we mean by the words "or" and "there exists" (along with common synonyms such as "there is" and "we have")?

A universal consensus may not be possible. Perhaps depending on the sort of mathematics being done, one convention or the other may be more useful — or, perhaps, the choice of convention may be irrelevant. In this case, a remark at the beginning of a mathematical paper may suffice to inform the reader of the linguistic conventions in use therein. However, even after

one overall convention is chosen, the other sort of logic will usually arise at least occasionally, so we need a way to refer to it. More generally, one may consider replacing the propositional truncation with another operation on types that behaves similarly, such as the double negation operation $A \mapsto \neg\neg A$, or the n-truncations to be considered in Chapter 7. As an experiment in exposition, in what follows we will occasionally use *adverbs* to denote the application of such "modalities" as propositional truncation.

For instance, if untruncated logic is the default convention, we may use the adverb **merely** to denote propositional truncation. Thus the phrase

$$\text{"there merely exists an } x : A \text{ such that } P(x)\text{"}$$

indicates the type $\left\|\sum_{(x:A)} P(x)\right\|$. Similarly, we will say that a type A is **merely inhabited** to mean that its propositional truncation $\|A\|$ is inhabited (i.e. that we have an unnamed element of it). Note that this is a *definition* of the adverb "merely" as it is to be used in our informal mathematical English, in the same way that we define nouns like "group" and "ring", and adjectives like "regular" and "normal", to have precise mathematical meanings. We are not claiming that the dictionary definition of "merely" refers to propositional truncation; the choice of word is meant only to remind the mathematician reader that a mere proposition contains "merely" the information of a truth value and nothing more.

On the other hand, if truncated logic is the current default convention, we may use an adverb such as **purely** or **constructively** to indicate its absence, so that

$$\text{"there purely exists an } x : A \text{ such that } P(x)\text{"}$$

would denote the type $\sum_{(x:A)} P(x)$. We may also use "purely" or "actually" just to emphasize the absence of truncation, even when that is the default convention.

In this book we will continue using untruncated logic as the default convention, for a number of reasons.

(1) We want to encourage the newcomer to experiment with it, rather than sticking to truncated logic simply because it is more familiar.

(2) Using truncated logic as the default in type theory suffers from the same sort of "abuse of language" problems as set-theoretic foundations, which untruncated logic avoids. For instance, our definition of "$A \simeq B$" as the type of equivalences between A and B, rather than its propositional truncation, means that to prove a theorem of the form "$A \simeq B$" is literally to construct a particular such equivalence. This specific equivalence can then be referred to later on.

(3) We want to emphasize that the notion of "mere proposition" is not a fundamental part of type theory. As we will see in Chapter 7, mere propositions are just the second rung on an infinite ladder, and there are also many other modalities not lying on this ladder at all.

(4) Many statements that classically are mere propositions are no longer so in homotopy type theory. Of course, foremost among these is equality.

(5) On the other hand, one of the most interesting observations of homotopy type theory is that a surprising number of types are *automatically* mere propositions, or can be slightly

modified to become so, without the need for any truncation. (See Lemma 3.3.5 and Chapters 4, 7, 9 and 10.) Thus, although these types contain no data beyond a truth value, we can nevertheless use them to construct untruncated objects, since there is no need to use the induction principle of propositional truncation. This useful fact is more clumsy to express if propositional truncation is applied to all statements by default.

(6) Finally, truncations are not very useful for most of the mathematics we will be doing in this book, so it is simpler to notate them explicitly when they occur.

3.11 Contractibility

In Lemma 3.3.2 we observed that a mere proposition which is inhabited must be equivalent to unit, and it is not hard to see that the converse also holds. A type with this property is called *contractible*. Another equivalent definition of contractibility, which is also sometimes convenient, is the following.

Definition 3.11.1. A type A is **contractible**, or a **singleton**, if there is $a : A$, called the **center of contraction**, such that $a = x$ for all $x : A$. We denote the specified path $a = x$ by contr_x.

In other words, the type $\mathsf{isContr}(A)$ is defined to be

$$\mathsf{isContr}(A) :\equiv \sum_{(a:A)} \prod_{(x:A)} (a = x).$$

Note that under the usual propositions-as-types reading, we can pronounce $\mathsf{isContr}(A)$ as "A contains exactly one element", or more precisely "A contains an element, and every element of A is equal to that element".

Remark 3.11.2. We can also pronounce $\mathsf{isContr}(A)$ more topologically as "there is a point $a : A$ such that for all $x : A$ there exists a path from a to x". Note that to a classical ear, this sounds like a definition of *connectedness* rather than contractibility. The point is that the meaning of "there exists" in this sentence is a continuous/natural one. A more correct way to express connectedness would be $\sum_{(a:A)} \prod_{(x:A)} \|a = x\|$; we will come back to this later.

Lemma 3.11.3. *For a type A, the following are logically equivalent.*

(i) *A is contractible in the sense of Definition 3.11.1.*

(ii) *A is a mere proposition, and there is a point $a : A$.*

(iii) *A is equivalent to $\mathbf{1}$.*

Proof. If A is contractible, then it certainly has a point $a : A$ (the center of contraction), while for any $x, y : A$ we have $x = a = y$; thus A is a mere proposition. Conversely, if we have $a : A$ and A is a mere proposition, then for any $x : A$ we have $x = a$; thus A is contractible. And we showed (ii)⇒(iii) in Lemma 3.3.2, while the converse follows since $\mathbf{1}$ easily has property (ii). □

Lemma 3.11.4. *For any type A, the type $\mathsf{isContr}(A)$ is a mere proposition.*

Proof. Suppose given $c, c' : \mathsf{isContr}(A)$. We may assume $c \equiv (a, p)$ and $c' \equiv (a', p')$ for $a, a' : A$ and $p : \prod_{(x:A)}(a = x)$ and $p' : \prod_{(x:A)}(a' = x)$. By the characterization of paths in Σ-types, to show $c = c'$ it suffices to exhibit $q : a = a'$ such that $q_*(p) = p'$.

We choose $q :\equiv p(a')$. For the other equality, by function extensionality we must show that $(q_*(p))(x) = p'(x)$ for any $x : A$. For this, it will suffice to show that for any $x, y : A$ and $u : x = y$ we have $u = p(x)^{-1} \cdot p(y)$, since then we would have $(q_*(p))(x) = p(a')^{-1} \cdot p(x) = p'(x)$. But now we can invoke path induction to assume that $x \equiv y$ and $u \equiv \mathsf{refl}_x$. In this case our goal is to show that $\mathsf{refl}_x = p(x)^{-1} \cdot p(x)$, which is just the inversion law for paths. □

Corollary 3.11.5. *If A is contractible, then so is $\mathsf{isContr}(A)$.*

Proof. By Lemma 3.11.4 and Lemma 3.11.3(ii). □

Like mere propositions, contractible types are preserved by many type constructors. For instance, we have:

Lemma 3.11.6. *If $P : A \to \mathcal{U}$ is a type family such that each $P(a)$ is contractible, then $\prod_{(x:A)} P(x)$ is contractible.*

Proof. By Example 3.6.2, $\prod_{(x:A)} P(x)$ is a mere proposition since each $P(x)$ is. But it also has an element, namely the function sending each $x : A$ to the center of contraction of $P(x)$. Thus by Lemma 3.11.3(ii), $\prod_{(x:A)} P(x)$ is contractible. □

(In fact, the statement of Lemma 3.11.6 is equivalent to the function extensionality axiom. See §4.9.)

Of course, if A is equivalent to B and A is contractible, then so is B. More generally, it suffices for B to be a *retract* of A. By definition, a **retraction** is a function $r : A \to B$ such that there exists a function $s : B \to A$, called its **section**, and a homotopy $\epsilon : \prod_{(b:B)}(r(s(y)) = y)$; then we say that B is a **retract** of A.

Lemma 3.11.7. *If B is a retract of A, and A is contractible, then so is B.*

Proof. Let $a_0 : A$ be the center of contraction. We claim that $b_0 :\equiv p(a_0) : B$ is a center of contraction for B. Let $b : B$; we need a path $b = b_0$. But we have $\epsilon_b : p(s(b)) = b$ and $\mathsf{contr}_{s(b)} : s(b) = a_0$, so by composition

$$\epsilon_b{}^{-1} \cdot p\Big(\mathsf{contr}_{s(b)}\Big) : b = p(a_0) \equiv b_0.$$

□

Contractible types may not seem very interesting, since they are all equivalent to $\mathbf{1}$. One reason the notion is useful is that sometimes a collection of individually nontrivial data will collectively form a contractible type. An important example is the space of paths with one free endpoint. As we will see in §5.8, this fact essentially encapsulates the based path induction principle for identity types.

Lemma 3.11.8. *For any A and any $a : A$, the type $\sum_{(x:A)}(a = x)$ is contractible.*

Proof. We choose as center the point (a, refl_a). Now suppose $(x, p) : \sum_{(x:A)}(a = x)$; we must show $(a, \mathsf{refl}_a) = (x, p)$. By the characterization of paths in Σ-types, it suffices to exhibit $q : a = x$ such that $q_*(\mathsf{refl}_a) = p$. But we can take $q :\equiv p$, in which case $q_*(\mathsf{refl}_a) = p$ follows from the characterization of transport in path types. □

When this happens, it can allows us to simplify a complicated construction up to equivalence, using the informal principle that contractible data can be freely ignored. This principle consists of many lemmas, most of which we leave to the reader; the following is an example.

Lemma 3.11.9. *Let $P : A \to \mathcal{U}$ be a type family.*

(i) If each $P(x)$ is contractible, then $\sum_{(x:A)} P(x)$ is equivalent to A.
(ii) If A is contractible with center a, then $\sum_{(x:A)} P(x)$ is equivalent to $P(a)$.

Proof. In the situation of (i), we show that $\mathsf{pr}_1 : \sum_{(x:A)} P(x) \to A$ is an equivalence. For quasi-inverse we define $g(x) :\equiv (x, c_x)$ where c_x is the center of $P(x)$. The composite $\mathsf{pr}_1 \circ g$ is obviously id_A, whereas the opposite composite is homotopic to the identity by using the contractions of each $P(x)$.

We leave the proof of (ii) to the reader (see Exercise 3.18). □

Another reason contractible types are interesting is that they extend the ladder of n-types mentioned in §3.1 downwards one more step.

Lemma 3.11.10. *A type A is a mere proposition if and only if for all $x, y : A$, the type $x =_A y$ is contractible.*

Proof. For "if", we simply observe that any contractible type is inhabited. For "only if", we observed in §3.3 that every mere proposition is a set, so that each type $x =_A y$ is a mere proposition. But it is also inhabited (since A is a mere proposition), and hence by Lemma 3.11.3(ii) it is contractible. □

Thus, contractible types may also be called (-2)**-types**. They are the bottom rung of the ladder of n-types, and will be the base case of the inductive definition of n-types in Chapter 7.

Notes

The fact that it is possible to define sets, mere propositions, and contractible types in type theory, with all higher homotopies automatically taken care of as in §§3.1, 3.3 and 3.11, was first observed by Voevodsky. In fact, he defined the entire hierarchy of n-types by induction, as we will do in Chapter 7.

Theorem 3.2.2 and Corollary 3.2.7 rely in essense on a classical theorem of Hedberg, which we will prove in Chapter 7. The implication that the propositions-as-types form of LEM contradicts univalence was observed by Martín Escardó on the AGDA mailing list. The proof we have given of Theorem 3.2.2 is due to Thierry Coquand.

The propositional truncation was introduced in the extensional type theory of NuPRL in 1983 by Constable [Con85] as an application of "subset" and "quotient" types. What is here called the

"propositional truncation" was called "squashing" in the NuPRL type theory [CAB+86]. Rules characterizing the propositional truncation directly, still in extensional type theory, were given in [AB04]. The intensional version in homotopy type theory was constructed by Voevodsky using an impredicative quantification, and later by Lumsdaine using higher inductive types (see §6.9).

Voevodsky [Voe12] has proposed resizing rules of the kind considered in §3.5. These are clearly related to the notorious *axiom of reducibility* proposed by Russell in his and Whitehead's *Principia Mathematica* [WR62].

The adverb "purely" as used to refer to untruncated logic is a reference to the use of monadic modalities to model effects in programming languages; see §7.7 and the Notes to Chapter 7.

The principle of unique choice fails in the category of so-called setoids in COQ [Spi11], in logic enriched type theory [AG02] and in minimal type theory [MS05]. The univalence axiom thus makes our type theory behave more like the internal logic of a topos; see also Chapter 10.

Martin-Löf [ML06] provides a discussion on the history of axioms of choice. For background on constructive logic see for instance [TvD88a, TvD88b].

Exercises

Exercise 3.1. Prove that if $A \simeq B$ and A is a set, then so is B.

Exercise 3.2. Prove that if A and B are sets, then so is $A + B$.

Exercise 3.3. Prove that if A is a set and $B : A \to \mathcal{U}$ is a type family such that $B(x)$ is a set for all $x : A$, then $\sum_{(x:A)} B(x)$ is a set.

Exercise 3.4. Show that A is a mere proposition if and only if $A \to A$ is contractible.

Exercise 3.5. Show that if A is a mere proposition, then so is $A + (\neg A)$. Thus, there is no need to insert a propositional truncation in (3.4.1).

Exercise 3.6. More generally, show that if A and B are mere propositions and $\neg(A \times B)$, then $A + B$ is also a mere proposition.

Exercise 3.7. Assuming that some type $\mathsf{isequiv}(f)$ satisfies conditions (i)–(iii) of §2.4, show that the type $\|\mathsf{qinv}(f)\|$ satisfies the same conditions and is equivalent to $\mathsf{isequiv}(f)$.

Exercise 3.8. Show that if LEM holds, then the type $\mathsf{Prop} :\equiv \sum_{(A:\mathcal{U})} \mathsf{isProp}(A)$ is equivalent to $\mathbf{2}$.

Exercise 3.9. Show that if $\mathcal{U}_{i+1}$ satisfies LEM, then the canonical inclusion $\mathsf{Prop}_{\mathcal{U}_i} \to \mathsf{Prop}_{\mathcal{U}_{i+1}}$ is an equivalence.

Exercise 3.10. Show that it is not the case that for all $A : \mathcal{U}$ we have $\|A\| \to A$. (However, there can be particular types for which $\|A\| \to A$. Exercise 3.7 implies that $\mathsf{qinv}(f)$ is such.)

Exercise 3.11. Show that if LEM holds, then for all $A : \mathcal{U}$ we have $\big\|(\|A\| \to A)\big\|$. (This property is a very simple form of the axiom of choice, which can fail in the absence of LEM; see [KECA13].)

Exercise 3.12. We showed in Corollary 3.2.7 that the following naive form of LEM is inconsistent with univalence:

$$\prod_{A:\mathcal{U}} (A + (\neg A))$$

In the absence of univalence, this axiom is consistent. However, show that it implies the axiom of choice (3.8.1).

Exercise 3.13. Show that assuming LEM, the double negation $\neg\neg A$ has the same universal property as the propositional truncation $\|A\|$, and is therefore equivalent to it. Thus, under LEM, the propositional truncation can be defined rather than taken as a separate type former.

Exercise 3.14. Show that if we assume propositional resizing as in §3.5, then the type

$$\prod_{P:\mathsf{Prop}} \Big((A \to P) \to P\Big)$$

has the same universal property as $\|A\|$. Thus, we can also define the propositional truncation in this case.

Exercise 3.15. Assuming LEM, show that double negation commutes with universal quantification of mere propositions over sets. That is, show that if X is a set and each $Y(x)$ is a mere proposition, then LEM implies

$$\Big(\prod_{x:X} \neg\neg Y(x)\Big) \simeq \Big(\neg\neg \prod_{x:X} Y(x)\Big). \tag{3.11.11}$$

Observe that if we assume instead that each $Y(x)$ is a set, then (3.11.11) becomes equivalent to the axiom of choice (3.8.3).

Exercise 3.16. Show that the rules for the propositional truncation given in §3.7 are sufficient to imply a dependent version of the induction principle: for any type family $B : \|A\| \to \mathcal{U}$ such that each $B(x)$ is a mere proposition, if for every $a : A$ we have $B(|a|)$, then for every $x : \|A\|$ we have $B(x)$.

Exercise 3.17. Show that the law of excluded middle (3.4.1) and the law of double negation (3.4.2) are logically equivalent.

Exercise 3.18. Prove Lemma 3.11.9(ii): if A is contractible with center a, then $\sum_{(x:A)} P(x)$ is equivalent to $P(a)$.

Chapter 4

Equivalences

We now study in more detail the notion of *equivalence of types* that was introduced briefly in §2.4. Specifically, we will give several different ways to define a type $\mathsf{isequiv}(f)$ having the properties mentioned there. Recall that we wanted $\mathsf{isequiv}(f)$ to have the following properties, which we restate here:

(i) $\mathsf{qinv}(f) \to \mathsf{isequiv}(f)$.
(ii) $\mathsf{isequiv}(f) \to \mathsf{qinv}(f)$.
(iii) $\mathsf{isequiv}(f)$ is a mere proposition.

Here $\mathsf{qinv}(f)$ denotes the type of quasi-inverses to f:

$$\sum_{g:B\to A} \big((f \circ g \sim \mathsf{id}_B) \times (g \circ f \sim \mathsf{id}_A)\big)$$

By function extensionality, it follows that $\mathsf{qinv}(f)$ is equivalent to the type

$$\sum_{g:B\to A} \big((f \circ g = \mathsf{id}_B) \times (g \circ f = \mathsf{id}_A)\big).$$

We will define three different types having properties (i)–(iii), which we call

- half adjoint equivalences,
- bi-invertible maps, and
- contractible functions.

We will also show that all these types are equivalent. These names are intentionally somewhat cumbersome, because after we know that they are all equivalent and have properties (i)–(iii), we will revert to saying simply "equivalence" without needing to specify which particular definition we choose. But for purposes of the comparisons in this chapter, we need different names for each definition.

Before we examine the different notions of equivalence, however, we give a little more explanation of why a different concept than quasi-invertibility is needed.

4.1 Quasi-inverses

We have said that $\mathsf{qinv}(f)$ is unsatisfactory because it is not a mere proposition, but we have given no evidence of that. In this section we exhibit a specific counterexample.

Lemma 4.1.1. *If $f : A \to B$ is such that $\mathsf{qinv}(f)$ is inhabited, then*

$$\mathsf{qinv}(f) \simeq (\prod_{x:A} (x = x)).$$

Proof. By assumption, f is an equivalence; that is, we have $e : \mathsf{isequiv}(f)$ and so $(f, e) : A \simeq B$. By univalence, $\mathsf{idtoeqv} : (A = B) \to (A \simeq B)$ is an equivalence, so we may assume that (f, e) is of the form $\mathsf{idtoeqv}(p)$ for some $p : A = B$. Then by path induction, we may assume p is refl_A, in which case $\mathsf{idtoeqv}(p)$ is id_A. Thus we are reduced to proving $\mathsf{qinv}(\mathsf{id}_A) \simeq (\prod_{(x:A)}(x = x))$. Now by definition we have

$$\mathsf{qinv}(\mathsf{id}_A) \equiv \sum_{g:A\to A} \big((g \sim \mathsf{id}_A) \times (g \sim \mathsf{id}_A)\big).$$

By function extensionality, this is equivalent to

$$\sum_{g:A\to A} \big((g = \mathsf{id}_A) \times (g = \mathsf{id}_A)\big)$$

and thus to

$$\sum_{h:\sum_{(g:A\to A)}(g=\mathsf{id}_A)} (\mathsf{pr}_1(h) = \mathsf{id}_A)$$

However, $\sum_{(g:A\to A)}(g = \mathsf{id}_A)$ is contractible with center id_A, so by Lemma 3.11.9 this type is equivalent to $(\mathsf{id}_A = \mathsf{id}_A)$. And by function extensionality, $(\mathsf{id}_A = \mathsf{id}_A)$ is equivalent to $\prod_{(x:A)}(x = x)$. □

We remark that the use of univalence in this proof is nonessential (see exercises).

Thus, what we need is some A which admits a nontrivial element of $\prod_{(x:A)}(x = x)$. Thinking of A as a higher groupoid, an inhabitant of $\prod_{(x:A)}(x = x)$ is a natural transformation from the identity functor of A to itself. Such transformations are said to form the **center** of a category, since the naturality axiom requires that they commute with all morphisms. Classically, if A is simply a group regarded as a one-object groupoid, then this yields precisely its center in the usual group-theoretic sense. This provides some motivation for the following.

Lemma 4.1.2. *Suppose we have a type A with $a : A$ and $q : a = a$ such that*

(i) The type $a = a$ is a set.
(ii) For all $x : A$ we have $\|a = x\|$.
(iii) For all $p : a = a$ we have $p \cdot q = q \cdot p$.

Then there exists $f : \prod_{(x:A)}(x = x)$ with $f(a) = q$.

Proof. Let $g : \prod_{(x:A)} \|a = x\|$ be as given by (ii). First we observe that each type $x =_A y$ is a set. For since being a set is a mere proposition, we may apply induction on truncation and assume that $g(x) = |p|$ and $g(y) = |q|$ for $p : a = x$ and $q : a = y$, in which case composing with p and q^{-1} yields an equivalence $(x = y) \simeq (a = a)$. But $(a = a)$ is a set by (i), so $(x = y)$ is also a set.

Now, we would like to define f by assigning to each x the path $g(x)^{-1} \cdot q \cdot g(x)$, but this does not work because $g(x)$ does not inhabit $a = x$ but rather $\|a = x\|$, and the type $(x = x)$ may not be a mere proposition, so we cannot use induction on propositional truncation. Instead we can apply the technique mentioned in §3.9: we characterize uniquely the object we wish to construct. Let us define, for each $x : A$, the type

$$B(x) :\equiv \sum_{(r:x=x)} \prod_{(s:a=x)} (r = s^{-1} \cdot q \cdot s).$$

We claim that $B(x)$ is a mere proposition for each $x : A$. Since this claim is itself a mere proposition, we may again apply induction on truncation and assume that $g(x) = |p|$ for some $p : a = x$. Now suppose given (r, h) and (r', h') in $B(x)$; then we have

$$h(p) \cdot h'(p)^{-1} : r = r'.$$

It remains to show that h is identified with h' when transported along this equality, which by transport in identity types and function types, reduces to showing

$$h(s) = h(p) \cdot h'(p)^{-1} \cdot h'(s)$$

for any $s : a = x$. But each side of this is an equality between elements of $(x = x)$, so it follows from our above observation that $(x = x)$ is a set.

Thus, each $B(x)$ is a mere proposition; we claim that $\prod_{(x:A)} B(x)$. Given $x : A$, we may now invoke the induction principle of propositional truncation to assume that $g(x) = |p|$ for $p : a = x$. We define $r :\equiv p^{-1} \cdot q \cdot p$; to inhabit $B(x)$ it remains to show that for any $s : a = x$ we have $r = s^{-1} \cdot q \cdot s$. Manipulating paths, this reduces to showing that $q \cdot (p \cdot s^{-1}) = (p \cdot s^{-1}) \cdot q$. But this is just an instance of (iii). □

Theorem 4.1.3. *There exist types A and B and a function $f : A \to B$ such that $\mathsf{qinv}(f)$ is not a mere proposition.*

Proof. It suffices to exhibit a type X such that $\prod_{(x:X)}(x = x)$ is not a mere proposition. Define $X :\equiv \sum_{(A:\mathcal{U})} \|\mathbf{2} = A\|$, as in the proof of Lemma 3.8.5. It will suffice to exhibit an $f : \prod_{(x:X)}(x = x)$ which is unequal to the function $x \mapsto \mathsf{refl}_x$.

Let $a :\equiv (\mathbf{2}, |\mathsf{refl}_{\mathbf{2}}|) : X$, and let $q : a = a$ be the path corresponding to the nonidentity equivalence $e : \mathbf{2} \simeq \mathbf{2}$ defined by $e(1_{\mathbf{2}}) :\equiv 0_{\mathbf{2}}$ and $e(0_{\mathbf{2}}) :\equiv 1_{\mathbf{2}}$. We would like to apply Lemma 4.1.2 to build an f. By definition of X, equalities in subset types, and univalence, we have $(a = a) \simeq (\mathbf{2} \simeq \mathbf{2})$, which is a set, so (i) holds. Similarly, by definition of X and equalities in subset types we have (ii). Finally, Exercise 2.9 implies that every equivalence $\mathbf{2} \simeq \mathbf{2}$ is equal to either $\mathsf{id}_{\mathbf{2}}$ or e, so we can show (iii) by a four-way case analysis. □

More generally, Lemma 4.1.2 implies that any "Eilenberg–Mac Lane space" $K(G,1)$, where G is an abelian group, will provide a counterexample; see Chapter 8. The type X we used turns out to be equivalent to $K(\mathbb{Z}/2,1)$. In Chapter 6 we will see that the circle $\mathbb{S}^1 = K(\mathbb{Z},1)$ is another easy-to-describe example.

We now move on to describing better notions of equivalence.

4.2 Half adjoint equivalences

In §4.1 we concluded that $\mathsf{qinv}(f)$ is equivalent to $\prod_{(x:A)}(x = x)$ by discarding a contractible type. Roughly, the type $\mathsf{qinv}(f)$ contains three data g, η, and ϵ, of which two (g and η) could together be seen to be contractible when f is an equivalence. The problem is that removing these data left one remaining (ϵ). In order to solve this problem, the idea is to add one *additional* datum which, together with ϵ, forms a contractible type.

Definition 4.2.1. A function $f : A \to B$ is a **half adjoint equivalence** if there are $g : B \to A$ and homotopies $\eta : g \circ f \sim \mathsf{id}_A$ and $\epsilon : f \circ g \sim \mathsf{id}_B$ such that there exists a path

$$\tau : \prod_{x:A} f(\eta x) = \epsilon(fx)$$

Thus we have a type $\mathsf{ishae}(f)$, defined to be

$$\sum_{(g:B\to A)} \sum_{(\eta:g\circ f\sim \mathsf{id}_A)} \sum_{(\epsilon:f\circ g\sim \mathsf{id}_B)} \prod_{x:A} f(\eta x) = \epsilon(fx).$$

Note that in the above definition, the coherence condition relating η and ϵ only involves f. We might consider instead an analogous coherence condition involving g:

$$\upsilon : \prod_{y:B} g(\epsilon y) = \eta(gy)$$

and a resulting analogous definition $\mathsf{ishae}'(f)$.

Fortunately, it turns out each of the conditions implies the other one:

Lemma 4.2.2. *For $f : A \to B$, $g : B \to A$, $\eta : g \circ f = \mathsf{id}_A$ and $\epsilon : f \circ g = \mathsf{id}_B$, the following conditions are logically equivalent:*

- $\prod_{(x:A)} f(\eta x) = \epsilon(fx)$
- $\prod_{(y:B)} g(\epsilon y) = \eta(gy)$

Proof. It suffices to show one direction; the other one is obtained by swapping A/B, f/g, and η/ϵ. Let $\tau : \prod_{(x:A)} f(\eta x) = \epsilon(fx)$. Fix $y : B$. Using naturality of ϵ and applying g, we get the following commuting diagram :

$$\begin{array}{ccc} gfgfgy & \xrightarrow{gfg(\epsilon y)} & gfgy \\ {\scriptstyle g(\epsilon(fgy))}\downarrow & & \downarrow{\scriptstyle g(\epsilon y)} \\ gfgy & \xrightarrow[g(\epsilon y)]{} & gy \end{array}$$

Using $\tau(gy)$ on the left side of the diagram gives us

$$\begin{array}{ccc} gfgfgy & \xrightarrow{gfg(\epsilon y)} & gfgy \\ {\scriptstyle gf(\eta(gy))}\downarrow & & \downarrow{\scriptstyle g(\epsilon y)} \\ gfgy & \xrightarrow[g(\epsilon y)]{} & gy \end{array}$$

Using the commutativity of η with $g \circ f$ (Corollary 2.4.5), we have

$$\begin{array}{ccc} gfgfgy & \xrightarrow{gfg(\epsilon y)} & gfgy \\ {\scriptstyle \eta(gfgy)}\downarrow & & \downarrow{\scriptstyle g(\epsilon y)} \\ gfgy & \xrightarrow[g(\epsilon y)]{} & gy \end{array}$$

However, by naturality of η we also have

$$\begin{array}{ccc} gfgfgy & \xrightarrow{gfg(\epsilon y)} & gfgy \\ {\scriptstyle \eta(gfgy)}\downarrow & & \downarrow{\scriptstyle \eta(gy)} \\ gfgy & \xrightarrow[g(\epsilon y)]{} & gy \end{array}$$

Thus, canceling all but the right-hand homotopy, we have $g(\epsilon y) = \eta(gy)$ as desired. □

However, it is important that we do not include *both* τ and v in the definition of $\mathsf{ishae}(f)$ (whence the name "*half* adjoint equivalence"). If we did, then after canceling contractible types we would still have one remaining datum — unless we added another higher coherence condition. In general, we expect to get a well-behaved type if we cut off after an odd number of coherences.

Of course, it is obvious that we have $\mathsf{ishae}(f) \to \mathsf{qinv}(f)$: simply forget the coherence datum. The other direction is a version of a standard argument from homotopy theory and category theory.

Theorem 4.2.3. *For any $f : A \to B$ we have $\mathsf{qinv}(f) \to \mathsf{ishae}(f)$.*

Proof. Suppose that (g, η, ε) is a quasi-inverse for f. We have to provide a quadruple $(g', \eta', \varepsilon', \tau)$ witnessing that f is a half adjoint equivalence. To define g' and η', we can just make the obvious choice by setting $g' :\equiv g$ and $\eta' :\equiv \eta$. However, in the definition of ε' we need start worrying about the construction of τ, so we cannot just follow our nose and take ε' to be ε. Instead, we take

$$\varepsilon'(b) :\equiv (\varepsilon(b) \cdot f(\eta(g(b)))) \cdot \varepsilon(f(g(b)))^{-1}$$

Now we need to find

$$\tau(a) : (\varepsilon(f(a)) \cdot f(\eta(g(f(a))))) \cdot \varepsilon(f(g(f(a))))^{-1} = f(\eta(a))$$

Note first that by Corollary 2.4.5, we have $\eta(g(f(a))) = g(f(\eta(a)))$. Therefore, we can apply Lemma 2.4.4 to compute

$$\begin{aligned}\varepsilon(f(a)) \cdot f(\eta(g(f(a)))) &= \varepsilon(f(a)) \cdot f(g(f(\eta(a))))\\ &= f(\eta(a)) \cdot \varepsilon(f(g(f(a))))\end{aligned}$$

from which we get the desired path $\tau(a)$. □

Combining this with Lemma 4.2.2 (or symmetrizing the proof), we also have $\mathsf{qinv}(f) \to \mathsf{ishae}'(f)$.

It remains to show that $\mathsf{ishae}(f)$ is a mere proposition. For this, we will need to know that the fibers of an equivalence are contractible.

Definition 4.2.4. The **fiber** of a map $f : A \to B$ over a point $y : B$ is

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

In homotopy theory, this is what would be called the *homotopy fiber* of f. The path lemmas in §2.5 yield the following characterization of paths in fibers:

Lemma 4.2.5. *For any $f : A \to B$, $y : B$, and $(x, p), (x', p') : \mathsf{fib}_f(y)$, we have*

$$\big((x, p) = (x', p')\big) \simeq \sum_{\gamma : x = x'} f(\gamma) \cdot p' = p$$

Theorem 4.2.6. *If $f : A \to B$ is a half adjoint equivalence, then for any $y : B$ the fiber $\mathsf{fib}_f(y)$ is contractible.*

Proof. Let $(g, \eta, \epsilon, \tau) : \mathsf{ishae}(f)$, and fix $y : B$. As our center of contraction for $\mathsf{fib}_f(y)$ we choose $(gy, \epsilon y)$. Now take any $(x, p) : \mathsf{fib}_f(y)$; we want to construct a path from $(gy, \epsilon y)$ to (x, p). By Lemma 4.2.5, it suffices to give a path $\gamma : gy = x$ such that $f(\gamma) \cdot p = \epsilon y$. We put $\gamma :\equiv g(p)^{-1} \cdot \eta x$. Then we have

$$\begin{aligned}f(\gamma) \cdot p &= fg(p)^{-1} \cdot f(\eta x) \cdot p\\ &= fg(p)^{-1} \cdot \epsilon(fx) \cdot p\\ &= \epsilon y\end{aligned}$$

where the second equality follows by τx and the third equality is naturality of ϵ. □

We now define the types which encapsulate contractible pairs of data. The following types put together the quasi-inverse g with one of the homotopies.

Definition 4.2.7. Let $f : A \to B$.

- Given a function $g : B \to A$, we define the types

$$\mathsf{linv}(f) :\equiv \sum_{g : B \to A} (g \circ f \sim \mathsf{id}_A)$$
$$\mathsf{rinv}(f) :\equiv \sum_{g : B \to A} (f \circ g \sim \mathsf{id}_B)$$

  of **left inverses** and **right inverses** to f, respectively. We call f **left invertible** if $\mathsf{linv}(f)$ is inhabited, and similarly **right invertible** if $\mathsf{rinv}(f)$ is inhabited.

Lemma 4.2.8. *If $f : A \to B$ has a quasi-inverse, then so do*

$$(f \circ -) : (C \to A) \to (C \to B)$$
$$(- \circ f) : (B \to C) \to (A \to C).$$

Proof. If g is a quasi-inverse of f, then $(g \circ -)$ and $(- \circ g)$ are quasi-inverses of $(f \circ -)$ and $(- \circ f)$ respectively. □

Lemma 4.2.9. *If $f : A \to B$ has a quasi-inverse, then the types $\mathsf{rinv}(f)$ and $\mathsf{linv}(f)$ are contractible.*

Proof. By function extensionality, we have

$$\mathsf{linv}(f) \simeq \sum_{g:B\to A} (g \circ f = \mathsf{id}_A).$$

But this is the fiber of $(- \circ f)$ over id_A, and so by Lemma 4.2.8 and Theorems 4.2.3 and 4.2.6, it is contractible. Similarly, $\mathsf{rinv}(f)$ is equivalent to the fiber of $(f \circ -)$ over id_B and hence contractible. □

Next we define the types which put together the other homotopy with the additional coherence datum.

Definition 4.2.10. For $f : A \to B$, a left inverse $(g, \eta) : \mathsf{linv}(f)$, and a right inverse $(g, \epsilon) : \mathsf{rinv}(f)$, we denote

$$\mathsf{lcoh}_f(g, \eta) :\equiv \sum_{(\epsilon : f\circ g \sim \mathsf{id}_B)} \prod_{(y:B)} g(\epsilon y) = \eta(gy)$$
$$\mathsf{rcoh}_f(g, \epsilon) :\equiv \sum_{(\eta : g\circ f \sim \mathsf{id}_A)} \prod_{(x:A)} f(\eta x) = \epsilon(fx)$$

Lemma 4.2.11. *For any f, g, ϵ, η, we have*

$$\mathsf{rcoh}_f(g, \eta) \simeq \prod_{y:B} (fgy, \eta(gy)) =_{\mathsf{fib}_g(gy)} (y, \mathsf{refl}_{gy})$$
$$\mathsf{lcoh}_f(g, \epsilon) \simeq \prod_{x:A} (gfx, \epsilon(fx)) =_{\mathsf{fib}_f(fx)} (x, \mathsf{refl}_{fx})$$

Proof. Using Lemma 4.2.5. □

Lemma 4.2.12. *If f is a half adjoint equivalence, then for any $(g, \epsilon) : \mathsf{rinv}(f)$, the type $\mathsf{rcoh}_f(g, \epsilon)$ is contractible.*

Proof. By Lemma 4.2.11 and the fact that dependent function types preserve contractible spaces, it suffices to show that for each $x : A$, the type $(fgx, \epsilon(fx)) =_{\mathsf{fib}_f(fx)} (x, \mathsf{refl}_{fx})$ is contractible. But by Theorem 4.2.6, $\mathsf{fib}_f(fx)$ is contractible, and any path space of a contractible space is itself contractible. □

Theorem 4.2.13. *For any $f : A \to B$, the type $\mathsf{ishae}(f)$ is a mere proposition.*

Proof. By Lemma 3.11.3 it suffices to assume f to be a half adjoint equivalence and show that $\mathsf{ishae}(f)$ is contractible. Now by rearranging Σs, the type $\mathsf{ishae}(f)$ is equivalent to $\sum_{(u:\mathsf{rinv}(f))} \mathsf{rcoh}_f(\mathsf{pr}_1(u), \mathsf{pr}_2(u))$. But by Lemmas 4.2.9 and 4.2.12 and the fact that Σ preserves contractibility, the latter type is also contractible. □

Thus, we have shown that $\mathsf{ishae}(f)$ has all three desiderata for the type $\mathsf{isequiv}(f)$. In the next two sections we consider a couple of other possibilities.

4.3 Bi-invertible maps

Using the language introduced in §4.2, we can restate the definition proposed in §2.4 as follows.

Definition 4.3.1. We say $f : A \to B$ is **bi-invertible** if it has both a left inverse and a right inverse:

$$\mathsf{biinv}(f) :\equiv \mathsf{linv}(f) \times \mathsf{rinv}(f).$$

In §2.4 we proved that $\mathsf{qinv}(f) \to \mathsf{biinv}(f)$ and $\mathsf{biinv}(f) \to \mathsf{qinv}(f)$. What remains is the following.

Theorem 4.3.2. *For any $f : A \to B$, the type $\mathsf{biinv}(f)$ is a mere proposition.*

Proof. We may suppose f to be bi-invertible and show that $\mathsf{biinv}(f)$ is contractible. But since $\mathsf{biinv}(f) \to \mathsf{qinv}(f) \to \mathsf{ishae}(f)$, by Lemma 4.2.9 in this case both $\mathsf{linv}(f)$ and $\mathsf{rinv}(f)$ are contractible. It is easy to show that the product of contractible types is contractible. □

Note that this also fits the proposal made at the beginning of §4.2: we combine g and η into a contractible type and add an additional datum which combines with ϵ into a contractible type. The difference is that instead of adding a *higher* datum (a 2-dimensional path) to combine with ϵ, we add a *lower* one (a right inverse that is separate from the left inverse).

Corollary 4.3.3. *For any $f : A \to B$ we have $\mathsf{biinv}(f) \simeq \mathsf{ishae}(f)$.*

Proof. We have $\mathsf{biinv}(f) \to \mathsf{qinv}(f) \to \mathsf{ishae}(f)$ and $\mathsf{ishae}(f) \to \mathsf{qinv}(f) \to \mathsf{biinv}(f)$. Since both $\mathsf{isContr}(f)$ and $\mathsf{biinv}(f)$ are mere propositions, the equivalence follows from Lemma 3.3.3. □

4.4 Contractible fibers

Note that our proofs about $\mathsf{ishae}(f)$ and $\mathsf{biinv}(f)$ made essential use of the fact that the fibers of an equivalence are contractible. In fact, it turns out that this property is itself a sufficient definition of equivalence.

Definition 4.4.1 (Contractible maps)**.** A map $f : A \to B$ is **contractible** if for all $y : B$, the fiber $\mathsf{fib}_f(y)$ is contractible.

Thus, the type $\mathsf{isContr}(f)$ is defined to be

$$\mathsf{isContr}(f) :\equiv \prod_{y:B} \mathsf{isContr}(\mathsf{fib}_f(y)) \tag{4.4.2}$$

$$:\equiv \prod_{y:B} \mathsf{isContr}(\{\, x : A \mid f(x) = y \,\}). \tag{4.4.3}$$

Note that in §3.11 we defined what it means for a *type* to be contractible. Here we are defining what it means for a *map* to be contractible. Our terminology follows the general homotopy-theoretic practice of saying that a map has a certain property if all of its (homotopy) fibers have that property. Thus, a type A is contractible just when the map $A \to \mathbf{1}$ is contractible. From Chapter 7 onwards we will call contractible maps and types (-2)-*truncated*.

We have already shown in Theorem 4.2.6 that $\mathsf{ishae}(f) \to \mathsf{isContr}(f)$. Conversely:

Theorem 4.4.4. *For any $f : A \to B$ we have $\mathsf{isContr}(f) \to \mathsf{ishae}(f)$.*

Proof. Let $P : \mathsf{isContr}(f)$. We define an inverse mapping $g : B \to A$ by sending each $y : B$ to the center of contraction of the h-fiber at y:

$$g(y) :\equiv \mathsf{pr}_1(\mathsf{pr}_1(Py))$$

We can thus define the transformation ϵ by mapping y to the witness that $g(y)$ indeed belongs to the h-fiber at y:

$$\epsilon(y) :\equiv \mathsf{pr}_2(\mathsf{pr}_1(Py))$$

It remains to define η and τ. This of course amounts to giving an element of $\mathsf{Icoh}_f(g, \epsilon)$. By Lemma 4.2.11, this is the same as giving for each $x : A$ a path from (x, refl_{fx}) to $(gfx, \epsilon(fx))$ in the fiber at fx. But this is easy: for any $x : A$, the type $\mathsf{fib}_f(fx)$ is contractible by assumption, hence such a path must exist. We can construct it explicitly as

$$\big(P(fx)\ (x, \mathsf{refl}_{fx})\big)^{-1} \cdot \big(P(fx)\ (gfx, \epsilon(fx))\big). \qquad \square$$

It is also easy to see:

Lemma 4.4.5. *For any f, the type $\mathsf{isContr}(f)$ is a mere proposition.*

Proof. By Lemma 3.11.4, each type $\mathsf{isContr}(\mathsf{fib}_f(y))$ is a mere proposition. Thus, by Example 3.6.2, so is (4.4.3). $\square$

Theorem 4.4.6. *For any $f : A \to B$ we have $\mathsf{isContr}(f) \simeq \mathsf{ishae}(f)$.*

Proof. We have already established a logical equivalence $\mathsf{isContr}(f) \leftrightarrow \mathsf{ishae}(f)$, and both are mere propositions (Lemma 4.4.5 and Theorem 4.2.13). Thus, Lemma 3.3.3 applies. $\square$

Usually, we prove that a function is an equivalence by exhibiting a quasi-inverse, but sometimes this definition is more convenient. For instance, it implies that when proving a function to be an equivalence, we are free to assume that its codomain is inhabited.

Corollary 4.4.7. *If $f : A \to B$ is such that $B \to \mathsf{isequiv}(f)$, then f is an equivalence.*

Proof. To show f is an equivalence, it suffices to show that $\mathsf{fib}_f(y)$ is contractible for any $y : B$. But if $e : B \to \mathsf{isequiv}(f)$, then given any such y we have $e(y) : \mathsf{isequiv}(f)$, so that f is an equivalence and hence $\mathsf{fib}_f(y)$ is contractible, as desired. $\square$

4.5 On the definition of equivalences

We have shown that all three definitions of equivalence satisfy the three desirable properties and are pairwise equivalent:

$$\mathsf{isContr}(f) \simeq \mathsf{ishae}(f) \simeq \mathsf{biinv}(f).$$

(There are yet more possible definitions of equivalence, but we will stop with these three. See Exercise 3.7 and the exercises in this chapter for some more.) Thus, we may choose any one of them as "the" definition of $\mathsf{isequiv}(f)$. For definiteness, we choose to define

$$\mathsf{isequiv}(f) :\equiv \mathsf{ishae}(f).$$

This choice is advantageous for formalization, since $\mathsf{ishae}(f)$ contains the most directly useful data. On the other hand, for other purposes, $\mathsf{biinv}(f)$ is ofter easier to deal with, since it contains no 2-dimensional paths and its two symmetrical halves can be treated independently. However, for purposes of this book, the specific choice will make little difference.

In the rest of this chapter, we study some other properties and characterizations of equivalences.

4.6 Surjections and embeddings

When A and B are sets and $f : A \to B$ is an equivalence, we also call it as **isomorphism** or a **bijection**. (We avoid these words for types that are not sets, since in homotopy theory and higher category theory they often denote a stricter notion of "sameness" than homotopy equivalence.) In set theory, a function is a bijection just when it is both injective and surjective. The same is true in type theory, if we formulate these conditions appropriately. For clarity, when dealing with types that are not sets, we will speak of *embeddings* instead of injections.

Definition 4.6.1. Let $f : A \to B$.

(i) We say f is **surjective** if for every $b : B$ we have $\left\|\mathsf{fib}_f(b)\right\|$. In other words, every fiber of f is merely inhabited.

(ii) We say f is a **embedding** if for every $x, y : A$ the function $\mathsf{ap}_f : (x =_A y) \to (f(x) =_B f(y))$ is an equivalence.

Note that if A and B are sets, then by Lemma 3.3.3, f is an embedding just when

$$\prod_{x,y:A} (f(x) =_B f(y)) \to (x =_A y). \tag{4.6.2}$$

In this case we say that f is **injective**. We avoid that word for types that are not sets, because it might be interpreted as (4.6.2), which is an ill-behaved notion for non-sets. It is also true that any function between sets is surjective if and only if it is an *epimorphism* in a suitable sense, but this also fails for more general types, and surjectivity is generally the more important notion.

Theorem 4.6.3. *A function $f : A \to B$ is an equivalence if and only if it is both surjective and an embedding.*

Proof. If f is an equivalence, then each $\mathsf{fib}_f(b)$ is contractible, hence so is $\left\|\mathsf{fib}_f(b)\right\|$, so f is surjective. And we showed in Theorem 2.11.1 that any equivalence is an embedding.

Conversely, suppose f is a surjective embedding. Let $b : B$; we show that $\sum_{(x:A)}(f(x) = b)$ is contractible. Since f is surjective, there merely exists an $a : A$ such that $f(a) = b$. Thus, the fiber of f over b is inhabited; it remains to show it is a mere proposition. For this, suppose given $x, y : A$ with $p : f(x) = b$ and $q : f(y) = b$. Then since ap_f is an equivalence, there exists $r : x = y$ with $\mathsf{ap}_f(r) = p \cdot q^{-1}$. However, using the characterization of paths in Σ-types, the latter equality rearranges to $r_*(p) = q$. Thus, together with r it exhibits $(x, p) = (y, q)$ in the fiber of f over b. □

Corollary 4.6.4. *For any $f : A \to B$ we have*

$$\mathsf{isequiv}(f) \simeq (\mathsf{isEmbedding}(f) \times \mathsf{isSurjective}(f)).$$

Proof. Being a surjection and an embedding are both mere propositions; now apply Lemma 3.3.3. □

Of course, this cannot be used as a definition of "equivalence", since the definition of embeddings refers to equivalences. However, this characterization can still be useful; see §8.8. We will generalize it in Chapter 7.

4.7 Closure properties of equivalences

We have already seen in Lemma 2.4.12 that equivalences are closed under composition. Furthermore, we have:

Theorem 4.7.1 (The 2-out-of-3 property)**.** *Suppose $f : A \to B$ and $g : B \to C$. If any two of f, g, and $g \circ f$ are equivalences, so is the third.*

Sketch of proof. If $g \circ f$ and g are equivalences, then $(g \circ f)^{-1} \circ g$ is a quasi-inverse to f. Similarly, if $g \circ f$ and f are equivalences, then $f \circ (g \circ f)^{-1}$ is a quasi-inverse to g. □

This is a standard closure condition on equivalences from homotopy theory. Also well-known is that they are closed under retracts, in the following sense.

Definition 4.7.2. A function $g : A \to B$ is said to be a **retract** of a function $f : X \to Y$ if there is a diagram

$$\begin{array}{ccccc} A & \xrightarrow{s} & X & \xrightarrow{r} & A \\ {\scriptstyle g}\downarrow & & {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\ B & \xrightarrow[s']{} & Y & \xrightarrow[r']{} & B \end{array}$$

for which there are

(i) a homotopy $R : r \circ s \sim \mathsf{id}_A$.

(ii) a homotopy $R' : r' \circ s' \sim \mathsf{id}_B$.
(iii) a homotopy $L : f \circ s \sim s' \circ g$.
(iv) a homotopy $K : g \circ r \sim r' \circ f$.
(v) for every $a : A$, a path $H(a)$ witnessing the commutativity of the square

$$\begin{array}{ccc} g(r(s(a))) & \xrightarrow{K(s(a))} & r'(f(s(a))) \\ {\scriptstyle g(R(a))}\downarrow & & \downarrow{\scriptstyle r'(L(a))} \\ g(a) & \xrightarrow[R'(g(a))]{} & r'(s'(g(a))) \end{array}$$

Recall that in §3.11 we defined what it means for a type to be a retract of another. This is a special case of the above definition where B and Y are $\mathbf{1}$. Conversely, just as with contractibility, retractions of maps induce retractions of their fibers.

Lemma 4.7.3. *If a function $g : A \to B$ is a retract of a function $f : X \to Y$, then $\mathsf{fib}_g(b)$ is a retract of $\mathsf{fib}_f(s'(b))$ for every $b : B$, where $s' : B \to Y$ is as in Definition 4.7.2.*

Proof. Suppose that $g : A \to B$ is a retract of $f : X \to Y$. Then for any $b : B$ we have the functions

$$\begin{aligned} \varphi_b &: \mathsf{fib}_g(b) \to \mathsf{fib}_f(s'(b)) \\ \varphi_b(a,p) &:\equiv (s(a), s'(p) \cdot L(a)) \\ \psi_b &: \mathsf{fib}_f(s'(b)) \to \mathsf{fib}_g(b) \\ \psi_b(x,q) &:\equiv (r(x), R'(b) \cdot r'(q) \cdot K(x)) \end{aligned}$$

Then we have $\psi_b(\varphi_b(a,p)) \equiv (r(s(a)), R'(b) \cdot r'(s'(p) \cdot L(a)) \cdot K(s(a)))$. We claim ψ_b is a retraction with section φ_b for all $b : B$, which is to say that for all $(a,p) : \mathsf{fib}_g(b)$ we have $\psi_b(\varphi_b(a,p)) = (a,p)$. In other words, we want to show

$$\prod_{(b:B)} \prod_{(a:A)} \prod_{(p:f(a)=b)} \psi_b(\varphi_b(a,p)) = (a,p).$$

By reordering the first two Πs and applying a version of Lemma 3.11.9, this is equivalent to

$$\prod_{a:A} \psi_{f(a)}(\varphi_{f(a)}(a, \mathsf{refl}_{g(a)})) = (a, \mathsf{refl}_{g(a)})$$

By assumption, we have $R(a) : r(s(a)) = a$. So it is left to show that there is a path

$$R(a)_*\big(R'(g(a)) \cdot r'(L(a)) \cdot K(s(a))\big) = \mathsf{refl}_{g(a)}.$$

But this transportation computes as $R'(g(a)) \cdot r'(L(a)) \cdot K(s(a)) \cdot g(R(a))$, so the required path is given by $H(a)$. □

Theorem 4.7.4. *If g is a retract of an equivalence f, then g is also an equivalence.*

Proof. By Lemma 4.7.3, every fiber of g is a retract of a fiber of f. Thus, by Lemma 3.11.7, if the latter are all contractible, so are the former. □

Finally, we show that fiberwise equivalences can be characterized in terms of equivalences of total spaces. To explain the terminology, recall from §2.3 that a type family $P : A \to \mathcal{U}$ can be viewed as a fibration over A with total space $\sum_{(x:A)} P(x)$, the fibration being is the projection $\mathsf{pr}_1 : \sum_{(x:A)} P(x) \to A$. From this point of view, given two type families $P, Q : A \to \mathcal{U}$, we may refer to a function $f : \prod_{(x:A)}(P(x) \to Q(x))$ as a *fiberwise map* or a *fiberwise transformation*. Such a map induces a function on total spaces:

Definition 4.7.5. Given type families $P, Q : A \to \mathcal{U}$ and a map $f : \prod_{(x:A)} P(x) \to Q(x)$, we define

$$\mathsf{total}(f) :\equiv \lambda w.\, (\mathsf{pr}_1 w, f(\mathsf{pr}_1 w, \mathsf{pr}_2 w)) : \sum_{x:A} P(x) \to \sum_{x:A} Q(x).$$

Theorem 4.7.6. *Suppose that f is a fiberwise transformation between families P and Q over a type A and let $x : A$ and $v : Q(x)$. Then we have an equivalence*

$$\mathsf{fib}_{\mathsf{total}(f)}((x, v)) \simeq \mathsf{fib}_{f(x)}(v).$$

Proof. We calculate:

$$\begin{aligned}
\mathsf{fib}_{\mathsf{total}(f)}((x,v)) &\equiv \sum_{w:\sum_{(x:A)} P(x)} (\mathsf{pr}_1 w, f(\mathsf{pr}_1 w, \mathsf{pr}_2 w)) = (x, v) \\
&\simeq \sum_{(a:A)} \sum_{(u:P(a))} (a, f(a,u)) = (x,v) && \text{by (2.15.9)} \\
&\simeq \sum_{(a:A)} \sum_{(u:P(a))} \sum_{(p:a=x)} p_*(f(a,u)) = v && \text{by Theorem 2.7.2} \\
&\simeq \sum_{(a:A)} \sum_{(p:a=x)} \sum_{(u:P(a))} p_*(f(a,u)) = v \\
&\simeq \sum_{u:P(x)} f(x,u) = v && \text{by (2.15.10)} \\
&\equiv \mathsf{fib}_{f(x)}(v) && \square
\end{aligned}$$

Theorem 4.7.7. *Suppose that f is a fiberwise transformation between families P and Q over a type A. Then $\mathsf{total}(f)$ is an equivalence if and only if each $f(x) : P(x) \to Q(x)$ is an equivalence.*

Proof. Let f, P, Q and A be as in the statement of the theorem. By Theorem 4.7.6 it follows for all $x : A$ and $v : Q(x)$ that $\mathsf{fib}_{\mathsf{total}(f)}((x,v))$ is contractible if and only if $\mathsf{fib}_{f(x)}(v)$ is contractible. Thus, $\mathsf{fib}_{\mathsf{total}(f)}(w)$ is contractible for all $w : \sum_{(x:A)} Q(x)$ if and only if $\mathsf{fib}_{f(x)}(v)$ is contractible for all $x : A$ and $v : Q(x)$. □

4.8 The object classifier

In type theory we have a basic notion of *family of types*, namely a function $B : A \to \mathcal{U}$. We have seen that such families behave somewhat like *fibrations* in homotopy theory, with the fibration

being the projection $\mathsf{pr}_1 : \sum_{(a:A)} B(a) \to A$. A basic fact in homotopy theory is that every map is equivalent to a fibration. With univalence at our disposal, we can prove the same thing in type theory.

Lemma 4.8.1. *For any type family $B : A \to \mathcal{U}$, the fiber of $\mathsf{pr}_1 : \sum_{(x:A)} B(x) \to A$ over $a : A$ is equivalent to $B(a)$:*

$$\mathsf{fib}_{\mathsf{pr}_1}(a) \simeq B(a)$$

Proof. We have

$$\begin{aligned}
\mathsf{fib}_{\mathsf{pr}_1}(a) &:\equiv \sum_{u:\sum_{(x:A)} B(x)} \mathsf{pr}_1(u) = a \\
&\simeq \sum_{(x:A)} \sum_{(b:B(x))} (x = a) \\
&\simeq \sum_{(x:A)} \sum_{(p:x=a)} B(x) \\
&\simeq B(a)
\end{aligned}$$

using the left universal property of identity types. □

Lemma 4.8.2. *For any function $f : A \to B$, we have $A \simeq \sum_{(b:B)} \mathsf{fib}_f(b)$.*

Proof. We have

$$\begin{aligned}
\sum_{b:B} \mathsf{fib}_f(b) &:\equiv \sum_{(b:B)} \sum_{(a:A)} (f(a) = b) \\
&\simeq \sum_{(a:A)} \sum_{(b:B)} (f(a) = b) \\
&\simeq A
\end{aligned}$$

using the fact that $\sum_{(b:B)} (f(a) = b)$ is contractible. □

Theorem 4.8.3. *For any type B there is an equivalence*

$$\chi : \left(\sum_{A:\mathcal{U}} (A \to B) \right) \simeq (B \to \mathcal{U}).$$

Proof. We have to construct quasi-inverses

$$\begin{aligned}
\chi &: \big(\sum_{A:\mathcal{U}} (A \to B) \big) \to B \to \mathcal{U} \\
\psi &: (B \to \mathcal{U}) \to \big(\sum_{A:\mathcal{U}} (A \to B) \big).
\end{aligned}$$

We define χ by $\chi(f, b) :\equiv \mathsf{fib}_f(b)$, and ψ by $\psi(P) :\equiv ((\sum_{(b:B)} P(b)), \mathsf{pr}_1)$. Now we have to verify that $\chi \circ \psi \sim \mathsf{id}$ and that $\psi \circ \chi \sim \mathsf{id}$.

(i) Let $P : B \to \mathcal{U}$. By Lemma 4.8.1, $\mathsf{fib}_{\mathsf{pr}_1}(b) \simeq P(b)$ for any $b : B$, so it follows immediately that $P \sim \chi(\psi(P))$.

(ii) Let $f : A \to B$ be a function. We have to find a path

$$(\textstyle\sum_{(b:B)} \mathsf{fib}_f(b),\ \mathsf{pr}_1) = (A, f)$$

First note that by Lemma 4.8.2, we have $e : \sum_{(b:B)} \mathsf{fib}_f(b) \simeq A$ with $e(b, a, p) :\equiv a$ and $e^{-1}(a) :\equiv (f(a), a, \mathsf{refl}_{f(a)})$. It also follows that $e \cdot \mathsf{pr}_1 = \mathsf{pr}_1 \circ e^{-1}$. From this, we immediately read off that $(e \cdot \mathsf{pr}_1)(a) = f(a)$ for each $a : A$. □

In particular, this implies that we have an *object classifier* in the sense of higher topos theory.

Definition 4.8.4. Define

$$\mathcal{U}_\bullet :\equiv \sum_{A:\mathcal{U}} A$$

Thus, $\mathcal{U}_\bullet$ stands for the *pointed types*.

Theorem 4.8.5. *Let $f : A \to B$ be a function. Then the diagram*

$$\begin{array}{ccc} A & \xrightarrow{\vartheta_f} & \mathcal{U}_\bullet \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathsf{pr}_1} \\ B & \xrightarrow[\chi_f]{} & \mathcal{U} \end{array}$$

is a pullback diagram. Here, the function ϑ_f is defined by

$$\lambda a.\, (\mathsf{fib}_f(f(a)), (a, \mathsf{refl}_{f(a)})).$$

Proof. Note that we have the equivalences

$$\begin{aligned} A &\simeq \sum_{b:B} \mathsf{fib}_f(b) \\ &\simeq \sum_{(b:B)} \sum_{(X:\mathcal{U})} \sum_{(p:\mathsf{fib}_f(b)=X)} X \\ &\simeq \sum_{(b:B)} \sum_{(X:\mathcal{U})} \sum_{(x:X)} \mathsf{fib}_f(b) = X \\ &\equiv B \times_{\mathcal{U}} \mathcal{U}_\bullet. \end{aligned}$$

which gives us a composite equivalence $e : A \simeq B \times_{\mathcal{U}} \mathcal{U}_\bullet$. We may display the action of this composite equivalence step by step by

$$\begin{aligned} a &\mapsto (f(a), (a, \mathsf{refl}_{f(a)})) \\ &\mapsto (f(a), \mathsf{fib}_f(f(a)), \mathsf{refl}_{\mathsf{fib}_f(f(a))}, (a, \mathsf{refl}_{f(a)})) \\ &\mapsto (f(a), \mathsf{fib}_f(f(a)), (a, \mathsf{refl}_{f(a)}), \mathsf{refl}_{\mathsf{fib}_f(f(a))}) \end{aligned}$$

Therefore, we get homotopies $f \sim \mathsf{pr}_1 \circ e$ and $\vartheta_f \sim \mathsf{pr}_2 \circ e$. □

4.9 Univalence implies function extensionality

In the last section of this chapter we include a proof that the univalence axiom implies function extensionality. Thus, in this section we work *without* either the univalence axiom or function extensionality. The proof consists of two steps. First we show in Theorem 4.9.4 that the univalence axiom implies a weak form of function extensionality, defined in Definition 4.9.1 below. The principle of weak function extensionality in turn implies the usual function extensionality, and it does so without the univalence axiom (Theorem 4.9.5).

In this section, we assume that $\mathcal{U}$ is a universe. We will explicitly indicate where we assume that it is univalent.

Definition 4.9.1. The **weak function extensionality principle** asserts that there is a function

$$\left(\prod_{(x:A)} \mathsf{isContr}(P(x))\right) \to \mathsf{isContr}\left(\prod_{(x:A)} P(x)\right)$$

for any family $P : A \to \mathcal{U}$ of types over any type A.

Lemma 4.9.2. *Assuming $\mathcal{U}$ is univalent, for any $A, B, X : \mathcal{U}$ and any $e : A \simeq B$, there is an equivalence*

$$(X \to A) \simeq (X \to B)$$

of which the underlying map is given by postcomposition with the underlying function of e.

Proof. As in the proof of Lemma 4.1.1, we may assume that $e = \mathsf{idtoeqv}(p)$ for some $p : A = B$. Then by path induction, we may assume p is refl_A, so that $e = \mathsf{id}_A$. But in this case, postcomposition with e is the identity, hence an equivalence. □

Corollary 4.9.3. *Let $P : A \to \mathcal{U}$ be a family of contractible types, i.e. $\prod_{(x:A)} \mathsf{isContr}(P(x))$. Then the projection $\mathsf{pr}_1 : (\sum_{(x:A)} P(x)) \to A$ is an equivalence. Assuming $\mathcal{U}$ is univalent, it follows immediately that precomposition with pr_1 gives an equivalence*

$$\alpha : \left(A \to \sum_{(x:A)} P(x)\right) \simeq (A \to A).$$

Proof. In Exercise 4.4 it is asked to show that for $\mathsf{pr}_1 : \sum_{(x:A)} P(X) \to A$ and $x : A$ we have an equivalence

$$\mathsf{fib}_{\mathsf{pr}_1}(x) \simeq P(x).$$

Therefore pr_1 is an equivalence whenever each $P(x)$ is contractible. The assertion is now a consequence of Lemma 4.9.2. □

In particular, the homotopy fiber of the above equivalence at id_A is contractible. Therefore, we can show that univalence implies weak function extensionality by showing that the dependent function type $\prod_{(x:A)} P(x)$ is a retract of $\mathsf{fib}_\alpha(\mathsf{id}_A)$.

Theorem 4.9.4. *In a univalent universe $\mathcal{U}$, suppose that $P : A \to \mathcal{U}$ is a family of contractible types and let α be the function of Corollary 4.9.3. Then $\prod_{(x:A)} P(x)$ is a retract of $\mathsf{fib}_\alpha(\mathsf{id}_A)$. As a consequence, $\prod_{(x:A)} P(x)$ is contractible. In other words, the univalence axiom implies the weak function extensionality principle.*

Proof. Define the functions

$$\begin{aligned} \varphi &: \textstyle\prod_{(x:A)} P(x) \to \mathsf{fib}_\alpha(\mathsf{id}_A) \\ \varphi(f) &:\equiv (\lambda x.\, (x, f(x)), \mathsf{refl}_{\mathsf{id}_A}) \end{aligned}$$

and

$$\begin{aligned} \psi &: \mathsf{fib}_\alpha(\mathsf{id}_A) \to \textstyle\prod_{(x:A)} P(x) \\ \psi(g, p) &:\equiv \lambda x.\, p_*(\mathsf{pr}_2(g(x))) \end{aligned}$$

Then $\psi(\varphi(f)) = \lambda x.\, f(x)$, which is f by the η-rule for dependent function types. □

We now show that weak function extensionality implies the usual function extensionality. Recall from (2.9.2) the function $\mathsf{happly}(f, g) : (f = g) \to (f \sim g)$ which converts equality of functions to homotopy. In the proof that follows, the univalence axiom is not used.

Theorem 4.9.5. *Weak function extensionality implies the function extensionality Axiom 2.9.3.*

Proof. We want to show that

$$\prod_{(A:\mathcal{U})} \prod_{(P:A\to\mathcal{U})} \prod_{(f,g:\prod_{(x:A)} P(x))} \mathsf{isequiv}(\mathsf{happly}(f, g)).$$

Since a fiberwise map induces an equivalence on total spaces if and only if it is fiberwise an equivalence by Theorem 4.7.7, it suffices to show that the function of type

$$\left(\sum_{g:\prod_{(x:A)} P(x)} (f = g) \right) \to \sum_{g:\prod_{(x:A)} P(x)} (f \sim g)$$

induced by $\lambda(g : \prod_{(x:A)} P(x)).\, \mathsf{happly}(f, g)$ is an equivalence. Since the type on the left is contractible by Lemma 3.11.8, it suffices to show that the type on the right:

$$\sum_{(g:\prod_{(x:A)} P(x))} \prod_{(x:A)} f(x) = g(x) \tag{4.9.6}$$

is contractible. Now Theorem 2.15.7 says that this is equivalent to

$$\prod_{(x:A)} \sum_{(u:P(x))} f(x) = u. \tag{4.9.7}$$

The proof of Theorem 2.15.7 uses function extensionality, but only for one of the composites. Thus, without assuming function extensionality, we can conclude that (4.9.6) is a retract of (4.9.7). And (4.9.7) is a product of contractible types, which is contractible by the weak function extensionality principle; hence (4.9.6) is also contractible. □

Notes

The fact that the space of continuous maps equipped with quasi-inverses has the wrong homotopy type to be the "space of homotopy equivalences" is well-known in algebraic topology. In that context, the "space of homotopy equivalences" $(A \simeq B)$ is usually defined simply as the subspace of the function space $(A \to B)$ consisting of the functions that are homotopy equivalences. In type theory, this would correspond most closely to $\sum_{(f:A\to B)} \|\mathsf{qinv}(f)\|$; see Exercise 3.7.

The first definition of equivalence given in homotopy type theory was the one that we have called $\mathsf{isContr}(f)$, which was due to Voevodsky. The possibility of the other definitions was subsequently observed by various people. The basic theorems about adjoint equivalences such as Lemma 4.2.2 and Theorem 4.2.3 are adaptations of standard facts in higher category theory and homotopy theory. Using bi-invertibility as a definition of equivalences was suggested by André Joyal.

The properties of equivalences discussed in §§4.6 and 4.7 are well-known in homotopy theory. Most of them were first proven in type theory by Voevodsky.

The fact that every function is equivalent to a fibration is a standard fact in homotopy theory. The notion of object classifier in $(\infty, 1)$-category theory (the categorical analogue of Theorem 4.8.3) is due to Rezk (see [Rez05, Lur09]).

Finally, the fact that univalence implies function extensionality (§4.9) is due to Voevodsky. Our proof is a simplification of his.

Exercises

Exercise 4.1. Consider the type of "two-sided adjoint equivalence data" for $f : A \to B$,

$$\sum_{(g:B\to A)} \sum_{(\eta:g\circ f\sim \mathsf{id}_A)} \sum_{(\epsilon:f\circ g\sim \mathsf{id}_B)} \Big(\prod_{x:A} f(\eta x) = \epsilon(fx)\Big) \times \Big(\prod_{y:B} g(\epsilon y) = \eta(gy)\Big).$$

By Lemma 4.2.2, we know that if f is an equivalence, then this type is inhabited. Give a characterization of this type analogous to Lemma 4.1.1.

Can you give an example showing that this type is not generally a mere proposition? (This will be easier after Chapter 6.)

Exercise 4.2. Show that for any $f : A \to B$, the following type also satisfies the three desiderata of $\mathsf{isequiv}(f)$:

$$\sum_{R:A\to B\to \mathcal{U}} \Big(\prod_{a:A} \mathsf{isContr}\big(\sum_{b:B} R(a,b)\big)\Big) \times \Big(\prod_{b:B} \mathsf{isContr}\big(\sum_{a:A} R(a,b)\big)\Big).$$

Exercise 4.3. Reformulate the proof of Lemma 4.1.1 without using univalence.

Exercise 4.4. Let P be a family of types over A, and consider the function $\mathsf{pr}_1 : \sum_{(x:A)} P(x) \to A$. Show that there is an equivalence

$$\mathsf{fib}_{\mathsf{pr}_1}(x) \simeq P(x)$$

for each $x : A$. Conclude that pr_1 is an equivalence if and only if each type $P(x)$ is contractible.

Exercise 4.5. Prove that equivalences satisfy the *2-out-of-6 property*: given $f : A \to B$ and $g : B \to C$ and $h : C \to D$, if $g \circ f$ and $h \circ g$ are equivalences, so are f, g, h, and $h \circ g \circ f$.

Chapter 5

Induction

In Chapter 1, we introduced many ways to form new types from old ones. Except for (dependent) function types and universes, all these rules are special cases of the general notion of *inductive definition*. In this chapter we study inductive definitions more generally.

5.1 Introduction to inductive types

An *inductive type* X can be intuitively understood as a type "freely generated" by a certain finite collection of *constructors*, each of which is a function (of some number of arguments) with codomain X. This includes functions of zero arguments, which are simply elements of X.

When describing a particular inductive type, we list the constructors with bullets. For instance, the type $\mathbf{2}$ from §1.8 is inductively generated by the following constructors:

- $1_{\mathbf{2}} : \mathbf{2}$
- $0_{\mathbf{2}} : \mathbf{2}$

Similarly, $\mathbf{1}$ is inductively generated by the constructor:

- $\star : \mathbf{1}$

while $\mathbf{0}$ is inductively generated by no constructors at all. An example where the constructor functions take arguments is the coproduct $A + B$, which is generated by the two constructors

- $\mathsf{inl} : A \to A + B$
- $\mathsf{inr} : B \to A + B$.

And an example with a constructor taking multiple arguments is the cartesian product $A \times B$, which is generated by one constructor

- $(-,-) : A \to B \to A \times B$.

Crucially, we also allow constructors of inductive types that take arguments from the inductive type being defined. For instance, the type $\mathbb{N}$ of natural numbers has constructors

- $0 : \mathbb{N}$
- $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$.

Another useful example is the type $\mathsf{List}(A)$ of finite lists of elements of some type A, which has constructors

- $\mathsf{nil} : \mathsf{List}(A)$
- $\mathsf{cons} : A \to \mathsf{List}(A) \to \mathsf{List}(A)$.

Intuitively, we should understand an inductive type as being *freely generated by* its constructors. That is, the elements of an inductive type are exactly what can be obtained by starting from nothing and applying the constructors repeatedly. (We will see in §5.8 and Chapter 6 that this conception has to be modified slightly for more general kinds of inductive definitions, but for now it is sufficient.) For instance, in the case of $\mathbf{2}$, we should expect that the only elements are $1_{\mathbf{2}}$ and $0_{\mathbf{2}}$. Similarly, in the case of $\mathbb{N}$, we should expect that every element is either 0 or obtained by applying succ to some "previously constructed" natural number.

Rather than assert properties such as this directly, however, we express them by means of an *induction principle*, also called a *(dependent) elimination rule*. We have seen these principles already in Chapter 1. For instance, the induction principle for $\mathbf{2}$ is:

- When proving a statement $E : \mathbf{2} \to \mathcal{U}$ about *all* inhabitants of $\mathbf{2}$, it suffices to prove it for $1_{\mathbf{2}}$ and $0_{\mathbf{2}}$, i.e., to give proofs $e_t : E(1_{\mathbf{2}})$ and $e_f : E(0_{\mathbf{2}})$.

Furthermore, the resulting proof $\mathsf{ind}_{\mathbf{2}}(E, e_t, e_f) : \prod_{(b:\mathbf{2})} E(b)$ behaves as expected when applied to the constructors $1_{\mathbf{2}}$ and $0_{\mathbf{2}}$; this principle is expressed by the *computation rules*:

- We have $\mathsf{ind}_{\mathbf{2}}(E, e_t, e_f, 1_{\mathbf{2}}) \equiv e_t$.
- We have $\mathsf{ind}_{\mathbf{2}}(E, e_t, e_f, 0_{\mathbf{2}}) \equiv e_f$.

Thus, the induction principle for the type $\mathbf{2}$ of Booleans allow us to reason by *case analysis*. Since neither of the two constructors takes any arguments, this is all we need for Booleans.

For natural numbers, however, case analysis is generally not sufficient: in the case corresponding to the inductive step $s(n)$, we also want to presume that the statement being proven has already been shown for n. This gives us the following induction principle:

- When proving a statement $E : \mathbb{N} \to \mathcal{U}$ about *all* natural numbers, it suffices to prove it for 0 and for $\mathsf{succ}(n)$, assuming it holds for n. This entails giving the proofs $e_z : E(0)$ and $e_s : \prod_{(n:\mathbb{N})} \prod_{(y:E(n))} E(\mathsf{succ}(n))$.

The variable y represents our inductive hypothesis. As for Booleans, we also have the associated computation rules for the function $\mathsf{ind}_{\mathbb{N}}(E, e_z, e_s) : \prod_{(x:\mathbb{N})} E(x)$:

- We have $\mathsf{ind}_{\mathbb{N}}(E, e_z, e_s, 0) \equiv e_z$.
- For any $n : \mathbb{N}$, we have $\mathsf{ind}_{\mathbb{N}}(E, e_z, e_s, \mathsf{succ}(n)) \equiv e_s(n, \mathsf{rec}_{\mathbb{N}}(E, e_z, e_s, n))$.

The dependent function $\mathsf{ind}_{\mathbb{N}}(E, e_z, e_s)$ can thus be understood as being defined recursively on the argument $x : \mathbb{N}$, via the recurrences e_z and e_s. When x is zero, the function simply returns e_z. When x is the successor of another natural number n, the result is obtained by taking the recurrence e_s and substituting the specific predecessor n and the recursive call value $\mathsf{rec}_{\mathbb{N}}(E, e_z, e_s, n)$.

The induction principles for all the examples mentioned above share this family resemblance. In §5.6 we will discuss a general notion of "inductive definition" and how to derive an appropriate *induction principle* for it, but first we investigate various commonalities between inductive definitions.

For instance, we have remarked in every case in Chapter 1 that from the induction principle we can derive a *recursion principle* in which the codomain is a simple type (rather than a family). Both induction and recursion principles may seem odd, since they yield only the *existence* of a function without seeming to characterize it uniquely. However, in fact the induction principle is strong enough also to prove its own *uniqueness principle*. For example:

Theorem 5.1.1. *Let $f, g : \prod_{(x:\mathbb{N})} E(x)$ be two functions which satisfy the recurrences $e_z : E(0)$ and $e_s : \prod_{(n:\mathbb{N})} \prod_{(y:E(n))} E(\mathsf{succ}(n))$ up to propositional equality, i.e., such that*

$$f(0) = e_z$$
$$g(0) = e_z$$

and

$$\prod_{n:\mathbb{N}} f(\mathsf{succ}(n)) = e_s(n, f(n))$$
$$\prod_{n:\mathbb{N}} g(\mathsf{succ}(n)) = e_s(n, g(n))$$

Then f and g are equal.

Proof. We use induction on the type family $D(x) :\equiv f(x) = g(x)$. For the base case, we have

$$f(0) = e_z = g(0)$$

For the inductive case, assume $n : \mathbb{N}$ such that $f(n) = g(n)$. Then

$$f(\mathsf{succ}(n)) = e_s(n, f(n)) = e_s(n, g(n)) = g(\mathsf{succ}(n))$$

The first and last equality follow from the assumptions on f and g. The middle equality follows from the inductive hypothesis and the fact that application preserves equality. This gives us pointwise equality between f and g; invoking function extensionality finishes the proof. □

Note that the uniqueness principle applies even to functions that only satisfy the recurrences *up to propositional equality*, i.e. a path. Of course, the particular function obtained from the induction principle satisfies these recurrences judgmentally; we will return to this point in §5.5. On the other hand, the theorem itself only asserts a propositional equality between functions (see also Exercise 5.1). From a homotopical viewpoint it is natural to ask whether this path is *coherent*, i.e.

whether the equality $f = g$ is unique up to higher paths; in §5.4 we will see that this is in fact the case.

Of course, similar uniqueness theorems for functions can generally be formulated and shown for other inductive types as well. In the next section, we show how this uniqueness property, together with univalence, implies that an inductive type such as the natural numbers is completely characterized by its introduction, elimination, and computation rules.

5.2 Uniqueness of inductive types

We have defined "the" natural numbers to be a particular type $\mathbb{N}$ with particular inductive generators 0 and succ. However, by the general principle of inductive definitions in type theory described in the previous section, there is nothing preventing us from defining *another* type in an identical way. That is, suppose we let $\mathbb{N}'$ be the inductive type generated by the constructors

- $0' : \mathbb{N}'$
- $\mathsf{succ}' : \mathbb{N}' \to \mathbb{N}'$.

Then $\mathbb{N}'$ will have identical-looking induction and recursion principles to $\mathbb{N}$. When proving a statement $E : \mathbb{N}' \to \mathcal{U}$ for all of these "new" natural numbers, it suffices to give the proofs $e_z : E(0')$ and $e_s : \prod_{(n:\mathbb{N}')} \prod_{(x:E(n))} E(\mathsf{succ}'(n))$. And the function $\mathsf{rec}_{\mathbb{N}'}(E, e_z, e_s) : \prod_{(n:\mathbb{N}')} E(n)$ has the following computation rules:

- We have $\mathsf{rec}_{\mathbb{N}'}(E, e_z, e_s, 0') \equiv e_z$.
- For any $n : \mathbb{N}'$, we have $\mathsf{rec}_{\mathbb{N}'}(E, e_z, e_s, \mathsf{succ}'(n)) \equiv e_s(n, \mathsf{rec}_{\mathbb{N}'}(E, e_z, e_s, n))$.

But what is the relation between $\mathbb{N}$ and $\mathbb{N}'$?

This is not just an academic question, since structures that "look like" the natural numbers can be found in many other places. For instance, lists over type with one element (this is arguably the oldest appearance, found on walls of caves), in the non-negative integers, as a substructure of the rationals and the reals, and so on. And from a programming point of view, the "unary" representation of our natural numbers is very inefficient, so we might prefer sometimes to use a binary one instead. We would like to be able to identify all of these versions of "the natural numbers" with each other, in order to transfer constructions and results from one to another.

Of course, if two versions of the natural numbers satisfy identical induction principles, then they have identical induced structure. For instance, recall the example of the function double defined above. A similar function for our new natural numbers is readily defined by duplication and adding primes:

$$\mathsf{double}' :\equiv \mathsf{rec}_{\mathbb{N}'}(\mathbb{N}',\ 0',\ \lambda n.\, \lambda m.\, \mathsf{succ}'(\mathsf{succ}'(m)))$$

Simple as this may seem, it has the obvious drawback of leading to a proliferation of duplicates. Not only functions have to be duplicated, but also all lemmas and their proofs. For example, an easy result such as $\prod_{(n:\mathbb{N})} \mathsf{double}(\mathsf{succ}(n)) = \mathsf{succ}(\mathsf{succ}(\mathsf{double}(n)))$, as well as its proof by induction, also has to be "primed".

In traditional mathematics, one just proclaims that $\mathbb{N}$ and $\mathbb{N}'$ are obviously "the same", and can be substituted for each other whenever the need arises. This is usually unproblematic, but it sweeps a fair amount under the rug, widening the gap between informal mathematics and its precise description. In homotopy type theory, we can do better.

First observe that we have the following definable maps:

- $f :\equiv \mathsf{rec}_{\mathbb{N}}(\mathbb{N},\ 0',\ \lambda n.\, \mathsf{succ}') : \mathbb{N} \to \mathbb{N}'$,
- $g :\equiv \mathsf{rec}_{\mathbb{N}'}(\mathbb{N}',\ 0,\ \lambda n.\, , \mathsf{succ}) : \mathbb{N}' \to \mathbb{N}$.

Since the composition of g and f satisfies the same recurrences as the identity function on $\mathbb{N}$, Theorem 5.1.1 gives that $\prod_{(n:\mathbb{N})} g(f(n)) = n$, and the "primed" version of the same theorem gives $\prod_{(n:\mathbb{N}')} f(g(n)) = n$. Thus, f and g are quasi-inverses, so that $\mathbb{N} \simeq \mathbb{N}'$. We can now transfer functions on $\mathbb{N}$ directly to functions on $\mathbb{N}'$ (and vice versa) along this equivalence, e.g.

$$\mathsf{double}' :\equiv \lambda n.\, f(\mathsf{double}(g(n)))$$

It is an easy exercise to show that this version of double' is equal to the earlier one.

Of course, there is nothing surprising about this; such an isomorphism is exactly how a mathematician will envision "identifying" $\mathbb{N}$ with $\mathbb{N}'$. However, the mechanism of "transfer" across an isomorphism depends on the thing being transferred; it is not always as simple as pre- and post-composing a single function with f and g. Consider, for instance, a simple lemma such as

$$\prod_{n:\mathbb{N}'} \mathsf{double}'(\mathsf{succ}'(n)) = \mathsf{succ}'(\mathsf{succ}'(\mathsf{double}'(n))).$$

Inserting the correct fs and gs is only a little easier than re-proving it by induction on $n : \mathbb{N}'$ directly.

Here is where the univalence axiom steps in: since $\mathbb{N} \simeq \mathbb{N}'$, we also have $\mathbb{N} =_{\mathcal{U}} \mathbb{N}'$, i.e. $\mathbb{N}$ and $\mathbb{N}'$ are *equal* as types. Now the induction principle for identity guarantees that any construction or proof relating to $\mathbb{N}$ can automatically be transferred to $\mathbb{N}'$ in the same way. We simply consider the type of the function or theorem as a type-indexed family of types $P : \mathcal{U} \to \mathcal{U}$, with the given object being an element of $P(\mathbb{N})$, and transport along the path $\mathbb{N} = \mathbb{N}'$. This involves considerably less overhead.

For simplicity, we have described this method in the case of two types $\mathbb{N}$ and $\mathbb{N}'$ with *identical*-looking definitions. However, a more common situation in practice is when the definitions are not literally identical, but nevertheless one induction principle implies the other. Consider, for instance, the type of lists from a one-element type, $\mathsf{List}(\mathbf{1})$, which is generated by

- an element $\mathsf{nil} : \mathsf{List}(\mathbf{1})$, and
- a function $\mathsf{cons} : \mathbf{1} \times \mathsf{List}(\mathbf{1}) \to \mathsf{List}(\mathbf{1})$.

This is not identical to the definition of $\mathbb{N}$, and it does not give rise to an identical induction principle. The induction principle of $\mathsf{List}(\mathbf{1})$ says that for any $E : \mathsf{List}(\mathbf{1}) \to \mathcal{U}$ together with recurrence data $e_{\mathsf{nil}} : E(\mathsf{nil})$ and $e_{\mathsf{cons}} : \prod_{(u:\mathbf{1})} \prod_{(\ell:\mathsf{List}(\mathbf{1}))} E(\ell) \to E(\mathsf{cons}(u,\ell))$, there exists $f : \prod_{(\ell:\mathsf{List}(\mathbf{1}))} E(\ell)$ such that $f(\mathsf{nil}) \equiv e_{\mathsf{nil}}$ and $f(\mathsf{cons}(u,\ell)) \equiv e_{\mathsf{cons}}(u,\ell,f(\ell))$. (We will see how to derive the induction principle of an inductive definition in §5.6.)

Now suppose we define $0'' :\equiv \mathsf{nil} : \mathsf{List}(\mathbf{1})$, and $\mathsf{succ}'' : \mathsf{List}(\mathbf{1}) \to \mathsf{List}(\mathbf{1})$ by $\mathsf{succ}''(\ell) :\equiv \mathsf{cons}(\star, \ell)$. Then for any $E : \mathsf{List}(\mathbf{1}) \to \mathcal{U}$ together with $e_0 : E(0'')$ and $e_s : \prod_{(\ell:\mathsf{List}(\mathbf{1}))} E(\ell) \to E(\mathsf{succ}''(\ell))$, we can define

$$e_{\mathsf{nil}} :\equiv e_0$$
$$e_{\mathsf{cons}}(\star, \ell, x) :\equiv e_s(\ell, x).$$

(In the definition of e_{cons} we use the induction principle of $\mathbf{1}$ to assume that u is $\star$.) Now we can apply the induction principle of $\mathsf{List}(\mathbf{1})$, obtaining $f : \prod_{(\ell:\mathsf{List}(\mathbf{1}))} E(\ell)$ such that

$$f(0'') \equiv f(\mathsf{nil}) \equiv e_{\mathsf{nil}} \equiv e_0$$
$$f(\mathsf{succ}''(\ell)) \equiv f(cons(\star, \ell)) \equiv e_{\mathsf{cons}}(\star, \ell, f(\ell)) \equiv e_s(\ell, f(\ell)).$$

Thus, $\mathsf{List}(\mathbf{1})$ satisfies the same induction principle as $\mathbb{N}$, and hence (by the same arguments above) is equal to it.

Finally, these conclusions are not confined to the natural numbers: they apply to any inductive type. If we have an inductively defined type W, say, and some other type W' which satisfies the same induction principle as W, then it follows that $W \simeq W'$, and hence $W = W'$. We use the derived recursion principles for W and W' to construct maps $W \to W'$ and $W' \to W$, respectively, and then the induction principles for each to prove that both composites are equal to identities. For instance, in Chapter 1 we saw that the coproduct $A + B$ could also have been defined as $\sum_{(x:\mathbf{2})} \mathsf{rec}_{\mathbf{2}}(\mathcal{U}, A, B, x)$. The latter type satisfies the same induction principle as the former; hence they are canonically equivalent.

This is, of course, very similar to the familiar fact in category theory that if two objects have the same *universal property*, then they are equivalent. In §5.4 we will see that inductive types actually do have a universal property, so that this is a manifestation of that general principle.

5.3 W-types

Inductive types are very general, which is excellent for their usefulness and applicability, but makes them difficult to study as a whole. Fortunately, they can all be formally reduced to a few special cases. It is beyond the scope of this book to discuss this reduction — which is anyway irrelevant to the mathematician using type theory in practice — but we will take a little time to discuss the one of the basic special cases that we have not yet met. These are Martin-Löf's *W-types*, also known as the types of *well-founded trees*. W-types are a generalization of such types as natural numbers, lists, and binary trees, which are sufficiently general to encapsulate the "recursion" aspect of *any* inductive type.

A particular W-type is specified by giving two parameters $A : \mathcal{U}$ and $B : A \to \mathcal{U}$, in which case the resulting W-type is written $\mathsf{W}_{(a:A)} B(a)$. The type A represents the type of *labels* for $\mathsf{W}_{(a:A)} B(a)$, which function as constructors (however, we reserve that word for the actual functions which arise in inductive definitions). For instance, when defining natural numbers as a W-type, the type A would be the type $\mathbf{2}$ inhabited by the two elements $1_{\mathbf{2}}$ and $0_{\mathbf{2}}$, since there are precisely two ways to obtain a natural number—either it will be zero or a successor of another natural number.

The type family $B : A \to \mathcal{U}$ is used to record the arity of labels: a label $a : A$ will take a family of inductive arguments, indexed over $B(a)$. We can therefore think of the "$B(a)$-many" arguments of a. These arguments are represented by a function $f : B(a) \to \mathsf{W}_{(a:A)}B(a)$, with the understanding that for any $b : B(a)$, $f(b)$ is the "b-th" argument to the label a. The W-type $\mathsf{W}_{(a:A)}B(a)$ can thus be thought of as the type of well-founded trees, where nodes are labeled by elements of A and each node labeled by $a : A$ has $B(a)$-many branches.

In the case of natural numbers, the label $1_{\mathbf{2}}$ has arity 0, since it constructs the constant zero; the label $0_{\mathbf{2}}$ has arity 1, since it constructs the successor of its argument. We can capture this by using simple elimination on $\mathbf{2}$ to define a function $\mathsf{rec}_{\mathbf{2}}(\mathcal{U}, \mathbf{0}, \mathbf{1})$ into a universe of types; this function returns the empty type $\mathbf{0}$ for $1_{\mathbf{2}}$ and the unit type $\mathbf{1}$ for $0_{\mathbf{2}}$. We can thus define

$$\mathbf{N}^{\mathbf{w}} :\equiv \mathsf{W}(b : \mathbf{2}),\ \mathsf{rec}_{\mathbf{2}}(\mathcal{U}, \mathbf{0}, \mathbf{1})$$

where the superscript $\mathbf{w}$ serves to distinguish this version of natural numbers from the previously used one. Similarly, we can define the type of lists over A as a W-type with $\mathbf{1} + A$ many labels: one nullary label for the empty list, plus one unary label for each $a : A$, corresponding to appending a to the head of a list:

$$\mathsf{List}(A) :\equiv \mathsf{W}(x : \mathbf{1} + A),\ \mathsf{rec}_{\mathbf{1}+A}(\mathcal{U},\ \mathbf{0},\ \lambda a.\, \mathbf{1}).$$

In general, the W-type $\mathsf{W}_{(x:A)}B(x)$ specified by $A : \mathcal{U}$ and $B : A \to \mathcal{U}$ is the inductive type generated by the following constructor:

- $\mathsf{sup} : \prod_{(a:A)} \left(B(a) \to \mathsf{W}_{(x:A)}B(x)\right) \to \mathsf{W}_{(x:A)}B(x)$.

The constructor sup (short for supremum) takes a label $a : A$ and a function $f : B(a) \to \mathsf{W}_{(a:A)}B(a)$ representing the arguments to a, and constructs a new element of $\mathsf{W}_{(x:A)}B(x)$. Using our previous encoding of natural numbers as W-types, we can for instance define

$$0^{\mathbf{w}} :\equiv \mathsf{sup}(1_{\mathbf{2}},\ \lambda x.\, \mathsf{rec}_{\mathbf{0}}(\mathbf{N}^{\mathbf{w}}, x))$$

Put differently, we use the label $1_{\mathbf{2}}$ to construct $0^{\mathbf{w}}$. Then, $\mathsf{rec}_{\mathbf{2}}(\mathcal{U}, \mathbf{0}, \mathbf{1}, 1_{\mathbf{2}})$ evaluates to $\mathbf{0}$, as it should since $1_{\mathbf{2}}$ is a nullary label. Thus, we need to construct a function $f : \mathbf{0} \to \mathbf{N}^{\mathbf{w}}$, which represents the (zero) arguments supplied to $1_{\mathbf{2}}$. This is of course trivial, using simple elimination on $\mathbf{0}$ as shown. Similarly, we can define

$$1^{\mathbf{w}} :\equiv \mathsf{sup}(0_{\mathbf{2}},\ \lambda x.\, 0^{\mathbf{w}})$$
$$2^{\mathbf{w}} :\equiv \mathsf{sup}(0_{\mathbf{2}},\ \lambda x.\, 1^{\mathbf{w}})$$

and so on.

We have the following elimination rule for W types:

- When proving a statement $E : \left(\mathsf{W}_{(x:A)}B(x)\right) \to \mathcal{U}$ about *all* elements of the W-type $\mathsf{W}_{(x:A)}B(x)$, it suffices to prove it for $\mathsf{sup}(a, f)$, assuming it holds for all $f(b)$ with $b : B(a)$. In other words, it suffices to give a proof

$$e : \prod_{(a:A)} \prod_{(f:B(a)\to \mathsf{W}_{(x:A)}B(x))} \prod_{(g:\prod_{(b:B(a))} E(f(b)))} E(\mathsf{sup}(a, f))$$

The variable g represents our inductive hypothesis, namely that all arguments of a satisfy E. To state this, we quantify over all elements of type $B(a)$, since each $b : B(a)$ corresponds to one argument $f(b)$ of a.

How would we define the function double on natural numbers encoded as a W-type? We would like to use the (simple) elimination on $\mathbf{N}^{\mathbf{w}}$ into the type $\mathbf{N}^{\mathbf{w}}$ itself. We thus need to construct a suitable function

$$e : \prod_{(a:\mathbf{2})} \prod_{(f:B(a)\to\mathbf{N}^{\mathbf{w}})} \prod_{(g:B(a)\to\mathbf{N}^{\mathbf{w}})} \mathbb{N}$$

which will represent the recurrence for the double function; for simplicity we denote the type $\mathsf{rec}_{\mathbf{2}}(\mathcal{U}, \mathbf{0}, \mathbf{1})$ by B.

Clearly, e will be a function taking $a : \mathbf{2}$ as its first argument. The next step is to perform case analysis on a and proceed based on whether it is $1_{\mathbf{2}}$ or $0_{\mathbf{2}}$. This suggests the following form

$$e :\equiv \lambda a.\, \mathsf{rec}_{\mathbf{2}}(C, e_t, e_f, a)$$

where

$$C :\equiv \prod_{(f:B(a)\to\mathbf{N}^{\mathbf{w}})} \prod_{(g:B(a)\to\mathbf{N}^{\mathbf{w}})} \mathbf{N}^{\mathbf{w}}$$

If a is $1_{\mathbf{2}}$, the type $B(a)$ becomes $\mathbf{0}$. Thus, given $f : \mathbf{0} \to \mathbf{N}^{\mathbf{w}}$ and $g : \mathbf{0} \to \mathbf{N}^{\mathbf{w}}$, we want to construct an element of $\mathbf{N}^{\mathbf{w}}$. Since the label $1_{\mathbf{2}}$ represents $\mathbf{0}$, it needs zero inductive arguments and the variables f and g are irrelevant. We return $0^{\mathbf{w}}$ as a result:

$$e_t :\equiv \lambda f.\, \lambda g.\, 0^{\mathbf{w}}$$

Analogously, if a is $0_{\mathbf{2}}$, the type $B(a)$ becomes $\mathbf{1}$. Since the label $0_{\mathbf{2}}$ represents the successor operator, it needs one inductive argument—the predecessor—which is represented by the variable $f : \mathbf{1} \to \mathbf{N}^{\mathbf{w}}$. The value of the recursive call on the predecessor is represented by the variable $g : \mathbf{1} \to \mathbf{N}^{\mathbf{w}}$. Thus, taking this value (namely $g(\star)$) and applying the successor operator twice thus yields the desired result:

$$\begin{aligned} e_f :\equiv\ & \lambda f.\, \lambda g. \\ & \mathsf{sup}(0_{\mathbf{2}}, (\lambda x.\, \mathsf{sup}(0_{\mathbf{2}}, (\lambda y.\, g(\star))))) \end{aligned}$$

Putting this together, we thus have

$$\mathsf{double} :\equiv \mathsf{rec}_{\mathbf{N}^{\mathbf{w}}}(\mathbf{N}^{\mathbf{w}}, e)$$

with e as defined above.

The associated computation rule for the function $\mathsf{rec}_{\mathsf{W}_{(x:A)}B(x)}(E, e) : \prod_{(w:\mathsf{W}_{(x:A)}B(x))} E(w)$ is as follows.

- For any $a : A$ and $f : B(a) \to \mathsf{W}_{(x:A)}B(x)$ we have

$$\begin{aligned} & \mathsf{rec}_{\mathsf{W}_{(x:A)}B(x)}(E, e, \mathsf{sup}(a, f)) \equiv \\ & e(a, f, (\lambda b.\, \mathsf{rec}_{\mathsf{W}_{(x:A)}B(x)}(E, f(b)))) \end{aligned}$$

In other words, the function $\mathsf{rec}_{\mathsf{W}_{(x:A)}B(x)}(E,e)$ satisfies the recurrence e.

By the above computation rule, the function double behaves as expected:

$$\begin{aligned}
\mathsf{double}(0^{\mathbf{w}}) &\equiv \mathsf{rec}_{\mathbf{N}^{\mathbf{w}}}(\mathbf{N}^{\mathbf{w}}, e, \mathsf{sup}(1_{\mathbf{2}},\ \lambda x.\, \mathsf{rec}_{\mathbf{0}}(\mathbf{N}^{\mathbf{w}}, x))) \\
&\equiv e(1_{\mathbf{2}}, (\lambda x.\, \mathsf{rec}_{\mathbf{0}}(\mathbf{N}^{\mathbf{w}}, x)), (\lambda x.\, \mathsf{double}(\mathsf{rec}_{\mathbf{0}}(\mathbf{N}^{\mathbf{w}}, x)))) \\
&\equiv e_t((\lambda x.\, \mathsf{rec}_{\mathbf{0}}(\mathbf{N}^{\mathbf{w}}, x)), (\lambda x.\, \mathsf{double}(\mathsf{rec}_{\mathbf{0}}(\mathbf{N}^{\mathbf{w}}, x)))) \\
&\equiv 0^{\mathbf{w}}
\end{aligned}$$

$$\begin{aligned}
\mathsf{double}(1^{\mathbf{w}}) &\equiv \mathsf{rec}_{\mathbf{N}^{\mathbf{w}}}(\mathbf{N}^{\mathbf{w}}, e, \mathsf{sup}(0_{\mathbf{2}},\ \lambda x.\, 0^{\mathbf{w}})) \\
&\equiv e(0_{\mathbf{2}}, (\lambda x.\, 0^{\mathbf{w}}), (\lambda x.\, \mathsf{double}(0^{\mathbf{w}}))) \\
&\equiv e_f((\lambda x.\, 0^{\mathbf{w}}), (\lambda x.\, \mathsf{double}(0^{\mathbf{w}}))) \\
&\equiv \mathsf{sup}(0_{\mathbf{2}}, (\lambda x.\, \mathsf{sup}(0_{\mathbf{2}}, (\lambda y.\, \mathsf{double}(0^{\mathbf{w}}))))) \\
&\equiv \mathsf{sup}(0_{\mathbf{2}}, (\lambda x.\, \mathsf{sup}(0_{\mathbf{2}}, (\lambda y.\, 0^{\mathbf{w}})))) \\
&\equiv 2^{\mathbf{w}}
\end{aligned}$$

and so on.

Just as for natural numbers, we can prove a uniqueness theorem for W-types:

Theorem 5.3.1. *Let $g, h : \prod_{(w:\mathsf{W}_{(x:A)}B(x))} E(w)$ be two functions which satisfy the recurrence $e : \prod_{(a,f)}(\prod_{(b:B(a))} E(f(b))) \to E(\mathsf{sup}(a,f))$ up to propositional equality, i.e., such that*

$$\prod_{a,f} g(\mathsf{sup}(a,f)) = e(a,f,\lambda b.\, g(f(b)))$$
$$\prod_{a,f} h(\mathsf{sup}(a,f)) = e(a,f,\lambda b.\, h(f(b)))$$

Then g and h are equal.

5.4 Inductive types are initial algebras

As suggested earlier, inductive types also have a category-theoretic universal property. They are *homotopy-initial algebras*: initial objects (up to coherent homotopy) in a category of "algebras" determined by the specified constructors. As a simple example, consider the natural numbers. The appropriate sort of "algebra" here is a type equipped with the same structure that the constructors of $\mathbb{N}$ give to it.

Definition 5.4.1. A **$\mathbb{N}$-algebra** is a type C with two elements $c_0 : C$, $c_s : C \to C$. Thus,

$$\mathbb{N}\mathsf{Alg} :\equiv \sum_{C:\mathcal{U}} C \times (C \to C)$$

Definition 5.4.2. Fix any $\mathbb{N}$-algebras (C, c_0, c_s) and (D, d_0, d_s). A **$\mathbb{N}$-homomorphism** between them is a function $h : C \to D$ such that $h(c_0) = d_0$ and $h(c_s(c)) = d_s(h(c))$ for any $c : C$. Thus,

$$\mathbb{N}\mathsf{Hom}((C,c_0,c_s),(D,d_0,d_s)) :\equiv \sum_{(h:C\to D)} (h(c_0) = d_0) \times \textstyle\prod_{(c:C)}(h(c_s(c)) = d_s(h(c)))$$

We thus have a category of $\mathbb{N}$-algebras and $\mathbb{N}$-homomorphisms, and the claim is that $\mathbb{N}$ is the initial object of this category. A category theorist will immediately recognize this as the definition of a *natural numbers object* in a category.

Of course, since our types behave like ∞-groupoids, we actually have an $(\infty, 1)$-category of $\mathbb{N}$-algebras, and we should ask $\mathbb{N}$ to be initial in the appropriate $(\infty, 1)$-categorical sense. Fortunately, we can formulate this without needing to define $(\infty, 1)$-categories.

Definition 5.4.3. A $\mathbb{N}$-algebra I is called **homotopy-initial**, or **h-initial** for short, if for any other $\mathbb{N}$-algebra C, the type of $\mathbb{N}$-homomorphisms from I to C is contractible. Thus,

$$\mathsf{isHinit}_{\mathbb{N}}(I) :\equiv \prod_{C:\mathbb{N}\mathsf{Alg}} \mathsf{isContr}(\mathbb{N}\mathsf{Hom}(I, C)).$$

When they exist, h-initial algebras are unique — not just up to isomorphism, as usual in category theory, but up to equality, by the univalence axiom.

Theorem 5.4.4. *Any two h-initial* $\mathbb{N}$*-algebras are equal. Thus, the type of h-initial* $\mathbb{N}$*-algebras is a mere proposition.*

Proof. Suppose I and J are h-initial $\mathbb{N}$-algebras. Then $\mathbb{N}\mathsf{Hom}(I, J)$ is contractible, hence inhabited by some $\mathbb{N}$-homomorphism $f : I \to J$, and likewise we have an $\mathbb{N}$-homomorphism $g : J \to I$. Now the composite $g \circ f$ is a $\mathbb{N}$-homomorphism from I to I, as is id_I; but $\mathbb{N}\mathsf{Hom}(I, I)$ is contractible, so $g \circ f = \mathsf{id}_I$. Similarly, $f \circ g = \mathsf{id}_J$. Hence $I \simeq J$, and so $I = J$. Since being contractible is a mere proposition and dependent products preserve mere propositions, it follows that being h-initial is itself a mere proposition. Thus any two proofs that I (or J) is h-initial are necessarily equal, which finishes the proof. □

We now have the following theorem.

Theorem 5.4.5. *The* $\mathbb{N}$*-algebra* $(\mathbb{N}, \mathbf{0}, \mathsf{succ})$ *is homotopy initial.*

Sketch of proof. Fix an arbitrary $\mathbb{N}$-algebra (C, c_0, c_s). The recursion principle of $\mathbb{N}$ yields a function $f : \mathbb{N} \to C$ defined by

$$f(0) :\equiv c_0$$
$$f(\mathsf{succ}(n)) :\equiv c_s(f(n)).$$

These two equalities make f an $\mathbb{N}$-homomorphism, which we can take as the center of contraction for $\mathbb{N}\mathsf{Hom}(\mathbb{N}, C)$. The uniqueness theorem (Theorem 5.1.1) then implies that any other $\mathbb{N}$-homomorphism is equal to f. □

To place this in a more general context, it is useful to consider the notion of *algebra for an endofunctor*. Note that to make a type C into a $\mathbb{N}$-algebra is the same as to give a function $c : C + \mathbf{1} \to C$, and a function $f : C \to D$ is a $\mathbb{N}$-homomorphism just when $f \circ c \sim d \circ (f + \mathbf{1})$. In categorical language, this means the $\mathbb{N}$-algebras are the algebras for the endofunctor $F(X) :\equiv X + 1$ of the category of types.

For a more generic case, consider the W-type associated to $A : \mathcal{U}$ and $B : A \to \mathcal{U}$. In this case we have an associated **polynomial functor**:

$$P(X) = \sum_{x:A} (B(x) \to X)\,. \tag{5.4.6}$$

Actually, this assignment is functorial only up to homotopy, but this makes no difference in what follows. By definition, a P**-algebra** is then a type C equipped a function $s_C : PC \to C$. By the universal property of Σ-types, this is equivalent to giving a function $\prod_{(a:A)}(B(a) \to C) \to C$. We will also call such objects W**-algebras** for A and B, and we write

$$\mathsf{WAlg}(A, B) :\equiv \sum_{(C:\mathcal{U})} \prod_{(a:A)} (B(a) \to C) \to C.$$

Similarly, for P-algebras (C, s_C) and (D, s_D), a **homomorphism** or **algebra map** between them $(f, s_f) : (C, s_C) \to (D, s_D)$ consists of a function $f : C \to D$ and a homotopy between maps $PC \to D$

$$s_f : f \circ s_C \;= s_D \circ Pf\,,$$

where $Pf : PC \to PD$ is the result of the easily-definable action of P on $f : C \to D$. Such an algebra homomorphism can be represented suggestively in the form:

$$\begin{array}{ccc} PC & \xrightarrow{Pf} & PD \\ {\scriptstyle s_C}\downarrow & {\scriptstyle s_f} & \downarrow{\scriptstyle s_D} \\ C & \xrightarrow[f]{} & D \end{array}$$

In terms of elements, f is a P-homomorphism (or W-*homomorphism*) if

$$f(s_C(a, h)) = s_D(a, \lambda b.\, f(h(b))).$$

We have the type of W-homomorphisms:

$$\mathsf{WHom}_{A,B}((C, c), (D, d)) :\equiv \sum_{(h:C\to D)} \prod_{(a:A)} \prod_{(f:B(a)\to C)} h(c(a, f)) = \lambda b.\, h(f(b))$$

Finally, a P-algebra (C, s_C) is said to be **homotopy-initial** if for every P-algebra (D, s_D), the type of all algebra maps is contractible. That is,

$$\mathsf{isHinit}_{\mathsf{W}}(A, B, I) :\equiv \prod_{C:\mathsf{WAlg}(A,B)} \mathsf{isContr}(\mathsf{WHom}_{A,B}(I, C)).$$

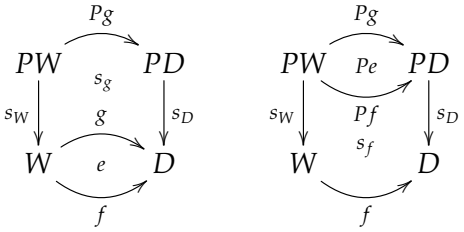
Now the analogous theorem to Theorem 5.4.5 is:

Theorem 5.4.7. *For any type $A : \mathcal{U}$ and type family $B : A \to \mathcal{U}$, the* W*-algebra* $(\mathsf{W}_{(x:A)}B(x), \mathsf{sup})$ *is h-initial.*

Sketch of proof. Suppose we have $A : \mathcal{U}$ and $B : A \to \mathcal{U}$, and consider the associated polynomial functor $P(X) :\equiv \sum_{(x:A)}(B(x) \to X)$. Let $W :\equiv \mathsf{W}_{(x:A)}B(x)$. Then using the W-introduction rule from §5.3, we have a structure map $s_W :\equiv \mathsf{sup} : PW \to W$. We want to show that the algebra (W, s_W) is h-initial. So, let us consider another algebra (C, s_C) and show that the type $T :\equiv \mathsf{WHom}_{A,B}((W, s_W), (C, s_C))$ of W-homomorphisms from (W, s_W) to (C, s_C) is contractible. To do so, observe that the W-elimination rule and the W-computation rule allow us to define a W-homomorphism $(f, s_f) : (W, s_W) \to (C, s_C)$, thus showing that T is inhabited. It is furthermore necessary to show that for every W-homomorphism $(g, s_g) : (W, s_W) \to (C, s_C)$, there is an identity proof

$$p : (f, s_f) = (g, s_g)\,. \tag{5.4.8}$$

This uses the fact that, in general, a type of the form $(f, s_f) = (g, s_g)$ is equivalent to the type of what we call **algebra 2-cells**, whose canonical elements are pairs of the form (e, s_e), where $e : f = g$ and s_e is a higher identity proof between the identity proofs represented by the following pasting diagrams:



In light of this fact, to prove that there exists an element as in (5.4.8), it is sufficient to show that there is an algebra 2-cell

$$(e, s_e) : (f, s_f) = (g, s_g)\,.$$

The identity proof $e : f = g$ is now constructed by function extensionality and W-elimination so as to guarantee the existence of the required identity proof s_e. □

5.5 Homotopy-inductive types

In §5.3 we showed how to encode natural numbers as W-types, with

$$\begin{aligned}\mathbf{N}^{\mathbf{w}} &:\equiv \mathsf{W}(b : \mathbf{2}),\ \mathsf{rec}_{\mathbf{2}}(\mathcal{U}, \mathbf{0}, \mathbf{1})\\ 0^{\mathbf{w}} &:\equiv \mathsf{sup}(1_{\mathbf{2}}, (\lambda x.\, \mathsf{rec}_{\mathbf{0}}(\mathbf{N}^{\mathbf{w}}, x)))\\ \mathbf{s}^{\mathbf{w}} &:\equiv \lambda n.\, \mathsf{sup}(0_{\mathbf{2}}, (\lambda x.\, n))\end{aligned}$$

We also showed how one can define a double function on $\mathbf{N}^{\mathbf{w}}$ using the recursion principle. When it comes to the induction principle, however, this encoding is no longer satisfactory: given $E : \mathbf{N}^{\mathbf{w}} \to \mathcal{U}$ and recurrences $e_z : E(0^{\mathbf{w}})$ and $e_s : \prod_{(n:\mathbf{N}^{\mathbf{w}})} \prod_{(y:E(n))} E(\mathbf{s}^{\mathbf{w}}(n))$, we can only construct a dependent function $r(E, e_z, e_s) : \prod_{(n:\mathbf{N}^{\mathbf{w}})} E(n)$ satisfying the given recurrences *propositionally*, i.e. up to a path. This means that the computation rules for natural numbers, which give judgmental equalities, cannot be derived from the rules for W-types in any obvious way.

This problem goes away if instead of the conventional inductive types we consider *homotopy-inductive types*, where all computation rules are stated up to a path, i.e. the symbol $\equiv$ is replaced by $=$. For instance, the computation rule for the homotopy version of W-types W^{h} becomes:

- For any $a : A$ and $f : B(a) \to \mathsf{W}^h_{(x:A)} B(x)$ we have

$$\mathsf{rec}_{\mathsf{W}^h_{(x:A)}B(x)}(E, \mathsf{sup}(a, f)) = e\Big(a, f, \big(\lambda b.\, \mathsf{rec}_{\mathsf{W}^h_{(x:A)}B(x)}(E, f(b))\big)\Big)$$

Homotopy-inductive types have an obvious disadvantage when it comes to computational properties—the behavior of any function constructed using the elimination principle can now only be characterized propositionally. But numerous other considerations drive us to consider homotopy-inductive types as well. For instance, while we showed in §5.4 that inductive types are homotopy-initial algebras, not every homotopy-initial algebra is an inductive type (i.e. satisfies the corresponding induction principle) — but every homotopy-initial algebra *is* a homotopy-inductive type. Similarly, we might want to apply the uniqueness argument from §5.2 when one (or both) of the types involved is only a homotopy-inductive type — for instance, to show that the W-type encoding of $\mathbb{N}$ is equivalent to the usual $\mathbb{N}$.

Additionally, the notion of a homotopy-inductive type is now internal to the type theory. For example, this means we can form a type of all natural numbers objects and make assertions about it. In the case of W-types, we can characterize a homotopy W-type $\mathsf{W}_{(x:A)}B(x)$ as any type endowed with a supremum function and a dependent eliminator satisfying the appropriate (propositional) computation rule:

$$\mathsf{W}_d(A, B) :\equiv \sum_{W:\mathcal{U}} \sum_{\mathsf{sup}:\prod_{(a)}(B(a)\to W)\to W} \prod_{E:W\to\mathcal{U}} \prod_{e:\prod_{(a,f)}(\prod_{(b:B(a))} E(f(b)))\to E(\mathsf{sup}(a,f))} \sum_{\mathsf{rec}:\prod_{(w:W)} E(w)} \prod_{a,f} \mathsf{rec}(\mathsf{sup}(a, f)) = e(a, \lambda b.\, \mathsf{rec}(f(b))).$$

In Chapter 6 we will see some other reasons why propositional computation rules are worth considering.

In this section, we will state some basic facts about homotopy-inductive types. We omit most of the proofs, which are somewhat technical.

Theorem 5.5.1. *For any $A : \mathcal{U}$ and $B : A \to \mathcal{U}$, the type $\mathsf{W}_d(A, B)$ is a mere proposition.*

It turns out that there is an equivalent characterization of W-types using simple elimination, plus certain *uniqueness* and *coherence* laws. First we give the rule for simple elimination:

- When constructing a function from the the W-type $\mathsf{W}^h_{(x:A)}B(x)$ into the type C, it suffices to give its value for $\mathsf{sup}(a, f)$, assuming we are given the values of all $f(b)$ with $b : B(a)$. In other words, it suffices to construct a function

$$c : \prod_{a:A} (B(a) \to C) \to C$$

The associated computation rule for $\mathsf{rec}_{\mathsf{W}^h_{(x:A)}B(x)}(C, c) : (\mathsf{W}_{(x:A)}B(x)) \to C$ is as follows:

- For any $a : A$ and $f : B(a) \to \mathsf{W}^h_{(x:A)}B(x)$ we have

$$\mathsf{rec}_{\mathsf{W}^h_{(x:A)}B(x)}(C, c, \mathsf{sup}(a, f)) =$$
$$c(a, \lambda b.\, \mathsf{rec}_{\mathsf{W}^h_{(x:A)}B(x)}(C, c, f(b)))$$

  i.e., there is a witness $\beta(C, c, a, f)$ for the above equality.

Furthermore, we have the following uniqueness rule, saying that any two functions defined by the same recurrence are equal:

- Let $C : \mathcal{U}$ and $c : \prod_{(a:A)}(B(a) \to C) \to C$ be given. Let $g, h : (\mathsf{W}^h_{(x:A)} B(x)) \to C$ be two functions which satisfy the recurrence c up to propositional equality, i.e., such that we have

$$\beta_g : \prod_{a,f} g(\mathsf{sup}(a, f)) = c(a, \lambda b.\, g(f(b)))$$
$$\beta_h : \prod_{a,f} h(\mathsf{sup}(a, f)) = c(a, \lambda b.\, h(f(b)))$$

  Then g and h are equal, i.e. there is an equality $\alpha(C, c, f, g, \beta_g, \beta_h) : g = h$.

We note that in the dependent case the uniqueness rule is derivable from dependent elimination (Theorem 5.3.1). When the eliminator is simple rather than dependent, this rule is no longer derivable—and in fact, the statement is not even true (exercise). We hence postulate it as an axiom, together with the following coherence law, which tells us how the proof of uniqueness behaves on canonical elements:

- For any $a : A$ and $f : B(a) \to C$, the following diagram commutes propositionally:

$$\begin{array}{ccc} g(\mathsf{sup}(x, f)) & \xrightarrow{\beta_g} & c(a, \lambda b.\, g(f(b))) \\ {\scriptstyle \alpha(\mathsf{sup}(x,f))}\downarrow & & \downarrow{\scriptstyle c(a,-)(\mathsf{funext}\ \lambda b.\, \alpha(f(b)))} \\ h(\mathsf{sup}(x, f)) & \xrightarrow[\beta_h]{} & c(a, \lambda b.\, h(f(b))) \end{array}$$

  where α abbreviates the path $\alpha(C, c, f, g, \beta_g, \beta_h) : g = h$.

Putting all of this data together yields another characterization of $\mathsf{W}_{(x:A)} B(x)$, as a type with a supremum function, satisfying simple elimination, computation, uniqueness, and coherence rules:

$$\begin{aligned} \mathsf{W}_s(A, B) :\equiv \sum_{W:\mathcal{U}} \sum_{\mathsf{sup}:\prod_{(a)}(B(a)\to W)\to W} \prod_{C:\mathcal{U}} \prod_{c:\prod_{(a)}(B(a)\to C)\to C} \sum_{\mathsf{rec}:W\to C} \\ \sum_{\beta:\prod_{(a,f)} \mathsf{rec}(\mathsf{sup}(a,f))=c(a,\lambda b.\, \mathsf{rec}(f(b)))} \prod_{g:W\to C} \prod_{h:W\to C} \prod_{\beta_g:\prod_{(a,f)} g(\mathsf{sup}(a,f))=c(a,\lambda b.\, g(f(b)))} \\ \prod_{\beta_h:\prod_{(a,f)} h(\mathsf{sup}(a,f))=c(a,\lambda b.\, h(f(b)))} \sum_{\alpha:\prod_{(w:W)} g(w)=h(w)} \\ \alpha(\mathsf{sup}(x, f)) \cdot \beta_h = \beta_g \cdot c(a, -)(\mathsf{funext}\ \lambda b.\, \alpha(f(b))) \end{aligned}$$

Theorem 5.5.2. *For any $A : \mathcal{U}$ and $B : A \to \mathcal{U}$, the type $\mathsf{W}_s(A, B)$ is a mere proposition.*

Finally, we have a third, very concise characterization of $\mathsf{W}_{(x:A)} B(x)$ as an h-initial W-algebra:

$$\mathsf{W}_h(A, B) :\equiv \sum_{I:\mathsf{WAlg}(A,B)} \mathsf{isHinit}_{\mathsf{W}}(A, B, I).$$

Theorem 5.5.3. *For any $A : \mathcal{U}$ and $B : A \to \mathcal{U}$, the type $\mathsf{W}_h(A, B)$ is a mere proposition.*

It turns out all three characterizations of W-types are in fact equivalent:

Lemma 5.5.4. *For any $A : \mathcal{U}$ and $B : A \to \mathcal{U}$, we have*

$$\mathsf{W}_d(A, B) \simeq \mathsf{W}_s(A, B) \simeq \mathsf{W}_h(A, B)$$

Indeed, we have the following theorem, which is an improvement over Theorem 5.4.7:

Theorem 5.5.5. *The types satisfying the formation, introduction, elimination, and propositional computation rules for W-types are precisely the homotopy-initial* W*-algebras.*

Sketch of proof. Inspecting the proof of Theorem 5.4.7, we see that only the *propositional* computation rule was required to establish the h-initiality of $\mathsf{W}_{(x:A)} B(x)$. For the converse implication, let us assume that the polynomial functor associated to $A : \mathcal{U}$ and $B : A \to \mathcal{U}$, has an h-initial algebra (W, s_W). To derive the W-formation rule, we let $(\mathsf{W}x : A)B(x) :\equiv W$. The W-introduction rule is equally simple to derive; namely, for $a : A$ and $t : B(a) \to W$, we define $\mathsf{sup}(a, t) : W$ as the result of applying the structure map $s_W : PW \to W$ to $(a, t) : PW$. For the W-elimination rule, let us assume its premisses and in particular that $C' : W \to \mathcal{U}$. Using the other premisses, one shows that the type $C :\equiv \sum_{(w:W)} C'(w)$ can be equipped with a structure map $s_C : PC \to C$. By the h-initiality of W, we obtain a weak homomorphism $(f, s_f) : (W, s_W) \to (C, s_C)$. Furthermore, the first projection $\pi_1 : C \to W$ can be equipped with the structure of a weak homomorphism, so that we obtain a diagram of the form

$$\begin{array}{ccccc} PW & \xrightarrow{Pf} & PC & \xrightarrow{P\pi_1} & PW \\ {\scriptstyle s_W}\downarrow & & \downarrow{\scriptstyle s_C} & & \downarrow{\scriptstyle s_W} \\ W & \xrightarrow[f]{} & C & \xrightarrow[\pi_1]{} & W\,. \end{array}$$

But the identity function $1_W : W \to W$ has a canonical structure of a weak algebra homomorphism and so, by the contractibility of the type of weak homorphisms from (W, s_W) to itself, there must be an identity proof between the composite of (f, s_f) with (π_1, s_{π_1}) and $(1_W, s_{1_W})$. This implies, in particular, that there is an identity proof $p : \pi_1 \circ f = 1_W$.

Since $(\pi_2 \circ f)w : C((\pi_1 \circ f)w)$, we can define

$$\mathsf{rec}(w, c) :\equiv p_*((\pi_2 \circ f)w) : C(w)$$

where the transport p_* is with respect to the family

$$\lambda u.\, C \circ u : (W \to W) \to W \to \mathcal{U}\,.$$

The verification of the propositional W-computation rule is a calculation, involving the naturality properties of operations of the form p_*. □

Finally, as desired, it can be shown that homotopy-natural numbers can be encoded as homotopy-W-types:

Theorem 5.5.6. *The rules for natural numbers with propositional computation rules can be derived from the rules for W-types with propositional computation rules.*

5.6 The general syntax of inductive definitions

So far, we have been discussing only particular inductive types: $\mathbf{0}$, $\mathbf{1}$, $\mathbf{2}$, $\mathbb{N}$, coproducts, products, Σ-types, W-types, etc. However, an important aspect of type theory is the ability to define *new* inductive types, rather than being restricted only to some particular fixed list of them. In order to be able to do this, however, we need to know what sorts of "inductive definitions" are valid or reasonable.

To see that not everything which "looks like an inductive definition" makes sense, consider the following "constructor" of a type C:

- $g : (C \to \mathbb{N}) \to C$.

The recursion principle for such a type C ought to say that given a type P, in order to construct a function $f : C \to P$, it suffices to consider the case when the input $c : C$ is of the form $g(\alpha)$ for some $\alpha : C \to \mathbb{N}$. Moreover, we would expect to be able to use the "recursive data" of f applied to α in some way. However, it is not at all clear how to "apply f to α", since both are functions with domain C.

We could write down a "recursion principle" for C by just supposing (unjustifiably) that there is some way to apply f to α and obtain a function $P \to \mathbb{N}$. Then the input to the recursion rule would ask for a type P together with a function

$$h : (C \to \mathbb{N}) \to (P \to \mathbb{N}) \to P \tag{5.6.1}$$

where the two arguments of h are α and "the result of applying f to α". However, what would the computation rule for the resulting function $f : C \to P$ be? Looking at other computation rules, we would expect something like "$f(g(\alpha)) \equiv h(\alpha, f(\alpha))$" for $\alpha : C \to \mathbb{N}$, but as we have seen, "$f(\alpha)$" does not make sense. The induction principle of C is even more problematic; it's not even clear how to write down the hypotheses. (See also Exercises 5.6 and 5.7.)

This example suggests one restriction on inductive definitions: the domains of all the constructors must be *covariant functors* of the type being defined, so that we can "apply f to them" to get the result of the "recursive call". In other words, if we replace all occurrences of the type being defined with a variable $X : \mathcal{U}$, then each domain of a constructor must be an expression that can be made into a covariant functor of X. This is the case for all the examples we have considered so far. For instance, with the constructor $\mathsf{inl} : A \to A + B$, the relevant functor is constant at A (i.e. $X \mapsto A$), while for the constructor $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$, the functor is the identity functor ($X \mapsto X$).

However, this necessary condition is also not sufficient. Covariance prevents the inductive type from occurring on the left of a single function type, as in the argument $C \to \mathbb{N}$ of the "constructor" g considered above, since this yields a contravariant functor rather than a covariant one. However, since the composite of two contravariant functors is covariant, *double* function types such as $((X \to \mathbb{N}) \to \mathbb{N})$ are once again covariant. This enables us to reproduce Cantorian-style paradoxes.

For instance, consider an "inductive type" D with the following constructor:

- $k : ((D \to \mathsf{Prop}) \to \mathsf{Prop}) \to D$.

Assuming such a type exists, we define functions

$$\begin{aligned} r &: D \to ((D \to \mathsf{Prop}) \to \mathsf{Prop}) \\ r(k(\theta)) &:\equiv \theta \\ f &: (D \to \mathsf{Prop}) \to D \\ f(\delta) &:\equiv k(\lambda x.\, (x = \delta)) \\ p &: (D \to \mathsf{Prop}) \to ((D \to \mathsf{Prop}) \to \mathsf{Prop}) \\ p(\delta) &:\equiv \lambda x.\, \delta(f(x)). \end{aligned}$$

Here r is defined by the recursion principle of D, while f and p are defined explicitly. Then for any $\delta : D \to \mathsf{Prop}$, we have $r(f(\delta)) = \lambda x.\, (x = \delta)$.

In particular, therefore, if $f(\delta) = f(\delta')$, then we have a path $s : (\lambda x.\, (x = \delta)) = (\lambda x.\, (x = \delta'))$. Thus, $\mathsf{happly}(s, \delta) : (\delta = \delta) = (\delta = \delta')$, and so in particular $\delta = \delta'$ holds. Hence, f is "injective" (although *a priori* D may not be a set). This already sounds suspicious — we have an "injection" of the "power set" of D into D — and with a little more work we can massage it into a contradiction.

Suppose given $\theta : (D \to \mathsf{Prop}) \to \mathsf{Prop}$, and define $\delta : D \to \mathsf{Prop}$ by

$$\delta(d) :\equiv \exists(\gamma : D \to \mathsf{Prop}).\, (f(\gamma) = d) \times \theta(\gamma). \tag{5.6.2}$$

We claim that $p(\delta) = \theta$. By function extensionality, it suffices to show $p(\delta)(\gamma) =_{\mathsf{Prop}} \theta(\gamma)$ for any $\gamma : D \to \mathsf{Prop}$. And by univalence, for this it suffices to show that each implies the other. Now by definition of p, we have

$$\begin{aligned} p(\delta)(\gamma) &\equiv \delta(f(\gamma)) \\ &\equiv \exists(\gamma' : D \to \mathsf{Prop}).\, (f(\gamma') = f(\gamma)) \times \theta(\gamma') \end{aligned}$$

Clearly this holds if $\theta(\gamma)$, since we may take $\gamma' :\equiv \gamma$. On the other hand, if we have γ' with $f(\gamma') = f(\gamma)$ and $\theta(\gamma')$, then $\gamma' = \gamma$ since f is injective, hence also $\theta(\gamma)$.

This completes the proof that $p(\delta) = \theta$. Thus, every element $\theta : (D \to \mathsf{Prop}) \to \mathsf{Prop}$ is the image under p of some element $\delta : D \to \mathsf{Prop}$. However, if we define θ by a classic diagonalization:

$$\theta(\gamma) :\equiv \neg p(\gamma)(\gamma) \quad \text{for all } \gamma : D \to \mathsf{Prop}$$

then from $\theta = p(\delta)$ we deduce $p(\delta)(\delta) = \neg p(\delta)(\delta)$. This is a contradiction: no proposition can be equivalent to its negation. (Supposing $P \leftrightarrow \neg P$, if P, then $\neg P$, and so $\mathbf{0}$; hence $\neg P$, but then P, and so $\mathbf{0}$.)

Remark 5.6.3. There is a question of universe size to be addressed. In general, an inductive type must live in a universe that already contains all the types going into its definition. Thus if in the definition of D, the ambiguous notation Prop means $\mathsf{Prop}_{\mathcal{U}}$, then we do not have $D : \mathcal{U}$ but only $D : \mathcal{U}'$ for some larger universe $\mathcal{U}'$ with $\mathcal{U} : \mathcal{U}'$. In a predicative theory, therefore, the right-hand side of (5.6.2) lives in $\mathsf{Prop}_{\mathcal{U}'}$, not $\mathsf{Prop}_{\mathcal{U}}$. So this contradiction does require the resizing rule for mere propositions mentioned in §3.5.

This counterexample suggests that we should ban an inductive type from ever appearing on the left of an arrow in the domain of its constructors, even if that appearance is nested in other arrows so as to eventually become covariant. (Similarly, we also forbid it from appearing in the domain of a dependent function type.) This restriction is called **strict positivity** (ordinary "positivity" being essentially covariance), and it turns out to suffice.

In conclusion, therefore, a valid inductive definition of a type W consists of a list of *constructors*. Each constructor is assigned a type that is a function type taking some number (possibly zero) of inputs (possibly dependent on one another) and returning an element of W. Finally, we allow W itself to occur in the input types of its constructors, but only strictly positively. This essentially means that each argument of a constructor is either a type not involving W, or some iterated function type with codomain W. For instance, the following is a valid constructor type:

$$c : (A \to W) \to (B \to C \to W) \to D \to W \to W. \tag{5.6.4}$$

All of these function types can also be dependent functions (Π-types).[1]

Note we require that an inductive definition is given by a *finite* list of constructors. This is simply because we have to write it down on the page. If we want an inductive type which behaves as if it has an infinite number of constructors, we can simply parametrize one constructor by some infinite type. For instance, a constructor such as $\mathbb{N} \to W \to W$ can be thought of as equivalent to countably many constructors of the form $W \to W$. (Of course, the infinity is now *internal* to the type theory, but this is as it should be for any foundational system.) Similarly, if we want a constructor that takes "infinitely many arguments", we can allow it to take a family of arguments parametrized by some infinite type, such as $(\mathbb{N} \to W) \to W$ which takes an infinite sequence of elements of W.

Now, once we have such an inductive definition, what can we do with it? Firstly, there is a *recursion principle* stating that in order to define a function $f : W \to P$, it suffices to consider the case when the input $w : W$ arises from one of the constructors, allowing ourselves to recursively call f on the inputs to that constructor. For the example constructor (5.6.4), we would require P to be equipped with a function of type

$$d : (A \to W) \to (A \to P) \to (B \to C \to W) \to (B \to C \to P) \to D \to W \to P \to P. \tag{5.6.5}$$

Under these hypotheses, the recursion principle yields $f : W \to P$, which moreover "preserves the constructor data" in the evident way — this is the computation rule, where we use covariance of the inputs. For instance, in the example (5.6.4), the computation rule says that for any $\alpha : A \to W$, $\beta : B \to C \to W$, $\delta : d$, and $\omega : W$, we have

$$f(c(\alpha, \beta, \delta, \omega)) \equiv d(\alpha, f \circ \alpha, \beta, f \circ \beta, \delta, \omega, f(\omega)). \tag{5.6.6}$$

As we have before in particular cases, when defining a particular function f, we may write these rules with $:\equiv$ as a way of specifying the data d, and say that f is defined by them.

[1] In the language of §5.4, the condition of strict positivity ensures that the relevant endofunctor is polynomial. It is well-known in category theory that not *all* endofunctors can have initial algebras; restricting to polynomial functors ensures consistency. One can consider various relaxations of this condition, but in this book we will restrict ourselves to strict positivity as defined here.

The *induction principle* for a general inductive type W is only a little more complicated. Of course, we start with a type family $P : W \to \mathcal{U}$, which we require to be equipped with constructor data "lying over" the constructor data of W. That means the "recursive call" arguments such as $A \to P$ above must be replaced by dependent functions with types such as $\prod_{(a:A)} P(\alpha(a))$. In the full example of (5.6.4), the corresponding hypothesis for the induction principle would require

$$d : \prod_{(\alpha:A\to W)} \left(\prod_{(a:A)} P(\alpha(a)) \right) \to \prod_{(\beta:B\to C\to W)} \left(\prod_{(b:B)} \prod_{(c:C)} P(\beta(b,c)) \right) \to \prod_{(\delta:D)} \prod_{(\omega:W)} P(\omega) \to P(c(\alpha,\beta,\delta,\omega)). \quad (5.6.7)$$

The corresponding computation rule looks identical to (5.6.6). Of course, the recursion principle is the special case of the induction principle where P is a constant family. As we have mentioned before, the induction principle is also called the **eliminator**, and the recursion principle the **non-dependent eliminator**.

As discussed in §1.10, we also allow ourselves to invoke the induction and recursion principles implicitly, writing a definitional equation for each expression that would be the hypotheses of the induction principle. This is called giving a definition by (dependent) **pattern matching**. In our running example, this means we could define $f : \prod_{(w:W)} P(w)$ by

$$f(c(\alpha,\beta,\delta,\omega)) :\equiv \cdots$$

where $\alpha : A \to W$ and $\beta : B \to C \to W$ and $\delta : D$ and $\omega : W$ are variables that are bound in the right-hand side. Moreover, the right-hand side may involve recursive calls to f of the form $f(\alpha(a))$, $f(\beta(b,c))$, and $f(\omega)$. When this definition is repackaged in terms of the induction principle, we replace such recursive calls by $\bar{\alpha}(a)$, $\bar{\beta}(b,c)$, and $\bar{\omega}$, respectively, for new variables

$$\begin{aligned} \bar{\alpha} &: \prod_{a:A} P(\alpha(a)) \\ \bar{\beta} &: \prod_{(b:B)} \prod_{(c:C)} P(\beta(b,c)) \\ \bar{\omega} &: P(\omega). \end{aligned}$$

Then we could write

$$f :\equiv \mathsf{ind}_W(P,\ \lambda\alpha.\,\lambda\bar{\alpha}.\,\lambda\beta.\,\lambda\bar{\beta}.\,\lambda\delta.\,\lambda\omega.\,\lambda\bar{\omega}.\,\cdots)$$

where the second argument to ind_W has the type of (5.6.7).

We will not attempt to give a formal presentation of the grammar of a valid inductive definition and its resulting induction and recursion principles and pattern matching rules. This is possible to do (indeed, it is necessary to do if implementing a computer proof assistant), but provides no additional insight. With practice, one learns to automatically deduce the induction and recursion principles for any inductive definition, and to use them without having to think twice.

5.7 Generalizations of inductive types

The notion of inductive type has been studied in type theory for many years, and admits of many, many generalizations: inductive type families, mutual inductive types, inductive-inductive types, inductive-recursive types, etc. In this section we give an overview of some of these, a few of which will be used later in the book. (In Chapter 6 we will study in more depth a very different generalization of inductive types, which is particular to *homotopy* type theory.)

Most of these generalizations involve allowing ourselves to define more than one type by induction at the same time. One very simple example of this, which we have already seen, is the coproduct $A + B$. It would be tedious indeed if we had to write down separate inductive definitions for $\mathbb{N} + \mathbb{N}$, for $\mathbb{N} + \mathbf{2}$, for $\mathbf{2} + \mathbf{2}$, and so on every time we wanted to consider the coproduct of two types. Instead, we make one definition in which A and B are variables standing for types; in type theory they are called **parameters**. Thus technically speaking, what results from the definition is not a single type, but a family of types $+ : \mathcal{U} \to \mathcal{U} \to \mathcal{U}$, taking two types as input and producing their coproduct. Similarly, the type $\mathsf{List}(A)$ of lists is a family $\mathsf{List}(-) : \mathcal{U} \to \mathcal{U}$ in which the type A is a parameter.

In mathematics, this sort of thing is so obvious as to not be worth mentioning, but we bring it up in order to contrast it with the next example. Note that each type $A + B$ is *independently* defined inductively, as is each type $\mathsf{List}(A)$. By contrast, we might also consider defining a whole type family $B : A \to \mathcal{U}$ by induction *together*. The difference is that now the constructors may change the index $a : A$, and as a consequence we cannot say that the individual types $B(a)$ are inductively defined, only that the entire family is inductively defined.

The standard example is the type of *lists of specified length*, traditionally called *vectors*. We fix a parameter type A, and define a type family $\mathsf{Vec}_n(A)$, for $n : \mathbb{N}$, generated by the following constructors.

- There is a vector $\mathsf{nil} : \mathsf{Vec}_0(A)$ of length zero.
- For any $n : \mathbb{N}$, there is a function $\mathsf{cons} : A \to \mathsf{Vec}_n(A) \to \mathsf{Vec}_{\mathsf{succ}(n)}(A)$.

In contrast to lists, vectors (with elements from a fixed type A) form a family of types indexed by their length. While A is a parameter, we say that $n : \mathbb{N}$ is an **index** of the inductive family. An individual type such as $\mathsf{Vec}_3(A)$ is not inductively defined: the constructors which build elements of $\mathsf{Vec}_3(A)$ take input from a different type in the family, such as $\mathsf{cons} : A \to \mathsf{Vec}_2(A) \to \mathsf{Vec}_3(A)$.

In particular, the induction principle must refer to the entire type family as well; thus the hypotheses and the conclusion must quantify over the indices appropriately. In the case of vectors, the induction principle states that given a type family $C : \prod_{(n:\mathbb{N})} \mathsf{Vec}_n(A) \to \mathcal{U}$, together with

- an element $c_{\mathsf{nil}} : C(0, \mathsf{nil})$, and
- a function $c_{\mathsf{cons}} : \prod_{(n:\mathbb{N})} \prod_{(a:A)} \prod_{(\ell:\mathsf{Vec}_n(A))} C(n, \ell) \to C(\mathsf{succ}(n), \mathsf{cons}(a, \ell))$

there exists a function $f : \prod_{(n:\mathbb{N})} \prod_{(\ell:\mathsf{Vec}_n(A))} C(n, \ell)$ such that

$$f(0, \mathsf{nil}) \equiv c_{\mathsf{nil}}$$
$$f(\mathsf{succ}(n), \mathsf{cons}(a, \ell)) \equiv c_{\mathsf{cons}}(n, a, \ell, f(\ell)).$$

One use of inductive families is to define *predicates* inductively. For instance, we might define the predicate $\mathsf{iseven} : \mathbb{N} \to \mathcal{U}$ as an inductive family indexed by $\mathbb{N}$, with the following constructors:

- An element $\mathsf{even}_0 : \mathsf{iseven}(0)$
- A function $\mathsf{even}_{ss} : \prod_{(n:\mathbb{N})} \mathsf{iseven}(n) \to \mathsf{iseven}(\mathsf{succ}(\mathsf{succ}(n)))$.

In other words, we stipulate that 0 is even, and that if n is even then so is $\mathsf{succ}(\mathsf{succ}(n))$. These constructors "obviously" give no way to construct an element of, say, $\mathsf{iseven}(1)$, and since iseven is supposed to be freely generated by these constructors, there must be no such element. (Actually proving that $\neg\mathsf{iseven}(1)$ is not entirely trivial, however). The induction principle for iseven says that to prove something about all even natural numbers, it suffices to prove it for 0 and verify that it is preserved by adding two.

Inductively defined predicates are much used in computer formalization of mathematics and software verification. But we will not have much use for them, with one exception in §10.3.

Another important special case is when the indexing type of an inductive family is finite. In this case, we can equivalently express the inductive definition as a finite collection of types defined by *mutual induction*. For instance, we might define the types even and odd of even and odd natural numbers by mutual induction, where even is generated by constructors

- $0 : \mathsf{even}$ and
- $\mathsf{esucc} : \mathsf{odd} \to \mathsf{even}$

while odd is generated by the one constructor

- $\mathsf{osucc} : \mathsf{even} \to \mathsf{odd}$.

Note that even and odd are simple types (not type families), but their constructors can refer to each other. If we expressed this definition as an inductive type family $\mathsf{paritynat} : \mathbf{2} \to \mathcal{U}$, with $\mathsf{paritynat}(0_{\mathbf{2}})$ and $\mathsf{paritynat}(1_{\mathbf{2}})$ representing even and odd respectively, it would instead have constructors:

- $0 : \mathsf{paritynat}(0_{\mathbf{2}})$
- $\mathsf{esucc} : \mathsf{paritynat}(0_{\mathbf{2}}) \to \mathsf{paritynat}(1_{\mathbf{2}})$
- $\mathsf{oesucc} : \mathsf{paritynat}(1_{\mathbf{2}}) \to \mathsf{paritynat}(0_{\mathbf{2}})$

When expressed explicitly as a mutual inductive definition, the induction principle for even and odd says that given $C : \mathsf{even} \to \mathcal{U}$ and $D : \mathsf{odd} \to \mathcal{U}$, along with

- $c_0 : C(0)$
- $c_s : \prod_{(n:\mathsf{odd})} D(n) \to C(\mathsf{esucc}(n))$
- $d_s : \prod_{(n:\mathsf{even})} C(n) \to D(\mathsf{osucc}(n))$

there exist $f : \prod_{(n:\mathsf{even})} C(n)$ and $g : \prod_{(n:\mathsf{odd})} D(n)$ such that

$$\begin{aligned} f(0) &\equiv c_0 \\ f(\mathsf{esucc}(n)) &\equiv c_s(g(n)) \\ g(\mathsf{osucc}(n)) &\equiv d_s(f(n)). \end{aligned}$$

In particular, just as we can only induct over an inductive family "all at once", we have to induct on even and odd simultaneously. We will not have much use for mutual inductive definitions in this book either.

A further, more radical, generalization is to allow definition of a type family $B : A \to \mathcal{U}$ in which not only the types $B(a)$, but the type A itself, is defined as part of one big induction. In other words, not only do we specify constructors for the $B(a)$s which can take inputs from other $B(a')$s, as with inductive families, we also at the same time specify constructors for A itself, which can take inputs from the $B(a)$s. This can be regarded as an inductive family in which the indices are inductively defined simultaneously with the indexed types, or as a mutual inductive definition in which one of the types can depend on the other. More complicated dependency structures are also possible. In general, these are called **inductive-inductive definitions**. For the most part, we will not use them in this book, but their higher variant (see Chapter 6) will appear in a couple of experimental examples in Chapter 11.

The last generalization we wish to mention is **inductive-recursive definitions**, in which a type is defined inductively at the same time as a *recursive* function on it. That is, we fix a known type P, and give constructors for an inductive type A and at the same time define a function $f : A \to P$ using the recursion principle for A resulting from its constructors — with the twist that the constructors of A are allowed to refer also to the values of f. We do not yet know how to justify such definitions from a homotopical perspective, and we will not use any of them in this book.

5.8 Identity types and identity systems

We now wish to point out that the *identity types*, which play so central a role in homotopy type theory, may also be considered to be defined inductively. Specifically, they are an "inductive family" with indices, in the sense of §5.7. In fact, there are *two* ways to describe identity types as an inductive family, resulting in the two induction principles described in Chapter 1, path induction and based path induction.

In both definitions, the type A is a parameter. For the first definition, we inductively define a family $=_A \colon A \to A \to \mathcal{U}$, with two indices belonging to A, by the following constructor:

- For any $a : A$, an element $\mathsf{refl}_A : a =_A a$.

By analogy with the other inductive families, we may extract the induction principle from this definition. It states that given any $C : \prod_{(a,b:A)}(a =_A b) \to \mathcal{U}$, along with $d : \prod_{(a:A)} C(a, a, \mathsf{refl}_a)$, there exists $f : \prod_{(a,b:A)} \prod_{(p:a=_A b)} C(a, b, p)$ such that $f(a, a, \mathsf{refl}_a) \equiv d(a)$. This is exactly the path induction principle for identity types.

For the second definition, we consider one element $a_0 : A$ to be a parameter along with $A : \mathcal{U}$, and we inductively define a family $(a_0 =_A -) : A \to \mathcal{U}$, with *one* index belonging to A, by the following constructor:

- An element $\mathsf{refl}_{a_0} : a_0 =_A a_0$.

Note that because $a_0 : A$ was fixed as a parameter, the constructor refl_{a_0} does not appear inside the inductive definition as a function, but only an element. The induction principle for this

definition says that given $C : \prod_{(b:A)}(a_0 =_A b) \to \mathcal{U}$ along with an element $d : C(a_0, \mathsf{refl}_{a_0})$, there exists $f : \prod_{(b:A)} \prod_{(p:a_0=_A b)} C(b,p)$ with $f(a_0, \mathsf{refl}_{a_0}) \equiv d$. This is exactly the based path induction principle for identity types.

The view of identity types as inductive types has historically caused some confusion, because of the intuition mentioned in §5.1 that all the elements of an inductive type should be obtained by repeatedly applying its constructors. For ordinary inductive types such as $\mathbf{2}$ and $\mathbb{N}$, this is the case: we saw in **??** that indeed every element of $\mathbf{2}$ is either $1_{\mathbf{2}}$ or $0_{\mathbf{2}}$, and similarly one can prove that every element of $\mathbb{N}$ is either 0 or a successor.

However, this is *not* true for identity types: there is only one constructor refl, but not every path is equal to the constant path. More precisely, we cannot prove, using only the induction principle for identity types (either one), that every inhabitant of $a =_A a$ is equal to refl_a. In order to actually exhibit a counterexample, we need some additional principle such as the univalence axiom — recall that in Example 3.1.9 we used univalence to exhibit a particular path $\mathbf{2} =_{\mathcal{U}} \mathbf{2}$ which is not equal to $\mathsf{refl}_{\mathbf{2}}$.

The point is that, as validated by the study of homotopy-initial algebras, an inductive definition should be regarded as *freely generated* by its constructors. Of course, a freely generated structure may contain elements other than its generators: for instance, the free group on two symbols x and y contains not only x and y but also words such as xy, $yx^{-1}y$, and $x^3y^2x^{-2}yx$. In general, the elements of a free structure are obtained by applying not only the generators, but also the operations of the ambient structure, such as the group operations if we are talking about free groups.

In the case of inductive types, we are talking about freely generated *types* — so what are the "operations" of the structure of a type? If types are viewed as like *sets*, as was traditionally the case in type theory, then there are no such operations, and hence we expect there to be no elements in an inductive type other than those resulting from its constructors. In homotopy type theory, we view types as like *spaces* or ∞-groupoids, in which case there are many operations on the *paths* (concatenation, inversion, etc.) — this will be important in Chapter 6 — but there are still no operations on the *objects* (elements). Thus, it is still true for us that, e.g., every element of $\mathbf{2}$ is either $1_{\mathbf{2}}$ or $0_{\mathbf{2}}$, and every element of $\mathbb{N}$ is either 0 or a successor.

However, as we saw in Chapter 2, viewing types as ∞-groupoids entails also viewing functions as functors, and this includes type families $B : A \to \mathcal{U}$. Thus, the identity type $(a_0 =_A -)$, viewed as an inductive type family, is actually a *freely generated functor* $A \to \mathcal{U}$. Specifically, it is the functor $F : A \to \mathcal{U}$ freely generated by one element $\mathsf{refl}_{a_0} : F(a_0)$. And a functor does have operations on objects, namely the action of the morphisms (paths) of A.

In category theory, the *Yoneda lemma* tells us that for any category A and object a_0, the functor freely generated by an element of $F(a_0)$ is the representable functor $\hom_A(a_0, -)$. Thus, we should expect the identity type $(a_0 =_A -)$ to be this representable functor, and this is indeed exactly how we view it: $(a_0 =_A b)$ is the space of morphisms (paths) in A from a_0 to b.

One reason for viewing identity types as inductive families is to apply the uniqueness principles of §§5.2 and 5.5. Specifically, we can characterize the family of identity types of a type A, up to equivalence, by giving another family of types over $A \times A$ satisfying the same induction principle. This suggests the following definitions and theorem.

Definition 5.8.1. Let A be a type and $a_0 : A$ an element.

- A **pointed predicate** over (A, a_0) is a family $R : A \to \mathcal{U}$ equipped with an element $r_0 : R(a_0)$.
- For pointed predicates (R, r_0) and (S, s_0), a family of maps $g : \prod_{(b:A)} R(b) \to S(b)$ is **pointed** if $g(r_0) = s_0$. We have

$$\mathsf{ppmap}(R, S) :\equiv \sum_{g:\prod_{(b:A)} R(b)\to S(b)} (g(r_0) = s_0).$$

- An **identity system at** a_0 is a pointed predicate (R, r_0) such that for any type family $D : \prod_{(b:A)} R(b) \to \mathcal{U}$ and $d : D(a_0, r_0)$, there exists a function $f : \prod_{(b:A)} \prod_{(r:R(b))} D(b, r)$ such that $f(a_0, r_0) = d$.

Theorem 5.8.2. *For a pointed predicate (R, r_0), the following are logically equivalent.*

(i) *(R, r_0) is an identity system at a_0.*
(ii) *For any pointed predicate (S, s_0), the type $\mathsf{ppmap}(R, S)$ is contractible.*
(iii) *For any $b : A$, the function $\mathsf{transport}^R(-, r_0) : (a_0 =_A b) \to R(b)$ is an equivalence.*
(iv) *The type $\sum_{(b:A)} R(b)$ is contractible.*

Note that the equivalences (i)⇔(ii)⇔(iii) are a version of Lemma 5.5.4 for identity types $a_0 =_A -$, regarded as inductive families varying over one element of A. Of course, (ii)–(iv) are mere propositions, so that logical equivalence implies actual equivalence. (Condition (i) is also a mere proposition, but we will not prove this.)

Proof. First, assume (i) and let (S, s_0) be a pointed predicate. Define $D(b, r) :\equiv S(b)$ and $d :\equiv s_0 : S(a_0) \equiv D(a_0, r_0)$. Since R is an identity system, we have $f : \prod_{(b:A)} R(b) \to S(b)$ with $f(a_0, r_0) = s_0$; hence $\mathsf{ppmap}(R, S)$ is inhabited. Now suppose $(f, f_r), (g, g_r) : \mathsf{ppmap}(R, S)$, and define $D(b, r) :\equiv (f(b, r) = g(b, r))$, and let $d :\equiv f_r \cdot {g_r}^{-1} : f(a_0, r_0) = s_0 = g(a_0, r_0)$. Then again since R is an identity system, we have $h : \prod_{(b:A)} \prod_{(r:R(b))} D(b, r)$ such that $h(a_0, r_0) = f_r \cdot {g_r}^{-1}$. By the characterization of paths in Σ-types and path types, these data yield an equality $(f, f_r) = (g, g_r)$. Hence $\mathsf{ppmap}(R, S)$ is an inhabited mere proposition, and thus contractible; so (ii) holds.

Now suppose (ii), and define $S(b) :\equiv (a_0 = b)$ with $s_0 :\equiv \mathsf{refl}_{a_0} : S(a_0)$. Then (S, s_0) is a pointed predicate, and $\lambda b.\, \lambda p.\, \mathsf{transport}^R(p, r) : \prod_{(b:A)} S(b) \to R(b)$ is a pointed family of maps from S to R. By assumption, $\mathsf{ppmap}(R, S)$ is contractible, hence inhabited, so there also exists a pointed family of maps from R to S. And the composites in either direction are pointed families of maps from R to R and from S to S, respectively, hence equal to identities since $\mathsf{ppmap}(R, R)$ and $\mathsf{ppmap}(S, S)$ are contractible. Thus (iii) holds.

Now supposing (iii), condition (iv) follows from Lemma 3.11.8, using the fact that Σ-types respect equivalences (the "if" direction of Theorem 4.7.7).

Finally, assume (iv), and let $D : \prod_{(b:A)} R(b) \to \mathcal{U}$ and $d : D(a_0, r_0)$. We can equivalently express D as a family $D' : \sum_{(b:A)} R(b) \to \mathcal{U}$. Now since $\sum_{(b:A)} R(b)$ is contractible, we have

$$p : \prod_{u:\sum_{(b:A)} R(b)} (a_0, r_0) = u.$$

Moreover, since the path types of a contractible type are again contractible, we have $p((a_0, r_0)) = \mathsf{refl}_{(a_0,r_0)}$. Define $f(u) :\equiv \mathsf{transport}^{D'}(p(u), d)$, yielding $f : \prod_{(u:\sum_{(b:A)} R(b))} D'(u)$, or equivalently $f : \prod_{(b:A)} \prod_{(r:R(b))} D(b, r)$. Finally, we have

$$f(a_0, r_0) \equiv \mathsf{transport}^{D'}(p((a_0, r_0)), d) = \mathsf{transport}^{D'}(\mathsf{refl}_{(a_0,r_0)}, d) = d.$$

Thus, (i) holds. □

We can deduce a similar result for identity types $- =_A -$, regarded as a family varying over two elements of A.

Definition 5.8.3. An **identity system** over a type A is a family $R : A \to A \to \mathcal{U}$ equipped with a function $r_0 : \prod_{(a:A)} R(a, a)$ such that for any type family $D : \prod_{(a,b:A)} R(a, b) \to \mathcal{U}$ and $d : \prod_{(a:A)} D(a, a, r_0(a))$, there exists a function $f : \prod_{(a,b:A)} \prod_{(r:R(b))} D(a, b, r)$ such that $f(a, a, r_0(a)) = d(a)$ for all $a : A$.

Theorem 5.8.4. *For $R : A \to A \to \mathcal{U}$ equipped with $r_0 : \prod_{(a:A)} R(a, a)$, the following are logically equivalent.*

(i) *(R, r_0) is an identity system over A.*

(ii) *For all $a_0 : A$, the pointed predicate $(R(a_0), r_0(a_0))$ is an identity system at a_0.*

(iii) *For any $S : A \to A \to \mathcal{U}$ and $s_0 : \prod_{(a:A)} S(a, a)$, the type*

$$\sum_{(g:\prod_{(a,b:A)} R(a,b)\to S(a,b))} \prod_{(a:A)} g(a, a, r_0(a)) = s_0(a)$$

is contractible.

(iv) *For any $a, b : A$, the map $\mathsf{transport}^{R(a)}(-, r_0(a)) : (a =_A b) \to R(a, b)$ is an equivalence.*

(v) *For any $a : A$, the type $\sum_{(b:A)} R(a, b)$ is contractible.*

Proof. The equivalence (i)⇔(ii) follows exactly the proof of equivalence between the path induction and based path induction principles for identity types; see §1.12. The equivalence with (iv) and (v) then follows from Theorem 5.8.2, while (iii) is straightforward. □

One reason this characterization is interesting is that it provides an alternative way to state univalence and function extensionality. The univalence axiom for a universe $\mathcal{U}$ says exactly that the type family

$$(- \simeq -) : \mathcal{U} \to \mathcal{U} \to \mathcal{U}$$

together with $\mathsf{id}_- : \prod_{(A:\mathcal{U})}(A \simeq A)$ satisfies Theorem 5.8.4(iv). Therefore, it is equivalent to the corresponding version of (i), which we can state as follows.

Corollary 5.8.5 (Equivalence induction)**.** *Given any type family $D : \prod_{(A,B:\mathcal{U})}(A \simeq B) \to \mathcal{U}$ and function $d : \prod_{(A:\mathcal{U})} D(A, A, \mathsf{id}_A)$, there exists $f : \prod_{(A,B:\mathcal{U})} \prod_{(e:A\simeq B)} D(A, B, e)$ such that $f(A, A, \mathsf{id}_A) = d(A)$ for all $A : \mathcal{U}$.*

In other words, to prove something about all equivalences, it suffices to prove it about identity maps. We have already used this principle (without stating it in generality) in Lemma 4.1.1.

Similarly, function extensionality says that for any $B : A \to \mathcal{U}$, the type family

$$(- \sim -) : \left(\prod_{a:A} B(a)\right) \to \left(\prod_{a:A} B(a)\right) \to \mathcal{U}$$

together with $\lambda f.\, \lambda a.\, \mathsf{refl}_{f(a)}$ satisfies Theorem 5.8.4(iv). Thus, it is also equivalent to the corresponding version of (i).

Corollary 5.8.6 (Homotopy induction)**.** *Given any* $D : \prod_{(f,g:\prod_{(a:A)} B(a))} (f \sim g) \to \mathcal{U}$ *and* $d : \prod_{(f:\prod_{(a:A)} B(a))} D(f, f, \lambda x.\, \mathsf{refl}_{f(x)})$*, there exists* $k : \prod_{(f,g:\prod_{(a:A)} B(a))} \prod_{(h:f\sim g)} D(f, g, h)$ *such that* $k(f, f, \lambda x.\, \mathsf{refl}_{f(x)}) = d(f)$ *for all* f*.*

Notes

Inductive definitions have a long pedigree in mathematics, arguably going back at least to Frege and Peano's axioms for the natural numbers. More general "inductive predicates" are not uncommon, but in set theoretic foundations they are usually constructed explicitly, either as an intersection of an appropriate class of subsets or using transfinite iteration along the ordinals, rather than regarded as a basic notion.

In type theory, particular cases of inductive definitions date back to Martin-Löf's original papers: [ML71] presents a general notion of inductively defined predicates and relations; the notion of inductive type was present (but only with instances, not as a general notion) in Martin-Löf's first papers in type theory [ML75a]; and then as a general notion with W-types in [ML82].

A general notion of inductive type was introduced in 1985 by Constable and Mendler [CM85]. A general schema for inductive types in intensional type theory was suggested in [PPM90]. Further developments included [CP90, Dyb91].

The notion of inductive-recursive definition appears in [Dyb00]. An important type-theoretic notion is the notion of tree types (a general expression of the notion of Post system in type theory) which appears in [PS89].

The universal property of the natural numbers as an initial object of the category of $\mathbb{N}$-algebras is due to Lawvere [Law06]. This was later generalized to a description of W-types as initial algebras for polynomial endofunctors by [MP00]. The coherently homotopy-theoretic equivalence between such universal properties and the corresponding induction principles (§§5.4 and 5.5) is due to [AGS12].

For actual constructions of inductive types in homotopy-theoretic semantics of type theory, see [KLV12, MvdB13, LS13b].

Exercises

Exercise 5.1. Construct two functions on natural numbers which satisfy the same recurrence (e_z, e_s) but are not definitionally equal.

Exercise 5.2. Construct two different recurrences (e_z, e_s) on the same type E which are both satisfied by the same function $f : \mathbb{N} \to E$.

Exercise 5.3. Show that for any type family $E : \mathbf{2} \to \mathcal{U}$, the induction operator

$$\mathsf{ind}_{\mathbf{2}}(E) : \big(E(1_{\mathbf{2}}) \times E(0_{\mathbf{2}})\big) \to \prod_{b:\mathbf{2}} E(b)$$

is an equivalence.

Exercise 5.4. Show that the analogous statement to Exercise 5.3 for $\mathbb{N}$ fails.

Exercise 5.5. Show that if we assume simple instead of dependent elimination for W-types, the uniqueness property (analogue of Theorem 5.3.1) fails to hold. That is, exhibit a type satisfying the recursion principle of a W-type, but for which functions are not determined uniquely by their recurrence.

Exercise 5.6. Suppose that in the "inductive definition" of the type C at the beginning of §5.6, we replace the type $\mathbb{N}$ by $\mathbf{0}$. Using only a "recursion principle" for such a definition with hypotheses analogous to (5.6.1), construct an element of $\mathbf{0}$.

Exercise 5.7. Similarly to the previous exercise, derive a contradiction from an "inductive type" D with one constructor $(D \to D) \to D$.

Chapter 6

Higher inductive types

6.1 Introduction

Like the general inductive types we discussed in Chapter 5, *higher inductive types* are a general schema for defining new types generated by some constructors. But unlike ordinary inductive types, in defining a higher inductive type we may have "constructors" which generate not only *points* of that type, but also *paths* and higher paths in that type. For instance, we can consider the higher inductive type $\mathbb{S}^1$ generated by

- A point $\mathsf{base} : \mathbb{S}^1$, and
- A path $\mathsf{loop} : \mathsf{base} =_{\mathbb{S}^1} \mathsf{base}$.

This should be regarded as entirely analogous to the definition of, for instance, $\mathbf{2}$, as being generated by

- A point $0_{\mathbf{2}} : \mathbf{2}$ and
- A point $1_{\mathbf{2}} : \mathbf{2}$,

or the definition of $\mathbb{N}$ as generated by

- A point $0 : \mathbb{N}$ and
- A function $s : \mathbb{N} \to \mathbb{N}$.

When we think of types as higher groupoids, the more general notion of "generation" is very natural: since a higher groupoid is a "multi-sorted object" with paths and higher paths as well as points, we should allow "generators" in all dimensions.

We will refer to the ordinary sort of constructors (such as base) as *point constructors* or *ordinary constructors*, and to the others (such as loop) as *path constructors* or *higher constructors*. Note that a path constructor such as loop generates a *new* inhabitant of an identity type, which is not (at least, not *a priori*) equal to any previously existing such inhabitant. In particular, loop is not *a priori* equal to $\mathsf{refl}_{\mathsf{base}}$ (although proving that they are definitely unequal takes a little thought; see Lemma 6.4.1). This is what distinguishes $\mathbb{S}^1$ from the ordinary inductive type $\mathbf{1}$.

There are some important points to be made regarding this generalization.

First of all, the word "generation" should be taken seriously, in the same sense that a group can be freely generated by some set. In particular, because a higher groupoid comes with *operations* on paths and higher paths, when such an object is "generated" by certain constructors, the operations will create more paths that don't come directly from the constructors themselves. For instance, in the higher inductive type $\mathbb{S}^1$, the constructor loop is not the only nontrivial path from base to base; we have also $\mathsf{loop} \cdot \mathsf{loop}$ and $\mathsf{loop} \cdot \mathsf{loop} \cdot \mathsf{loop}$ and so on, as well as loop^{-1}, etc., all of which are different. This may seem so obvious as to be not worth mentioning, but it is a departure from the behavior of "ordinary" inductive types, where one can expect to see nothing in the inductive type except what was "put in" directly by the constructors.

Secondly, this generation is really *free* generation: higher inductive types do not technically allow us to impose "axioms", such as forcing $\mathsf{loop} \cdot \mathsf{loop}$ to equal $\mathsf{refl}_{\mathsf{base}}$. However, in the world of ∞-groupoids, there is little difference between "free generation" and "presentation", since we can make two paths equal *up to homotopy* by adding a new 2-dimensional generator relating them (e.g. a path $\mathsf{loop} \cdot \mathsf{loop} = \mathsf{refl}_{\mathsf{base}}$ in $\mathsf{base} = \mathsf{base}$). We do then, of course, have to worry about whether this new generator should satisfy its own "axioms", and so on, but in principle any "presentation" can be transformed into a "free" one by making axioms into constructors. As we will see, by adding "truncation constructors" we can use higher inductive types to express classical notions such as group presentations as well.

Thirdly, even though a higher inductive type contains "constructors" which generate *paths in* that type, it is still an inductive definition of a *single* type. In particular, as we will see, it is the higher inductive type itself which will be given a universal property (expressed, as usual, by an induction principle), and *not* its identity types. The identity type of a higher inductive type retains the usual induction principle of any identity type (i.e. path induction), and does not acquire any new induction principle.

Thus, it may be nontrivial to identify the identity types of a higher inductive type in a concrete way, in contrast to how in Chapter 2 we were able to give explicit descriptions of the behavior of identity types under all the traditional type forming operations. For instance, are there any paths from base to base in $\mathbb{S}^1$ which are not simply composites of copies of loop and its inverse? Intuitively, it seems that the answer should be no (and it is), but proving this is not trivial. Indeed, such questions bring us rapidly to problems such as calculating the homotopy groups of spheres, a long-standing problem in algebraic topology for which no simple formula is known. Homotopy type theory brings a new and powerful viewpoint to bear on such questions, but it also requires type theory to become as complex as the answers to these questions.

Fourthly, the "dimension" of the constructors (i.e. whether they output points, paths, paths between paths, etc.) does not have a direct connection to which dimensions the resulting type has nontrivial homotopy in. As a simple example, if an inductive type B has a constructor of type $A \to B$, then any paths and higher paths in A will result in paths and higher paths in B, even though the constructor is not a "higher" constructor at all. The same thing happens with higher constructors too: having a constructor of type $A \to (x =_B y)$ means not only that points of A yield paths from x to y in B, but that paths in A yield paths between these paths, and so on. As we will see, this possibility is responsible for much of the power of higher inductive types.

On the other hand, it is even possible for constructors *without* higher types in their inputs to generate "unexpected" higher paths. For instance, in the 2-dimensional sphere $\mathbb{S}^2$ generated by

- A point $\mathsf{base} : \mathbb{S}^2$, and
- A 2-dimensional path $\mathsf{surf} : \mathsf{refl}_{\mathsf{base}} = \mathsf{refl}_{\mathsf{base}}$ in $\mathsf{base} = \mathsf{base}$,

there is a nontrivial *3-dimensional path* from $\mathsf{refl}_{\mathsf{refl}_{\mathsf{base}}}$ to itself. Topologists will recognize this path as an incarnation of the *Hopf fibration*. From a category-theoretic point of view, this is the same sort of phenomenon as the fact mentioned above that $\mathbb{S}^1$ contains not only loop but also $\mathsf{loop} \cdot \mathsf{loop}$ and so on: it's just that in a *higher* groupoid, there are *operations* which raise dimension. Indeed, we saw many of these operations back in §2.1: the associativity and unit laws are not just properties, but operations, whose inputs are 1-paths and whose outputs are 2-paths.

6.2 Induction principles and dependent paths

When we describe a higher inductive type such as the circle as being generated by certain constructors, we have to explain what this means by giving rules analogous to those for the basic type constructors from Chapter 1. The constructors themselves give the *introduction* rules, but it requires a bit more thought to explain the *elimination* rules. In this book we will not attempt to give a general formulation of what constitutes a "higher inductive definition" and how to extract the elimination rule from such a definition — indeed, this is a subtle question and the subject of current research. Instead we will rely on some general informal discussion and numerous examples.

The non-dependent form of the eliminator is usually easy to describe: given any type equipped with the same structure with which the constructors equip the higher inductive type in question, there is a function which maps the constructors to that structure. For instance, in the case of $\mathbb{S}^1$, the non-dependent eliminator says that given any type B equipped with a point $b : B$ and a path $\ell : b = b$, there is a function $f : \mathbb{S}^1 \to B$ such that $f(\mathsf{base}) = b$ and $\mathsf{ap}_f(\mathsf{loop}) = \ell$.

The latter two equalities are the *computation rules*. There is, however, a question of whether these computation rules are judgmental equalities or propositional equalities (paths). For ordinary inductive types, we had no qualms about making them judgmental, although we saw in Chapter 5 that making them propositional would still yield the same type up to equivalence. In the ordinary case, one may argue that the computation rules are really *definitional* equalities, in the intuitive sense described in the Introduction.

For higher inductive types, this is less clear. Moreover, since the operation ap_f is not really a fundamental part of the type theory, but something that we *defined* using the induction principle of identity types (and which we might have defined in some other, equivalent, way), it seems odd to refer to it explicitly in a *judgmental* equality.

It does seem unproblematic, both philosophically and semantically, to make the computational rules for the *point* constructors of a higher inductive type judgmental. In the example above, this means we have $f(\mathsf{base}) \equiv b$, judgmentally. Moreover, this choice greatly simplifies our lives, since otherwise the second computation rule $\mathsf{ap}_f(\mathsf{loop}) = \ell$ would not even be well-typed as a propositional equality; we would have to compose one side or the other with the specified identification of $f(\mathsf{base})$ with b. (Such problems do arise eventually, of course, when we come to talk about paths of higher dimension, but that will not be of great concern to us here. See also §6.7.) Thus, we will take the point-constructor computation rules to be judgmental, and

those for paths and higher paths to be propositional.[1]

Remark 6.2.1. Recall that for ordinary inductive types, we regard the computation rules for a recursively defined function as not merely judgmental equalities, but *definitional* ones, and thus we may use the notation $:\equiv$ for them. For instance, the truncated predecessor function $p : \mathbb{N} \to \mathbb{N}$ is defined by $p(0) :\equiv 0$ and $p(\mathsf{succ}(n)) :\equiv n$. In the case of higher inductive types, this sort of notation is reasonable for the point constructors (e.g. $f(\mathsf{base}) :\equiv b$), but for the path constructors it could be misleading, since equalities such as $f(\mathsf{loop}) = \ell$ are not judgmental. Thus, we hybridize the notations, writing instead $f(\mathsf{loop}) := \ell$ for this sort of "propositional equality by definition".

Now, what about the dependent eliminator (the induction principle)? Recall that for an ordinary inductive type W, to prove by induction that $\prod_{(x:W)} P(x)$, we must specify for each constructor of W, an operation on P which acts on the "fibers" above that constructor in W. For instance, if W is the natural numbers $\mathbb{N}$, then to prove by induction that $\prod_{(x:\mathbb{N})} P(x)$, we must specify

- An element $b : P(0)$ in the fiber over the constructor $0 : \mathbb{N}$, and
- For each $n : \mathbb{N}$, a function $P(n) \to P(\mathsf{succ}(n))$.

The second can be viewed as a function "$P \to P$" lying *over* the constructor $s : \mathbb{N} \to \mathbb{N}$, generalizing how $b : P(0)$ lies over the constructor $0 : \mathbb{N}$.

By analogy, therefore, to prove that $\prod_{(x:\mathbb{S}^1)} P(x)$, we should specify

- An element $b : P(\mathsf{base})$ in the fiber over the constructor $\mathsf{base} : \mathbb{N}$, and
- A path from b to b "lying over the constructor $\mathsf{loop} : \mathsf{base} = \mathsf{base}$".

Note that even though $\mathbb{S}^1$ contains paths other than loop (such as $\mathsf{refl}_{\mathsf{base}}$ and $\mathsf{loop} \cdot \mathsf{loop}$), we only need to specify a path lying over the constructor *itself*. This expresses the intuition that $\mathbb{S}^1$ is "freely generated" by its constructors.

The question, however, is what it means to have a path "lying over" another path. It definitely does *not* mean simply a path $b = b$, since that would be a path in the fiber $P(\mathsf{base})$ (topologically, a path lying over the *constant* path at base). Actually, however, we have already answered this question in Chapter 2: in the discussion preceeding Lemma 2.3.4 we concluded that a path from $u : P(x)$ to $v : P(y)$ lying over $p : x = y$ can be represented by a path $p_*(u) = v$ in the fiber $P(y)$. Since we will have a lot of use for such *dependent paths* in this chapter, we introduce a special notation for them:

$$(u =^P_p v) :\equiv (\mathsf{transport}^P(p, u) = v).$$

Remark 6.2.2. There are other possible ways to define dependent paths. For instance, instead of $p_*(u) = v$ we could consider $u = (p^{-1})_*(v)$. We could also obtain it as a special case of a more general "heterogeneous equality", or with a direct definition as an inductive type family.

[1] In particular, in the language of §1.1, this means that our higher inductive types are a mix of *rules* (specifying how we can introduce such types and their elements, their induction principle, and their computation rules for point-constructors) and *axioms* (the computation rules for path-constructors, which assert that certain identity types are inhabited by otherwise unspecified terms). We hope that eventually, there will be a better type theory in which higher inductive types, like univalence, will be presented using only rules and no axioms.

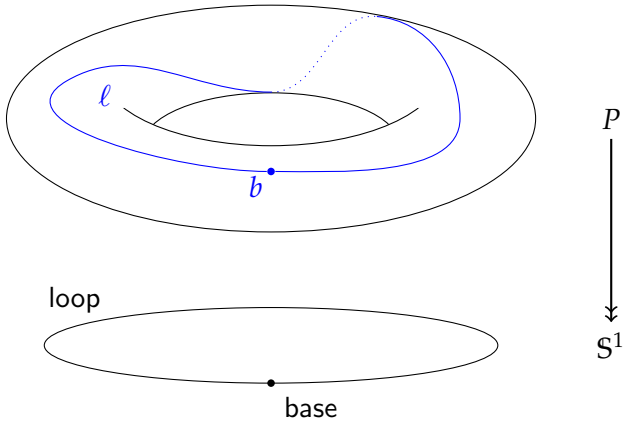


Figure 6.1: The topological induction principle for $\mathbb{S}^1$

All these definitions result in equivalent types, so in that sense it doesn't much matter which we pick. However, choosing $p_*(u) = v$ as the definition makes it easiest to conclude other things about dependent paths, such as the fact that apd_f produces them, or that we can compute them in particular type families using the transport lemmas in §2.5.

With the notion of dependent paths in hand, we can now state more precisely the induction principle for $\mathbb{S}^1$: given $P : \mathbb{S}^1 \to \mathcal{U}$ and

- An element $b : P(\mathsf{base})$, and
- A path $\ell : b =^P_{\mathsf{loop}} b$,

there is a function $f : \prod_{(x:\mathbb{S}^1)} P(x)$ such that $f(\mathsf{base}) \equiv b$ and $\mathsf{apd}_f(\mathsf{loop}) = \ell$. As in the non-dependent case, we speak of defining f by $f(\mathsf{base}) :\equiv b$ and $\mathsf{apd}_f(\mathsf{loop}) := \ell$.

Topologically, the induction principle for $\mathbb{S}^1$ can be visualized as shown in Figure 6.1. Given a fibration over the circle (which in the picture is a torus), to define a section of this fibration is the same as to give a point b in the fiber over base along with a path from b to b lying over loop. The way we interpret this type-theoretically, using our definition of dependent paths, is shown in Figure 6.2: the path from b to b over loop is represented by a path from $\mathsf{loop}_*(b)$ to b in the fiber over base.

Of course, we expect to be able to prove the non-dependent elimination rule from the dependent one, by taking P to be a constant type family. This is in fact the case, although deriving the non-dependent computation rule for loop (which refers to ap_f) from the dependent one (which refers to apd_f) is surprisingly a little tricky.

Lemma 6.2.3. *If A is a type together with $a : A$ and $p : a =_A a$, then there is a function $f : \mathbb{S}^1 \to A$ with*

$$f(\mathsf{base}) :\equiv a$$
$$\mathsf{ap}_f(\mathsf{loop}) := p.$$

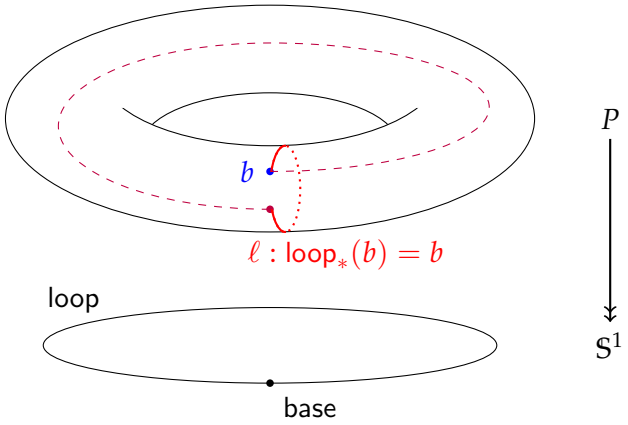


Figure 6.2: The type-theoretic induction principle for $\mathbb{S}^1$

Proof. The type $(a =^{x\mapsto A}_{\mathsf{loop}} a) \equiv (\mathsf{transport}^{x\mapsto A}(\mathsf{loop}, a) = a)$ is not the same as the type $a =_A a$. But it is equivalent to it, because by Lemma 2.3.5 we have $\mathsf{transportconst}^A_{\mathsf{loop}}(a) : \mathsf{transport}^{x\mapsto A}(\mathsf{loop}, a) = a$. Thus, given $a : A$ and $p : a = a$, we can consider the composite

$$\mathsf{transportconst}^A_{\mathsf{loop}}(a) \cdot p : (a =^{x\mapsto A}_{\mathsf{loop}} a).$$

Applying the induction principle, we obtain $f : \mathbb{S}^1 \to A$ such that

$$f(\mathsf{base}) \equiv a \qquad \text{and} \tag{6.2.4}$$
$$\mathsf{apd}_f(\mathsf{loop}) = \mathsf{transportconst}^A_{\mathsf{loop}}(a) \cdot p. \tag{6.2.5}$$

It remains to derive the equality $\mathsf{ap}_f(\mathsf{loop}) = p$. However, by Lemma 2.3.8, we have

$$\mathsf{apd}_f(\mathsf{loop}) = \mathsf{transportconst}^A_{\mathsf{loop}}(f(\mathsf{base})) \cdot \mathsf{ap}_f(\mathsf{loop}).$$

Combining this with (6.2.5) and canceling the occurrences of transportconst (which are the same by (6.2.4)), we obtain $\mathsf{ap}_f(\mathsf{loop}) = p$. □

We also have an η-rule.

Lemma 6.2.6. *If A is a type and $f, g : \mathbb{S}^1 \to A$ are two maps together with two equalities p, q:*

$$p : f(\mathsf{base}) =_A g(\mathsf{base})$$
$$q : f(\mathsf{loop}) =^{\lambda x.\, x =_A x}_{p} g(\mathsf{loop})$$

Then for all $x : \mathbb{S}^1$ we have $f(x) = g(x)$.

Proof. This is just the induction principle for the type family $P(x) :\equiv (f(x) = g(x))$. □

These two lemmas imply the expected universal property of the circle:

Lemma 6.2.7. *For any type A we have a natural equivalence*

$$(\mathbb{S}^1 \to A) \simeq \sum_{x:A} (x = x).$$

Proof. We have a canonical function $f : (\mathbb{S}^1 \to A) \to \sum_{(x:A)}(x = x)$ defined by $f(g) :\equiv (g(\mathsf{base}), g(\mathsf{loop}))$. The non-dependent eliminator shows that the fibers of f are inhabited, while the η-rule shows that they are mere propositions. Hence they are contractible, so f is an equivalence. □

As in §5.5, we can show that the conclusion of Lemma 6.2.7 is equivalent to having a dependent eliminator with propositional computation rules. Other higher inductive types also satisfy lemmas analogous to Lemmas 6.2.3 and 6.2.7; we will generally leave their proofs to the reader. We now proceed to consider many examples.

6.3 The interval

The **interval**, which we denote I, is perhaps an even simpler higher inductive type than the circle. It is generated by:

- a point $0_I : I$,
- a point $1_I : I$, and
- a path $\mathsf{seg} : 0_I =_I 1_I$.

The recursion principle for the interval says that given a type B along with

- a point $b_0 : B$,
- a point $b_1 : B$, and
- a path $s : b_0 = b_1$,

there is a function $f : I \to B$ such that $f(0_I) \equiv b_0$, $f(1_I) \equiv b_1$, and $f(\mathsf{seg}) = s$. The induction principle says that given $P : I \to \mathcal{U}$ along with

- a point $b_0 : P(0_I)$,
- a point $b_1 : P(1_I)$, and
- a path $s : b_0 =^P_{\mathsf{seg}} b_1$,

there is a function $f : \prod_{(x:I)} P(x)$ such that $f(0_I) \equiv b_0$, $f(1_I) \equiv b_1$, and $\mathsf{apd}_f(\mathsf{seg}) = s$.

From the point of view of homotopy theory, of course, the interval is not really interesting:

Lemma 6.3.1. *The type I is contractible.*

Proof. We will prove that for all $x : I$ we have $x =_I 1_I$. In other words we want a function f of type $\prod_{(x:I)}(x =_I 1_I)$. We begin to define f in the following way:

$$f(0_I) :\equiv \mathsf{seg} \ : 0_I =_I 1_I$$
$$f(1_I) :\equiv \mathsf{refl}_{1_I} : 1_I =_I 1_I$$

It remains to define $\mathsf{apd}_f(\mathsf{seg})$, which must have type $\mathsf{seg} =^{\lambda x.\, x =_I 1_I}_{\mathsf{seg}} \mathsf{refl}_{1_I}$. By definition this type is $\mathsf{seg}_*(\mathsf{seg}) =_{1_I =_I 1_I} \mathsf{refl}_{1_I}$, which in turn is equivalent to $\mathsf{seg}^{-1} \cdot \mathsf{seg} = \mathsf{refl}_{1_I}$. But there is a canonical element of that type, namely the proof that path inverses are in fact inverses. □

However, type-theoretically the interval does still have some interesting features. For instance, it enables us to give an easy proof of function extensionality.

Lemma 6.3.2. *If $f, g : A \to B$ are two functions such that $f(x) = g(x)$ for every $x : A$, then $f = g$ in the type $A \to B$.*

Proof. Let's call the proof we have $p : \prod_{(x:A)} (f(x) = g(x))$. For all $x : A$ we define a function $\widetilde{p}_x : I \to B$ by

$$\begin{aligned} \widetilde{p}_x(0_I) &= f(x) \\ \widetilde{p}_x(1_I) &= g(x) \\ \widetilde{p}_x(\mathsf{seg}) &= p(x) \end{aligned}$$

We now define $q : I \to (A \to B)$ by

$$q(i) :\equiv (\lambda x.\, \widetilde{p}_x(i))$$

Then $q(0_I)$ is the function $\lambda x.\, \widetilde{p}_x(0_I)$, which is equal to f because $\widetilde{p}_x(0_I)$ is defined by $f(x)$. Similarly, we have $q(1_I) = g$, and hence

$$q(\mathsf{seg}) : f =_{(A \to B)} g$$

□

6.4 Circles and spheres

We have already discussed the circle $\mathbb{S}^1$ as the higher inductive type generated by

- A point $\mathsf{base} : \mathbb{S}^1$, and
- A path $\mathsf{loop} : \mathsf{base} =_{\mathbb{S}^1} \mathsf{base}$.

Its induction principle says that given $P : \mathbb{S}^1 \to \mathcal{U}$ along with $b : P(\mathsf{base})$ and $\ell : b =^P_{\mathsf{loop}} b$, we have $f : \prod_{(x:\mathbb{S}^1)} P(x)$ with $f(\mathsf{base}) \equiv b$ and $\mathsf{apd}_f(\mathsf{loop}) = \ell$. Its non-dependent recursion principle says that given B with $b : B$ and $\ell : b = b$, we have $f : \mathbb{S}^1 \to B$ with $f(\mathsf{base}) \equiv b$ and $f(\mathsf{loop}) = \ell$.

We observe that the circle is nontrivial.

Lemma 6.4.1. $\mathsf{loop} \neq \mathsf{refl}_{\mathsf{base}}$.

Proof. Suppose that $\mathsf{loop} = \mathsf{refl}_{\mathsf{base}}$. Then since for any type A with $x : A$ and $p : x = x$, there is a function $f : \mathbb{S}^1 \to A$ defined by $f(\mathsf{base}) :\equiv x$ and $f(\mathsf{loop}) := p$, we have

$$p = f(\mathsf{loop}) = f(\mathsf{refl}_{\mathsf{base}}) = \mathsf{refl}_x.$$

But this implies that every type is a set, which as we have seen is not the case (see Example 3.1.9). □

The circle also has the following interesting property, which is useful as a source of counterexamples.

Lemma 6.4.2. *There exists an element of* $\prod_{(x:\mathbb{S}^1)}(x = x)$ *which is not equal to* $x \mapsto \mathsf{refl}_x$.

Proof. We define $f : \prod_{(x:\mathbb{S}^1)}(x = x)$ by $\mathbb{S}^1$-induction. When $x \equiv \mathsf{base}$, we let $f(\mathsf{base}) :\equiv \mathsf{loop}$. Now when x varies along loop, we must show that $\mathsf{transport}^{x\mapsto x=x}(\mathsf{loop}, \mathsf{loop}) = \mathsf{loop}$. However, in §2.11 we observed that $\mathsf{transport}^{x\mapsto x=x}(p, q) = p^{-1} \cdot q \cdot p$, so what we have to show is that $\mathsf{loop}^{-1} \cdot \mathsf{loop} \cdot \mathsf{loop} = \mathsf{loop}$. But this is clear by canceling an inverse.

To show that $f \neq (x \mapsto \mathsf{refl}_x)$, it suffices by function extensionality to show that $f(\mathsf{base}) \neq \mathsf{refl}_x$. But $f(\mathsf{base}) = \mathsf{loop}$, so this is just the previous lemma. □

For instance, this enables us to show that any universe which contains the circle cannot be a 1-type. (Recall that we observed in §3.1 that any universe which contains **2** cannot be a set, i.e. a 0-type.)

Corollary 6.4.3. *If the type* $\mathbb{S}^1$ *belongs to some universe* $\mathcal{U}$*, then* $\mathcal{U}$ *is not a 1-type.*

Proof. The type $\mathbb{S}^1 = \mathbb{S}^1$ in $\mathcal{U}$ is, by univalence, equivalent to the type $\mathbb{S}^1 \simeq \mathbb{S}^1$ of autoequivalences of $\mathbb{S}^1$, so it suffices to show that $\mathbb{S}^1 \simeq \mathbb{S}^1$ is not a set. Since being an equivalence is a mere proposition, it will suffice to show that the type $\mathbb{S}^1 \to \mathbb{S}^1$ is not a set. For this, it will suffice to show that its equality type $\mathsf{id}_{\mathbb{S}^1} = \mathsf{id}_{\mathbb{S}^1}$ is not a mere proposition. But by function extensionality, this is equivalent to $\prod_{(x:\mathbb{S}^1)}(x = x)$, which as we have seen in Lemma 6.4.2 contains two unequal elements. □

We have also mentioned that the 2-sphere $\mathbb{S}^2$ should be the higher inductive type generated by

- A point $\mathsf{base} : \mathbb{S}^2$, and
- A 2-dimensional path $\mathsf{surf} : \mathsf{refl}_{\mathsf{base}} = \mathsf{refl}_{\mathsf{base}}$ in $\mathsf{base} = \mathsf{base}$.

The non-dependent recursion principle for $\mathbb{S}^2$ is not hard: it says that given B with $b : B$ and $s : \mathsf{refl}_b = \mathsf{refl}_b$, we have $f : \mathbb{S}^2 \to B$ with $f(\mathsf{base}) \equiv b$ and $f(\mathsf{surf}) = s$. Here by "$f(\mathsf{surf})$" we mean an extension of the functorial action of f to two-dimensional paths, which can be stated precisely as follows.

Lemma 6.4.4. *Given* $f : A \to B$ *and* $x, y : A$ *and* $p, q : x = y$*, and* $r : p = q$*, we have* $f(r) : f(p) = f(q)$.

Proof. By path induction, we may assume $p \equiv q$ and r is reflexivity. But then we may define $f(\mathsf{refl}_p) :\equiv \mathsf{refl}_{f(p)}$. □

In order to state the general induction principle, we need a version of this lemma for dependent functions, which in turn requires a notion of dependent two-dimensional paths. As before, there are many ways to define such a thing; one is by way of a two-dimensional version of transport.

Lemma 6.4.5. *Given $P : A \to \mathcal{U}$ and $x, y : A$ and $p, q : x = y$ and $r : p = q$, for any $u : P(x)$ we have $r_{**}(u) : p_*(u) = q_*(u)$.*

Proof. By path induction. □

Now suppose given $x, y : A$ and $p, q : x = y$ and $r : p = q$ and also points $u : P(x)$ and $v : P(y)$ and dependent paths $h : u =^P_p v$ and $k : u =^P_q v$. By our definition of dependent paths, this means $h : p_*(u) = v$ and $k : q_*(u) = v$. Thus, it is reasonable to define the type of dependent 2-paths over r to be

$$(h =^P_r k) :\equiv (h = r_{**}(u) \cdot k).$$

We can now state the dependent version of Lemma 6.4.4.

Lemma 6.4.6. *Given $P : A \to \mathcal{U}$ and $x, y : A$ and $p, q : x = y$ and $r : p = q$ and a function $f : \prod_{(x:A)} P(x)$, we have $\mathsf{apd}^2_f(r) : \mathsf{apd}_f(p) =^P_r \mathsf{apd}_f(q)$.*

Proof. Path induction. □

Now we can state the induction principle for $\mathbb{S}^2$: given $P : \mathbb{S}^2 \to P$ with $b : P(\mathsf{base})$ and $s : \mathsf{refl}_b =^P_{\mathsf{surf}} \mathsf{refl}_b$, there is a function $f : \prod_{(x:\mathbb{S}^2)} P(x)$ such that $f(\mathsf{base}) \equiv b$ and $\mathsf{apd}^2_f(\mathsf{surf}) = s$.

Of course, this explicit approach will get more and more complicated as we go up in dimension. Thus, if we want to define n-spheres for all n, we need some more systematic idea. One approach is to work with n-fold loops directly, rather than n-fold paths.

Recall from §2.1 the definitions of *pointed types* $\mathcal{U}_*$, and the n-fold loop-space $\Omega^n : \mathcal{U}_* \to \mathcal{U}_*$ (Definitions 2.1.7 and 2.1.8). Now we can define the n-sphere $\mathbb{S}^n$ to be the higher inductive type generated by

- A point $\mathsf{base} : \mathbb{S}^n$, and
- An n-fold loop $\mathsf{loop}_n : \Omega^n(\mathbb{S}^n, \mathsf{base})$.

In order to write down the induction principle for this presentation, we would need to define a notion of "dependent n-loop", along with the action of dependent functions on n-loops. We leave this to the reader (see Exercise 6.3); in the next section we will discuss a different way to define the spheres that is sometimes more tractable.

6.5 Suspensions

The **suspension** of a type A is the universal way of making the points of A into paths (and hence the paths in A into 2-paths, and so on). It is a type ΣA defined by the following generators:

- a point $\mathsf{N} : \Sigma A$,
- a point $\mathsf{S} : \Sigma A$, and
- a function $\mathsf{merid} : A \to (\mathsf{N} =_{\Sigma A} \mathsf{S})$.

The names are intended to suggest a "globe" of sorts, with a north pole, a south pole, and an A's worth of meridians from one to the other. Indeed, as we will see, if $A = \mathbb{S}^1$, then its suspension is equivalent to the surface of an ordinary sphere, $\mathbb{S}^2$.

The recursion principle for ΣA says that given a type B together with points $n, s : B$ and a function $m : A \to (n = s)$, we have a function $f : \Sigma A \to B$ such that $f(\mathsf{N}) \equiv n$ and $f(\mathsf{S}) \equiv s$, and for all $a : A$ we have $f(\mathsf{merid}(a)) = m(a)$. The induction principle says that given $P : \Sigma A \to \mathcal{U}$ together with

- a point $n : P(\mathsf{N})$,
- a point $s : P(\mathsf{S})$, and
- for each $a : A$, a path $m(a) : n =^P_{\mathsf{merid}(a)} s$,

there exists a function $f : \prod_{(x:\Sigma A)} P(x)$ such that $f(\mathsf{N}) \equiv n$ and $f(\mathsf{S}) \equiv s$ and for each $a : A$ we have $\mathsf{apd}_f(\mathsf{merid}(a)) = m(a)$.

Our first observation about suspension is that it gives another way to define the circle.

Lemma 6.5.1. $\Sigma\mathbf{2} \simeq \mathbb{S}^1$.

Proof. Define $f : \Sigma\mathbf{2} \to \mathbb{S}^1$ by recursion such that $f(\mathsf{N}) :\equiv \mathsf{base}$ and $f(\mathsf{S}) :\equiv \mathsf{base}$, while $f(\mathsf{merid}(0_\mathbf{2})) := \mathsf{loop}$ but $f(\mathsf{merid}(1_\mathbf{2})) := \mathsf{refl}_{\mathsf{base}}$. Define $g : \mathbb{S}^1 \to \Sigma\mathbf{2}$ by recursion such that $g(\mathsf{base}) :\equiv \mathsf{N}$ and $g(\mathsf{loop}) := \mathsf{merid}(0_\mathbf{2}) \cdot \mathsf{merid}(1_\mathbf{2})^{-1}$. We now show that f and g are quasi-inverses.

First we show by induction that $g(f(x)) = x$ for all $x : \Sigma\mathbf{2}$. If $x \equiv \mathsf{N}$, then $g(f(\mathsf{N})) \equiv g(\mathsf{base}) \equiv \mathsf{N}$, so we have $\mathsf{refl}_\mathsf{N} : g(f(\mathsf{N})) = \mathsf{N}$. If $x \equiv \mathsf{S}$, then $g(f(\mathsf{S})) \equiv g(\mathsf{base}) \equiv \mathsf{N}$, and we choose the equality $\mathsf{merid}(1_\mathbf{2}) : g(f(\mathsf{S})) = \mathsf{S}$. It remains to show that for any $y : \mathbf{2}$, these equalities are preserved as x varies along $\mathsf{merid}(y)$, which is to say that when refl_N is transported along $\mathsf{merid}(y)$ it yields $\mathsf{merid}(1_\mathbf{2})$. By transport in path spaces and pulled back fibrations, this means we are to show that

$$g(f(\mathsf{merid}(y)))^{-1} \cdot \mathsf{refl}_\mathsf{N} \cdot \mathsf{merid}(y) = \mathsf{merid}(1_\mathbf{2}).$$

Of course, we may cancel refl_N. Now by $\mathbf{2}$-induction, we may assume either $y \equiv 0_\mathbf{2}$ or $y \equiv 1_\mathbf{2}$. If $y \equiv 0_\mathbf{2}$, then we have

$$\begin{aligned} g(f(\mathsf{merid}(0_\mathbf{2})))^{-1} \cdot \mathsf{merid}(0_\mathbf{2}) &= g(\mathsf{loop})^{-1} \cdot \mathsf{merid}(0_\mathbf{2}) \\ &= \mathsf{merid}(0_\mathbf{2}) \cdot \mathsf{merid}(1_\mathbf{2})^{-1^{-1}} \cdot \mathsf{merid}(0_\mathbf{2}) \\ &= \mathsf{merid}(1_\mathbf{2}) \cdot \mathsf{merid}(0_\mathbf{2})^{-1} \cdot \mathsf{merid}(0_\mathbf{2}) \\ &= \mathsf{merid}(1_\mathbf{2}) \end{aligned}$$

while if $y \equiv 1_\mathbf{2}$, then we have

$$\begin{aligned} g(f(\mathsf{merid}(1_\mathbf{2})))^{-1} \cdot \mathsf{merid}(1_\mathbf{2}) &= g(\mathsf{refl}_{\mathsf{base}})^{-1} \cdot \mathsf{merid}(1_\mathbf{2}) \\ &= \mathsf{refl}_\mathsf{N}{}^{-1} \cdot \mathsf{merid}(1_\mathbf{2}) \\ &= \mathsf{merid}(1_\mathbf{2}). \end{aligned}$$

Thus, for all $x : \Sigma\mathbf{2}$, we have $g(f(x)) = x$.

Now we show by induction that $f(g(x)) = x$ for all $x : \mathbb{S}^1$. If $x \equiv \mathsf{base}$, then $f(g(\mathsf{base})) \equiv f(\mathsf{N}) \equiv \mathsf{base}$, so we have $\mathsf{refl}_{\mathsf{base}} : f(g(\mathsf{base})) = \mathsf{base}$. It remains to show that this equality is preserved as x varies along loop, which is to say that it is transported along loop to itself. Again, by transport in path spaces and pulled back fibrations, this means to show that

$$f(g(\mathsf{loop}))^{-1} \cdot \mathsf{refl}_{\mathsf{base}} \cdot \mathsf{loop} = \mathsf{refl}_{\mathsf{base}}.$$

However, we have

$$\begin{aligned} f(g(\mathsf{loop})) &= f\Big(\mathsf{merid}(0_{\mathbf{2}}) \cdot \mathsf{merid}(1_{\mathbf{2}})^{-1}\Big) \\ &= f(\mathsf{merid}(0_{\mathbf{2}})) \cdot f(\mathsf{merid}(1_{\mathbf{2}}))^{-1} \\ &= \mathsf{loop} \cdot \mathsf{refl}_{\mathsf{base}} \end{aligned}$$

so this follows easily. □

Topologically, the two-point space $\mathbf{2}$ is also known as the *0-dimensional sphere*, $\mathbb{S}^0$. (For instance, it is the space of points at distance 1 from the origin in $\mathbb{R}^1$, just as the topological 1-sphere is the space of points at distance 1 from the origin in $\mathbb{R}^2$.) Thus, Lemma 6.5.1 can be phrased suggestively as $\Sigma\mathbb{S}^0 \simeq \mathbb{S}^1$. In fact, this pattern continues: we can define all the spheres inductively by

$$\begin{aligned} \mathbb{S}^0 &:\equiv \mathbf{2} \\ \mathbb{S}^{n+1} &:\equiv \Sigma\mathbb{S}^n. \end{aligned} \tag{6.5.2}$$

In fact, we can start one dimension lower by defining $\mathbb{S}^{-1} :\equiv \mathbf{0}$, observing that $\Sigma\mathbf{0} \simeq \mathbf{2}$.

To prove that this is correct would require making explicit the definition of $\mathbb{S}^n$ from the previous section, but we can motivate it as follows. If (A, a_0) and (B, b_0) are pointed types (with basepoints often left implicit), let $\mathsf{Map}_*(A, B)$ denote the type of based maps:

$$\mathsf{Map}_*(A, B) :\equiv \sum_{f:A\to B} (f(a_0) = b_0).$$

Note that any type A gives rise to a pointed type $A_+ :\equiv A + \mathbf{1}$ with basepoint $\mathsf{inr}(\star)$; this is called *adjoining a disjoint basepoint*.

Lemma 6.5.3. *For a type A and a pointed type (B, b_0), we have*

$$\mathsf{Map}_*(A_+, B) \simeq (A \to B)$$

Note that on the right we have the ordinary type of *unbased* functions from A to B.

Proof. From left to right, given $f : A_+ \to B$ with $p : f(\mathsf{inr}(\star)) = b_0$, we have $f \circ \mathsf{inl} : A \to B$. And from right to left, given $g : A \to B$ we define $g' : A_+ \to B$ by $g'(\mathsf{inl}(a)) :\equiv g(a)$ and $g'(\mathsf{inr}(u)) :\equiv b_0$. We leave it to the reader to show that these are quasi-inverse operations. □

In particular, note that $\mathbf{2} \simeq \mathbf{1}_+$. Thus, for any pointed type B we have

$$\mathsf{Map}_*(\mathbf{2}, B) \simeq (\mathbf{1} \to B) \simeq B.$$

Now recall that the loop space operation Ω acts on pointed types, with definition $\Omega(A, a_0) :\equiv (a_0 =_A a_0, \mathsf{refl}_{a_0})$. We can also make the suspension Σ act on pointed types, by $\Sigma(A, a_0) :\equiv (\Sigma A, \mathsf{N})$.

Lemma 6.5.4. *For pointed types (A, a_0) and (B, b_0) we have*

$$\mathsf{Map}_*(\Sigma A, B) \simeq \mathsf{Map}_*(A, \Omega B).$$

Proof. From left to right, given $f : \Sigma A \to B$ with $p : f(\mathsf{N}) = b_0$, we define $g : A \to \Omega B$ by

$$g(a) :\equiv p^{-1} \cdot f\Big(\mathsf{merid}(a) \cdot \mathsf{merid}(a_0)^{-1}\Big) \cdot p.$$

Then we have

$$\begin{aligned} g(a_0) &\equiv p^{-1} \cdot f\Big(\mathsf{merid}(a_0) \cdot \mathsf{merid}(a_0)^{-1}\Big) \cdot p \\ &= p^{-1} \cdot f(\mathsf{refl}_{\mathsf{N}}) \cdot p \\ &= p^{-1} \cdot p \\ &= \mathsf{refl}_{b_0}. \end{aligned}$$

Thus, denoting this path by $q : g(a_0) = \mathsf{refl}_{b_0}$, we have $(g, q) : \mathsf{Map}_*(A, \Omega B)$.

On the other hand, from right to left, given $g : A \to \Omega B$ and $q : g(a_0) = \mathsf{refl}_{b_0}$, we define $f : \Sigma A \to B$ by Σ-recursion, such that $f(\mathsf{N}) :\equiv b_0$ and $f(\mathsf{S}) :\equiv b_0$ and

$$f(\mathsf{merid}(a)) :\equiv g(a).$$

Then we can simply take p to be $\mathsf{refl}_{b_0} : f(\mathsf{N}) = b_0$.

Now given (f, p), by passing back and forth we obtain (f', p') where f' is defined by $f'(\mathsf{N}) \equiv b_0$ and $f'(\mathsf{S}) \equiv b_0$ and

$$f'(\mathsf{merid}(a)) = p^{-1} \cdot f\Big(\mathsf{merid}(a) \cdot \mathsf{merid}(a_0)^{-1}\Big) \cdot p,$$

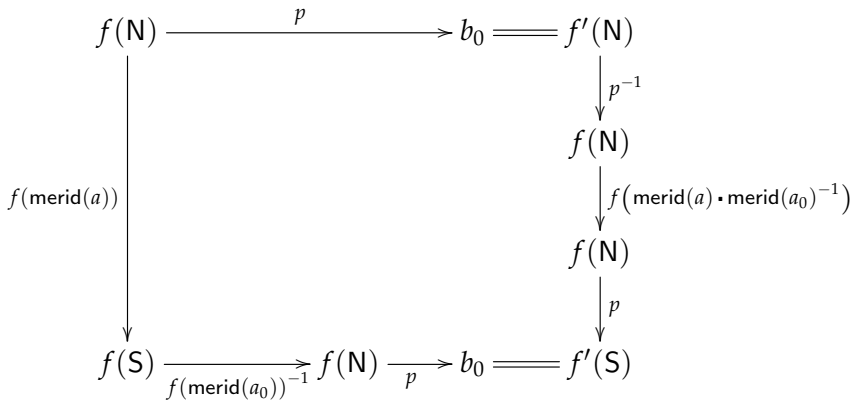
while $p' \equiv \mathsf{refl}_{b_0}$. To show $f = f'$, by function extensionality it suffices to show $f(x) = f'(x)$ for all $x : \Sigma A$, so we can use Σ-induction. First, we have

$$f(\mathsf{N}) \overset{p}{=} b_0 \equiv f'(\mathsf{N}) \tag{6.5.5}$$

Second, we have

$$f(\mathsf{S}) \overset{f(\mathsf{merid}(a_0))^{-1}}{=} f(\mathsf{N}) \overset{p}{=} b_0 \equiv f'(\mathsf{S}).$$

And thirdly, as x varies along $\mathsf{merid}(a)$ we must show that the following diagram commutes (invoking the definition of $f'(\mathsf{merid}(a))$):



This is clear. Thus, to show that $(f, p) = (f', p')$, it remains only to show that p is identified with p' when transported along this equality $f = f'$. Since the type of p is $f(\mathsf{N}) = b_0$, this means essentially that when p is composed on the left with the inverse of the equality (6.5.5), it becomes p'. But this is obvious, since (6.5.5) is just p itself, while p' is reflexivity.

On the other side, suppose given (g, q). By passing back and forth we obtain (g', q') with

$$\begin{aligned} g'(a) &= \mathsf{refl}_{b_0}{}^{-1} \cdot g(a) \cdot g(a_0)^{-1} \cdot \mathsf{refl}_{b_0} \\ &= g(a) \cdot g(a_0)^{-1} \\ &= g(a) \end{aligned}$$

using $q : g(a_0) = \mathsf{refl}_{b_0}$ in the last equality. Thus, $g' = g$ by function extensionality, so it remains to show that when transported along this equality q is identified with q'. At a_0, the induced equality $g(a_0) = g'(a_0)$ consists essentially of q itself, while the definition of q' involves only canceling inverses and reflexivities. Thus, some tedious manipulations of naturality finish the proof. □

In particular, for the spheres defined as in (6.5.2) we have

$$\mathsf{Map}_*(\mathbb{S}^n, B) \simeq \mathsf{Map}_*(\mathbb{S}^{n-1}, \Omega B) \simeq \cdots \simeq \mathsf{Map}_*(\mathbf{2}, \Omega^n B) \simeq \Omega^n B.$$

Thus, these spheres $\mathbb{S}^n$ have the universal property that we would expect from the spheres defined directly in terms of n-fold loops as in §6.4.

6.6 Cell complexes

In classical topology, a *cell complex* is a space obtained by successively attaching discs along their boundaries. It is called a *CW complex* if the boundary of an n-dimensional disc is constrained to lie in the discs of dimension strictly less than n (the $(n-1)$-skeleton).

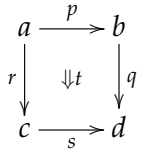
Any finite CW complex can be presented as a higher inductive type, by turning n-dimensional discs into n-dimensional paths and partitioning the image of the attaching map into a "source"

and a "target", with each written as a composite of lower-dimensional paths. Our explicit definitions of $\mathbb{S}^1$ and $\mathbb{S}^2$ in §6.4 had this form.

Another example is the torus T^2, which is generated by:

- a point $b : T^2$,
- a path $p : b = b$,
- another path $q : b = b$, and
- a 2-path $t : p \cdot q = q \cdot p$.

Perhaps the easiest way to see that this is a torus is to start with a rectangle, having four corners a, b, c, d, four edges p, q, r, s, and an interior which is manifestly a 2-path t from $p \cdot q$ to $r \cdot s$:



Now identify the edge r with q and the edge s with p, resulting in also identifying all four corners. Topologically, this identification can be seen to produce a torus.

The induction principle for the torus is the trickiest of any we've written out so far. Given $P : T^2 \to \mathcal{U}$, for a section $\prod_{(x:T^2)} P(x)$ we require

- a point $b' : P(b)$,
- a path $p' : b =^P_p b$,
- a path $q' : b =^P_q b$, and
- a 2-path t' between the "composites" $p' \cdot q'$ and $q' \cdot p'$, lying over t.

In order to make sense of this last datum, we need a composition operation for dependent paths, but this is not hard to define. Then the induction principle gives a function $f : \prod_{(x:T^2)} P(x)$ such that $f(b) \equiv b'$ and $\mathsf{apd}_f(p) = p'$ and $\mathsf{apd}_f(q) = q'$ and something like "$\mathsf{apd}_f(t) = t'$". However, this is not well-typed as it stands, firstly because the equalities $\mathsf{apd}_f(p) = p'$ and $\mathsf{apd}_f(q) = q'$ are not judgmental, and secondly because apd_f only preserves path concatenation up to homotopy. We leave the details to the reader (see Exercise 6.1).

Of course, another definition of the torus is $T^2 :\equiv \mathbb{S}^1 \times \mathbb{S}^1$. The cell-complex definition, however, generalizes easily to other spaces without such descriptions, such as the Klein bottle, the projective plane, etc. It does, however, get increasingly difficult to write down the induction principles, requiring us to define notions of dependent n-paths and of apd acting on n-paths. Fortunately, once we have the spheres in hand, there is a way around this.

6.7 Hubs and spokes

In topology, one usually speaks of building CW complexes by attaching n-dimensional discs along their $(n-1)$-dimensional boundary spheres. However, another way to express this is by gluing in the *cone* on an $(n-1)$-dimensional sphere. That is, we regard a disc as consisting of a

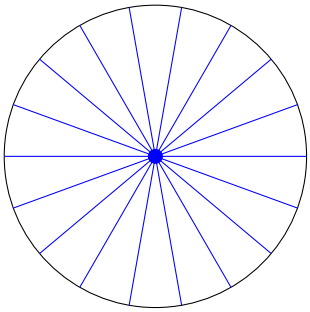


Figure 6.3: A 2-disc made out of a hub and spokes

cone point (or "hub"), with meridians (or "spokes") connecting that point to every point on the boundary, continuously, as shown in Figure 6.3.

We can use this idea to express higher inductive types containing n-dimensional paths for $n > 1$ in terms of ones containing only 1-dimensional paths. The point is that we can obtain an n-dimensional path as a continuous family of 1-dimensional paths parametrized by an $(n-1)$-dimensional object. The simplest $(n-1)$-dimensional object to use is the $(n-1)$-sphere, although in some cases a different one may be preferable. (Recall that we were able to define the spheres in §6.5 inductively using suspensions, which involve only 1-dimensional path constructors. Indeed, suspension can also be regarded as an instance of this idea, since it involves a family of 1-dimensional paths parametrized by the type being suspended.)

For instance, the torus T^2 from the previous section could be defined instead to be generated by:

- a point $b : T^2$,
- a path $p : b = b$,
- another path $q : b = b$,
- a point $h : T^2$, and
- for each $x : \mathbb{S}^1$, a path $s(x) : f(x) = h$, where $f : \mathbb{S}^1 \to T^2$ is defined by $f(\mathsf{base}) :\equiv b$ and $f(\mathsf{loop}) := p \cdot q \cdot p^{-1} \cdot q^{-1}$.

The induction principle for this version of the torus says that given $P : T^2 \to \mathcal{U}$, for a section $\prod_{(x:T^2)} P(x)$ we require

- a point $b' : P(b)$,
- a path $p' : b =^P_p b$,
- a path $q' : b =^P_q b$,
- a point $h' : P(h)$, and
- for each $x : \mathbb{S}^1$, a path $g(x) =^P_{s(x)} h'$, where $g : \prod_{(x:\mathbb{S}^1)} P(f(x))$ is defined by $g(\mathsf{base}) :\equiv b'$ and $\mathsf{apd}_g(\mathsf{loop}) := p' \cdot q' \cdot (p')^{-1} \cdot (q')^{-1}$.

Note that there is no need for dependent 2-paths or apd acting on 2-paths. We leave it to the reader to write out the computation rules.

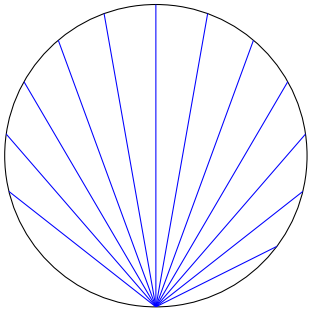


Figure 6.4: Hubless spokes

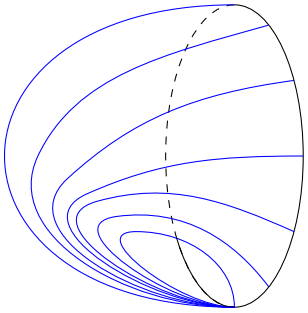


Figure 6.5: Hubless spokes, II

Remark 6.7.1. One might question the need for introducing the hub point h; why couldn't we instead simply add paths continuously relating the boundary of the disc to a point *on* that boundary, as shown in Figure 6.4? This does work, but not as well. For if, given some $f : \mathbb{S}^1 \to X$, we give a path constructor connecting each $f(x)$ to $f(\mathsf{base})$, then what we end up with is more like the picture in Figure 6.5 of a cone whose vertex is twisted around and glued to some point on its base. The problem is that the specified path from $f(\mathsf{base})$ to itself may not be reflexivity. We could add a 2-dimensional path constructor ensuring this, but using a separate hub avoids the need for any path constructors of dimension abbove 1.

Remark 6.7.2. Note also that this "translation" of higher paths into 1-paths does not preserve judgmental computation rules for these paths, though it does preserve propositional ones.

6.8 Pushouts

From a category-theoretic point of view, one of the important aspects of any foundational system is the ability to construct limits and colimits. In set-theoretic foundations, these are limits and colimits of sets, whereas in our case they will be limits and colimits of *types*. We have seen in §2.15 that cartesian product types have the correct universal property of a categorical product of types, and in Exercise 2.8 that coproduct types likewise have their expected universal property.

More general limits can be constructed using identity types and Σ-types, e.g. the pullback of $f : A \to C$ and $g : B \to C$ can be defined by $\sum_{(a:A)} \sum_{(b:B)} (f(a) = g(b))$. However, more general *colimits* require identifying elements coming from different types, for which higher inductives are well-adapted. Since all our constructions are homotopy-invariant, our colimits will all be *homotopy colimits*, but we drop the ubiquitous adjective in the interests of conciseness.

In this section we discuss *pushouts*, as perhaps the simplest and one of the most useful colimits. Indeed, one expects all finite colimits (for a suitable homotopical definition of "finite") to be constructible from pushouts and finite coproducts. It is also possible to give a direct construction of more general colimits using higher inductive types, but this is somewhat technical, and also not completely satisfactory since we do not yet have a good fully general notion of homotopy coherent diagrams.

Suppose given a span of types and functions:

$$\mathscr{D} = \quad \begin{array}{ccc} C & \xrightarrow{g} & B \\ {\scriptstyle f}\downarrow & & \\ A & & \end{array}$$

The **pushout** of this span is the higher inductive type $A \sqcup^C B$ presented by

- a function $\mathsf{inl} : A \to A \sqcup^C B$,
- a function $\mathsf{inr} : B \to A \sqcup^C B$, and
- for each $c : C$ a path $\mathsf{glue}(c) : (\mathsf{inl}(f(c)) = \mathsf{inr}(g(c)))$.

In other words, $A \sqcup^C B$ is the disjoint union of A and B, together with for every $c : C$ a witness that $f(c)$ and $g(c)$ are equal. The recursion principle says that if D is another type, we can define a map $s : A \sqcup^C B \to D$ by defining

- for each $a : A$, the value of $s(\mathsf{inl}(a)) : D$,
- for each $b : B$, the value of $s(\mathsf{inr}(b)) : D$, and
- for each $c : C$, the value of $\mathsf{ap}_s(\mathsf{glue}(c)) : s(\mathsf{inl}(f(c))) = s(\mathsf{inr}(g(c)))$.

We leave it to the reader to formulate the induction principle. It also implies the η-rule that if $s, s' : A \sqcup^C B \to D$ are two maps such that

$$\begin{aligned} s(\mathsf{inl}(a)) &= s'(\mathsf{inl}(a)) \\ s(\mathsf{inr}(b)) &= s'(\mathsf{inr}(b)) \\ \mathsf{ap}_s(\mathsf{glue}(c)) &= \mathsf{ap}_{s'}(\mathsf{glue}(c)) \qquad \text{(modulo the previous two equalities)} \end{aligned}$$

for every a, b, c, then $s = s'$.

To formulate the universal property of a pushout, we introduce the following.

Definition 6.8.1. Given functions $A \xleftarrow{f} C \xrightarrow{g} B$ and a type D, a **cocone under $\mathscr{D}$ with base D** consists of functions $i : A \to D$ and $j : B \to D$ and a homotopy $h : \prod_{(c:C)} (i(f(c)) = j(g(c)))$:

$$\begin{array}{ccc} C & \xrightarrow{g} & B \\ {\scriptstyle f}\downarrow & \swarrow{\scriptstyle h} & \downarrow{\scriptstyle j} \\ A & \xrightarrow[i]{} & D \end{array}$$

We denote by $\mathrm{cocone}_{\mathscr{D}}(D)$ the type of all such cocones, i.e.

$$\mathrm{cocone}_{\mathscr{D}}(D) :\equiv \sum_{(i:A\to D)} \sum_{(j:B\to D)} \prod_{c:C} (i(f(c)) = j(g(c))).$$

Of course, there is a canonical cocone under $\mathscr{D}$ with base $A \sqcup^C B$ consisting of inl, inr, and glue.

$$\begin{array}{ccc} C & \xrightarrow{g} & B \\ {\scriptstyle f}\downarrow & \swarrow_{\mathsf{glue}} & \downarrow{\scriptstyle \mathsf{inr}} \\ A & \xrightarrow[\mathsf{inl}]{} & A \sqcup^C B \end{array}$$

The following lemma says that this is the universal such cocone.

Lemma 6.8.2. *For any type E, there is an equivalence*

$$(A \sqcup^C B \to E) \;\simeq\; \mathrm{cocone}_{\mathscr{D}}(E).$$

Proof. Let's consider an arbitrary type $E : \mathcal{U}$. There is a canonical function

$$\begin{cases} (A \sqcup^C B \to E) & \longrightarrow & \mathrm{cocone}_{\mathscr{D}}(E) \\ t & \longmapsto & t \circ c_{\sqcup} \end{cases}$$

defined by sending (i, j, h) to $(t \circ i, t \circ j, \mathsf{ap}_t \circ h)$. We will show that this is an equivalence.

Firstly, given a $c = (i, j, h) : \mathrm{cocone}_{\mathscr{D}}(E)$, we need to construct a map $\mathsf{s}(c)$ from $A \sqcup^C B$ to E.

$$\begin{array}{ccc} C & \xrightarrow{g} & B \\ {\scriptstyle f}\downarrow & \swarrow_{h} & \downarrow{\scriptstyle j} \\ A & \xrightarrow[i]{} & E \end{array}$$

The map $\mathsf{s}(c)$ is defined in the following way

$$\begin{aligned} \mathsf{s}(c)(\mathsf{inl}(a)) &= i(a) \\ \mathsf{s}(c)(\mathsf{inr}(b)) &= j(b) \\ \mathsf{ap}_{\mathsf{s}(c)}(\mathsf{glue}(x)) &= h(x) \end{aligned}$$

We have defined a map

$$\begin{cases} \mathrm{cocone}_{\mathscr{D}}(E) & \longrightarrow & (A \sqcup^B C \to E) \\ c & \longmapsto & \mathsf{s}(c) \end{cases}$$

and we need to prove that this map is an inverse to $t \mapsto t \circ c_{\sqcup}$. On the one hand, if $c = (i, j, h) : \mathrm{cocone}_{\mathscr{D}}(E)$, we have

$$\begin{aligned} \mathsf{s}(c) \circ c_{\sqcup} &= (\mathsf{s}(c) \circ \mathsf{inl}, \mathsf{s}(c) \circ \mathsf{inr}, \mathsf{ap}_{\mathsf{s}(c)} \circ \mathsf{glue}) \\ &= (\lambda a.\, \mathsf{s}(c)(\mathsf{inl}(a)),\ \lambda b.\, \mathsf{s}(c)(\mathsf{inr}(b)),\ \lambda x.\, \mathsf{ap}_{\mathsf{s}(c)}(\mathsf{glue}(x))) \\ &= (\lambda a.\, i(a),\ \lambda b.\, j(b),\ \lambda x.\, h(x)) \\ &\equiv (i, j, h) \\ &= c \end{aligned}$$

On the other hand, if $t : A \sqcup^B C \to E$, we want to prove that $\mathsf{s}(t \circ c_{\sqcup}) = t$. For $a : A$, we have

$$\mathsf{s}(t \circ c_{\sqcup})(\mathsf{inl}(a)) = t(\mathsf{inl}(a))$$

because the first component of $t \circ c_{\sqcup}$ is $t \circ \mathsf{inl}$. In the same way, for $b : B$ we have

$$\mathsf{s}(t \circ c_{\sqcup})(\mathsf{inr}(b)) = t(\mathsf{inr}(b))$$

and for $x : C$ we have

$$\mathsf{ap}_{\mathsf{s}(t \circ c_{\sqcup})}(\mathsf{glue}(x)) = \mathsf{ap}_t(\mathsf{glue}(x))$$

hence $\mathsf{s}(t \circ c_{\sqcup}) = t$.

This proves that $c \mapsto \mathsf{s}(c)$ is a quasi-inverse to $t \mapsto t \circ c_{\sqcup}$, as desired. □

A number of standard homotopy-theoretic constructions can be expressed as (homotopy) pushouts.

- The pushout of the span $\mathbf{1} \leftarrow A \to \mathbf{1}$ is the **suspension** ΣA (see §6.5).
- The pushout of $A \xleftarrow{\mathsf{pr}_1} A \times B \xrightarrow{\mathsf{pr}_2} B$ is called the **join** of A and B, written $A * B$.
- The pushout of $\mathbf{1} \leftarrow A \xrightarrow{f} B$ is the **cone** or **cofiber** of f.
- If A and B are equipped with basepoints $a_0 : A$ and $b_0 : B$, then the pushout of $A \xleftarrow{a_0} \mathbf{1} \xrightarrow{b_0} B$ is the **wedge** $A \vee B$.
- If A and B are pointed as before, define $f : A \vee B \to A \times B$ by $f(\mathsf{inl}(a)) :\equiv (a, b_0)$ and $f(\mathsf{inr}(b)) :\equiv (a_0, b)$, with $f(\mathsf{glue}) := \mathsf{refl}_{(a_0,b_0)}$. Then the cone of f is called the **smash product** $A \wedge B$.

We will discuss pushouts further in Chapters 7 and 8.

Remark 6.8.3. As remarked in §3.7, the notations $\wedge$ and $\vee$ for the smash product and wedge of pointed spaces are also used in logic for "and" and "or", respectively. Since types in homotopy type theory can behave either like spaces or like propositions, there is technically a potential for conflict — but since they rarely do both at once, context generally disambiguates. Furthermore, the smash product and wedge only apply to *pointed* spaces, whil the only pointed mere proposition is $\top \equiv \mathbf{1}$ — and we have $\mathbf{1} \wedge \mathbf{1} = \mathbf{1}$ and $\mathbf{1} \vee \mathbf{1} = \mathbf{1}$ for either meaning of $\wedge$ and $\vee$.

Remark 6.8.4. Note that colimits do not in general preserve truncation. For instance, $\mathbb{S}^0$ and $\mathbf{1}$ are both sets, but the pushout of $\mathbf{1} \leftarrow \mathbb{S}^0 \to \mathbf{1}$ is $\mathbb{S}^1$, which is not a set. If we are interested in colimits in the category of n-types, therefore (and, in particular, in the category of sets), we need to "truncate" the colimit somehow. We will return to this point in §6.9 and Chapters 7 and 10.

6.9 Truncations

In §3.7 we introduced the propositional truncation as a new type forming operation; we now observe that it can be obtained as special cases of higher inductive types. This reduces the problem of understanding truncations to the problem of understanding higher inductives, which at least are amenable to a systematic treatment. It is also interesting because it provides our first

example of a higher inductive type which is truly *recursive*, in that its constructors take inputs from the type being defined (as does the successor $s : \mathbb{N} \to \mathbb{N}$).

Let A be a type; we define its propositional truncation $\|A\|$ to be the higher inductive type generated by:

- A function $|-| : A \to \|A\|$, and
- For each $x, y : \|A\|$, a path $x = y$.

The first constructor is by definition a function $A \to \|A\|$, while the second constructor is by definition the assertion that $\|A\|$ is a mere proposition. Thus, the definition of $\|A\|$ can be interpreted as saying that $\|A\|$ is freely generated by a function $A \to \|A\|$ and the fact that it is a mere proposition.

The recursion principle for this higher inductive definition is easy to write down: it says that given any type B together with

- A function $g : A \to B$, and
- For any $x, y : B$, a path $x =_B y$,

there exists a function $f : \|A\| \to B$ such that

- $f(|a|) \equiv g(a)$ for all $a : A$, and
- for any $x, y : \|A\|$, the function ap_f takes the specified path $x = y$ in $\|A\|$ to the specified path $f(x) = f(y)$ in B (propositionally).

These are exactly the hypotheses that we stated in §3.7 for the recursion principle of propositional truncation — a function $A \to B$ such that B is a mere proposition — and the first part of the conclusion is exactly what we stated there as well. The second part (the action of ap_f) was not mentioned previously, but it turns out to be uninteresting in this case because B is a mere proposition, so *any* two paths in it are automatically equal.

There is also an induction principle for $\|A\|$, which says that given any $B : \|A\| \to \mathcal{U}$ together with

- a function $g : \prod_{(a:A)} B(|a|)$, and
- for any $x, y : \|A\|$ and $u : B(x)$ and $v : B(y)$, a dependent path $q : u =^B_{p(x,y)} v$, where $p(x, y)$ is the path coming from the second constructor of $\|A\|$,

there exists $f : \prod_{(x:\|A\|)} B(x)$ such that $f(|a|) \equiv a$ for $a : A$, and also another computation rule. However, because there can be at most one function between any two mere propositions (up to homotopy), this induction principle is not really useful (see also Exercise 3.16).

We can, however, extend this idea to construct similar truncations landing in n-types, for any n. For instance, we might define the *0-truncation* $\|A\|_0$ to be generated by

- A function $|-|_0 : A \to \|A\|_0$, and
- For each $x, y : \|A\|_0$ and each $p, q : x = y$, a path $p = q$.

Then $\|A\|_0$ would be freely generated by a function $A \to \|A\|_0$ together with the assertion that $\|A\|_0$ is a set. A natural induction principle for it would say that given $B : \|A\|_0 \to \mathcal{U}$ together with

- a function $g : \prod_{(a:A)} B(|a|_0)$, and
- for any $x, y : \|A\|_0$ with $z : B(x)$ and $w : B(y)$, and each $p, q : x = y$ with $r : z =^B_p w$ and $s : z =^B_q w$, a 2-path $v : p =^B_{u(x,y,p,q)} q$, where $u(x, y, p, q) : p = q$ is obtained from the second constructor of $\|A\|_0$,

there exists $f : \prod_{(x:\|A\|_0)} B(x)$ such that $f(|a|_0) \equiv g(a)$ for all $a : A$, and also $\mathsf{apd}_f(u(x, y, p, q))$ is the 2-path specified above. (As in the propositional case, the latter condition turns out to be uninteresting.) From this, however, we can prove a more useful induction principle.

Lemma 6.9.1. *Suppose given $B : \|A\|_0 \to \mathcal{U}$ together with $g : \prod_{(a:A)} B(|a|_0)$, and assume that each $B(x)$ is a set. Then there exists $f : \prod_{(x:\|A\|_0)} B(x)$ such that $f(|a|_0) \equiv g(a)$ for all $a : A$.*

Proof. It suffices to construct, for any x, y, z, w, p, q, r, s as above, a 2-path $v : p =^B_{u(x,y,p,q)} q$. However, by the definition of dependent 2-paths, this is an ordinary 2-path in the fiber $B(y)$. Since $B(y)$ is a set, a 2-path exists with any given source and target. □

This implies the expected universal property.

Lemma 6.9.2. *For any set B and any type A, composition with $|-|_0 : A \to \|A\|_0$ determines an equivalence*

$$(\|A\|_0 \to B) \simeq (A \to B).$$

Proof. The special case of Lemma 6.9.1 when B is the constant family gives a map from right to left, which is a right inverse to the "compose with $|-|_0$" function from left to right. To show that it is also a left inverse, let $h : \|A\|_0 \to B$, and define $h' : \|A\|_0 \to B$ by applying Lemma 6.9.1 to the composite $a \mapsto h(|a|_0)$. Thus, $h'(|a|_0) = h(|a|_0)$.

However, since B is a set, for any $x : \|A\|_0$ the type $h(x) = h'(x)$ is a mere proposition, and hence also a set. Therefore, by Lemma 6.9.1, the observation that $h'(|a|_0) = h(|a|_0)$ for any $a : A$ implies $h(x) = h'(x)$ for any $x : \|A\|_0$, and hence $h = h'$. □

For instance, this enables us to construct colimits of sets. We have seen that if $A \xleftarrow{f} C \xrightarrow{g} B$ is a span of sets, then the pushout $A \sqcup^C B$ may no longer be a set. (For instance, if A and B are $\mathbf{1}$ and C is $\mathbf{2}$, then the pushout is $\mathbb{S}^1$.) However, we can construct a pushout that is a set, and has the expected universal property with respect to other sets, by truncating.

Lemma 6.9.3. *Let $A \xleftarrow{f} C \xrightarrow{g} B$ be a span of sets. Then for any set E, there is a canonical equivalence*

$$\left(\left\|A \sqcup^C B\right\|_0 \to E\right) \simeq \mathrm{cocone}_{\mathscr{D}}(E).$$

Proof. Compose the equivalences in Lemmas 6.8.2 and 6.9.2. □

We will refer to $\left\|A \sqcup^C B\right\|_0$ as the **set-pushout** of f and g, to distinguish it from the (homotopy) pushout $A \sqcup^C B$. Alternatively, we could modify the definition of the pushout in §6.8 to include the 0-truncation constructor directly, avoiding the need to truncate afterwards. Similar remarks apply to any sort of colimit of sets; we will explore this further in Chapter 10.

However, while the above definition of the 0-truncation works — it gives what we want, and is consistent — it has a couple of issues. Firstly, it doesn't fit so nicely into the general theory of higher inductive types. In general, it is tricky to deal directly with constructors such as the second one we have given for $\|A\|_0$, whose *inputs* involve not only elements of the type being defined, but paths in it.

This can be gotten round fairly easily, however. Recall in §5.1 we mentioned that we can allow a constructor of an inductive type W to take "infinitely many arguments" of type W by having it take a single argument of type $\mathbb{N} \to W$. There is a general principle behind this: to model a constructor with funny-looking inputs, use an auxiliary inductive type (such as $\mathbb{N}$) to parametrize them, reducing the input to a simple function with inductive domain.

For the 0-truncation, we can consider the auxiliary *higher* inductive type S generated by two points $a, b : S$ and two paths $p, q : a = b$. Then the fishy-looking constructor of $\|A\|_0$ can be replaced by the unobjectionable

- For every $f : S \to A$, a path $\mathsf{ap}_f(p) = \mathsf{ap}_f(q)$.

Since to give a map out of S is the same as to give two points and two parallel paths between them, this will yield the same induction principle.

A more serious problem with our current definition of 0-truncation, however, is that it doesn't generalize very well. If we want to describe a notion of definition of "n-truncation" into n-types uniformly for all $n : \mathbb{N}$, then this approach is unfeasible, since the second constructor would need a number of arguments that increases with n. In §7.3, therefore, we will use a different idea to construct these, based on the observation that the type S introduced above is equivalent to the circle $\mathbb{S}^1$. This will include the 0-truncation as a special case, and satisfy generalized versions of Lemmas 6.9.1 and 6.9.2.

6.10 Quotients

A particularly important sort of colimit of sets is the *quotient* by a relation. That is, let A be a set and $R : A \times A \to \mathsf{Prop}$ a family of mere propositions (a "mere relation"). Its quotient should be the set-coequalizer of the two projections

$$\textstyle\sum_{(a,b:A)} R(a,b) \rightrightarrows A.$$

We can also describe this directly, as the higher inductive type A/R generated by

- A function $q : A \to A/R$;
- For each $a, b : A$ such that $R(a,b)$, an equality $q(a) = q(b)$; and
- The 0-truncation constructor: for all $x, y : A/R$ and $r, s : x = y$, we have $r = s$.

We may sometimes refer to A/R as the **set-quotient** of A by R, to emphasize that it produces a set by definition. (There are more general notions of "quotient" in homotopy theory, but they are mostly beyond the scope of this book. However, in §9.9 we will consider the "quotient" of a type by a 1-groupoid, which is the next level up from set-quotients.)

Remark 6.10.1. It is not actually necessary for the definition of set-quotients, and most of their properties, that A be a set. However, this is generally the case of most interest.

Lemma 6.10.2. *The function $q : A \to A/R$ is surjective.*

Proof. We must show that for any $x : A/R$ there merely exists an $a : A$ with $q(a) = x$. We use the induction principle of A/R. The first case is trivial: if x is $q(a)$, then of course there merely exists an a such that $q(a) = q(a)$. And since the goal is a mere proposition, it automatically respects all path constructors, so we are done. □

Lemma 6.10.3. *For any set B, precomposing with q yields an equivalence*

$$(A/R \to B) \simeq \Big(\sum_{(f:A\to B)} \prod_{(a,b:A)} R(a,b) \to (f(a) = f(b))\Big)$$

Proof. The quasi-inverse of $- \circ q$, going from right to left, is just the recursion principle for A/R. That is, given $f : A \to B$ such that $\prod_{(a,b:A)} R(a,b) \to (f(a) = f(b))$, we define $\bar{f} : A/R \to B$ by $\bar{f}(q(a)) :\equiv f(a)$. This defining equation says precisely that $(f \mapsto \bar{f})$ is a right inverse to $(- \circ q)$.

For it to also be a left inverse, we must show that for any $g : A/R \to B$ and $x : A/R$ we have $g(x) = \overline{g \circ q}$. However, by Lemma 6.10.2 there merely exists a such that $q(a) = x$. Since our desired equality is a mere proposition, we may assume there purely exists such an a, in which case $g(x) = g(q(a)) = \overline{g \circ q}(q(a)) = \overline{g \circ q}(x)$. □

Of course, classically the usual case to consider is when R is an **equivalence relation**, i.e. we have

$$\prod_{a:A} R(a,a)$$
$$\prod_{a,b:A} R(a,b) \to R(b,a)$$
$$\prod_{a,b,c:C} R(a,b) \times R(b,c) \to R(a,c).$$

In this case, the set-quotient A/R has additional good properties, as we will see in §10.1: for instance, we have $R(a,b) \simeq (q(a) =_{A/R} q(b))$.

The quotient by an equivalence relation can also be constructed in other ways. The set theoretic approach is to consider the set of *equivalence classes*, as a subset of the power set of A. We can mimic this in type theory if we assume the impredicativity of mere propositions (see §3.5).

Definition 6.10.4. A predicate $P : A \to \mathsf{Prop}$ is an **equivalence class** of R if there merely exists an $a : A$ such that for all $b : A$ we have $R(a,b) \leftrightarrow P(b)$.

Of course, for any $a : A$ we have the canonical equivalence class $P_a(b) :\equiv R(a,b)$.

Definition 6.10.5. We define

$$A /\!\!/ R :\equiv \{\, P : A \to \mathsf{Prop} \mid P \text{ is an equivalence class of } R \,\}$$

The function $q' : A \to A /\!\!/ R$ is defined by $q'(a) :\equiv P_a$.

Theorem 6.10.6. *For any equivalence relation R on A, the two set-quotients A/R and $A /\!\!/ R$ are equivalent.*

Proof. First, note that if $R(a,b)$, then since R is an equivalence relation we have $R(a,c) \leftrightarrow R(b,c)$ for any $c : A$. Thus, $R(a,c) = R(b,c)$ by univalence, hence $P_a = P_b$ by function extensionality, i.e. $q'(a) = q'(b)$. Therefore, by Lemma 6.10.3 we have an induced map $f : A/R \to A /\!\!/ R$ such that $f \circ q = q'$.

We will show that f is injective and surjective, hence an equivalence. Surjectivity follows immediately from the fact that q' is surjective, which in turn is true essentially by definition of $A /\!\!/ R$. For injectivity, if $f(x) = f(y)$, then to show the mere proposition $x = y$, by surjectivity of q we may assume $x = q(a)$ and $y = q(b)$ for some $a, b : A$. Then $R(a,c) = f(q(a))(c) = f(q(b))(c) = R(b,c)$ for any $c : A$, and in particular $R(a,b) = R(b,b)$. But $R(b,b)$ is inhabited, since R is an equivalence relation, hence so is $R(a,b)$. Thus $q(a) = q(b)$ and so $x = y$. □

In §10.1.3 we will give an alternative proof of this theorem.

Remark 6.10.7. The previous two constructions provide quotients in generality, but in particular cases there may be easier constructions. For instance, we may define the integers $\mathbb{Z}$ as a set-quotient

$$\mathbb{Z} :\equiv (\mathbb{N} \times \mathbb{N})/\sim$$

where

$$(a,b) \sim (c,d) :\equiv (a + d = b + c).$$

In other words, a pair (a,b) represents the integer $a - b$. In this case, however, there are *canonical representatives* of the equivalence classes: those of the form $(n,0)$ or $(0,n)$.

The following lemma says that when this sort of thing happens, we don't need either general construction of quotients. (A function $r : A \to A$ is called **idempotent** if $r \circ r = r$.)

Lemma 6.10.8. *Suppose $\sim$ is an equivalence relation on a set A, and there exists an idempotent $r : A \to A$ such that, for all $x, y \in A$, $(r(x) = r(y)) \simeq (x \sim y)$. Then the type*

$$(A/\sim) :\equiv \sum_{x:A} r(x) = x$$

is the set-quotient of $\sim$ by A. In other words, there is a map $q : A \to (A/\sim)$ such that for every set B, the type $(A/\sim) \to B$ is equivalent to

$$\sum_{(g:A\to B)} \prod_{(x,y:A)} (x \sim y) \to (g(x) = g(y)) \tag{6.10.9}$$

with the map being induced by precomposition with q.

Proof. Let $i : \prod_{(x:A)} r(r(x)) = r(x)$ witness idempotence of r. The map $q : A \to A/\sim$ is defined by $q(x) :\equiv (r(x), i(x))$. An equivalence e from $A/\sim \to B$ to (6.10.9) is defined by

$$e(f) :\equiv (f \circ q, \ldots),$$

where ... is the following proof: if $x, y : A$ and $x \sim y$ then by assumption $r(x) = r(y)$, hence $(r(x), i(x)) = (r(y), i(y))$ as A is a set, therefore $f(q(x)) = f(q(y))$. To see that e is an equivalence, consider the map e' in the opposite direction,

$$e'(g, p)(x, q) \equiv g(x).$$

Given any $f : A/\sim \to B$,

$$e'(e(f))(x, p) \equiv f(q(x)) \equiv f(r(x), i(x)) = f(x, p)$$

where the last equality holds because $p : r(x) = x$ and so $(x, p) = (r(x), i(x))$ because A is a set. Similarly we compute

$$e(e'(g, p)) \equiv e(g \circ \mathsf{pr}_1) \equiv (f \circ \mathsf{pr}_1 \circ q, \ldots).$$

Because B is a set we need not worry about the ... part, while for the first component we have

$$f(\mathsf{pr}_1(q(x))) :\equiv f(r(x)) = f(x),$$

where the last equation holds because $r(x) \sim x$ and f respects $\sim$ by assumption. □

Corollary 6.10.10. *Suppose $p : A \to B$ is a retraction between sets. Then B is the quotient of A by the equivalence relation $\sim$ defined by*

$$(a_1 \sim a_2) :\equiv (p(a_1) = p(a_2)).$$

Proof. Suppose $s : B \to A$ is a section of p. Then $s \circ p : A \to A$ is an idempotent which satisfies the condition of Lemma 6.10.8 for this $\sim$, and s induces an isomorphism from B to its set of fixed points. □

Remark 6.10.11. Lemma 6.10.8 applies to $\mathbb{Z}$ with the idempotent $r : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ defined by

$$r(a, b) = \begin{cases} (a - b, 0) & \text{if } a \geq b, \\ (0, b - a) & \text{otherwise.} \end{cases}$$

Thus a non-negative integer is canonically represented as $(k, 0)$ and a non-positive one by $(0, m)$, for $k, m : \mathbb{N}$. This division into cases implies the following induction principle for integers, which will be useful in Chapter 8. (As usual, we identify natural numbers with the corresponding non-negative integers.)

Lemma 6.10.12. *Suppose $P : \mathbb{Z} \to \mathcal{U}$ is a type family and that we have*

- $d_0 : P(0)$,
- $d_+ : \prod_{(n:\mathbb{N})} P(n) \to P(\mathsf{succ}(n))$, *and*
- $d_- : \prod_{(n:\mathbb{N})} P(-n) \to P(-\mathsf{succ}(n))$.

Then we have $f : \prod_{(z:\mathbb{Z})} P(z)$ such that $f(0) \equiv d_0$ and $f(\mathsf{succ}(n)) \equiv d_+(f(n))$, and $f(-\mathsf{succ}(n)) \equiv d_-(f(-n))$ for all $n : \mathbb{N}$.

Proof. We identify $\mathbb{Z}$ with $\sum_{(x:\mathbb{N}\times\mathbb{N})}(r(x) = x)$, where r is the above idempotent. Now define $Q :\equiv P \circ r : \mathbb{N} \times \mathbb{N} \to \mathcal{U}$. We can construct $g : \prod_{(x:\mathbb{N}\times\mathbb{N})} Q(x)$ by double induction on n:

$$\begin{aligned} g(0,0) &:\equiv d_0 \\ g(\mathsf{succ}(n),0) &:\equiv d_+(g(n,0)) \\ g(0,\mathsf{succ}(m)) &:\equiv d_-(g(0,m)) \\ g(\mathsf{succ}(n),\mathsf{succ}(m)) &:\equiv g(n,m) \end{aligned}$$

Let f be the restriction of g to $\mathbb{Z}$. □

For example, we can define the n-fold concatenation of a loop for any integer n.

Corollary 6.10.13. *Let A be a type with $a : A$ and $p : a = a$. There is a function $\prod_{(n:\mathbb{Z})}(a = a)$, denoted $n \mapsto p^n$, defined by*

$$\begin{aligned} p^0 &:\equiv \mathsf{refl}_{\mathsf{base}} \\ p^{n+1} &:\equiv p^n \cdot p \qquad n \geq 0 \\ p^{n-1} &:\equiv p^n \cdot p^{-1} \quad n \leq 0. \end{aligned}$$

We will discuss the integers further in §§6.11 and 11.1.

6.11 Algebra

In addition to constructing higher-dimensional objects such as spheres and cell complexes, higher inductive types are also very useful even when working only with sets. We have seen one example already in Lemma 6.9.3: they allow us to construct the colimit of any diagram of sets, which is not possible in the base type theory of Chapter 1. Higher inductive types are also very useful when we study sets with algebraic structure.

As a running example in this section, we consider *groups*, which are familiar to most mathematicians and exhibit the essential phenomena (and will be needed in later chapters). However, most of what we say applies equally well to any sort of algebraic structure.

Definition 6.11.1. A **monoid** is a set G together with

- a function $G \times G \to G$, written infix as $(x, y) \mapsto x \cdot y$; and
- an element $e : G$; such that
- for any $x : G$, we have $x \cdot e = x$ and $e \cdot x = x$; and
- for any $x, y, z : G$, we have $x \cdot (y \cdot z) = (x \cdot y) \cdot z$.

A **group** is a monoid G together with

- a function $i : G \to G$, written $x \mapsto x^{-1}$; such that
- for any $x : G$ we have $x \cdot x^{-1} = e$ and $x^{-1} \cdot x = e$.

Remark 6.11.2. Note that we require a group to be a set. We could consider a more general notion of "∞-group" which is not a set — for instance, the loop spaces $\Omega(A)$ defined in §2.1 are such — but this would take us further afield than is appropriate at the moment. With our current definition, we may expect the resulting "group theory" to behave similarly to the way it does in set-theoretic mathematics (with the caveat that, unless we assume LEM, it will be "constructive" group theory).

Example 6.11.3. The natural numbers $\mathbb{N}$ are a monoid under addition, with unit 0, and also under multiplication, with unit 1. If we define the arithmetical operations on the integers $\mathbb{Z}$ in the obvious way, then as usual they are a group under addition and a monoid under multiplication (and, of course, a ring). For instance, if $u, v \in \mathbb{Z}$ are represented by (a, b) and (c, d), respectively, then $u + v$ is represented by $(a + c, b + d)$, $-u$ is represented by (b, a), and uv is represented by $(ac + bd, ad + bc)$.

Example 6.11.4. We essentially observed in §2.1 that if (A, a) is a pointed type, then its loop space $\Omega(A, a) :\equiv (a =_A a)$ has all the structure of a group, except that it is not in general a set. We define the **fundamental group** of A based at $a : A$ to be

$$\pi_1(A, a) :\equiv \|\Omega(A, a)\|_0.$$

This inherits a group structure; for instance, the multiplication $\pi_1(A, a) \times \pi_1(A, a) \to \pi_1(A, a)$ is defined by double induction on truncation from the concatenation of paths.

More generally, the n^{th} **homotopy group** of (A, a) is $\pi_n(A, a) :\equiv \|\Omega^n(A, a)\|_0$. Then $\pi_n(A, a) = \pi_1(\Omega^{n-1}(A, a))$ for $n \geq 1$, so it is also a group. (When $n = 0$, we have $\pi_0(A) \equiv \|A\|_0$, which is not a group.) Moreover, the Eckmann–Hilton argument (Theorem 2.1.6) implies that if $n \geq 2$, then $\pi_n(A, a)$ is a *abelian* group, i.e. we have $x \cdot y = y \cdot x$ for all x, y. Chapter 8 will be largely the study of these groups.

One important notion in group theory is that of the *free group* generated by a set, or more generally of a group *presented* by generators and relations. It is well-known in type theory that *some* free algebraic objects can be defined using *ordinary* inductive types. For instance, the free monoid on a set A can be identified with the type $\mathsf{List}(A)$ of *finite lists* of elements of A, which is inductively generated by

- a constructor $\mathsf{nil} : \mathsf{List}(A)$, and
- for each $\ell : \mathsf{List}(A)$ and $a : A$, an element $\mathsf{cons}(a, \ell) : \mathsf{List}(A)$.

We have an obvious inclusion $\eta : A \to \mathsf{List}(A)$ defined by $a \mapsto \mathsf{cons}(a, \mathsf{nil})$. The monoid operation on $\mathsf{List}(A)$ is concatenation, defined inductively by

$$\begin{aligned} \mathsf{nil} \cdot \ell &:\equiv \ell \\ \mathsf{cons}(a, \ell_1) \cdot \ell_2 &:\equiv \mathsf{cons}(a, \ell_1 \cdot \ell_2). \end{aligned}$$

It is straightforward to prove, using the induction principle for $\mathsf{List}(A)$, that $\mathsf{List}(A)$ is a set and that concatenation of lists is associative and has nil as a unit. Thus, $\mathsf{List}(A)$ is a monoid.

Lemma 6.11.5. *For any set A, the type $\mathsf{List}(A)$ is the free monoid on A. In other words, for any monoid G, composition with η is an equivalence*

$$\hom_{\mathrm{Monoid}}(\mathsf{List}(A), G) \simeq (A \to G),$$

where $\hom_{\mathrm{Monoid}}(-,-)$ denotes the set of monoid homomorphisms (functions which preserve the multiplication and unit).

Proof. Given $f : A \to G$, we define $\bar{f} : \mathsf{List}(A) \to G$ by recursion:

$$\bar{f}(\mathsf{nil}) :\equiv e$$
$$\bar{f}(\mathsf{cons}(a, \ell)) :\equiv f(a) \cdot \bar{f}(\ell).$$

It is straightforward to prove by induction that $\bar{f}$ is a monoid homomorphism, and that $f \mapsto \bar{f}$ is a quasi-inverse of $(- \circ \eta)$; see Exercise 6.7. □

This construction of the free monoid is possible because elements of the free monoid have computable canonical forms (namely, finite lists). However, elements of other free (and presented) algebraic structures — such as groups — do not in general have *computable* canonical forms. For instance, equality of words in group presentations is algorithmically undecidable. However, we can still describe free algebraic objects as *higher* inductive types, by simply asserting all the axiomatic equations as path constructors.

For example, let A be a set, and define a higher inductive type $F(A)$ with the following generators.

- A function $\eta : A \to F(A)$.
- A function $m : F(A) \times F(A) \to F(A)$.
- An element $e : F(A)$.
- A function $i : F(A) \to F(A)$.
- For each $x, y, z : F(A)$, an equality $m(x, m(y, z)) = m(m(x, y), z)$.
- For each $x : F(A)$, equalities $m(x, e) = x$ and $m(e, x) = x$.
- For each $x : F(A)$, equalities $m(x, i(x)) = e$ and $m(i(x), x) = e$.
- The 0-truncation constructor: for any $x, y : F(A)$ and $p, q : x = y$, we have $p = q$.

The first constructor says that A maps to $F(A)$. The next three give $F(A)$ the operations of a group: multiplication, an identity element, and inversion. The three constructors after that assert the axioms of a group: associativity, unitality, and inverses. Finally, the last constructor asserts that $F(A)$ is a set.

Therefore, $F(A)$ is a group. It is also straightforward to prove:

Theorem 6.11.6. *$F(A)$ is the free group on A. In other words, for any (set) group G, composition with $\eta : A \to F(A)$ determines an equivalence*

$$\hom_{\mathrm{Group}}(F(A), G) \simeq (A \to G)$$

where $\hom_{\mathrm{Group}}(-,-)$ denotes the set of group homomorphisms between two groups.

Proof. The recursion principle of the higher inductive type $F(A)$ says *precisely* that if G is a group and we have $f : A \to G$, then we have $\bar{f} : F(A) \to G$. Its computation rules say that $\bar{f} \circ \eta \equiv f$, and that $\bar{f}$ is a group homomorphism. Thus, $(- \circ \eta) : \hom_{\mathrm{Group}}(F(A), G) \to (A \to G)$ has a right inverse. It is straightforward to use the induction principle of $F(A)$ to show that this is also a left inverse. □

It is worth taking a step back to consider what we have just done. We have proven that the free group on any set exists *without* giving an explicit construction of it. Essentially all we had to do was write down the universal property that it should satisfy. In set theory, we could achieve a similar result by appealing to black boxes such as the adjoint functor theorem; type theory builds such constructions into the foundations of mathematics.

Of course, it is sometimes also useful to have a concrete description of free algebraic structures. In the case of free groups, we can provide one, using quotients. Consider $\mathsf{List}(A + A)$, where in $A + A$ we write $\mathsf{inl}(a)$ as a, and $\mathsf{inr}(a)$ as $\hat{a}$ (intended to stand for the formal inverse of a). The elements of $\mathsf{List}(A + A)$ are *words* for the free group on A.

Theorem 6.11.7. *Let A be a set, and let $F'(A)$ be the set-quotient of $\mathsf{List}(A + A)$ by the following relations.*

$$(\dots, a_1, a_2, \widehat{a_2}, a_3, \dots) = (\dots, a_1, a_3, \dots)$$
$$(\dots, a_1, \widehat{a_2}, a_2, a_3, \dots) = (\dots, a_1, a_3, \dots).$$

Then $F'(A)$ is also the free group on the set A.

Proof. First we show that $F'(A)$ is a group. We have seen that $\mathsf{List}(A + A)$ is a monoid; we claim that the monoid structure descends to the quotient. We define $F'(A) \times F'(A) \to F'(A)$ by double quotient recursion; it suffices to check that the equivalence relation generated by the given relations is preserved by concatenation of lists. Similarly, we prove the associativity and unit laws by quotent induction.

In order to define inverses in $F'(A)$, we first define $\mathsf{reverse} : \mathsf{List}(B) \to \mathsf{List}(B)$ by recursion on lists:

$$\mathsf{reverse}(\mathsf{nil}) :\equiv \mathsf{nil}$$
$$\mathsf{reverse}(\mathsf{cons}(b, \ell)) :\equiv \mathsf{reverse}(\ell) \cdot \mathsf{cons}(b, \mathsf{nil})$$

Now we define $i : F'(A) \to F'(A)$ by quotient recursion, acting on a list $\ell : \mathsf{List}(A + A)$ by switching the two copies of A and reversing the list. This preserves the relations, hence descends to the quotient. And we can prove that $i(x) \cdot x = e$ for $x : F'(A)$ by induction. First, quotient induction allows us to assume x comes from $\ell : \mathsf{List}(A + A)$, and then we can do list induction:

$$\begin{aligned}
i(\mathsf{nil}) \centerdot \mathsf{nil} &= \mathsf{nil} \centerdot \mathsf{nil} \\
&= \mathsf{nil} \\
i(\mathsf{cons}(a, \ell)) \centerdot \mathsf{cons}(a, \ell) &= i(\ell) \centerdot \mathsf{cons}(\hat{a}, \mathsf{nil}) \centerdot \mathsf{cons}(a, \ell) \\
&= i(\ell) \centerdot \mathsf{cons}(\hat{a}, \mathsf{cons}(a, \ell)) \\
&= i(\ell) \centerdot \ell \\
&= \mathsf{nil} \qquad \text{by the inductive hypothesis.}
\end{aligned}$$

(We have omitted a number of fairly evident lemmas about the behavior of concatenation of lists, etc.)

This completes the proof that $F'(A)$ is a group. Now if G is any group with a function $f : A \to G$, we can define $A + A \to G$ to be f on the first copy of A and f composed with the inversion map of G on the second copy. Now the fact that G is a monoid yields a monoid homomorphism $\mathsf{List}(A + A) \to G$. And since G is a group, this map respects the relations, hence descends to a map $F'(A) \to G$. It is straightforward to prove that this is a group homomorphism, and the unique one which restricts to f on A. □

If A has decidable equality (such as if we assume LEM), then the quotient defining $F'(A)$ can be obtained from an idempotent as in Lemma 6.10.8. We define a word (that is, an element of $\mathsf{List}(A + A)$) to be **reduced** if it contains no adjacent pairs of the form $(a, \hat{a})$ or $(\hat{a}, a)$. When A has decidable equality, it is straightforward to define the **reduction** of a word, which is an idempotent generating the appropriate quotient; we leave the details to the reader.

If $A :\equiv \mathbf{1}$, which has decidable equality, a reduced word must consist either entirely of $\star$'s or entirely of $\hat{\star}$'s. Thus, the free group on $\mathbf{1}$ is equivalent to the integers $\mathbb{Z}$, with 0 corresponding to nil, the positive integer n corresponding to a reduced word of n $\star$'s, and the negative integer $(-n)$ corresponding to a reduced word of n $\hat{\star}$'s. One could also, of course, show directly that $\mathbb{Z}$ has the universal property of $F(\mathbf{1})$.

Remark 6.11.8. Nowhere in the construction of $F(A)$ and $F'(A)$, and the proof of their universal properties, did we use the assumption that A is a set. Thus, we can actually construct the free group on an arbitrary type. Comparing universal properties, we conclude that $F(A) \simeq F(\|A\|_0)$.

We can also use higher inductive types to construct colimits of algebraic objects. For instance, suppose $f : G \to H$ and $g : G \to K$ are group homomorphisms. Their pushout in the category of groups, called the **amalgamated free product** $H *_G K$, can be constructed as the higher inductive type generated by

- Functions $h : H \to H *_G K$ and $k : K \to H *_G K$.
- The operations and axioms of a group, as in the definition of $F(A)$.
- Axioms asserting that h and k are group homomorphisms.
- For $x : G$, we have $h(f(x)) = k(g(x))$.
- The 0-truncation constructor.

On the other hand, it can also be constructed explicitly, as the set-quotient of $\mathsf{List}(H + K)$ by the following relations:

$$\begin{aligned}
(\dots, x_1, x_2, \dots) &= (\dots, x_1 \cdot x_2, \dots) && \text{for } x_1, x_2 : H \\
(\dots, y_1, y_2, \dots) &= (\dots, y_1 \cdot y_2, \dots) && \text{for } y_1, y_2 : K \\
(\dots, 1_G, \dots) &= (\dots, \dots) \\
(\dots, 1_H, \dots) &= (\dots, \dots) \\
(\dots, f(x), \dots) &= (\dots, g(x), \dots) && \text{for } x : G
\end{aligned}$$

We leave the proofs to the reader. In the special case that G is the trivial group, the last relation is unnecessary, and we obtain the **free product** $H * K$, the coproduct in the category of groups. (This

notation unfortunately clashes with that for the *join* of types, as in §6.8, but context generally disambiguates.)

Note that groups defined by *presentations* can be regarded as a special case of colimits. Suppose given a set (or more generally a type) A, and a pair of functions $R \rightrightarrows F(A)$. We regard R as the type of "relations", with the two functions assigning to each relation the two words that it sets equal. For instance, in the presentation $\langle a \mid a^2 = e \rangle$ we would have $A :\equiv \mathbf{1}$ and $R :\equiv \mathbf{1}$, with the two morphisms $R \rightrightarrows F(A)$ picking out the list (a, a) and the empty list nil, respectively. Then by the universal property of free groups, we obtain a pair of group homomorphisms $F(R) \rightrightarrows F(A)$. Their coequalizer in the category of groups, which can be built just like the pushout, is the group *presented* by this presentation.

Note that all these sorts of construction only apply to *algebraic* theories, which are theories whose axioms are (universally quantified) equations referring to variables, constants, and operations from a given signature. They can be modified to apply also to what are called *essentially algebraic theories*: those whose operations are partially defined on a domain specified by equalities between previous operations. But they do not apply, for instance, to the theory of fields, in which the "inversion" operation is partially defined on a domain $\{ x \mid x \# 0 \}$ specified by an *apartness* $\#$ between previous operations, see Theorem 11.2.4. And indeed, it is well-known that the category of fields has no initial object.

On the other hand, these constructions do apply just as well to *infinitary* algebraic theories, whose "operations" can take infinitely many inputs. In such cases, there may not be any presentation of free algebras or colimits of algebras as a simple quotient, unless we assume the axiom of choice. This means that higher inductive types represent a significant strengthening of constructive type theory (not necessarily in terms of proof-theoretic strength, but in terms of practical power), and indeed are stronger in some ways than Zermelo–Fraenkel set theory (without choice). We will see an example of this in §10.3.

6.12 The flattening lemma

As we will see in Chapter 8, amazing things happen when we combine higher inductive types with univalence. The principal way this comes about is that if W is a higher inductive type and $\mathcal{U}$ is a type universe, then we can define a type family $P : W \to \mathcal{U}$ by using the recursion principle for W. When we come to the clauses of the recursion principle dealing with the path constructors of W, we will need to supply paths in $\mathcal{U}$, and this is where univalence comes in.

For example, suppose we have a type X and a self-equivalence $e : X \simeq X$. Then we can define a type family $P : \mathbb{S}^1 \to \mathcal{U}$ by using $\mathbb{S}^1$-recursion:

$$P(\mathsf{base}) :\equiv X$$
$$P(\mathsf{loop}) := \mathsf{ua}(e).$$

The type X thus appears as the fiber $P(\mathsf{base})$ of P over the basepoint. The self-equivalence e is a little more hidden in P, but the following lemma says that it can be extracted by transporting along loop.

Lemma 6.12.1. *Given $B : A \to \mathcal{U}$ and $x, y : A$, with a path $p : x = y$ and an equivalence $e : P(x) \simeq P(y)$ such that $B(p) = \mathsf{ua}(e)$, then for any $u : P(x)$ we have*

$$\mathsf{transport}^B(p, u) = e(u).$$

Proof. Applying Lemma 2.10.5, we have

$$\begin{aligned}\mathsf{transport}^B(p, u) &= \mathsf{idtoeqv}(B(p))(u)\\ &= \mathsf{idtoeqv}(\mathsf{ua}(e))(u)\\ &= e(u).\end{aligned}$$

□

We have seen type families defined by recursion before: in §§2.12 and 2.13 we used them to characterize the identity types of (ordinary) inductive types. In Chapter 8, we will use similar ideas to calculate homotopy groups of higher inductive types.

In this section, we describe a general lemma about type families of this sort which will be useful later on. We call it the **flattening lemma**: it says that if $P : W \to \mathcal{U}$ is defined recursively as above, then its total space $\sum_{(x:W)} P(x)$ is equivalent to a "flattened" higher inductive type, whose constructors may be deduced from those of W and the definition of P. From a category-theoretic point of view, $\sum_{(x:W)} P(x)$ is the "Grothendieck construction" of P, and this expresses its universal property as a "lax colimit".

We will prove here one general case of the flattening lemma, which directly implies many particular cases and suggests the method to prove others. Suppose we have $A, B : \mathcal{U}$ and $f, g : B \to A$, and that the higher inductive type W is generated by

- $\mathsf{c} : A \to W$ and
- $\mathsf{p} : \prod_{(b:B)} (\mathsf{c}(f(b)) =_W \mathsf{c}(g(b)))$.

Using binary sums (coproducts) and dependent sums (Σ-types), a lot of interesting nonrecursive higher inductive types can be represented in this form. All point constructors have to be bundled in the type A and all path constructors in the type B. For instance:

- The circle S^1 can be represented by taking $A :\equiv \mathbf{1}$ and $B :\equiv \mathbf{1}$, with f and g the identity.
- The pushout of $j : X \to Y$ and $k : X \to Z$ can be represented by taking $A :\equiv Y + Z$ and $B :\equiv X$, with $f :\equiv \mathsf{inl} \circ j$ and $g :\equiv \mathsf{inr} \circ k$.

Now suppose in addition that

- $C : A \to \mathcal{U}$ is a family of types over A, and
- $D : \prod_{(b:B)} C(f(b)) \simeq C(g(b))$ is a family of equivalences over B.

Define a type family $P : W \to \mathcal{U}$ inductively by

$$\begin{aligned}P(\mathsf{c}(a)) &:\equiv C(a)\\ P(\mathsf{p}(b)) &:= \mathsf{ua}(D(b)).\end{aligned}$$

Let $\widetilde{W}$ be the higher inductive type generated by

- $\widetilde{\mathsf{c}} : \prod_{(a:A)} C(a) \to \widetilde{W}$ and
- $\widetilde{\mathsf{p}} : \prod_{(b:B)} \prod_{(y:C(f(b)))} (\widetilde{\mathsf{c}}(f(b), y) =_{\widetilde{W}} \widetilde{\mathsf{c}}(g(b), D(b)(y)))$.

The flattening lemma is:

Lemma 6.12.2 (Flattening lemma)**.** *In the above situation, we have*

$$\big(\sum_{x:W} P(x)\big) \;\simeq\; \widetilde{W}.$$

As remarked above, this equivalence can be seen as expressing the universal property of $\sum_{(x:W)} P(x)$ as a "lax colimit" of P over W. It can also be seen as part of the *stability and descent* property of colimits, which characterizes higher toposes.

The proof of Lemma 6.12.2 will occupy the rest of this section. It is somewhat technical and can be skipped on a first reading. But it is also a good example of "proof-relevant mathematics", so we recommend it on a second reading.

The idea is to show that $\sum_{(x:W)} P(x)$ has the same universal property as $\widetilde{W}$. We begin by showing that it comes with analogues of the constructors $\widetilde{\mathsf{c}}$ and $\widetilde{\mathsf{p}}$.

Lemma 6.12.3. *There are functions*

- $\widetilde{\mathsf{c}}' : \prod_{(a:A)} C(a) \to \sum_{(x:W)} P(x)$ *and*
- $\widetilde{\mathsf{p}}' : \prod_{(b:B)} \prod_{(y:C(f(b)))} \Big(\widetilde{\mathsf{c}}'(f(b), y) =_{\sum_{(w:W)} P(w)} \widetilde{\mathsf{c}}'(g(b), D(b)(y))\Big)$.

Proof. The first is easy; we define $\widetilde{\mathsf{c}}'(a, x) :\equiv (\mathsf{c}(a), x)$, noting that $P(\mathsf{c}(a)) \equiv C(a)$ by definition. For the second, suppose given $b : B$ and $y : C(f(b))$; we must give an equality

$$(\mathsf{c}(f(b)), y) = (\mathsf{c}(g(b), D(b)(y))).$$

Since we have $\mathsf{p}(b) : f(b) = g(b)$, by equalities in Σ-types it suffices to give an equality $\mathsf{p}(b)_*(y) = D(b)(y)$. But this follows from Lemma 6.12.1, using the definition of P. □

Now the following lemma says to define a section of a type family over $\sum_{(w:W)} P(w)$, it suffices to give analogous data as in the case of $\widetilde{W}$.

Lemma 6.12.4. *Suppose $Q : \big(\sum_{(x:W)} P(x)\big) \to \mathcal{U}$ is a type family and that we have*

- $c : \prod_{(a:A)} \prod_{(x:C(a))} Q(\widetilde{\mathsf{c}}'(a, x))$ *and*
- $p : \prod_{(b:B)} \prod_{(y:C(f(b)))} \Big(\widetilde{\mathsf{p}}'(b, y)_*(c(f(b), y)) = c(g(b), D(b)(y))\Big)$.

Then there exists $f : \prod_{(z:\sum_{(w:W)} P(w))} Q(z)$ such that $f(\widetilde{\mathsf{c}}'(a, x)) \equiv c(a, x)$.

Proof. Suppose given $w : W$ and $x : P(w)$; we must produce an element $f(w, x) : Q(w, x)$. By induction on w, it suffices to consider two cases. When $w \equiv \mathsf{c}(a)$, then we have $x : C(a)$, and so $c(a, x) : Q(\mathsf{c}(a), x)$ as desired. (This part of the definition also ensures that the stated computational rule holds.)

Now we must show that this definition is preserved by transporting along $\mathsf{p}(b)$ for any $b : B$. Since what we are defining, for all $w : W$, is a function of type $\prod_{(x:P(w))} Q(w,x)$, by Lemma 2.9.7 it will suffice to show that for any $y : C(f(b))$, we have

$$\mathsf{transport}^Q(\mathsf{pair}^=(\mathsf{p}(b), \mathsf{refl}_{\mathsf{p}(b)_*(y)}), c(f(b),y)) = c(g(b), \mathsf{p}(b)_*(y)).$$

Let $q : \mathsf{p}(b)_*(y) = D(b)(y)$ be the path obtained from Lemma 6.12.1. Then we have

$$\begin{aligned} c(g(b), \mathsf{p}(b)_*(y)) &= \mathsf{transport}^{x\mapsto Q(c(g(b),x))}(q^{-1}, c(g(b), D(b)(y))) && \text{(by } \mathsf{apd}_{x\mapsto c(g(b),x)}(q^{-1})\text{)} \\ &= \mathsf{transport}^Q(\mathsf{ap}_{x\mapsto c(g(b),x)}(q^{-1}), c(g(b), D(b)(y))). && \text{(by Lemma 2.3.10)} \end{aligned}$$

Thus, it will suffice to show

$$\mathsf{transport}^Q\Big(\mathsf{pair}^=(\mathsf{p}(b), \mathsf{refl}_{\mathsf{p}(b)_*(y)}),\ c(f(b),y)\Big) = \mathsf{transport}^Q\Big(\mathsf{ap}_{x\mapsto c(g(b),x)}(q^{-1}),\ c(g(b), D(b)(y))\Big).$$

Moving the right-hand transport to the other side, and combining two transports, this is equivalent to

$$\mathsf{transport}^Q\Big(\mathsf{ap}_{x\mapsto c(g(b),x)}(q) \cdot \mathsf{pair}^=(\mathsf{p}(b), \mathsf{refl}_{\mathsf{p}(b)_*(y)}),\ c(f(b),y)\Big) = c(g(b), D(b)(y)).$$

However, we have

$$\mathsf{ap}_{x\mapsto c(g(b),x)}(q) \cdot \mathsf{pair}^=(\mathsf{p}(b), \mathsf{refl}_{\mathsf{p}(b)_*(y)}) = \mathsf{pair}^=(\mathsf{refl}_{g(b)}, q) \cdot \mathsf{pair}^=(\mathsf{p}(b), \mathsf{refl}_{\mathsf{p}(b)_*(y)}) = \mathsf{pair}^=(\mathsf{p}(b), q) = \widetilde{\mathsf{p}}'(b,y)$$

so the construction is completed by the assumption $p(b,y)$ of type

$$\mathsf{transport}^Q(\widetilde{\mathsf{p}}'(b,y), c(f(b),y)) = c(g(b), D(b)(y)). \qquad \square$$

Lemma 6.12.4 *almost* gives $\sum_{(w:W)} P(w)$ the same induction principle as $\widetilde{W}$. What is missing is the (propositional) computation rule $\mathsf{apd}_f(\widetilde{\mathsf{p}}'(b,y)) = p(b,y)$. In order to prove this, we would need to analyze the proof of Lemma 6.12.4, which of course is the definition of f.

It should be possible to do this, but it turns out that we will only need the computation rule for the non-dependent recursion principle. Thus, we now give a somewhat simpler direct construction of the recursor, and a proof of its computation rule.

Lemma 6.12.5. *Suppose Q is a type and that we have*

- $c : \prod_{(a:A)} C(a) \to Q$ *and*
- $p : \prod_{(b:B)} \prod_{(y:C(f(b)))} \Big(c(f(b),y) =_Q c(g(b), D(b)(y))\Big)$.

Then there exists $f : \big(\sum_{(w:W)} P(w)\big) \to Q$ such that $f(\widetilde{\mathsf{c}}'(a,x)) \equiv c(a,x)$.

Proof. As in Lemma 6.12.4, we define $f(w,x)$ by induction on $w : W$. When $w \equiv \mathsf{c}(a)$, we define $f(\mathsf{c}(a),x) :\equiv c(a,x)$. Now by Lemma 2.9.6, it suffices to consider, for $b : B$ and $y : C(f(b))$, the composite path

$$\begin{aligned}
\mathsf{transport}^{x\mapsto Q}(\mathsf{p}(b), c(f(b),y)) &= c(f(b),y) && \text{(by Lemma 2.3.5)} \\
&= c(g(b), D(b)(y)) && \text{(by } p(b,y)\text{)} \\
&= c(g(b), \mathsf{transport}^{P}(\mathsf{p}(b),y)) && \text{(by Lemma 6.12.1)}
\end{aligned} \tag{6.12.6}$$

The computation rule $f(\widetilde{\mathsf{c}}'(a,x)) \equiv c(a,x)$ follows by definition, as before. □

For the second computation rule, we will need the following lemma.

Lemma 6.12.7. *Let $Y : X \to \mathcal{U}$ be a type family and let $f : (\sum_{(x:X)} Y(x)) \to Z$ be defined componentwise by $f(x,y) :\equiv d(x)(y)$ for a curried function $d : \prod_{(x:X)} Y(x) \to Z$. Then for any $s : x_1 =_X x_2$ and any $y_1 : P(x_1)$ and $y_2 : P(x_2)$ with a path $r : s_*(y_1) = y_2$, the path*

$$\mathsf{ap}_f(\mathsf{pair}^{=}(s,r)) : f(x_1,y_1) = f(x_2,y_2)$$

is equal to the composite

$$\begin{aligned}
f(x_1,y_1) &\equiv d(x_1)(y_1) \\
&= \mathsf{transport}^{x\mapsto Q}(s, d(x_1)(y_1)) && \textit{by (Lemma 2.3.5)}^{-1} \\
&= \mathsf{transport}^{x\mapsto Q}(s, d(x_1)({s^{-1}}_*(s_*(y_1)))) \\
&= \left(\mathsf{transport}^{x\mapsto (Y(x)\to Z)}(s, d(x_1))\right)(s_*(y_1)) && \textit{by (2.9.4)}^{-1} \\
&= d(x_2)(s_*(y_1)) && \textit{by } \mathsf{happly}(\mathsf{apd}_d(s))(s_*(y_1)) \\
&= d(x_2)(y_2) && \textit{by } \mathsf{ap}_{d(x_2)}(r) \\
&\equiv f(x_2,y_2).
\end{aligned}$$

Proof. After path induction on s and r, both equalities reduce to reflexivities. □

At first it may seem surprising that Lemma 6.12.7 has such a complicated statement, while it can be proven so simply. The reason for the complication is to ensure that the statement is well-typed: both $\mathsf{ap}_f(\mathsf{pair}^{=}(s,r))$ and the composite path it is claimed to be equal to must have the same start and end points. Once we have managed this, the proof is easy by path induction.

Lemma 6.12.8. *In the situation of Lemma 6.12.5, we have $\mathsf{ap}_f(\widetilde{\mathsf{p}}'(b,y)) = p(b,y)$.*

Proof. Recall that $\widetilde{\mathsf{p}}'(b,y) :\equiv \mathsf{pair}^{=}(\mathsf{p}(b),q)$ where $q : \mathsf{p}(b)_*(y) = D(b)(y)$ comes from Lemma 6.12.1. Thus, since f is defined componentwise, we may compute $\mathsf{ap}_f(\widetilde{\mathsf{p}}'(b,y))$ by Lemma 6.12.7, with

$$\begin{aligned}
x_1 &:\equiv \mathsf{c}(f(b)) & y_1 &:\equiv y \\
x_2 &:\equiv \mathsf{c}(g(b)) & y_2 &:\equiv D(b)(y) \\
s &:\equiv \mathsf{p}(b) & r &:\equiv q.
\end{aligned}$$

The curried function $d : \prod_{(w:W)} P(w) \to Q$ was defined by induction on $w : W$; to apply Lemma 6.12.7 we need to understand $\mathsf{ap}_{d(x_2)}(r)$ and $\mathsf{happly}(\mathsf{apd}_d(s), s_*(y_1))$.

For the first, since $d(\mathsf{c}(a), x) \equiv c(a, x)$, we have

$$\mathsf{ap}_{d(x_2)}(r) \equiv \mathsf{ap}_{c(g(b),-)}(q).$$

For the second, the computation rule for the induction principle of W tells us that $\mathsf{apd}_d(\mathsf{p}(b))$ is equal to the composite (6.12.6), passed across the equivalence of Lemma 2.9.6. Thus, the computation rule given in Lemma 2.9.6 implies that $\mathsf{happly}(\mathsf{apd}_d(\mathsf{p}(b)), \mathsf{p}(b)_*(y))$ is equal to the composite

$$\begin{aligned}
\big(\mathsf{p}(b)_*(c(f(b),-))\big)(\mathsf{p}(b)_*(y)) &= \mathsf{p}(b)_*\Big(c(f(b), \mathsf{p}(b)^{-1}{}_*(\mathsf{p}(b)_*(y)))\Big) && \text{by (2.9.4)} \\
&= \mathsf{p}(b)_*(c(f(b), y)) \\
&= c(f(b), y) && \text{by Lemma 2.3.5} \\
&= c(f(b), D(b)(y)) && \text{by } p(b,y) \\
&= c(f(b), \mathsf{p}(b)_*(y)) && \text{by } \mathsf{ap}_{c(g(b),-)}(q)^{-1}.
\end{aligned}$$

Finally, substituting these values of $\mathsf{ap}_{d(x_2)}(r)$ and $\mathsf{happly}(\mathsf{apd}_d(s), s_*(y_1))$ into Lemma 6.12.7, we see that all the paths cancel out in pairs, leaving only $p(b, y)$. □

Now we are finally ready to prove the flattening lemma.

Proof of Lemma 6.12.2. We define $h : \widetilde{W} \to \sum_{(w:W)} P(w)$ by using the recursion principle for $\widetilde{W}$, with $\widetilde{\mathsf{c}}'$ and $\widetilde{\mathsf{p}}'$ as input data. Similarly, we define $k : (\sum_{(w:W)} P(w)) \to \widetilde{W}$ by using the recursion principle of Lemma 6.12.5, with $\widetilde{\mathsf{c}}$ and $\widetilde{\mathsf{p}}$ as input data.

On the one hand, we must show that for any $z : \widetilde{W}$, we have $k(h(z)) = z$. By induction on z, it suffices to consider the two constructors of $\widetilde{W}$. But we have

$$k(h(\widetilde{\mathsf{c}}(a, x))) \equiv k(\widetilde{\mathsf{c}}'(a, x)) \equiv \widetilde{\mathsf{c}}(a, x)$$

by definition, while similarly

$$k(h(\widetilde{\mathsf{p}}(b, y))) = k\big(\widetilde{\mathsf{p}}'(b, y)\big) = \widetilde{\mathsf{p}}(b, y)$$

using the propositional computation rule for $\widetilde{W}$ and Lemma 6.12.8.

On the other hand, we must show that for any $z : \sum_{(w:W)} P(w)$, we have $h(k(z)) = z$. But this is essentially identical, using Lemma 6.12.4 for "induction on $\sum_{(w:W)} P(w)$" and the same computation rules. □

6.13 The general syntax of higher inductive definitions

In Chapter 5, we discussed the conditions on a putative "inductive definition" which make it acceptable, namely that all inductive occurrences of the type in its constructors are "strictly positive". In this section, we will say something about the additional conditions required for *higher* inductive definitions. Finding a general syntactic description of valid higher inductive definitions is an area of current research, and all of the solutions proposed to date are somewhat

technical in nature; thus we will only give a general description and not a precise definition. Fortunately, the corner cases never seem to arise in practice.

Like an ordinary inductive definition, a higher inductive definition is specified by a list of *constructors*, each of which is a (dependent) function. For simplicity, we may require the inputs of each constructor to satisfy the same condition as the inputs for constructors of ordinary inductive types. In particular, they may contain the type being defined only strictly positively. Note that this excludes definitions such as the 0-truncation as presented in §6.9, where the input of a constructor contains not only the inductive type being defined, but its identity type as well. It may be possible to extend the syntax to allow such definitions; but also, in §7.3 we will give a different construction of the 0-truncation whose constructors do satisfy the more restrictive condition.

The only difference between an ordinary inductive definition and a higher one, then, is that the *output* type of a constructor may be, not the type being defined (W, say), but some identity type of it, such as $u =_W v$, or more generally an iterated identity type such as $p =_{(u=_W v)} q$. Thus, when we give a higher inductive definition, we have to specify not only the inputs of each constructor, but the expressions u and v (or u, v, p, and q, etc.) which determine the source and target of the path being constructed.

Importantly, these expressions may refer to *other* constructors of W. For instance, in the definition of $\mathbb{S}^1$, the constructor loop has both u and v being base, the previous constructor. To make sense of this, we require the constructors of a higher inductive type to be specified *in order*, and we allow the source and target expressions u and v of each constructor to refer to previous constructors, but not later ones. (Of course, in practice the constructors of any inductive definition are written down in some order, but for ordinary inductive types that order is irrelevant.)

Note that this order is not necessarily the order of "dimension": in principle, a 1-dimensional path constructor could refer to a 2-dimensional one and hence need to come after it. However, we have not given the 0-dimensional constructors (point constructors) any way to refer to previous constructors, so they might as well all come first. And if we use the hub-and-spoke construction (§6.7) to reduce all constructors to points and 1-paths, then we might assume that all point constructors come first, followed by all 1-path constructors — but the order among the 1-path constructors continues to matter.

The remaining question is, what sort of expressions can u and v be? We might hope that they could be any expression at all involving the previous constructors. However, the following example shows that a naive approach to this idea does not work.

Example 6.13.1. Suppose that we have some family of functions $f : \prod_{(X:\mathcal{U})}(X \to X)$. Of course, f_X might be just id_X for all X, but other such fs may also exist. For instance, nothing prevents $f_{\mathbf{2}} : \mathbf{2} \to \mathbf{2}$ from being the nonidentity automorphism (see Exercise 6.8).

Now suppose that we attempt to define a higher inductive type K generated by:

- two elements $a, b : K$, and
- a path $\sigma : f_K(a) = f_K(b)$.

What would the induction principle for K say? We would assume a type family $P : K \to \mathcal{U}$, and of course we would need $x : P(a)$ and $y : P(b)$. The remaining datum should be a dependent path in P living over σ, which must therefore connect some element of $P(f_K(a))$ to some element of

$P(f_K(b))$. But what could these elements possibly be? We know that $P(a)$ and $P(b)$ are inhabited by x and y, respectively, but this tells us nothing about $P(f_K(a))$ and $P(f_K(b))$.

Clearly some condition on u and v is required in order for the definition to be sensible. It seems that, just as the domain of each constructor is required to be (among other things) a *covariant functor*, the appropriate condition on the expressions u and v is that they define *natural transformations*. Making precise sense of this requirement is beyond the scope of this book, but informally it means that u and v must only involve operations which are preserved by all functions between types.

For instance, it is permissible for u and v to refer to concatenation of paths, as in the case of the final constructor of the torus in §6.6, since all functions in type theory preserve path concatenation (up to homotopy). However, it is not permissible for them to refer to an operation like the function f in Example 6.13.1, which is not necessarily natural: there might be some function $g : X \to Y$ such that $f_Y \circ g \neq g \circ f_X$. (Univalence implies that f_X must be natural with respect to all *equivalences*, but not necessarily with respect to functions that are not equivalences.)

The intuition of naturality supplies only a rough guide for when a higher inductive definition is permissible. Even if it were possible to give a precise specification of permissible forms of such definitions in this book, such a specification would probably be out of date quickly, as new extensions to the theory are constantly being explored. For instance, the presentation of n-spheres in terms of "dependent n-loops" referred to in §6.4, and the "higher inductive-recursive definitions" used in Chapter 11, were innovations introduced while this book was being written. We encourage the reader to experiment — with caution.

Notes

The general idea of higher inductive types was conceived in discussions between Peter Lumsdaine, Mike Shulman, Andrej Bauer, and Michael Warren at the Oberwolfach meeting in 2011, although there are some suggestions of some special cases in earlier work. So far, not much has been written about them except in blog posts. A general discussion of their syntax, and their semantics in higher-categorical models, will appear in [LS13b]. Dan Licata and Guillaume Brunerie have also contributed substantially to the general theory, especially by finding convenient ways to represent them in computer proof assistants and to do homotopy theory with them (see Chapter 8).

The description of higher spheres using loop spaces and suspensions in §§6.4 and 6.5 is largely due to Dan Licata and Guillaume Brunerie; Favonia has given an alternative description that uses a basic notion of n-dimensional path. The reduction of higher paths to 1-dimensional paths with hubs and spokes (§6.7) is due to Peter Lumsdaine and Mike Shulman. The description of truncation as a higher inductive type is due to Peter Lumsdaine; the (-1)-truncation is closely related to the "bracket types" of [AB04]. The flattening lemma was first formulated in generality by Guillaume Brunerie.

Quotient types are unproblematic in extensional type theory, such as NuPRL [CAB+86]. However, quotients are a trickier issue in intensional type theory (the starting point for homotopy type theory), because the additional propositional equalities added by quotients interfere with the computational interpretation of intensional type theory as a programming language.

Some solutions to this problem have been studied [Hof95, Alt99, AMS07], and several different notions of quotient types for quotients have been considered. The construction of set-quotients using higher-inductives provides an argument for our particular notion (which is very similar to ones that have been considered in the literature), because it arises as an instance of a general mechanism. Our construction does not yet provide a new solution to the computational problems with quotients, since we still lack a good computational understanding of higher inductive types in general—but it does mean that ongoing work on the computational interpretation of higher inductives applies to the quotients as well. The construction of quotients in terms of equivalence classes is, of course, a standard set-theoretic idea, and a well-known aspect of elementary topos theory; its applicability in type theory (which depends on the univalence axiom, at least for mere propositions) was proposed by Vladimir Voevodsky. The fact that quotient types in intensional type theory imply function extensionality was proved by [Hof95], inspired by the work of [Car95] on exact completions; Lemma 6.3.2 is an adaptation of such arguments.

Exercises

Exercise 6.1. Define concatenation of dependent paths, prove that application of dependent functions preserves concatenation, and write out the precise induction principle for the torus T^2 with its computation rules.

Exercise 6.2. Prove that $\Sigma\mathbb{S}^1 \simeq \mathbb{S}^2$, using the explicit definition of $\mathbb{S}^2$ in terms of base and surf given in §6.4.

Exercise 6.3. Define dependent n-loops and the action of dependent functions on n-loops, and write down the induction principle for the n-spheres as defined at the end of §6.4.

Exercise 6.4. Prove that $\Sigma\mathbb{S}^n \simeq \mathbb{S}^{n+1}$, using the definition of $\mathbb{S}^n$ in terms of Ω^n from §6.4.

Exercise 6.5. Prove that if the type $\mathbb{S}^2$ belongs to some universe $\mathcal{U}$, then $\mathcal{U}$ is not a 2-type.

Exercise 6.6. Prove that if G is a monoid and $x : G$, then $\sum_{(y:G)}((x \cdot y = e) \times (y \cdot x = e))$ is a mere proposition. Conclude, using the principle of unique choice (Corollary 3.9.2), that it would be equivalent to define a group to be a monoid such that for every $x : G$, there merely exists a $y : G$ such that $x \cdot y = e$ and $y \cdot x = e$.

Exercise 6.7. Prove that if A is a set, then $\mathsf{List}(A)$ is a monoid. Then complete the proof of Lemma 6.11.5.

Exercise 6.8. Assuming LEM, construct a family of functions $f : \prod_{(X:\mathcal{U})}(X \to X)$ such that $f_{\mathbf{2}} : \mathbf{2} \to \mathbf{2}$ is the nonidentity automorphism.

Chapter 7

Homotopy n-types

One of the basic notions of homotopy theory is that of a *homotopy n-type*: a space containing no interesting homotopy above dimension n. For instance, a homotopy 0-type is essentially a set, containing no nontrivial paths, while a homotopy 1-type may contain nontrivial paths, but no nontrivial paths between paths. Homotopy n-types are also called *n-truncated spaces*. We have mentioned this notion already in §3.1; our first goal in this chapter is to give it a formal definition in homotopy type theory.

A dual notion to truncatedness is connectedness: a space is *n-connected* if it has no interesting homotopy in dimensions n and *below*. For instance, a space is 0-connected if it has only one connected component, and 1-connected (also called "simply connected") if it also has no nontrivial loops (though it may have nontrivial higher loops between loops).

The duality between truncatedness and connectedness is most easily seen by extending both notions to maps. We call a map *n-truncated* or *n-connected* if all its fibers are so. Then n-connected and n-truncated maps form the two classes of maps in an *orthogonal factorization system*, i.e. every map factors uniquely as an n-connected map followed by an n-truncated one.

In the case $n = -1$, the n-truncated maps are the embeddings and the n-connected maps are the surjections, as defined in §4.6. Thus, the n-connected factorization system is a massive generalization of the standard image factorization of a function between sets into a surjection followed by an injection. At the end of this chapter, we sketch briefly an even more general theory: any type-theoretic *modality* gives rise to an analogous factorization system.

7.1 Definition of n-types

As mentioned in §§3.1 and 3.11, it turns out to be convenient to define n-types starting two levels below zero, with the (-1)-types being the mere propositions and the (-2)-types the contractible ones.

Definition 7.1.1. Define the predicate $\mathsf{is}\text{-}n\text{-}\mathsf{type} : \mathcal{U} \to \mathcal{U}$ for $n \geq -2$ by recursion as follows:

$$\mathsf{is}\text{-}n\text{-}\mathsf{type}(X) :\equiv \begin{cases} \mathsf{isContr}(X) & \text{if } n = -2, \\ \prod_{(x,y:X)} \mathsf{is}\text{-}n'\text{-}\mathsf{type}(x =_X y) & \text{if } n = n' + 1 \end{cases}$$

We say that X **is an n-type** if is-n-type(X) is inhabited.

Remark 7.1.2. The number n in Definition 7.1.1 ranges over all integers greater than or equal to -2. We could make sense of this formally by defining a type $\mathbb{Z}_{\geq -2}$ of such integers (a type whose induction principle is identical to that of $\mathbb{N}$), or instead defining a predicate is-$(k-2)$-type for $k : \mathbb{N}$. Either way, we can prove theorems about n-types by induction on n, with $n = -2$ as the base case.

Example 7.1.3. We saw in Lemma 3.11.10 that X is a (-1)-type if and only if it is a mere proposition. Therefore, X is a 0-type if and only if it is a set.

We have also seen that there are types which are not sets (Example 3.1.9). So far, however, we have not shown for any $n > 0$ that there exist types which are not n-types. In Chapter 8, however, we will show that the $(n+1)$-sphere $\mathbb{S}^{n+1}$ is not an n-type. (Kraus has also shown that the n^{th} nested univalent universe is also not an n-type, without using any higher inductive types.) Moreover, in §8.8 will give an example of a type that is not an n-type for *any* (finite) number n.

We begin the general theory of n-types by showing they are closed under certain operations and constructors.

Theorem 7.1.4. *Let $p : X \to Y$ be a retraction and suppose that X is an n-type, for any $n \geq -2$. Then Y is also an n-type.*

Proof. We proceed by induction on n. The base case $n = -2$ is Lemma 3.11.7.

For the inductive step, assume that any retract of an n-type is an n-type, and that X is an $(n+1)$-type. Let $y, y' : Y$; we must show that $y = y'$ is an n-type. Let s be a section of p, and let ϵ be a homotopy $\epsilon : p \circ s \sim 1$. Since X is an $(n+1)$-type, $s(y) =_X s(y')$ is an n-type. We claim that $y = y'$ is a retract of $s(y) =_X s(y')$. For the section, we take

$$\mathsf{ap}_s : (y = y') \to (s(y) = s(y')).$$

For the retraction, we define $t : (s(y) = s(y')) \to (y = y')$ by

$$t(q) :\equiv {\epsilon_y}^{-1} \cdot p(q) \cdot \epsilon_{y'}.$$

To show that t is a retraction of ap_s, we must show that

$${\epsilon_y}^{-1} \cdot p(s(r)) \cdot \epsilon_{y'} = q$$

for any $r : y = y'$. But this follows from Lemma 2.4.4. □

As an immediate corollary we obtain the stability of n-types under equivalence (which is also immediate from univalence):

Corollary 7.1.5. *If $X \simeq Y$ and X is an n-type, then so is Y.*

Recall also the notion of embedding from §4.6.

Theorem 7.1.6. *If $f : X \to Y$ is an embedding and Y is an n-type for some $n \geq -1$, then so is X.*

Proof. Let $x, x' : X$; we must show that $x =_X x'$ is an $(n-1)$-type. But since f is an embedding, we have $(x =_X x') \simeq (f(x) =_Y f(x'))$, and the latter is an $(n-1)$-type by assumption. □

Note that this theorem fails when $n = -2$: the map $\mathbf{0} \to \mathbf{1}$ is an embedding, but $\mathbf{1}$ is a (-2)-type while $\mathbf{0}$ is not.

Theorem 7.1.7. *The hierarchy of n-types is cumulative in the following sense: given a number $n \geq -2$, if X is an n-type, then it is also an $(n+1)$-type.*

Proof. We proceed by induction on n.

For $n = -2$, we need to show that a contractible type, say, A, has contractible path spaces. Let $a_0 : A$ be the center of contraction of A, and let $x, y : A$. We show that $x =_A y$ is contractible. By contractibility of A we have a path $\mathsf{contr}_x \cdot \mathsf{contr}_y^{-1} : x = y$, which we choose as the center of contraction for $x = y$. Given any $p : x = y$, we need to show $p = \mathsf{contr}_x \cdot \mathsf{contr}_y^{-1}$. By identity elimination, it suffices to show that $\mathsf{refl}_x = \mathsf{contr}_x \cdot \mathsf{contr}_x^{-1}$, which is trivial.

For the inductive step, we need to show that $x =_X y$ is an $(n+1)$-type, provided that X is an $(n+1)$-type. Applying the induction hypothesis to $x =_X y$ yields the desired result. □

We now show that n-types are preserved by most type forming operations.

Theorem 7.1.8. *Let $n \geq -2$, and let $A : \mathcal{U}$ and $B : A \to \mathcal{U}$. If A is an n-type and for all $a : A$, $B(a)$ is an n-type, then so is $\sum_{(x:A)} B(x)$.*

Proof. We proceed by induction on n.

For $n = -2$, we choose the center of contraction for $\sum_{(x:A)} B(x)$ to be the pair (a_0, b_0), where $a_0 : A$ is the center of contraction of A and $b_0 : B(a_0)$ is the center of contraction of $B(a_0)$. Given any other element (a, b) of $\sum_{(x:A)} B(x)$, we provide a path $(a, b) = (a_0, b_0)$ by contractibility of A and $B(a_0)$, respectively.

For the inductive step, suppose that A is an $(n+1)$-type and for any $a : A$, $B(a)$ is an $(n+1)$-type. We show that $\sum_{(x:A)} B(x)$ is an $(n+1)$-type: fix (a_1, b_1) and (a_2, b_2) in $\sum_{(x:A)} B(x)$, we show that $(a_1, b_1) = (a_2, b_2)$ is an n-type. By Theorem 2.7.2 we have

$$((a_1, b_1) = (a_2, b_2)) \simeq \sum_{p : a_1 = a_2} (p_*(b_1) =_{B(a_2)} b_2)$$

and by preservation of n-types under equivalences (Corollary 7.1.5) it suffices to prove that the latter is an n-type. This follows from the induction hypothesis. □

As a special case, if A and B are n-types, so is $A \times B$. Note also that Theorem 7.1.7 implies that if A is an n-type, then so is $x =_A y$ for any $x, y : A$. Combining this with Theorem 7.1.8, we see that for any functions $f : A \to C$ and $g : B \to C$ between n-types, the **pullback**

$$A \times_C B :\equiv \sum_{(x:A)} \sum_{(y:B)} (f(x) = g(y))$$

is also an n-type. More generally, n-types are closed under all *limits*.

Theorem 7.1.9. *Let $n \geq -2$, and let $A : \mathcal{U}$ and $B : A \to \mathcal{U}$. If for all $a : A$, $B(a)$ is an n-type, then so is $\prod_{(x:A)} B(x)$.*

Proof. We proceed by induction on n. For $n = -2$, the result is simply Lemma 3.11.6.

For the inductive step, assume the result is true for n-types, and that each $B(a)$ is an $n+1$-type. Let $f, g : \prod_{(a:A)} B(a)$. We need to show that $f = g$ is an n-type. By function extensionality and closure of n-types under equivalence, it suffices to show that $\prod_{(a:A)}(f(a) =_{B(a)} g(a))$ is an n-type. This follows from the inductive hypothesis. □

As a special case of the above theorem, the function space $A \to B$ is an n-type provided that B is an n-type. We can now generalize our observations in Chapter 2 that $\mathsf{isSet}(A)$ and $\mathsf{isProp}(A)$ are mere propositions.

Theorem 7.1.10. *For any $n \geq -2$ and any type X, the type is-n-$\mathsf{type}(X)$ is a mere proposition.*

Proof. We proceed by induction with respect to n.

For the base case, we need to show that for any X, the type $\mathsf{isContr}(X)$ is a mere proposition. By Lemma 3.11.3, it suffices to show that if X is contractible, then $\mathsf{isContr}(X)$ is a mere proposition. But this follows from Corollary 3.11.5.

For the inductive step we need to show

$$\prod_{X:\mathcal{U}} \mathsf{isProp}(\mathsf{is}\text{-}n\text{-}\mathsf{type}(X)) \to \prod_{X:\mathcal{U}} \mathsf{isProp}(\mathsf{is}\text{-}(n+1)\text{-}\mathsf{type}(X))$$

To show the conclusion of this implication, we need to show that for any type X, the type

$$\prod_{x,x':X} \mathsf{is}\text{-}n\text{-}\mathsf{type}(x = x')$$

is a mere proposition. By Theorem 7.1.9 it suffices to show that for any $x, x' : X$, the type is-n-$\mathsf{type}(x =_X x')$ is a mere proposition. But this follows from the induction hypothesis applied to the type $(x =_X x')$. □

Finally, we show that the type of n-types is itself an $(n+1)$-type. We define this to be:

$$n\text{-}\mathsf{Type} :\equiv \sum_{X:\mathcal{U}} \mathsf{is}\text{-}n\text{-}\mathsf{type}(X)$$

If necessary, we may specify the universe $\mathcal{U}$ by writing n-$\mathsf{Type}_{\mathcal{U}}$. In particular, we have $\mathsf{Prop} :\equiv (-1)$-$\mathsf{Type}$ and $\mathsf{Set} :\equiv 0$-Type, as defined in Chapter 2. Note that just as for Prop and Set, because is-n-$\mathsf{type}(X)$ is a mere proposition, by Lemma 3.5.1 for any $(X, p), (X', p') : n$-Type we have

$$\begin{aligned}\Big((X, p) =_{n\text{-}\mathsf{Type}} (X', p')\Big) &\simeq (X =_{\mathcal{U}} X') \\ &\simeq (X \simeq X').\end{aligned}$$

Theorem 7.1.11. *For any $n \geq -2$, the type n-Type is an $(n+1)$-type.*

Proof. Let $(X, p), (X', p') : n$-Type; we need to show that $(X, p) = (X', p')$ is an n-type. By the above observation, this type is equivalent to $X \simeq X'$. Next, we observe that the projection

$$(X \simeq X') \hookrightarrow (X \to X').$$

is an embedding, so that if $n \geq -1$, then by Theorem 7.1.6 it suffices to show that $X \to X'$ is an n-type. But since n-types are preserved under the arrow type, this reduces to an assumption that X' is an n-type.

In the case $n = -2$, this argument shows that $X \simeq X'$ is a (-1)-type — but it is also inhabited, since any two contractible types are equivalent to $\mathbf{1}$, and hence to each other. Thus, $X \simeq X'$ is also a (-2)-type. □

7.2 Uniqueness of identity proofs and Hedberg's theorem

In §3.1 we defined a type X to be a *set* if for all $x, y : X$ and $p, q : x =_X y$ we have $p = q$. In conventional type theory, this property goes by the name of **uniqueness of identity proofs (UIP)**. We have seen also that it is equivalent to being a 0-type in the sense of the previous section. Here is another equivalent characterization, involving Streicher's "axiom K" [Str93]:

Theorem 7.2.1. *A type X is a set if and only if it satisfies **Axiom K**: for all $x : X$ and $p : (x =_A x)$ we have $p = \mathsf{refl}_x$.*

Proof. Clearly Axiom K is a special case of UIP. Conversely, if X satisfies Axiom K, let $x, y : X$ and $p, q : (x = y)$; we want to show $p = q$. But induction on q reduces this goal precisely to Axiom K. □

We stress that *we* are not assuming the K principle as an axiom! It is simply a property which a particular type may or may not satisfy (which is equivalent to being a set).

The following theorem is another useful way to show that types are sets.

Theorem 7.2.2. *Suppose R is a reflexive mere relation on a type X implying identity. Then X is a set, and $R(x, y)$ is equivalent to $x =_X y$ for all $x, y : X$.*

Proof. Let $\rho : \prod_{(x:X)} R(x, x)$ witness the reflexivity of R, and let $f : \prod_{(x,y:X)} R(x, y) \to (x =_X y)$ be a witness that R implies identity. Note first that the two statements in the theorem are equivalent. For on one hand, if X is a set, then $x =_X y$ is a mere proposition, and since it is logically equivalent to the mere proposition $R(x, y)$ by hypothesis, it must also be equivalent to it. On the other hand, if $x =_X y$ is equivalent to $R(x, y)$, then like the latter it is a mere proposition for all $x, y : X$, and hence X is a set.

We give two proofs of this theorem. The first shows directly that X is a set; the second shows directly that $R(x, y) \simeq (x = y)$.

First proof: we show that X is a set. The idea is the same as that of Lemma 3.3.4: the function f must be continuous in its arguments x and y. However, it is slightly more notationally complicated because we have to deal with the additional argument of type $R(x, y)$.

Firstly, for any $x : X$ and $p : x =_X x$, consider $\mathsf{apd}_{f(x)}(p)$. This is a dependent path from $f(x, x)$ to itself. Since $f(x, x)$ is still a function $R(x, x) \to (x =_X y)$, by Lemma 2.9.6 this yields a path

$$p_*(f(x, x, r)) = f(x, x, p_*(r)).$$

On the left-hand side, we have transport in an identity type, which is concatenation. And on the right-hand side, we have $p_*(r) = r$, since both lie in the mere proposition $R(x,x)$. Thus, substituting $r :\equiv \rho(x)$, we obtain

$$f(x,x,\rho(x)) \cdot p = f(x,x,\rho(x)).$$

By cancellation, $p = \mathsf{refl}_x$. So X satisfies Axiom K, and hence is a set.

Second proof: we show that each $f(x,y) : R(x,y) \to x =_X y$ is an equivalence. By Theorem 4.7.7, it suffices to show that f induces an equivalence of total spaces:

$$\big(\sum_{y:X} R(x,y)\big) \simeq \big(\sum_{y:X} x =_X y\big).$$

By Lemma 3.11.8, the type on the right is contractible, so it suffices to show that the type on the left is contractible. As the center of contraction we take the pair $(x,\rho(x))$. It remains to show, for every $y : X$ and every $H : R(x,y)$ that

$$(x,\rho(x)) = (y,H).$$

But since $R(x,y)$ is a mere proposition, by Theorem 2.7.2 it suffices to show that $x =_X y$, which we get from $f(H)$. □

Corollary 7.2.3. *If a type X has the property that $\neg\neg(x = y) \to (x = y)$ for any $x, y : X$, then X is a set.*

Another convenient way to show that a type is a set is by way of the following property.

Definition 7.2.4. A type X has **decidable equality** if for all $x, y : X$ we have

$$(x =_X y) + \neg(x =_X y).$$

In the propositions-as-types language, we can say that X has decidable equality if for every $x, y : X$, either $x = y$ or $x \neq y$. Note that this is a very constructive form of "or". LEM implies (using Example 3.6.2) that *any* type X has *decidable mere equality*, meaning

$$\prod_{x,y:X} \big(\|x = y\| + \|x \neq y\|\big).$$

Definition 7.2.4 asserts moreover that a path $x = y$ can be chosen, when it exists, continuously (or computably, or functorially) in x and y. This turns out to imply that X is a set, by way of Theorem 7.2.2 and the following lemma.

Lemma 7.2.5. *For any type A we have $(A + \neg A) \to (\neg\neg A \to A)$.*

Proof. Suppose $x : A + \neg A$. We have two cases to consider. If x is $\mathsf{inl}(a)$ for some $a : A$, then we have the constant function $\neg\neg A \to A$ which maps everything to a. If x is $\mathsf{inr}(f)$ for some $t : \neg A$, we have $g(t) : \mathbf{0}$ for every $g : \neg\neg A$. Hence we may use *ex falso quodlibet*, that is $\mathsf{rec}_{\mathbf{0}}$, to obtain an element of A for any $g : \neg\neg A$. □

Theorem 7.2.6 (Hedberg). *If X has decidable equality, then X is a set.*

Proof. If X has decidable equality, it follows that $\neg\neg(x = y) \to (x = y)$ for any $x, y : X$. Therefore, Hedberg's theorem follows from Corollary 7.2.3. □

There is, of course, a strong connection between this theorem and Corollary 3.2.7. The form of LEM denied by Corollary 3.2.7 clearly implies that every type has decidable equality, and hence is a set; which we know is not the case.

Recall that in Example 3.1.4 we observed that $\mathbb{N}$ is a set, using our characterization of its equality types in §2.13. A more traditional proof of this theorem uses only (2.13.2) and (2.13.3), rather than the full characterization of Theorem 2.13.1, with Theorem 7.2.6 to fill in the blanks.

Theorem 7.2.7. *The type $\mathbb{N}$ of natural numbers has decidable equality, and hence is a set.*

Proof. Let $x, y : \mathbb{N}$ be given; we proceed by induction on x and case analysis on y to prove $(x = y) + \neg(x = y)$. If $x \equiv 0$ and $y \equiv 0$, we take $\mathsf{inl}(\mathsf{refl}(0))$. If $x \equiv 0$ and $y \equiv \mathsf{succ}(n)$, then by (2.13.2) we get $\neg(0 = \mathsf{succ}(n))$.

For the inductive step, let $x \equiv \mathsf{succ}(n)$. If $y \equiv 0$, we use (2.13.2) again. Finally, if $y \equiv \mathsf{succ}(m)$, the induction hypothesis gives $(m = n) + \neg(m = n)$. In the first case, if $p : m = n$, then $\mathsf{succ}(p) : \mathsf{succ}(m) = \mathsf{succ}(n)$. And in the second case, (2.13.3) yields $\neg(\mathsf{succ}(m) = \mathsf{succ}(n))$. □

Although Hedberg's theorem appears rather special to sets (0-types), "Axiom K" generalizes naturally to n-types. Note that the ordinary Axiom K (as a property of a type X) states that for all $x : X$, the loop space $\Omega(X, x)$ (see Definition 2.1.8) is contractible. Since $\Omega(X, x)$ is always inhabited (by refl_x), this is equivalent to its being a mere proposition (a (-1)-type). Since $0 = (-1) + 1$, this suggests the following generalization.

Theorem 7.2.8. *For any $n \geq -1$, a type X type is an $(n+1)$-type if and only if for all $x : X$, the type $\Omega(X, x)$ is an n-type.*

Before proving this, we prove an auxiliary lemma:

Lemma 7.2.9. *Given $n \geq -1$ and $X : \mathcal{U}$. If, given any inhabitant of X it follows that X is an n-type, then X is an n-type.*

Proof. Let $f : X \to \mathsf{is}\text{-}n\text{-}\mathsf{type}(X)$ be the given map. We need to show that for any $x, x' : X$, the type $x = x'$ is an $(n-1)$-type. But then $f(x)$ shows that X is an n-type, hence all its path spaces are $(n-1)$-types. □

Proof of Theorem 7.2.8. The "only if" direction is obvious, since $\Omega(X, x) :\equiv (x =_X x)$. Conversely, in order to show that X is an $(n+1)$-type, we need to show that for any $x, x' : X$, the type $x = x'$ is an n-type. Following Lemma 7.2.9 it suffices to give a map

$$(x = x') \to \mathsf{is}\text{-}n\text{-}\mathsf{type}(x = x').$$

By path induction, it suffices to do this when $x \equiv x'$, in which case it follows from the assumption that $\Omega(X, x)$ is an n-type. □

By induction and some slightly clever whiskering, we can obtain a generalization of the K property to $n > 0$.

Theorem 7.2.10. *For every $n \geq 0$, a type A is an n-type if and only if $\Omega^{n+1}(A, a)$ is contractible for all $a : A$.*

Proof. The case $n = 0$ is Theorem 7.2.1. By induction, suppose the statement holds for $n : \mathbb{N}$. By Theorem 7.2.8, A is an $(n+1)$-type iff $\Omega(A, a)$ is an n-type for all $a : A$. By the inductive hypothesis, the latter is equivalent to saying that $\Omega^{n+1}(\Omega(A, a), p)$ is contractible for all $p : \Omega(A, a)$.

Since $\Omega^{n+2}(A, a) :\equiv \Omega^{n+1}(\Omega(A, a), \mathsf{refl}_a)$, and $\Omega^{n+1} = \Omega^n \circ \Omega$, it will suffice to show that $\Omega(\Omega(A, a), p)$ is equal to $\Omega(\Omega(A, a), \mathsf{refl}_a)$, in the type $\mathcal{U}_\bullet$ of pointed types. For this, it suffices to give an equivalence

$$g : \Omega(\Omega(A, a), p) \simeq \Omega(\Omega(A, a), \mathsf{refl}_a)$$

which carries the basepoint refl_p to the basepoint $\mathsf{refl}_{\mathsf{refl}_a}$. For $q : p = p$, define $g(q) : \mathsf{refl}_a = \mathsf{refl}_a$ to be the following composite:

$$\mathsf{refl}_a = p \cdot p^{-1} \stackrel{q}{=} p \cdot p^{-1} = \mathsf{refl}_a,$$

where the path labeled "q" is actually $\mathsf{ap}_{\lambda r.\, r \cdot p^{-1}}(q)$. Then g is an equivalence because it is a composite of equivalences

$$(p = p) \xrightarrow{\mathsf{ap}_{\lambda r.\, r \cdot p^{-1}}} (p \cdot p^{-1} = p \cdot p^{-1}) \xrightarrow{i \cdot - \cdot i^{-1}} (\mathsf{refl}_a = \mathsf{refl}_a).$$

using Example 2.4.9 and Theorem 2.11.1, where $i : \mathsf{refl}_a = p \cdot p^{-1}$ is the canonical equality. And it is evident that $g(\mathsf{refl}_p) = \mathsf{refl}_{\mathsf{refl}_a}$. □

7.3 Truncations

In §3.7 we introduced the propositional truncation, which makes the "best approximation" of a type that is a mere proposition, i.e. a (-1)-type. In §6.9 we constructed this truncation as a higher inductive type, and gave one way to generalize it to a 0-truncation. We now explain a better generalization of this, which truncates any type into an n-type for any $n \geq -2$.

The idea is to make use of Theorem 7.2.10, which states that A is an n-type just when $\Omega^{n+1}(A, a)$ is contractible for all $a : A$, and Lemma 6.5.4, which implies that $\Omega^{n+1}(A, a) \simeq \mathsf{Map}_*(\mathbb{S}^{n+1}, (A, a))$, where $\mathbb{S}^{n+1}$ is equipped with some basepoint which we may as well call base. However, contractibility of $\mathsf{Map}_*(\mathbb{S}^{n+1}, (A, a))$ is something that we can ensure directly by giving path constructors.

We might first of all try to define $\|A\|_n$ to be generated by a function $|-|_n : A \to \|A\|_n$, together with for each $r : \mathbb{S}^{n+1} \to \|A\|_n$ and each $x : \mathbb{S}^{n+1}$, a path $s_r(x) : r(x) = r(\mathsf{base})$. But this does not quite work, for the same reason that Remark 6.7.1 fails. Instead, we use the full "hub and spoke" construction as in §6.7.

Thus, we take $\|A\|_n$ to be the higher inductive type generated by:

- a function $|-|_n : A \to \|A\|_n$,

- for each $r : \mathbb{S}^{n+1} \to \|A\|_n$, a *hub* point $h(r) : \|A\|_n$, and
- for each $r : \mathbb{S}^{n+1} \to \|A\|_n$ and each $x : \mathbb{S}^{n+1}$, a *spoke* path $s_r(x) : r(x) = h(r)$.

The existence of these constructors is now enough to show:

Lemma 7.3.1. *$\|A\|_n$ is an n-type.*

Proof. By Theorem 7.2.10, it suffices to show that $\Omega^{n+1}(\|A\|_n, b)$ is contractible for all $b : \|A\|_n$, which by Lemma 6.5.4 is equivalent to $\mathsf{Map}_*(\mathbb{S}^{n+1}, (\|A\|_n, b))$. As center of contraction for the latter, we choose the function $c_b : \mathbb{S}^{n+1} \to \|A\|_n$ which is constant at b, together with $\mathsf{refl}_b : c_b(\mathsf{base}) = b$.

Now, an arbitrary element of $\mathsf{Map}_*(\mathbb{S}^{n+1}, (\|A\|_n, b))$ consists of a map $r : \mathbb{S}^{n+1} \to \|A\|_n$ together with a path $p : r(\mathsf{base}) = b$. By function extensionality, to show $r = c_b$ it suffices to give, for each $x : \mathbb{S}^{n+1}$, a path $r(x) = c_b(x) \equiv b$. We choose this to be the composite $s_r(x) \cdot s_r(\mathsf{base})^{-1} \cdot p$, where $s_r(x)$ is the spoke at x.

Finally, we must show that when transported along this equality $r = c_b$, the path p becomes refl_b. By transport in path types, this means we need

$$(s_r(\mathsf{base}) \cdot s_r(\mathsf{base})^{-1} \cdot p)^{-1} \cdot p = \mathsf{refl}_b.$$

But this is immediate from path operations. □

To show the desired universal property of the n-truncation, we need the induction principle. We extract this from the constructors in the usual way; it says that given $P : \|A\|_n \to \mathcal{U}$ together with

- For each $a : A$, an element $g(a) : P(|a|_n)$,
- For each $r : \mathbb{S}^{n+1} \to \|A\|_n$ and $r' : \prod_{(x:\mathbb{S}^{n+1})} P(r(x))$, an element $h'(r, r') : P(h(r))$.
- For each $r : \mathbb{S}^{n+1} \to \|A\|_n$ and $r' : \prod_{(x:\mathbb{S}^{n+1})} P(r(x))$, and each $x : \mathbb{S}^{n+1}$, a dependent path $r'(x) =^P_{s_r(x)} h'(r, r')$.

there exists a section $f : \prod_{(x:\|A\|_n)} P(x)$ with $f(|a|_n) \equiv g(a)$ for all $a : A$. To make this more useful, we reformulate it as follows.

Theorem 7.3.2. *For any type family $P : \|A\|_n \to \mathcal{U}$ such that each $P(x)$ is an n-type, and any function $g : \prod_{(a:A)} P(|a|_n)$, there exists a section $f : \prod_{(x:\|A\|_n)} P(x)$ such that $f(|a|_n) :\equiv g(a)$ for all $a : A$.*

Proof. It will suffice to construct the second and third data listed above, since g has exactly the type of the first datum. Given $r : \mathbb{S}^{n+1} \to \|A\|_n$ and $r' : \prod_{(x:\mathbb{S}^{n+1})} P(r(x))$, we have $h(r) : \|A\|_n$ and $s_r : \prod_{(x:\mathbb{S}^{n+1})}(r(x) = h(r))$. Define $t : \mathbb{S}^{n+1} \to P(h(r))$ by $t(x) :\equiv s_r(x)_*(r'(x))$. Then since $P(h(r))$ is n-truncated, there exists a point $u : P(h(r))$ and a contraction $v : \prod_{(x:\mathbb{S}^{n+1})}(t(x) = u)$. Define $h'(r, r') :\equiv u$, giving the second datum. Then (recalling the definition of dependent paths), v has exactly the type required of the third datum. □

In particular, if E is some n-type, we can consider the constant family of types equal to E for every point of A. Thus, every map $f : A \to E$ can be extended to a map $\mathsf{ext}(f) : \|A\|_n \to E$ defined by $\mathsf{ext}(f)(|a|_n) :\equiv f(a)$; this is the *recursion principle* for $\|A\|_n$.

The induction principle also implies a uniqueness principle for functions of this form. Namely, if E is an n-type and $g, g' : \|A\|_n \to E$ are such that $g(|a|_n) = g'(|a|_n)$ for every $a : A$, then $g(x) = g'(x)$ for all $x : \|A\|_n$, since the type $g(x) = g'(x)$ is an n-type. Thus, $g = g'$. This yields the following universal property.

Lemma 7.3.3 (Universal property of truncations)**.** *Let $n \geq -2$, $A : \mathcal{U}$ and $B : n\text{-Type}$. The following map is an equivalence:*

$$\begin{cases} (\|A\|_n \to B) & \longrightarrow & A \to B \\ g & \longmapsto & g \circ |-|_n \end{cases}$$

Proof. Given that B is n-truncated, any $f : A \to B$ can be extended to a map $\mathsf{ext}(f) : \|A\|_n \to B$. The map $\mathsf{ext}(f) \circ |-|_n$ is equal to f, because for every $a : A$ we have $\mathsf{ext}(f)(|a|_n) = f(a)$ by definition. And the map $\mathsf{ext}(g \circ |-|_n)$ is equal to g, because they both send $|a|_n$ to $g(|a|_n)$. □

In categorical language, this says that the n-types form a *reflective subcategory* of the category of types. (To state this fully precisely, one ought to use the language of $(\infty, 1)$-categories.) In particular, this implies that the n-truncation is functorial: given $f : A \to B$, applying the recursion principle to the composite $A \xrightarrow{f} B \to \|B\|_n$ yields a map $\|f\|_n : \|A\|_n \to \|B\|_n$. By definition, we have a homotopy

$$\mathsf{nat}_n^f : \prod_{a:A} \|f\|_n(|a|_n) = |f(a)|_n, \tag{7.3.4}$$

expressing *naturality* of the maps $|-|_n$.

Uniqueness implies functoriality laws such as $\|g \circ f\|_n = \|g\|_n \circ \|f\|_n$ and $\|\mathsf{id}_A\|_n = \mathsf{id}_{\|A\|_n}$, with attendant coherence laws. We also have higher functoriality, for instance:

Lemma 7.3.5. *Given $f, g : A \to B$ and a homotopy $h : f \sim g$, there is an induced homotopy $\|h\|_n : \|f\|_n \sim \|g\|_n$ such that the composite*

$$|f(a)|_n \overset{\mathsf{nat}_n^f(a)^{-1}}{=\!=} \|f\|_n(|a|_n) \overset{\|h\|_n(|a|_n)}{=\!=} \|g\|_n(|a|_n) \overset{\mathsf{nat}_n^g(a)}{=\!=} |g(a)|_n \tag{7.3.6}$$

is equal to $\mathsf{ap}_{|-|_n}(h(a))$.

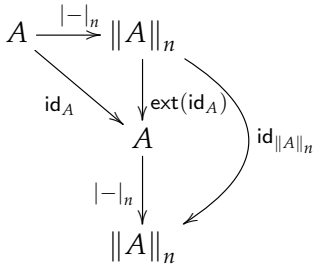
Proof. First, we indeed have a homotopy with components $\mathsf{ap}_{|-|_n}(h(a)) : |f(a)|_n = |g(a)|_n$. Composing on either sides with the paths $|f(a)|_n = \|f\|_n(|a|_n)$ and $|g(a)|_n = \|g\|_n(|a|_n)$, which arise from the definitions of $\|f\|_n$ and $\|g\|_n$, we obtain a homotopy $(\|f\|_n \circ |-|_n) \sim (\|g\|_n \circ |-|_n)$, and hence an equality by function extensionality. But since $(- \circ |-|_n)$ is an equivalence, there must be a path $\|f\|_n = \|g\|_n$ inducing it, and the coherence laws for function extensionality imply (7.3.6). □

The following observation about reflective subcategories is also standard.

Corollary 7.3.7. *A type A is an n-type if and only if the map $|-|_n : A \to \|A\|_n$ is an equivalence.*

Proof. "If" follows from closure of n-types under equivalence. On the other hand, if A is an n-type, we can define $\mathsf{ext}(\mathsf{id}_A) : \|A\|_n \to A$. Then we have $\mathsf{ext}(\mathsf{id}_A) \circ |-|_n = \mathsf{id}_A : A \to A$ by

definition. In order to prove that $|-|_n \circ \mathsf{ext}(\mathsf{id}_A) = \mathsf{id}_{\|A\|_n}$, we only need to prove that $|-|_n \circ \mathsf{ext}(\mathsf{id}_A) \circ |-|_n = \mathsf{id}_{\|A\|_n} \circ |-|_n$. This is again true:



□

The category of n-types also has some special properties not possesed by all reflective subcategories. For instance, the reflector $\|-\|_n$ preserves finite products.

Theorem 7.3.8. *For any types A and B, the induced map $\|A \times B\|_n \to \|A\|_n \times \|B\|_n$ is an equivalence.*

Proof. It suffices to show that $\|A\|_n \times \|B\|_n$ has the same universal property as $\|A \times B\|_n$. Thus, let C be an n-type; we have

$$\begin{aligned}(\|A\|_n \times \|B\|_n \to C) &= (\|A\|_n \to (\|B\|_n \to C)) \\ &= (\|A\|_n \to (B \to C)) \\ &= (A \to (B \to C)) \\ &= (A \times B \to C)\end{aligned}$$

using the universal properties of $\|B\|_n$ and $\|A\|_n$, along with the fact that $B \to C$ is an n-type since C is. It is straightforward to verify that this equivalence is given by composing with $|-|_n \times |-|_n$, as needed. □

We can characterize the path spaces of a truncation using the same method that we used in §§2.12 and 2.13 for coproducts and natural numbers (and which we will use in Chapter 8 to calculate homotopy groups). Unsurprisingly, the path spaces in the $(n+1)$-truncation of A are the n-truncations of the path spaces of A. Indeed, for any $x, y : A$ there is a canonical map

$$f : \|x =_A y\|_n \to \left(|x|_{n+1} =_{\|A\|_{n+1}} |y|_{n+1}\right) \tag{7.3.9}$$

defined by

$$f(|p|_n) :\equiv \mathsf{ap}_{|-|_{n+1}}(p).$$

This definition uses the recursion principle for $\|-\|_n$, which is correct because $\|A\|_{n+1}$ is $(n+1)$-truncated, so that the codomain of f is n-truncated.

Theorem 7.3.10. *For any A and $x, y : A$ and $n \geq -2$, the map (7.3.9) is an equivalence; thus we have*

$$\|x =_A y\|_n \simeq \left(|x|_{n+1} =_{\|A\|_{n+1}} |y|_{n+1}\right).$$

Proof. The proof is a simple application of the encode-decode method: As in previous situations, we cannot directly define a quasi-inverse to (7.3.9) because there is no way to induct on an equality between $|x|_{n+1}$ and $|y|_{n+1}$. Thus, instead we generalize its type, in order to have general elements of the type $\|A\|_{n+1}$ instead of $|x|_{n+1}$ and $|y|_{n+1}$. Define $P : \|A\|_{n+1} \to \|A\|_{n+1} \to n\text{-Type}$ by

$$P(|x|_{n+1}, |y|_{n+1}) :\equiv \|x =_A y\|_n$$

This definition is correct because $\|x =_A y\|_n$ is n-truncated, and n-Type is $(n+1)$-truncated by Theorem 7.1.11. Now for every $u, v : \|A\|_{n+1}$, there is a map

$$\mathsf{encode} : P(u, v) \to (u =_{\|A\|_{n+1}} v)$$

defined for $u = |x|_{n+1}$ and $v = |y|_{n+1}$ and $p : x = y$ by

$$\mathsf{encode}(|p|_n) :\equiv \mathsf{ap}_{|-|_{n+1}}(p).$$

Since the codomain of encode is n-truncated, it suffices to define it only for u and v of this form, and then it's just the same definition as before. We also define a function

$$r : \prod_{u:\|A\|_{n+1}} P(u, u)$$

by induction on u, where $r(|x|_{n+1}) :\equiv |\mathsf{refl}_x|_n$.

Now we can define an inverse map

$$\mathsf{decode} : (u =_{\|A\|_{n+1}} v) \to P(u, v)$$

by

$$\mathsf{decode}(p) :\equiv \mathsf{transport}^{v \mapsto P(u,v)}(p, r(u)).$$

To show that the composite

$$(u =_{\|A\|_{n+1}} v) \xrightarrow{\mathsf{decode}} P(u, v) \xrightarrow{\mathsf{encode}} (u =_{\|A\|_{n+1}} v)$$

is the identity function, by path induction it suffices to check it for $\mathsf{refl}_u : u = u$, in which case what we need to know is that $\mathsf{decode}(r(u)) = \mathsf{refl}_u$. But since this is an n-type, hence also an $(n+1)$-type, we may assume $u \equiv |x|_{n+1}$, in which case it follows by definition of r and decode. Finally, to show that

$$P(u, v) \xrightarrow{\mathsf{encode}} (u =_{\|A\|_{n+1}} v) \xrightarrow{\mathsf{decode}} P(u, v)$$

is the identity function, since this goal is again an n-type, we may assume that $u = |x|_{n+1}$ and $v = |y|_{n+1}$ and that we are considering $|p|_n : P(|x|_{n+1}, |y|_{n+1})$ for some $p : x = y$. Then we have

$$\begin{aligned}
\mathsf{decode}(\mathsf{encode}(|p|_n)) &= \mathsf{decode}(\mathsf{ap}_{|-|_{n+1}}(p)) \\
&= \mathsf{transport}^{v \mapsto P(|x|_{n+1}, v)}(\mathsf{ap}_{|-|_{n+1}}(p), |\mathsf{refl}_x|_n) \\
&= \mathsf{transport}^{v \mapsto \|u=v\|_n}(p, |\mathsf{refl}_x|_n) \\
&= \left|\mathsf{transport}^{v \mapsto (u=v)}(p, \mathsf{refl}_x)\right|_n \\
&= |p|_n.
\end{aligned}$$

This completes the proof that encode and decode are quasi-inverses. The stated result is then the special case where $u = |x|_{n+1}$ and $v = |y|_{n+1}$. □

Corollary 7.3.11. *Let $n \geq -2$ and (A, a) be a pointed type. Then*

$$\left\|\Omega(A,a)\right\|_n = \Omega\left(\|(A,a)\|_{n+1}\right)$$

Proof. This is a special case of the previous lemma where $x = y = a$. □

Corollary 7.3.12. *Let $n \geq -2$ and $k \geq 0$ and (A, a) a pointed type. Then*

$$\left\|\Omega^k(A,a)\right\|_n = \Omega^k\left(\|(A,a)\|_{n+k}\right).$$

Proof. By induction on k, using the recursive definition of Ω^k. □

We also observe that "truncations are cumulative": if we truncate to an n-type and then to a k-type with $k \leq n$, then we might as well have truncated directly to a k-type.

Lemma 7.3.13. *Let $k, n \geq -2$ with $k \leq n$ and $A : \mathcal{U}$. Then $\|\|A\|_n\|_k = \|A\|_k$.*

Proof. We define two maps $f : \|\|A\|_n\|_k \to \|A\|_k$ and $g : \|A\|_k \to \|\|A\|_n\|_k$ by

$$f(||a|_n|_k) :\equiv |a|_k \qquad \text{and} \qquad g(|a|_k) :\equiv ||a|_n|_k.$$

The map f is well-defined because $\|A\|_k$ is k-truncated and also n-truncated (because $k \leq n$), and the map g is well-defined because $\|\|A\|_n\|_k$ is k-truncated.

The composition $f \circ g : \|A\|_k \to \|A\|_k$ satisfies $(f \circ g)(|a|_k) = |a|_k$, hence $f \circ g = \mathsf{id}_{\|A\|_k}$. Similarly, we have $(g \circ f)(||a|_n|_k) = ||a|_n|_k$ and hence $g \circ f = \mathsf{id}_{\|\|A\|_n\|_k}$. □

7.4 Colimits of n-types

Recall that in §6.8, we used higher inductive types to define pushouts of types, and proved their universal property. In general, a (homotopy) colimit of n-types may no longer be an n-type. However, if we n-truncate it, we obtain an n-type which satisfies the correct universal property with respect to other n-types.

In this section we prove this formally for the case of pushouts, which is the most important and nontrivial one. Recall the following definitions from §6.8.

Definition 7.4.1. A **span** is a 5-tuple $\mathscr{D} = (A, B, C, f, g)$ with $f : C \to A$ and $g : C \to B$.

$$\mathscr{D} = \quad \begin{array}{ccc} C & \xrightarrow{g} & B \\ {\scriptstyle f}\downarrow & & \\ A & & \end{array}$$

Definition 7.4.2. Given a span $\mathscr{D} = (A, B, C, f, g)$ and a type D, a **cocone under $\mathscr{D}$ with base D** is a triple (i, j, h) with $i : A \to D$, $j : B \to D$ and $h : \prod_{(c:C)} i(f(c)) = j(g(c))$:

$$\begin{array}{ccc} C & \xrightarrow{g} & B \\ {\scriptstyle f}\downarrow & \swarrow^{h} & \downarrow{\scriptstyle j} \\ A & \xrightarrow[i]{} & D \end{array}$$

We denote by $\mathrm{cocone}_{\mathscr{D}}(D)$ the type of all such cocones.

The type of cocones is (covariantly) functorial. For instance, given D, E and a map $t : D \to E$, there is a map

$$\begin{cases} \mathrm{cocone}_{\mathscr{D}}(D) & \longrightarrow & \mathrm{cocone}_{\mathscr{D}}(E) \\ c & \longmapsto & t \circ c \end{cases}$$

defined by:

$$t \circ (i, j, h) = (t \circ i, t \circ j, \mathsf{ap}_t \circ h)$$

And given D, E, F, functions $t : D \to E$, $u : E \to F$ and $c : \mathrm{cocone}_{\mathscr{D}}(D)$, we have

$$\mathsf{id}_D \circ c = c \tag{7.4.3}$$

$$(u \circ t) \circ c = u \circ (t \circ c) \tag{7.4.4}$$

Definition 7.4.5. Given a span $\mathscr{D}$ of n-types, an n-type D, and a cocone $c : \mathrm{cocone}_{\mathscr{D}}(D)$, the pair (D, c) is said to be a **pushout of $\mathscr{D}$ in n-types** if for every n-type E, the map

$$\begin{cases} (D \to E) & \longrightarrow & \mathrm{cocone}_{\mathscr{D}}(E) \\ t & \longmapsto & t \circ c \end{cases}$$

is an equivalence.

In order to construct pushouts of n-types, we need to explain how to reflect spans and cocones.

Definition 7.4.6. Let

$$\mathscr{D} = \begin{array}{ccc} C & \xrightarrow{g} & B \\ {\scriptstyle f}\downarrow & & \\ A & & \end{array}$$

be a span. We denote by $\|\mathscr{D}\|_n$ the following span of n-types:

$$\|\mathscr{D}\|_n :\equiv \begin{array}{ccc} \|C\|_n & \xrightarrow{\|g\|_n} & \|B\|_n \\ {\scriptstyle \|f\|_n}\downarrow & & \\ \|A\|_n & & \end{array}$$

Definition 7.4.7. Let $D : \mathcal{U}$ and $c = (i, j, h) : \mathrm{cocone}_{\mathscr{D}}(D)$. We define

$$\|c\|_n = (\|i\|_n, \|j\|_n, \|h\|_n) : \mathrm{cocone}_{\|\mathscr{D}\|_n}(\|D\|_n)$$

where $\|h\|_n : \|i\|_n \circ \|f\|_n \sim \|j\|_n \circ \|g\|_n$ is defined as in Lemma 7.3.5.

We now observe that the maps from each type to its n-truncation assemble into a map of spans, in the following sense.

Definition 7.4.8. Let

$$\mathscr{D} = \begin{array}{ccc} C & \xrightarrow{g} & B \\ {\scriptstyle f}\downarrow & & \\ A & & \end{array} \qquad \text{and} \qquad \mathscr{D}' = \begin{array}{ccc} C' & \xrightarrow{g'} & B' \\ {\scriptstyle f'}\downarrow & & \\ A' & & \end{array}$$

be spans. A **map of spans** $\mathscr{D} \to \mathscr{D}'$ consists of functions $\alpha : A \to A'$, $\beta : B \to B'$, and $\gamma : C \to C'$ and homotopies $\phi : \alpha \circ f \sim f' \circ \gamma$ and $\psi : \beta \circ g \sim g' \circ \gamma$.

Thus, for any span $\mathscr{D}$, we have a map of spans $|-|_n^{\mathscr{D}} : \mathscr{D} \to \|\mathscr{D}\|_n$ consisting of $|-|_n^A$, $|-|_n^B$, $|-|_n^C$, and the naturality homotopies nat_n^f and nat_n^g from (7.3.4).

We also need to know that maps of spans behave functorially. Namely, if $(\alpha, \beta, \gamma, \phi, \psi) : \mathscr{D} \to \mathscr{D}'$ is a map of spans and D any type, then we have

$$\begin{cases} \mathrm{cocone}_{\mathscr{D}'}(D) & \longrightarrow & \mathrm{cocone}_{\mathscr{D}}(D) \\ (i, j, h) & \longmapsto & (i \circ \alpha, j \circ \beta, k) \end{cases}$$

where $k : \prod_{(z:C)} i(\alpha(f(z))) = j(\beta(g(z)))$ is the composite

$$i(\alpha(f(z))) \overset{\mathsf{ap}_i(\phi)}{=\!=} i(f'(\gamma(z))) \overset{h(\gamma(z))}{=\!=} j(g'(\gamma(z))) \overset{\mathsf{ap}_j(\psi)}{=\!=} j(\beta(g(z))). \tag{7.4.9}$$

We denote this cocone by $(i, j, h) \circ (\alpha, \beta, \gamma, \phi, \psi)$. Moreover, this functorial action commutes with the other functoriality of cocones:

Lemma 7.4.10. *Given $(\alpha, \beta, \gamma, \phi, \psi) : \mathscr{D} \to \mathscr{D}'$ and $t : D \to E$, the following diagram commutes:*

$$\begin{array}{ccc} \mathrm{cocone}_{\mathscr{D}'}(D) & \xrightarrow{t\circ -} & \mathrm{cocone}_{\mathscr{D}'}(E) \\ \downarrow & & \downarrow \\ \mathrm{cocone}_{\mathscr{D}}(D) & \xrightarrow[t\circ -]{} & \mathrm{cocone}_{\mathscr{D}}(E) \end{array}$$

Proof. Given $(i, j, h) : \mathrm{cocone}_{\mathscr{D}'}(D)$, note that both composites yield a cocone whose first two components are $t \circ i \circ \alpha$ and $t \circ j \circ \beta$. Thus, it remains to verify that the homotopies agree. For the top-right composite, the homotopy is (7.4.9) with (i, j, h) being replaced by $(t \circ i, t \circ j, \mathsf{ap}_t \circ h)$:

$$t(i(\alpha(f(z)))) \overset{\mathsf{ap}_{t\circ i}(\phi)}{=\!=} t(i(f'(\gamma(z)))) \overset{\mathsf{ap}_t(h(\gamma(z)))}{=\!=} t(j(g'(\gamma(z)))) \overset{\mathsf{ap}_{t\circ j}(\psi)}{=\!=} t(j(\beta(g(z)))).$$

On the other hand, for the left-bottom composite, the homotopy is ap_t applied to (7.4.9). Since ap respects path-concatenation, this is equal to

$$t(i(\alpha(f(z)))) \overset{\mathsf{ap}_t(\mathsf{ap}_i(\phi))}{=\!=} t(i(f'(\gamma(z)))) \overset{\mathsf{ap}_t(h(\gamma(z)))}{=\!=} t(j(g'(\gamma(z)))) \overset{\mathsf{ap}_t(\mathsf{ap}_j(\psi))}{=\!=} t(j(\beta(g(z)))).$$

But $\mathsf{ap}_t \circ \mathsf{ap}_i = \mathsf{ap}_{t\circ i}$ and similarly for j, so these two homotopies are equal. □

Finally, note that since we defined $\|c\|_n : \mathrm{cocone}_{\|\mathscr{D}\|_n}(\|D\|_n)$ using Lemma 7.3.5, the additional condition (7.3.6) implies

$$|-|^D_n \circ c = \|c\|_n \circ |-|^{\mathscr{D}}_n. \tag{7.4.11}$$

for any $c : \mathrm{cocone}_{\mathscr{D}}(D)$. Now we can prove our desired theorem.

Theorem 7.4.12. *Let $\mathscr{D}$ be a span and (D, c) its pushout. Then $(\|D\|_n, \|c\|_n)$ is a pushout of $\|\mathscr{D}\|_n$ in n-types.*

Proof. Let E be an n-type, and consider the following diagram:

$$\begin{array}{ccc}
(\|D\|_n \to E) & \xrightarrow{-\circ|-|^D_n} & (D \to E) \\
{\scriptstyle -\circ\|c\|_n}\downarrow & & \downarrow{\scriptstyle -\circ c} \\
\mathrm{cocone}_{\|\mathscr{D}\|_n}(E) & \xrightarrow{-\circ|-|^{\mathscr{D}}_n} & \mathrm{cocone}_{\mathscr{D}}(E) \\
{\scriptstyle \ell_1}\uparrow & & \uparrow{\scriptstyle \ell_2} \\
(\|A\|_n \to E) \times_{(\|C\|_n \to E)} (\|B\|_n \to E) & \longrightarrow & (A \to E) \times_{(C\to E)} (B \to E)
\end{array}$$

The upper horizontal arrow is an equivalence since E is an n-type, while $- \circ c$ is an equivalence since c is a pushout cocone. Thus, by the 2-out-of-3 property, to show that $- \circ \|c\|_n$ is an equivalence, it will suffice to show that the upper square commutes and that the middle horizontal arrow is an equivalence. To see that the upper square commutes, let $t : \|D\|_n \to E$; then

$$\begin{aligned}
\left(t \circ \|c\|_n\right) \circ |-|^{\mathscr{D}}_n &= t \circ \left(\|c\|_n \circ |-|^{\mathscr{D}}_n\right) && \text{by Lemma 7.4.10} \\
&= t \circ \left(|-|^D_n \circ c\right) && \text{by (7.4.11)} \\
&= \left(t \circ |-|^D_n\right) \circ c && \text{by (7.4.4).}
\end{aligned}$$

To show that the middle horizontal arrow is an equivalence, consider the lower square. The two lower vertical arrows are simply applications of happly:

$$\begin{aligned}
\ell_1(i, j, p) &:\equiv (i, j, \mathsf{happly}(p)) \\
\ell_2(i, j, p) &:\equiv (i, j, \mathsf{happly}(p))
\end{aligned}$$

and hence are equivalences by function extensionality. The lowest horizontal arrow is defined by

$$(i, j, p) \mapsto \left(i \circ |-|^A_n,\ \ j \circ |-|^B_n,\ \ q\right)$$

where q is the composite

$$\begin{aligned} i \circ |-|_n^A \circ f &= i \circ \|f\|_n \circ |-|_n^C && \text{by } \mathsf{funext}(\lambda z.\, \mathsf{ap}_i(\mathsf{nat}_n^f(z))) \\ &= j \circ \|g\|_n \circ |-|_n^C && \text{by } \mathsf{ap}_{-\circ|-|_n^C}(p) \\ &= j \circ |-|_n^B \circ g && \text{by } \mathsf{funext}(\lambda z.\, \mathsf{ap}_j(\mathsf{nat}_n^g(z))). \end{aligned}$$

This is an equivalence, because it is induced by an equivalence of cospans. Thus, by 2-out-of-3, it will suffice to show that the lower square commutes. But the two composites around the lower square agree definitionally on the first two components, so it suffices to show that for (i, j, p) in the lower-left corner and $z : C$, the path

$$\mathsf{happly}(q, z) : i(|f(z)|_n) = j(|g(z)|_n)$$

(with q as above) is equal to the composite

$$\begin{aligned} i(|f(z)|_n) &= i(\|f\|_n(|z|_n)) && \text{by } \mathsf{ap}_i(\mathsf{nat}_n^f(z)) \\ &= j(\|g\|_n(|z|_n)) && \text{by } \mathsf{happly}(p, |z|_n) \\ &= j(|g(z)|_n) && \text{by } \mathsf{ap}_j(\mathsf{nat}_n^g(z)). \end{aligned}$$

However, since happly is functorial, it suffices to check equality for the three component paths:

$$\begin{aligned} \mathsf{happly}(\mathsf{funext}(\lambda z.\, \mathsf{ap}_i(\mathsf{nat}_n^f(z))), z) &= \mathsf{ap}_i(\mathsf{nat}_n^f(z)) \\ \mathsf{happly}(\mathsf{ap}_{-\circ|-|_n^C}(p), z) &= \mathsf{happly}(p, |z|_n) \\ \mathsf{happly}(\mathsf{funext}(\lambda z.\, \mathsf{ap}_j(\mathsf{nat}_n^g(z))), z) &= \mathsf{ap}_j(\mathsf{nat}_n^g(z)). \end{aligned}$$

The first and third of these are just the fact that happly is quasi-inverse to funext, while the second is an easy general lemma about happly and precomposition. □

7.5 Connectedness

An n-type is one that has no interesting information above dimension n. By contrast, an *n-connected type* is one that has no interesting information *below* dimension n. It turns out to be natural to study a more general notion for functions as well.

Definition 7.5.1. A function $f : A \to B$ is said to be **n-connected** if for all $b : B$, the type $\left\|\mathsf{fib}_f(b)\right\|_n$ is contractible:

$$\mathsf{conn}_n(f) :\equiv \prod_{b:B} \mathsf{isContr}(\left\|\mathsf{fib}_f(b)\right\|_n).$$

A type A is said to be **n-connected** if the unique function $A \to \mathbf{1}$ is n-connected, i.e. if $\|A\|_n$ is contractible.

Thus, a function $f : A \to B$ is n-connected if and only if $\mathsf{fib}_f(b)$ is n-connected for every $b : B$. Of course, every function is (-2)-connected. At the next level, we have:

Lemma 7.5.2. *A function f is (-1)-connected if and only if it is surjective in the sense of §4.6.*

Proof. We defined f to be surjective if $\left\|\mathsf{fib}_f(b)\right\|$ is inhabited for all b. But since it is a mere proposition, inhabitation is equivalent to contractibility. □

Thus, n-connectedness of a function for $n \geq 0$ can be thought of as a strong form of surjectivity. Category-theoretically, (-1)-connectedness corresponds to essential surjectivity on objects, while n-connectedness corresponds to essential surjectivity on k-morphisms for $k \leq n+1$.

Remark 7.5.3. While our notion of n-connectedness for types agrees with the standard notion in homotopy theory, our notion of n-connectedness for *functions* is off by one from a common indexing in classical homotopy theory. Whereas we say a function f is n-connected if all its fibers are n-connected, some classical homotopy theorists would call such a function $(n+1)$-connected. (This is due to a historical focus on *cofibers* rather than fibers.)

We now observe a few closure properties of connected maps.

Lemma 7.5.4. *Suppose that g is a retract of a n-connected function f. Then g is n-connected.*

Proof. This is a direct consequence of Lemma 4.7.3. □

Corollary 7.5.5. *If g is homotopic to a n-connected function f, then g is n-connected.*

Lemma 7.5.6. *Suppose that $f : A \to B$ is n-connected. Then $g : B \to C$ is n-connected if and only if $g \circ f$ is n-connected.*

Proof. Let $c : C$. We have $\mathsf{fib}_{g\circ f}(c) \simeq \sum_{(w:\mathsf{fib}_g(c))} \mathsf{fib}_f(\mathsf{pr}_1 w)$ and, so

$$\begin{aligned}\left\|\mathsf{fib}_{g\circ f}(c)\right\|_n &\simeq \Big\| \sum_{w:\mathsf{fib}_g(c)} \mathsf{fib}_f(\mathsf{pr}_1 w) \Big\|_n \\ &\simeq \Big\| \sum_{w:\mathsf{fib}_g(c)} \left\|\mathsf{fib}_f(\mathsf{pr}_1 w)\right\|_n \Big\|_n \\ &\simeq \left\|\mathsf{fib}_g(c)\right\|_n.\end{aligned}$$

It follows that $\left\|\mathsf{fib}_g(c)\right\|_n$ is contractible if and only if $\left\|\mathsf{fib}_{g\circ f}(c)\right\|_n$ is contractible. □

Importantly, n-connected functions can be equivalently characterized as those which satisfy an "induction principle" with respect to n-types. This idea will lead directly into our proof of the Freudenthal suspension theorem in §8.6.

Lemma 7.5.7. *For $f : A \to B$ and $P : B \to \mathcal{U}$, consider the following function:*

$$\lambda s.\, s \circ f : \Big(\prod_{b:B} P(b)\Big) \to \Big(\prod_{a:A} P(f(a))\Big).$$

For a fixed f and $n \geq -2$, the following are equivalent.

(i) f is n-connected.

(ii) For every $P : B \to n\text{-}\mathsf{Type}$, the map $\lambda s.\, s \circ f$ is an equivalence.

(iii) For every $P : B \to n\text{-}\mathsf{Type}$, the map $\lambda s.\, s \circ f$ has a section.

Proof. Suppose that f is n-connected and let $P : B \to n\text{-Type}$. Then we have the equivalences

$$\begin{aligned}\prod_{b:B} P(b) &\simeq \prod_{b:B} \left\|\mathsf{fib}_f(b)\right\|_n \to P(b)\\ &\simeq \prod_{b:B} \mathsf{fib}_f(b) \to P(b)\\ &\simeq \prod_{(b:B)} \prod_{(a:A)} \prod_{(p:f(a)=b)} P(b)\\ &\simeq \prod_{a:A} P(f(a)).\end{aligned}$$

We omit the proof that this equivalence is indeed given by $\lambda s.\, s \circ f$. Thus, (i)⇒(ii), and clearly (ii)⇒(iii). To show (iii)⇒(i), consider the type family

$$P(b) :\equiv \left\|\mathsf{fib}_f(b)\right\|_n.$$

Then (iii) yields a function $c : \prod_{(b:B)} \left\|\mathsf{fib}_f(b)\right\|_n$ with $c(f(a)) = \left|(a, \mathsf{refl}_{f(a)})\right|_n$. To show that each $\left\|\mathsf{fib}_f(b)\right\|_n$ is contractible, we will find a function of type

$$\prod_{(b:B)} \prod_{(w:\left\|\mathsf{fib}_f(b)\right\|_n)} w = c(b).$$

By Theorem 7.3.2, for this it suffices to find a function of type

$$\prod_{(b:B)} \prod_{(a:A)} \prod_{(p:f(a)=b)} |(a,p)|_n = c(b).$$

But by rearranging variables and path induction, this is equivalent to the type

$$\prod_{a:A} \left|(a, \mathsf{refl}_{f(a)})\right|_n = c(f(a)).$$

This property holds by our choice of $c(f(a))$. □

Corollary 7.5.8. *For any A, the canonical function $|-|_n : A \to \|A\|_n$ is n-connected.*

Proof. By Theorem 7.3.2 and the associated uniqueness principle, the condition of Lemma 7.5.7 holds. □

For instance, when $n = -1$, Corollary 7.5.8 says that the map $A \to \|A\|$ from a type to its propositional truncation is surjective.

Corollary 7.5.9. *A type A is n-connected if and only if the map*

$$\lambda b.\, \lambda a.\, b : B \to (A \to B)$$

is an equivalence for every n-type B. In other words, "every map from A to an n-type is constant".

Proof. By Lemma 7.5.7 applied to a function with codomain $\mathbf{1}$. □

Lemma 7.5.10. *Let B be an n-type and let $f : A \to B$ be a function. Then the induced function $g : \|A\|_n \to B$ is an equivalence if and only if f is n-connected.*

Proof. Note that f is homotopic to $g \circ |-|_n$. By Corollary 7.5.8, $|-|_n$ is n-connected, so by Lemma 7.5.6 f is n-connected if and only if g is n-connected. But since g is a function between n-types, its fibers are also n-types. Thus, g is n-connected if and only if it is an equivalence. □

We can also characterize connected pointed types in terms of connectivity of the inclusion of their basepoint.

Lemma 7.5.11. *Let A be a type and $a_0 : \mathbf{1} \to A$ a basepoint, with $n \geq -1$. Then A is n-connected if and only if the map a_0 is $(n-1)$-connected.*

Proof. First suppose $a_0 : \mathbf{1} \to A$ is $(n-1)$-connected and let B be an n-type; we will use Corollary 7.5.9. The map $\lambda b.\, \lambda a.\, b : B \to (A \to B)$ has a retraction given by $f \mapsto f(a_0)$, so it suffices to show it also has a section, i.e. that for any $f : A \to B$ there is $b : B$ such that $f = \lambda a.\, b$. We choose $b :\equiv f(a_0)$. Define $P : A \to \mathcal{U}$ by $P(a) :\equiv (f(a) = f(a_0))$. Then P is a family of $(n-1)$-types and we have $P(a_0)$; hence we have $\prod_{(a:A)} P(a)$ since $a_0 : \mathbf{1} \to A$ is $(n-1)$-connected. Thus, $f = \lambda a.\, f(a_0)$ as desired.

Now suppose A is n-connected, and let $P : A \to (n-1)\text{-Type}$ and $u : P(a_0)$ are given. By Lemma 7.5.7, it will suffice to construct $f : \prod_{(a:A)} P(a)$ such that $f(a_0) = u$. Now $(n-1)$-Type is an n-type and A is n-connected, so by Corollary 7.5.9, there is an n-type B such that $P = \lambda a.\, B$. Hence, we have a family of equivalences $g : \prod_{(a:A)} (P(a) \simeq B)$. Define $f(a) :\equiv {g_a}^{-1}(g_{a_0}(u))$; then $f : \prod_{(a:A)} P(a)$ and $f(a_0) = u$ as desired. □

A useful variation on Lemma 7.5.6 is:

Lemma 7.5.12. *Let $f : A \to B$ be a function and $P : A \to \mathcal{U}$ and $Q : B \to \mathcal{U}$ be type families. Suppose that $g : \prod_{(a:A)} P(a) \to Q(f(a))$ is a fiberwise n-connected family of functions, i.e. each $g(a)$ is n-connected. Then the function*

$$\varphi : \Big(\sum_{a:A} P(a)\Big) \to \Big(\sum_{b:B} Q(b)\Big)$$
$$\varphi(a,u) :\equiv (f(a), g(u))$$

is n-connected if and only if f is n-connected.

Proof. For $b : B$ and $v : Q(b)$ we have

$$\begin{aligned}
\big\|\mathsf{fib}_\varphi((b,v))\big\|_n &\simeq \Big\| \sum_{(a:A)} \sum_{(u:P(a))} \sum_{(p:f(a)=b)} f(p)_*(g(u)) = v \Big\|_n \\
&\simeq \Big\| \sum_{(w:\mathsf{fib}_f(b))} \sum_{(u:P(\mathsf{pr}_1(w)))} g(u) = f(p)^{-1}{}_*(v) \Big\|_n \\
&\simeq \Big\| \sum_{w:\mathsf{fib}_f(b)} \Big\| \mathsf{fib}_{g(\mathsf{pr}_1 w)}(f(p)^{-1}{}_*(v)) \Big\|_n \Big\|_n \\
&\simeq \big\|\mathsf{fib}_f(b)\big\|_n
\end{aligned}$$

where the transportations along $f(p)$ and $f(p)^{-1}$ are with respect to Q. Therefore, if either is contractible, so is the other. □

In the other direction, we have

Lemma 7.5.13. *Let $P, Q : A \to \mathcal{U}$ be dependent types and consider a fiberwise transformation*

$$f : \prod_{a:A} P(a) \to Q(a)$$

from P to Q. Then the induced map $\mathsf{total}(f) : \sum_{(a:A)} P(a) \to \sum_{(a:A)} Q(a)$ is n-connected if and only if each $f(a)$ is n-connected.

Proof. By Theorem 4.7.6, we have $\mathsf{fib}_{\mathsf{total}(f)}((x, v)) \simeq \mathsf{fib}_{f(x)}(v)$ for each $x : A$ and $v : Q(x)$. Hence $\left\|\mathsf{fib}_{\mathsf{total}(f)}((x, v))\right\|_n$ is contractible if and only if $\left\|\mathsf{fib}_{f(x)}(v)\right\|_n$ is contractible. □

Another useful fact about connected maps is that they induce an equivalence on n-truncations:

Lemma 7.5.14. *If $f : A \to B$ is n-connected, then it induces an equivalence $\|A\|_n \simeq \|B\|_n$.*

Proof. Let c be the proof that f is n-connected. From left to right, we use the map $\|f\|_n : \|A\|_n \to \|B\|_n$. To define the map from right to left, by the universal property of truncations, it suffices to give a map $\mathsf{back} : B \to \|A\|_n$. We can define this map as follows:

$$\mathsf{back}(y) :\equiv \|\mathsf{pr}_1\|_n(\mathsf{pr}_1(c(y)))$$

By definition, $c(y)$ has type $\mathsf{isContr}(\left\|\mathsf{fib}_f(y)\right\|_n)$, so its first component has type $\left\|\mathsf{fib}_f(y)\right\|_n$, and we can obtain an element of $\|A\|_n$ from this by projection.

Next, we show that the composites are the identity. In both directions, because the goal is a path in an n-truncated type, it suffices to cover the case of the constructor $|-|_n$.

In one direction, we must show that for all $x : A$,

$$\|\mathsf{pr}_1\|_n(\mathsf{pr}_1(c(f(x)))) = |x|_n$$

But $|(x, \mathsf{refl})|_n : \left\|\mathsf{fib}_f(y)\right\|_n$, and $c(y)$ says that this type is contractible, so

$$\mathsf{pr}_1(c(f(x))) = |(x, \mathsf{refl})|_n$$

Applying $\|\mathsf{pr}_1\|_n$ to both sides of this equation gives the result.

In the other direction, we must show that for all $y : B$,

$$\|f\|_n(\|\mathsf{pr}_1\|_n(\mathsf{pr}_1(c(y)))) = |y|_n$$

$\mathsf{pr}_1(c(y))$ has type $\left\|\mathsf{fib}_f(y)\right\|_n$, and the path we want is essentially the second component of the $\mathsf{fib}_f(y)$, but we need to make sure the truncations work out.

In general, suppose we are given $p : \left\|\sum_{(x:A)} B(x)\right\|_n$ and wish to prove $P(\|\mathsf{pr}_1\|_n(p))$. By truncation induction, it suffices to prove $P(|a|_n)$ for all $a : A$ and $b : B(a)$. Applying this principle in this case, it suffices to prove

$$\|f\|_n(|a|_n) = |y|_n$$

given $a : A$ and $b : f(a) = y$. But the left-hand side equals $|f(a)|_n$, so applying $|-|_n$ to both sides of b gives the result. □

One might guess that this fact characterizes the n-connected maps, but in fact being n-connected is a bit stronger than this. For instance, the inclusion $0_{\mathbf{2}} : \mathbf{1} \to \mathbf{2}$ induces an equivalence on (-1)-truncations, but is not surjective (i.e. (-1)-connected). In §8.4 we will see that the difference in general is an analogous extra bit of surjectivity.

7.6 Orthogonal factorization

In set theory, the surjections and the injections form a unique factorization system: every function factors essentially uniquely as a surjection followed by an injection. We have seen that surjections generalize naturally to n-connected maps, so it is natural to inquire whether these also participate in a factorization system. Here is the corresponding generalization of injections.

Definition 7.6.1. A function $f : A \to B$ is **n-truncated** if the fiber $\mathsf{fib}_f(b)$ is an n-type for all $b : B$.

In particular, f is (-2)-truncated if and only if it is an equivalence. And of course, A is an n-type if and only if $A \to \mathbf{1}$ is n-truncated. Moreover, n-truncated maps could equivalently be defined recursively, like n-types.

Lemma 7.6.2. *For any $n \geq -2$, a function $f : A \to B$ is $(n+1)$-truncated if and only if for all $x, y : A$, the map $\mathsf{ap}_f : (x = y) \to (f(x) = f(y))$ is n-truncated. In particular, f is (-1)-truncated if and only if it is an embedding in the sense of §4.6.*

Proof. Note that for any $(x, p), (y, q) : \mathsf{fib}_f(b)$, we have

$$\begin{aligned}\big((x,p) = (y,q)\big) &= \sum_{r:x=y} (p = f(r) \cdot q) \\ &= \sum_{r:x=y} (\mathsf{ap}_f(r) = p \cdot q^{-1}) \\ &= \mathsf{fib}_{\mathsf{ap}_f}(p \cdot q^{-1}).\end{aligned}$$

Thus, any path space in any fiber of f is a fiber of ap_f. On the other hand, choosing $b :\equiv f(y)$ and $q :\equiv \mathsf{refl}_{f(y)}$ we see that any fiber of ap_f is a path space in a fiber of f. The result follows, since f is $(n+1)$-connected if all path spaces of its fibers are n-types. □

We can now construct the factorization, in a fairly obvious way.

Definition 7.6.3. Let $f : A \to B$ be a function. The **n-image** of f is defined as

$$\mathsf{im}_n(f) :\equiv \sum_{b:B} \big\|\mathsf{fib}_f(b)\big\|_n$$

When $n = -1$, we write simply $\mathsf{im}(f)$.

Lemma 7.6.4. *For any function $f : A \to B$, the canonical function $\tilde{f} : A \to \mathsf{im}_n(f)$ is n-connected. Consequently, any function factors as an n-connected function followed by a modal function.*

Proof. Note that $A \simeq \sum_{(b:B)} \mathsf{fib}_f(b)$. The function $\tilde{f}$ is the function on total spaces induced by the fiberwise transformation

$$\prod_{b:B} \mathsf{fib}_f(b) \to \left\|\mathsf{fib}_f(b)\right\|_n.$$

Since this is fiberwise n-connected by Corollary 7.5.8, the statement follows from Lemma 7.5.13. □

In the following lemma we set up some machinery to prove the unique factorization theorem.

Lemma 7.6.5. *Suppose we have a commutative diagram of functions*

$$\begin{array}{ccccc} & \overset{g_1}{\nearrow} & X_1 & \overset{h_1}{\searrow} & \\ A & & & & B \\ & \underset{g_2}{\searrow} & X_2 & \underset{h_2}{\nearrow} & \end{array}$$

with $H : h_1 \circ g_1 \sim h_2 \circ g_2$, where g_1 and g_2 are n-connected and where h_1 and h_2 are n-truncated. Then there is an equivalence

$$E(H, b) : \mathsf{fib}_{h_1}(b) \simeq \mathsf{fib}_{h_2}(b)$$

for any $b : B$, such that for any $a : A$ we have

$$E(H, h_1(g_1(a)))((g_1(a), \mathsf{refl}_{h_1(g_1(a))})) = (g_2(a), H(a)^{-1}).$$

Proof. Let $b : B$. Then we have the following equivalences:

$$\begin{aligned} \mathsf{fib}_{h_1}(b) &\simeq \sum_{w:\mathsf{fib}_{h_1}(b)} \left\|\mathsf{fib}_{g_1}(\mathsf{pr}_1 w)\right\|_n \\ &\simeq \left\| \sum_{w:\mathsf{fib}_{h_1}(b)} \mathsf{fib}_{g_1}(\mathsf{pr}_1 w) \right\|_n \\ &\simeq \left\|\mathsf{fib}_{h_1 \circ g_1}(b)\right\|_n \end{aligned}$$

and likewise for h_2 and g_2. Here, the first equivalence holds because g_1 is n-connected; the second equivalence holds because h_1 is n-truncated; and the third equivalence holds by **??**. Also, since we have a homotopy $H : h_1 \circ g_1 \sim h_2 \circ g_2$, there is an obvious equivalence $\mathsf{fib}_{h_1 \circ g_1}(b) \simeq \mathsf{fib}_{h_2 \circ g_2}(b)$. Hence we obtain

$$\mathsf{fib}_{h_1}(b) \simeq \mathsf{fib}_{h_2}(b)$$

for any $b : B$. By analyzing the underlying functions, we get the following representation of what happens to the term $(g_1(a), \mathsf{refl}_{h_1(g_1(a))})$ after applying each of the equivalences of which E

is composed:

$$\begin{aligned}(g_1(a), \mathsf{refl}_{h_1(g_1(a))}) &\mapsto ((g_1(a), \mathsf{refl}_{h_1(g_1(a))}), \eta((a, \mathsf{refl}_{g_1(a)})))\\ &\mapsto \eta(((g_1(a), \mathsf{refl}_{h_1(g_1(a))}), (a, \mathsf{refl}_{g_1(a)}))\\ &\mapsto \eta((a, \mathsf{refl}_{h_1(g_1(a))}))\\ &\mapsto \eta((a, H(a)^{-1}))\\ &\mapsto \eta(((g_2(a), H(a)^{-1}), (a, \mathsf{refl}_{g_2(a)}))\\ &\mapsto ((g_2(a), H(a)^{-1}), \eta((a, \mathsf{refl}_{g_2(a)})))\\ &\mapsto (g_2(a), H(a)^{-1})\end{aligned}$$

□

The equivalences $E(H, b)$ are such that $E(H^{-1}, b) = E(H, b)^{-1}$.

Combining Lemmas 7.6.4 and 7.6.5, we have the following unique factorization result:

Theorem 7.6.6. *For each $f : A \to B$, the space $\mathsf{fact}_n(f)$ defined by*

$$\sum_{X:\mathcal{U}} (g : A \to X)(h : X \to B),\ (h \circ g \sim f) \times \mathsf{conn}_n(g) \times \mathsf{trunc}_n(h).$$

is contractible. By Lemma 7.6.4 we know that there is the element

$$(\mathsf{im}(f), \tilde{f}, \mathsf{pr}_1, \theta, \varphi, \psi) : \mathsf{fact}_n(f)$$

where $\theta : \mathsf{pr}_1 \circ \tilde{f} \sim f$ is the canonical homotopy, where φ is the proof of Lemma 7.6.4, and where ψ is the obvious proof that $\mathsf{pr}_1 : \mathsf{im}(f) \to B$ has n-truncated fibers.

In the following proof we use the symbols $\cdot_\ell$ and $\cdot_r$ to denote the whisker operations. Recall that if we have paths $p, p' : x = y$, $s : p = p'$ and $q : y = z$, then left whisker operation provides a path $q \cdot p = q \cdot p'$, wich is denoted by $q \cdot_\ell s$. Likewise, if $f, f' : X \to Y$ and $g : Y \to X$ are functions and if $H : f \sim f'$ is a homotopy, then there is a homotopy $g \cdot_\ell H : g \circ f \sim g \circ f'$.

Proof. By Lemma 7.6.4 we know that there is an element of $\mathsf{fact}_n(f)$, hence it is enough to show that $\mathsf{fact}_n(f)$ is a mere proposition. Suppose we have two n-factorizations

$$(X_1, g_1, h_1, H_1, \varphi_1, \psi_1) \qquad \text{and} \qquad (X_2, g_2, h_2, H_2, \varphi_2, \psi_2)$$

of f. Then we have the homotopy $H :\equiv H_2^{-1} \circ H_1 : h_1 \circ g_1 \sim h_2 \circ g_2$. By the univalence axiom, it suffices to show that

(i) there is an equivalence $e : X_1 \simeq X_2$,
(ii) there is a homotopy $\zeta : \underline{e} \circ g_1 \sim g_2$,
(iii) there is a homotopy $\eta : h_2 \circ \underline{e} \sim h_1$,
(iv) there is a homotopy $H_1 \circ (h_1 \cdot_\ell \zeta)^{-1} \circ (\eta \cdot_r g_2) \sim H_2$.

where $\underline{e}$ is the function underlying the equivalence. We prove these four assertions in that order.

(i) By Lemma 7.6.5, we have a fiberwise equivalence

$$\prod_{b:B} \mathsf{fib}_{h_1}(b) \to \mathsf{fib}_{h_2}(b).$$

This induces an equivalence of total spaces, i.e. we have

$$\sum_{b:B} \mathsf{fib}_{h_1}(b) \simeq \sum_{b:B} \mathsf{fib}_{h_2}(b)$$

Of course, we also have the familiar equivalences $X_1 \simeq \sum_{(b:B)} \mathsf{fib}_{h_1}(b)$ and $X_2 \simeq \sum_{(b:B)} \mathsf{fib}_{h_2}(b)$. This gives us our equivalence $e(H) : X_1 \simeq X_2$. The reader may verify that the underlying function $\underline{e}(H)$ of $e(H)$ is defined by

$$\underline{e}(H, x) :\equiv \mathsf{pr}_1 \underline{E}(H^{-1}, h_1(x))((x, \mathsf{id} h_1(x)))$$

(ii) By Lemma 7.6.5 we have $\underline{e}(H, g_1(a)) = g_2(a)$. Thus we get $\zeta(a) :\equiv \mathsf{id} g_2(a)$.

(iii) For every $x : X_1$, we have

$$\mathsf{pr}_2 \underline{E}(H^{-1}, h_1(x))((x, \mathsf{id} h_1(x))) : h_2(\underline{e}(H, x)) = h_1(x),$$

giving us a homotopy $\eta : h_2 \circ \underline{e} \sim h_1$.

(iv) By Lemma 7.6.5 we have $\eta(g_1(a)) = H(a)^{-1}$ and by *ii.* we have $h_2(\zeta(a)) = \mathsf{id} h_2(g_2(a))$. Thus we have

$$\begin{aligned}(H_1 \circ (h_2 \blacktriangleright_\ell \zeta)^{-1} \circ (\eta \blacktriangleright_r g_1))(a) &= H_1(a) \cdot h_2(\zeta(a))^{-1} \cdot \eta(g_1(a)) \\ &= H_1(a) \cdot H(a)^{-1} \\ &= H_2(a).\end{aligned}$$

□

By standard arguments, this yields the following orthogonality principle.

Theorem 7.6.7. *Let $e : A \to B$ be n-connected and $m : C \to D$ be n-truncated. Then the map*

$$\varphi : (B \to C) \to \sum_{(h:A\to C)} \sum_{(k:B\to D)} (m \circ h \sim k \circ e)$$

is an equivalence.

Sketch of proof. For any (h, k, H) in the codomain, let $h = h_2 \circ h_1$ and $k = k_2 \circ k_1$, where h_1 and k_1 are n-connected and h_2 and k_2 are n-truncated. Then $f = (m \circ h_2) \circ h_1$ and $f = k_2 \circ (k_1 \circ e)$ are both n-factorizations of $m \circ h = k \circ e$. Thus, there is a unique equivalence between them. It is straightforward (if a bit tedious) to extract from this that $\mathsf{fib}_\varphi((h, k, H))$ is contractible. □

We end by showing that images are stable under pullback.

Lemma 7.6.8. *Suppose that the square*

$$\begin{array}{ccc} A & \longrightarrow & C \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\ B & \underset{h}{\longrightarrow} & D \end{array}$$

is a pullback square and let $b : B$. Then $\mathsf{fib}_f(b) \simeq \mathsf{fib}_g(h(b))$.

Proof. This follows from pasting of pullbacks, since the type X in the diagram

have pasting of anywhere?

$$\begin{array}{ccccc} X & \longrightarrow & A & \longrightarrow & C \\ \downarrow & & {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\ \mathbf{1} & \underset{b}{\longrightarrow} & B & \underset{h}{\longrightarrow} & D \end{array}$$

is the pullback of the left square if and only if it is the pullback of the outer rectangle: $\mathsf{fib}_f(b)$ is the pullback of the square on the left and $\mathsf{fib}_g(h(b))$ is the pullback of the outer rectangle. □

Theorem 7.6.9. *Consider functions $f : A \to B$, $g : C \to D$ and the diagram*

$$\begin{array}{ccc} A & \longrightarrow & C \\ {\scriptstyle \tilde{f}_n}\downarrow & & \downarrow{\scriptstyle \tilde{g}_n} \\ \mathsf{im}_n(f) & \longrightarrow & \mathsf{im}_n(g) \\ {\scriptstyle \mathsf{pr}_1}\downarrow & & \downarrow{\scriptstyle \mathsf{pr}_1} \\ B & \underset{h}{\longrightarrow} & D \end{array}$$

Then the outer rectangle is a pullback if and only if the bottom square is a pullback. In either of these equivalent cases, the top square is also a pullback. Consequently, images are stable under pullbacks.

Proof. Suppose first that the outer square is a pullback. Note that we have the equivalences

$$\begin{aligned} B \times_D \mathsf{im}_n(g) &\equiv \sum_{(b:B)} \sum_{(w:\mathsf{im}_n(g))} h(b) = \mathsf{pr}_1 w \\ &\simeq \sum_{(b:B)} \sum_{(d:D)} \sum_{(w:\left\|\mathsf{fib}_g(d)\right\|_n)} h(b) = d \\ &\simeq \sum_{b:B} \left\|\mathsf{fib}_g(h(b))\right\|_n. \\ &\simeq \sum_{b:B} \left\|\mathsf{fib}_f(b)\right\|_n \\ &\equiv \mathsf{im}_n(f). \end{aligned}$$

In the last equivalence we have used Lemma 7.6.8.

Now suppose that the bottom square is a pullback, of which we denote the top arrow by ψ. By the pasting lemma for pullbacks, it suffices to show that the top square is a pullback. We have the equivalences

$$\begin{aligned}
\mathsf{im}(f) \times_{\mathsf{im}(g)} C &\equiv \sum_{(w:\mathsf{im}(f))} \sum_{(c:C)} \psi(w) = \tilde{g}_n(c) \\
&\simeq \sum_{(b:B)} \sum_{(w:\mathsf{im}(g))} \sum_{(p:h(b)=\mathsf{pr}_1 w)} \sum_{(c:C)} w = \tilde{g}_n(c) \\
&\simeq \sum_{(b:B)} \sum_{(c:C)} h(b) = g(c) \\
&\simeq \sum_{b:B} \mathsf{fib}_f(b) \\
&\simeq A.
\end{aligned}$$

□

7.7 Modalities

Nearly all of the theory of n-types and connectedness can be done in much greater generality. This section will not be used in the rest of the book.

Our first thought regarding generalizing the theory of n-types might be to take Lemma 7.3.3 as a definition.

Definition 7.7.1. A **reflective subuniverse** is a predicate $P : \mathcal{U} \to \mathsf{Prop}$ such that for every $A : \mathcal{U}$ we have a type $\bigcirc A$ such that $P(\bigcirc A)$ and a map $\eta_A : A \to \bigcirc A$, with the property that for every $B : \mathcal{U}$ with $P(B)$, the following map is an equivalence:

$$\begin{cases} (\bigcirc A \to B) & \longrightarrow & (A \to B) \\ f & \longmapsto & f \circ \eta_A \end{cases}.$$

We write $\mathcal{U}_P :\equiv \{\, A : \mathcal{U} \mid P(A) \,\}$, so $A : \mathcal{U}_P$ means that $A : \mathcal{U}$ and we have $P(A)$. We also write $\mathsf{rec}_\bigcirc$ for the quasi-inverse of the above map. The notation $\bigcirc$ may seem slightly odd, but it will make more sense soon.

For any reflective subuniverse, we can prove all the familiar facts about reflective subcategories from category theory, in the usual way. For instance, we have:

- A type A lies in $\mathcal{U}_P$ if and only if $\eta_A : A \to \bigcirc A$ is an equivalence.
- $\mathcal{U}_P$ is closed under retracts. In particular, A lies in $\mathcal{U}_P$ as soon as η_A admits a retraction.
- The operation $\bigcirc$ is a functor in a suitable up-to-coherent-homotopy sense, which we can make precise at as high levels as necessary.
- The types in $\mathcal{U}_P$ are closed under all limits such as products and pullbacks. In particular, for any $A : \mathcal{U}_P$ and $x, y : A$, the identity type $(x =_A y)$ is also in $\mathcal{U}_P$, since it is a pullback of two functions $\mathbf{1} \to A$.
- Colimits in $\mathcal{U}_P$ can be constructed by applying $\bigcirc$ to ordinary colimits of types.

Importantly, closure under products extends also to "infinite products", i.e. dependent function types.

Theorem 7.7.2. *If $B : A \to \mathcal{U}_P$ is any family of types in a reflective subuniverse $\mathcal{U}_P$, then $\prod_{(x:A)} B(x)$ is also in $\mathcal{U}_P$.*

Proof. For any $x : A$, consider the function $\mathsf{ev}_x : (\prod_{(x:A)} B(x)) \to B(x)$ defined by $\mathsf{ev}_x(f) :\equiv f(x)$. Since $B(x)$ lies in P, this extends to a function

$$\mathsf{rec}_{\bigcirc}(\mathsf{ev}_x) : \bigcirc\Big(\prod_{x:A} B(x)\Big) \to B(x).$$

Thus we can define $h : \bigcirc(\prod_{(x:A)} B(x)) \to \prod_{(x:A)} B(x)$ by $h(z)(x) :\equiv \mathsf{rec}_{\bigcirc}(\mathsf{ev}_x)(z)$. Then h is a retraction of $\eta_{\prod_{(x:A)} B(x)}$, so that $\prod_{(x:A)} B(x)$ is in $\mathcal{U}_P$. □

In particular, if $B : \mathcal{U}_P$ and A is any type, then $(A \to B)$ is in $\mathcal{U}_P$. In categorical language, this means that any reflective subuniverse is an **exponential ideal**. This, in turn, implies by a standard argument that the reflector preserves finite products.

Corollary 7.7.3. *For any types A and B and any reflective subuniverse, the induced map $\bigcirc(A \times B) \to \bigcirc(A) \times \bigcirc(B)$ is an equivalence.*

Proof. It suffices to show that $\bigcirc(A) \times \bigcirc(B)$ has the same universal property as $\bigcirc(A \times B)$. Thus, let $C : \mathcal{U}_P$; we have

$$\begin{aligned}(\bigcirc(A) \times \bigcirc(B) \to C) &= (\bigcirc(A) \to (\bigcirc(B) \to C)) \\ &= (\bigcirc(A) \to (B \to C)) \\ &= (A \to (B \to C)) \\ &= (A \times B \to C)\end{aligned}$$

using the universal properties of $\bigcirc(B)$ and $\bigcirc(A)$, along with the fact that $B \to C$ is in $\mathcal{U}_P$ since C is. It is straightforward to verify that this equivalence is given by composing with $\eta_A \times \eta_B$, as needed. □

It may seem odd that every reflective subcategory of types is automatically an exponential ideal, with a product-preserving reflector. However, this is also the case classically in the category of *sets*, for the same reasons. It's just that this fact is not usually remarked on, since the classical category of sets—in contrast to the category of homotopy types—does not have many interesting reflective subcategories.

Two basic properties of n-types are *not* shared by general reflective subuniverses: Theorem 7.1.8 (closure under Σ-types) and Theorem 7.3.2 (truncation induction). However, the analogues of these two properties are equivalent to each other.

Theorem 7.7.4. *For a reflective subuniverse $\mathcal{U}_P$, the following are logically equivalent.*

(i) If $A : \mathcal{U}_P$ and $B : A \to \mathcal{U}_P$, then $\sum_{(x:A)} B(x)$ is in $\mathcal{U}_P$.

(ii) for every $A : \mathcal{U}$, type family $B : \bigcirc A \to \mathcal{U}_P$, and function $g : \prod_{(a:A)} B(\eta(a))$, there exists $f : \prod_{(z:\bigcirc A)} B(z)$ such that $f(\eta(a)) = g(a)$ for all $a : A$.

Proof. Suppose (i). Then in the situation of (ii), the type $\sum_{(z:\bigcirc A)} B(z)$ lies in $\mathcal{U}_P$, and we have $g' : A \to \sum_{(z:\bigcirc A)} B(z)$ defined by $g'(a) :\equiv (\eta(a), g(a))$. Thus, we have $\mathsf{rec}_\bigcirc(g') : \bigcirc A \to \sum_{(z:\bigcirc A)} B(z)$ such that $\mathsf{rec}_\bigcirc(g')(\eta(a)) = (\eta(a), g(a))$.

Now consider the functions $\mathsf{pr}_2 \circ \mathsf{rec}_\bigcirc(g') : \bigcirc A \to \bigcirc A$ and $\mathsf{id}_{\bigcirc A}$. By assumption, these become equal when precomposed with η. Thus, by the universal property of $\bigcirc$, they are equal already, i.e. we have $p_z : \mathsf{pr}_2(\mathsf{rec}_\bigcirc(g')(z)) = z$ for all z. Now we can define $f(z) :\equiv p_{z*}(\mathsf{pr}_2(\mathsf{rec}_\bigcirc(g')(z)))$, and the second component of $\mathsf{rec}_\bigcirc(g')(\eta(a)) = (\eta(a), g(a))$ yields $f(\eta(a)) = g(a)$.

Conversely, suppose (ii), and that $A : \mathcal{U}_P$ and $B : A \to \mathcal{U}_P$. Let h be the composite

$$\bigcirc\Big(\sum_{x:A} B(x)\Big) \xrightarrow{\bigcirc(\mathsf{pr}_1)} \bigcirc A \xrightarrow{(\eta_A)^{-1}} A.$$

Then for $z : \sum_{(x:A)} B(x)$ we have

$$\begin{aligned} h(\eta(z)) &= \eta^{-1}(\bigcirc(\mathsf{pr}_1)(\eta(z))) \\ &= \eta^{-1}(\eta(\mathsf{pr}_1(z))) \\ &= \mathsf{pr}_1(z). \end{aligned}$$

Denote this path by p_z. Now if we define $C : \bigcirc(\sum_{(x:A)} B(x)) \to \mathcal{U}$ by $C(w) :\equiv B(h(w))$, we have

$$g :\equiv \lambda z.\, p_{z*}(\mathsf{pr}_2(z)) \;:\; \prod_{z:\sum_{(x:A)} B(x)} C(\eta(z)).$$

Thus, the assumption yields $f : \prod_{(w:\bigcirc(\sum_{(x:A)} B(x)))} C(w)$ such that $f(\eta(z)) = g(z)$. Together, h and f give a function $k : \bigcirc(\sum_{(x:A)} B(x)) \to \sum_{(x:A)} B(x)$ defined by $k(w) :\equiv (h(w), f(w))$, while p_z and the equality $f(\eta(z)) = g(z)$ show that k is a retraction of $\eta_{\sum_{(x:A)} B(x)}$. Therefore, $\sum_{(x:A)} B(x)$ is in $\mathcal{U}_P$. □

Note the similarity to the discussion in §5.5. The universal property of the reflector of a reflective subuniverse is like a recursion principle with its uniqueness property, while Theorem 7.7.4(ii) is like the corresponding induction principle. Unlike in §5.5, the two are not equivalent here, because of the restriction that we can only eliminate into types that lie in $\mathcal{U}_P$. Condition (i) of Theorem 7.7.4 is what fixes the disconnect.

Unsurprisingly, of course, if we have the induction principle, then we can derive the recursion principle. We can also derive its uniqueness property, as long as we allow ourselves to eliminate into path types. This suggests the following definition. Note that any reflective subuniverse can be characterized by the operation $\bigcirc : \mathcal{U} \to \mathcal{U}$ and the functions $\eta_A : A \to \bigcirc A$, since we have $P(A) = \mathsf{isequiv}(\eta_A)$.

Definition 7.7.5. A **modality** is an operation $\bigcirc : \mathcal{U} \to \mathcal{U}$ for which there are

(i) functions $\eta^\bigcirc_A : A \to \bigcirc(A)$ for every type A.

(ii) for every $A : \mathcal{U}$ and every type family $B : \bigcirc(A) \to \mathcal{U}$, a function

$$\mathsf{ind}_\bigcirc : \Big(\prod_{a:A} \bigcirc(B(\eta^\bigcirc_A(a)))\Big) \to \prod_{z:\bigcirc(A)} \bigcirc(B(z)).$$

(iii) A path $\mathsf{ind}_\bigcirc(f)(\eta^\bigcirc_A(a)) = f(a)$ for each $f : \prod_{(a:A)} \bigcirc(B(\eta^\bigcirc_A(a)))$.

(iv) For any $z, z' : \bigcirc(A)$, the function $\eta^\bigcirc_{z=z'} : (z = z') \to \bigcirc(z = z')$ is an equivalence.

We say that A is **modal** for $\bigcirc$ if $\eta^\bigcirc_A : A \to \bigcirc(A)$ is an equivalence, and we write

$$\mathcal{U}_\bigcirc :\equiv \{\, X : \mathcal{U} \mid X \text{ is } \bigcirc\text{-modal} \,\}$$

for the type of modal types.

Conditions (ii) and (iii) are very similar to Theorem 7.7.4(ii), but phrased using $\bigcirc B(z)$ rather than assuming B to be valued in $\mathcal{U}_P$. This allows us to state the condition purely in terms of the operation $\bigcirc$, rather than requiring the predicate $P : \mathcal{U} \to \mathsf{Prop}$ to be given in advance. (It is not entirely satisfactory, since we still have to refer to P not-so-subtly in clause (iv). We do not know whether (iv) follows from (i)–(iii).) However, the stronger-looking property of Theorem 7.7.4(ii) follows from Definition 7.7.5(ii) and (iii), since for any $C : \bigcirc A \to \mathcal{U}_\bigcirc$ we have $C(z) \simeq \bigcirc C(z)$, and we can pass back across this equivalence.

As with other induction principles, this implies a universal property.

Theorem 7.7.6. *Let A be a type and let $B : \bigcirc(A) \to \mathcal{U}_\bigcirc$. Then the function*

$$(- \circ \eta^\bigcirc_A) : \Big(\prod_{z:\bigcirc(A)} B(z)\Big) \to \Big(\prod_{a:A} B(\eta^\bigcirc_A(a))\Big)$$

is an equivalence.

Proof. By definition, the operation $\mathsf{ind}_\bigcirc$ is a right inverse to $(- \circ \eta^\bigcirc_A)$. Thus, we only need to find a homotopy

$$\prod_{z:\bigcirc(A)} s(z) = \mathsf{ind}_\bigcirc(s \circ \eta^\bigcirc_A)(z)$$

for each $s : \prod_{(z:\bigcirc(A))} B(z)$, exhibiting it as a left inverse as well. By assumption, each $B(z)$ is modal, and hence each type $s(z) = R^\bigcirc_X(s \circ \eta^\bigcirc_A)(z)$ is also modal. Thus, it suffices to find a function of type

$$\prod_{a:A} s(\eta^\bigcirc_A(a)) = \mathsf{ind}_\bigcirc(s \circ \eta^\bigcirc_A)(\eta^\bigcirc_A(a)).$$

which follows from Definition 7.7.5(iii). □

In particular, for every type A and every modal type B, we have an equivalence $(\bigcirc A \to B) \simeq (A \to B)$.

Corollary 7.7.7. *For any modality $\bigcirc$, the $\bigcirc$-modal types form a reflective subuniverse satisfying the equivalent conditions of Theorem 7.7.4.*

Thus, modalities can be identified with reflective subuniverses closed under Σ-types. The name *modality* comes, of course, from *modal logic*, which studies logic where we can form statements such as "possibly A" (usually written $\diamond A$) or "necessarily A" (usually written $\Box A$). The symbol $\bigcirc$ is somewhat common for an arbitrary modal operator. Under the propositions-as-types principle, a modality in the sense of modal logic corresponds to an operation on *types*, and Definition 7.7.5 seems a reasonable candidate for how such an operation should be defined. (More precisely, we should perhaps call these *idempotent, monadic* modalities; see the Notes.) As mentioned in §3.10, we may in general use adverbs to speak informally about such modalities, such as "merely" for the propositional truncation and "purely" for the identity modality (i.e. the one defined by $\bigcirc A :\equiv A$).

For any modality $\bigcirc$, we define a map $f : A \to B$ to be **$\bigcirc$-connected** if $\bigcirc(\mathsf{fib}_f(b))$ is contractible for all $b : B$, and to be **$\bigcirc$-truncated** if $\mathsf{fib}_f(b)$ is modal for all $b : B$. All of the theory of §§7.5 and 7.6 which doesn't involve relating n-types for different values of n applies verbatim in this generality. In particular, we have an orthogonal factorization system.

An important class of modalities which does *not* include the n-truncations is the *left exact* modalities: those for which the functor $\bigcirc$ preserves pullbacks as well as finite products. These are a categorification of "Lawvere-Tierney topologies" in elementary topos theory, and correspond in higher-categorical semantics to sub-∞-toposes. However, this is beyond the scope of this book.

Some particular examples of modalities other than n-truncation can be found in the exercises.

Notes

The notion of homotopy n-type in classical homotopy theory is quite old. It was Voevodsky who realized that the notion can be defined recursively in homotopy type theory, starting from contractibility.

The property "Axiom K" was so named by Thomas Streicher, as a property of identity types which comes after J, the latter being the traditional name for the eliminator of identity types. Theorem 7.2.6 is due to Hedberg [Hed98]; for more information and generalizations see [KECA13].

The notions of n-connected spaces and functions are also classical in homotopy theory, although as mentioned before, our indexing for connectedness of functions is off by one from the classical indexing. The importance of the resulting factorization system has been emphasized by recent work in higher topos theory by Rezk, Lurie, and others. In particular, the results of this chapter should be compared with [Lur09, §6.5.1]. In §8.6, the theory of n-connected maps will be crucial to our proof of the Freudenthal suspension theorem.

Modal operators in *simple* type theory have been studied extensively; see e.g. [dPGM04]. In the setting of dependent type theory, [AB04] treats the special case of propositional truncation ((-1)-truncation) as a modal operator. The development presented here greatly extends and generalizes this work, while drawing also on ideas from topos theory.

Generally, modal operators come in (at least) two flavors: those such as $\diamond$ ("possibly") for which $A \Rightarrow \diamond A$, and those such as $\Box$ ("necessarily") for which $\Box A \Rightarrow A$. When they are also *idempotent* (i.e. $\diamond A = \diamond\diamond A$ or $\Box A = \Box\Box A$), the former may be identified with reflective subcategories (or equivalently, idempotent monads), and the latter with coreflective subcategories

(or idempotent comonads). However, in dependent type theory it is trickier to deal with the comonadic sort, since they are more rarely stable under pullback, and thus cannot be interpreted as operations on the universe $\mathcal{U}$. Sometimes there are ways around this (see e.g. [SS12]), but for simplicity, here we stick to the monadic sort.

On the computational side, monads (and hence modalities) are used to model effects in functional programming [Mog89]. A computation is said to be *pure* if its execution results in no side effects (such as printing a message to the screen, playing music, or sending data over the Internet). There exist "purely functional" programming languages, such as Haskell, in which it is technically only possible to write pure functions: side effects are represented by applying "monads" to output types. For instance, a function of type $\mathsf{Int} \to \mathsf{Int}$ is pure, while a function of type $\mathsf{Int} \to \mathsf{IO}(\mathsf{Int})$ may perform input and output along the way to computing its result; the operation IO is a monad. (This is the origin of our use of the adverb "purely" for the identity monad, since it corresponds computationally to pure functions with no side-effects.) The modalities we have considered in this chapter are all idempotent, whereas those used in functional programming rarely are, but the ideas are still closely related.

Exercises

Exercise 7.1.

(i) Use Theorem 7.2.2 to show that if $\|A\| \to A$ for every type A, then every type is a set.

(ii) Show that if every surjective function (purely) splits, i.e. if $\prod_{(b:B)} \left\|\mathsf{fib}_f(b)\right\| \to \prod_{(b:B)} \mathsf{fib}_f(b)$ for every $f : A \to B$, then every type is a set.

Exercise 7.2. Use Lemma 7.5.13 to extend Lemma 7.5.11 to any section/retraction pair.

Exercise 7.3. Show that Corollary 7.5.9 also works as a characterization in the other direction: B is an n-type if and only if every map into B from an n-connected type is constant. Ideally, your proof should work for any modality as in §7.7.

Exercise 7.4. Prove that for $n \geq 0$, a type A is n-connected if and only if it is (-1)-connnected (i.e. merely inhabited) and for all $a, b : A$ the type $a =_A b$ is $(n-1)$-connected.

Exercise 7.5. For $-1 \leq n, m \leq \infty$, let $\mathsf{LEM}_{n,m}$ denote the statement

$$\prod_{A:n\text{-}\mathsf{Type}} \|A + \neg A\|_m,$$

where ∞-$\mathsf{Type} :\equiv \mathcal{U}$ and $\|X\|_\infty :\equiv X$. Show that:

(i) If $n = -1$ or $m = -1$, then $\mathsf{LEM}_{n,m}$ is equivalent to LEM from §3.4.

(ii) If $n \geq 0$ and $m \geq 0$, then $\mathsf{LEM}_{n,m}$ is inconsistent with univalence.

Exercise 7.6. For $-1 \leq n, m \leq \infty$, let $\mathsf{AC}_{n,m}$ denote the statement

$$\prod_{(X:\mathsf{Set})} \prod_{(Y:X\to n\text{-}\mathsf{Type})} \left(\prod_{x:X} \|Y(x)\|_m\right) \to \left\|\prod_{x:X} Y(x)\right\|_m,$$

with conventions as in Exercise 7.5. Thus $\mathsf{AC}_{0,-1}$ is the axiom of choice from §3.8, while $\mathsf{AC}_{\infty,\infty}$ is Theorem 2.15.7. It is known that $\mathsf{AC}_{\infty,-1}$ is consistent with univalence, since it holds in Voevodsky's simplicial model.

(i) Without using univalence, show that $\mathsf{LEM}_{n,\infty}$ implies $\mathsf{AC}_{n,m}$ for all m. (On the other hand, in §10.1.5 we will show that $\mathsf{AC} = \mathsf{AC}_{0,-1}$ implies $\mathsf{LEM} = \mathsf{LEM}_{-1,-1}$.)

(ii) Of course, $\mathsf{AC}_{n,m} \Rightarrow \mathsf{AC}_{k,m}$ if $k \leq n$. Are there any other implications between the principles $\mathsf{AC}_{n,m}$? Is $\mathsf{AC}_{n,m}$ consistent with univalence for any $m \geq -1$? (These are open questions.)

Exercise 7.7. Define the **n-connected axiom of choice** to be the statement

> If X is a set and $Y : X \to \mathcal{U}$ is a family of types such that each $Y(x)$ is n-connected, then $\prod_{(x:X)} Y(x)$ is n-connected.

Note that the (-1)-connected axiom of choice is $\mathsf{AC}_{\infty,-1}$ from Exercise 7.6.

(i) Prove that for every $n \geq -1$, the n-connected axiom of choice implies the $(n+1)$-connected axiom of choice.

(ii) Are there any other implications between the n-connected axioms of choice and the principles $\mathsf{AC}_{n,m}$? (This is an open question.)

Exercise 7.8. Show that the n-truncation modality is not left exact for any n. That is, exhibit a pullback which it fails to preserve.

Exercise 7.9. Show that $X \mapsto (\neg\neg X)$ is a modality.

Exercise 7.10. Let P be a mere proposition.

(i) Show that $X \mapsto (P \to X)$ is a left exact modality. This is called the **open modality** associated to P.

(ii) Show that $X \mapsto P * X$ is a left exact modality, where $*$ denotes the join (see §6.8). This is called the **closed modality** associated to P.

Exercise 7.11. Let $f : A \to B$ be a map; a type Z is **f-local** if $(- \circ f) : (B \to Z) \to (A \to Z)$ is an equivalence.

(i) Prove that the f-local types form a reflective subuniverse. You will want to use a higher inductive type to define the reflector (localization).

(ii) Prove that if $B = \mathbf{1}$, then this subuniverse is a modality.

Part II

Mathematics

Chapter 8

Homotopy theory

In this chapter, we develop some homotopy theory within type theory. We use the *synthetic approach* to homotopy theory introduced in Chapter 2: Spaces, points, paths, and homotopies are basic notions, which are represented by types and elements of types, particularly the identity type. The algebraic structure of paths and homotopies is represented by the natural ∞-groupoid structure on types, which is generated by the rules for the identity type. Using higher inductive types, as introduced in Chapter 6, we can describe spaces directly by their universal properties.

There are several interesting aspects of this synthetic approach. First, it combines advantages of working concretely with topological spaces with advantages of working abstractly with categorical frameworks for homotopy theory, such as Quillen model categories. On the one hand, the proofs feel elementary, and refer concretely to a space's points and paths and homotopies. At the same time, our approach abstracts from the concrete presentation of these objects in topological spaces (for example, associativity of path concatention is proved by path induction, rather than by reparametrization of maps $[0,1] \to X$), and our proofs apply in a variety of settings: for example, homotopy type theory also has an interpretation in Kan simplicial sets, which are one model of ∞-groupoids. Thus, our proofs apply to this model, and transferring them along the geometric realization functor from simplicial sets to topological spaces gives proofs of correponding theorems in classical homotopy theory. Moreover, though this is work in progress, it seems likely that we will be able to interpret type theory in a wide variety of other categories that look like the category of ∞-groupoids, such as $(\infty, 1)$-toposes. If so, proving a result in type theory will show that it holds in these settings as well. Additionally, our work so far suggests that type theory is a very convenient way to study the homotopy theory of ∞-groupoids: by using the rules for the identity type, we can avoid the complicated combinatorics involved in many definitions of ∞-groupoids, and explicate only as much of the structure as is needed in any particular proof.

Second, our synthetic approach has suggested some new type-theoretic methods and proofs. Some of the proofs that have been done are fairly direct transcriptions of classical proofs. Others have a more type-theoretic feel, and consist mainly of calculations with ∞-groupoid operations, in a style that is very similar to how computer-scientists use type theory to reason about computer programs. One thing that seems to have permitted these new proofs is the fact that type theory emphasizes different aspects of homotopy theory than other approaches: while tools like

path induction and the universal properties of higher inductives are available in a setting like Kan simplicial sets, type theory elevates their importance, because they are the *only* primitive tools available for working with these types. Focusing on these tools had led to new descriptions of familiar constructions such as the universal cover of the circle and the Hopf fibration, using just the recursion principles for higher inductive types. These descriptions are very direct, and many of the proofs in this chapter involve computational calculations with such fibrations. Another new aspect of our proofs is that they are constructive (assuming univalence and higher inductives types are constructive); we describe an application of this to homotopy groups of spheres in §8.10.

Third, our synthetic approach is very amenable to computer-checked proofs in proof assistants such as COQ and AGDA. Almost all of the proofs described in this chapter have been computer-checked, and many of these proofs were first given in a proof assistant, and then "unformalized" for this book. The computer-checked proofs are comparable in length and effort to the informal proofs presented here, and in some cases they are even shorter and easier to do.

Before turning to the presentation of our results, we briefly review some basic concepts and theorems from homotopy theory for the benefit of the reader who is not familiar with them. We also give an overview of the results proved in this chapter.

Homotopy theory is a branch of algebraic topology, and uses tools from abstract algebra, such as group theory, to investigate properties of spaces. One question homotopy theorists investigate is how to tell whether two spaces are the same, where "the same" means *homotopy equivalence* (continuous maps back and forth that compose to the identity up to homotopy—this gives the opportunity to "correct" maps that don't exactly compose to the identity.). One common way to tell whether two spaces are the same is to calculate *algebraic invariants* associated with a space, which include its *homotopy groups* and *homology* and *cohomology groups*. Equivalent spaces have isomorphic homotopy/(co)homology groups, so if two spaces have different groups, then they are not equivalent. Thus, these algebraic invariants provide global information about a space, which can be used to tell spaces apart, and complements the local information provided by notions such as continuity. For example, the torus locally looks like the 2-sphere, but it has a global difference, because it as a hole in it, and this difference is visible in the homotopy groups of these two spaces.

The simplest example of a homotopy group is the *fundamental group* of a space, which is written $\pi_1(X, x_0)$: Given a space X and a point x_0 in it, one can make a group whose elements are loops at x_0 (continuous paths from x_0 to x_0), considered up to homotopy, with the group operations given by the identity path (standing still), path concatenation, and path reversal. For example, the fundamental group of the 2-sphere is trivial, but the fundamental group of the torus is not, which shows that the sphere and the torus are not homotopy equivalent. The intuition is that every loop on the sphere is homotopic to the identity, because its inside can be filled in. In contrast, a loop on the torus that goes through the donut's hole is not homotopic to the identity, so there are non-trivial elements in the fundamental group.

The *higher homotopy groups* provide additional information about a space. Fix a point x_0 in X, and consider the constant path refl_{x_0}. Then the homotopy classes of homotopies between refl_{x_0} and itself form a group $\pi_2(X, x_0)$, which tells you something about the two-dimensional structure of the space. Then $\pi_3(X, x_0)$ is the group of homotopy classes of homotopies between

homotopies, and so on. One of the basic problems of algebraic topology is *calculating the homotopy groups of a space* X, which means giving a group isomorphism between $\pi_k(X, x_0)$ and some more direct description of a group (e.g., by a multiplication table or presentation). Somewhat surprisingly, this is a very difficult question, even for spaces as simple as the spheres. As can be seen from Table 8.1, some patterns emerge in the higher homotopy groups of spheres, but there is no general formula, and many homotopy groups of spheres are currently still unknown.

	$\mathbb{S}^0$	$\mathbb{S}^1$	$\mathbb{S}^2$	$\mathbb{S}^3$	$\mathbb{S}^4$	$\mathbb{S}^5$	$\mathbb{S}^6$	$\mathbb{S}^7$	$\mathbb{S}^8$
π_1	0	$\mathbb{Z}$	0	0	0	0	0	0	0
π_2	0	0	$\mathbb{Z}$	0	0	0	0	0	0
π_3	0	0	$\mathbb{Z}$	$\mathbb{Z}$	0	0	0	0	0
π_4	0	0	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}$	0	0	0	0
π_5	0	0	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}$	0	0	0
π_6	0	0	$\mathbb{Z}_{12}$	$\mathbb{Z}_{12}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}$	0	0
π_7	0	0	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}\times\mathbb{Z}_{12}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}$	0
π_8	0	0	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_2^2$	$\mathbb{Z}_{24}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}$
π_9	0	0	$\mathbb{Z}_3$	$\mathbb{Z}_3$	$\mathbb{Z}_2^2$	$\mathbb{Z}_2$	$\mathbb{Z}_{24}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$
π_{10}	0	0	$\mathbb{Z}_{15}$	$\mathbb{Z}_{15}$	$\mathbb{Z}_{24}\times\mathbb{Z}_3$	$\mathbb{Z}_2$	0	$\mathbb{Z}_{24}$	$\mathbb{Z}_2$
π_{11}	0	0	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_{15}$	$\mathbb{Z}_2$	$\mathbb{Z}$	0	$\mathbb{Z}_{24}$
π_{12}	0	0	$\mathbb{Z}_2^2$	$\mathbb{Z}_2^2$	$\mathbb{Z}_2$	$\mathbb{Z}_{30}$	$\mathbb{Z}_2$	0	0
π_{13}	0	0	$\mathbb{Z}_{12}\times\mathbb{Z}_2$	$\mathbb{Z}_{12}\times\mathbb{Z}_2$	$\mathbb{Z}_2^3$	$\mathbb{Z}_2$	$\mathbb{Z}_{60}$	$\mathbb{Z}_2$	0

Table 8.1: Homotopy groups of spheres [Wik13]. The k^{th} homotopy group π_k of the n-dimensional sphere $\mathbb{S}^n$ is isomorphic to the group listed in each entry, where $\mathbb{Z}$ is the additive group of integers, and $\mathbb{Z}_m$ is the cyclic group of order m.

One way of understanding this complexity is through the correspondence between spaces and ∞-groupoids introduced in Chapter 2. As discussed in §6.4, the 2-sphere is presented by a higher inductive type with one point and one 2-dimensional loop. Thus, one might wonder why $\pi_3(\mathbb{S}^2)$ is $\mathbb{Z}$, when the type $\mathbb{S}^2$ has no generators creating 3-dimensional cells. It turns out that the generating element of $\pi_3(\mathbb{S}^2)$ is constructed using the interchange law described in the proof of Theorem 2.1.6: the algebraic structure of an ∞-groupoid includes non-trivial interactions between levels, and these interactions create elements of higher homotopy groups.

Type theory provides a natural setting for investigating this structure. We can easily define the higher homotopy groups as groups on the set of n-dimensional loops in a space. Recall Definition 2.1.8 of the iterated loop space from Section 2.1: For $n : \mathbb{N}$, the n-fold iterated loop space of a pointed type (A, a) is defined recursively by:

$$\Omega^0(A, a) = (A, a)$$
$$\Omega^{n+1}(A, a) = \Omega^n(\Omega(A, a)).$$

This gives a *space* of n-dimensional loops, which itself has higher homotopies. We obtain the set of n-dimensional loops by truncation (this was also defined as an example in §6.11):

Definition 8.0.1 (Homotopy Groups)**.** Given $n \geq 1$ and (A, a) a pointed type, we define the **homotopy groups** of A at a by

$$\pi_n(A, a) :\equiv \left\| \Omega^n(A, a) \right\|_0$$

Since $n \geq 1$, the path concatenation and inversion operations on $\Omega^n(A)$ induce operations on $\pi_n(A)$ making it into a group in a straightforward way. If $n \geq 2$, then the group $\pi_n(A)$ is abelian, by the Eckmann–Hilton argument (Theorem 2.1.6). It is sometimes convenient to also write $\pi_0(A) :\equiv \|A\|_0$, but this case behaves somewhat differently: not only is it not a group, it is defined without reference to any basepoint in A.

This definition is a suitable one for investigating homotopy groups because the (higher) inductive definition of a type X presents X as a free type, analogous to a free ∞-groupoid, and this presentation *determines* but does not *explicitly describe* the higher identity types of X. The identity types are populated by both the generators (loop, for the circle), and all of the groupoid operations (identity, composition, inverses, associativity, interchange, ...). Thus, the higher-inductive presentation of a space allows you to pose the question "what does the identity type of X really turn out to be?", though it can take some significant mathematics to answer it. This is a higher-dimensional generalization of a familiar fact in type theory: characterizing the identity type of X can take some work, even if X is an ordinary inductive type, such as the natural numbers or booleans. For example, the theorem that true is different from false does not follow immediately from the definition; see §2.12.

The univalence axiom plays an essential role in calculating homotopy groups (without univalence, type theory is compatible with an interpretation where all paths, including, for example, the loop on the circle, are reflexivity). You can see this in the calculation of the the fundamental group of the circle below: the map from $\mathsf{base} =_{S^1} \mathsf{base}$ to $\mathbb{Z}$ is defined by mapping a loop on the circle to an automorphism of the set $\mathbb{Z}$, so that, for example, $\mathsf{loop} \cdot \mathsf{loop}^{-1}$ is sent to $\mathsf{successor} \cdot \mathsf{predecessor}$ (where $\mathsf{successor}$ and $\mathsf{predecessor}$ are automorphisms of $\mathbb{Z}$ viewed, by univalence, as paths in the universe), and then applying the automorphism to 0. Univalence produces non-trivial paths in the universe, and this is used to extract information from paths in higher inductive types.

In this chapter, we first calculate some homotopy groups of spheres, including $\pi_1(S^1)$ (§8.1), $\pi_{k<n}(S^n)$ (§8.3), $\pi_2(S^2)$ and $\pi_3(S^2)$ by way of the Hopf fibration (§8.5) and a long-exact-sequence argument (§8.4), and $\pi_n(S^n)$ by way of the Freudenthal suspension theorem (§8.6). Next, we discuss the van Kampen theorem (§8.7), which characterizes the fundamental group of a pushout, and the status of Whitehead's principle (when is a map that induces an equivalence on all homotopy groups an equivalence?) (§8.8). Finally, we include brief summaries of additional results that are not included in the book, such as $\pi_{n+1}(S^n)$ for $n \geq 3$, the Blakers-Massey theorem, and a construction of Eilenberg-Mac Lane spaces (§8.10).

8.1 $\pi_1(S^1)$

In this section, our goal is to show that $\pi_1(\mathbb{S}^1) = \mathbb{Z}$. In fact, we will show that the loop space $\Omega(\mathbb{S}^1)$ is equivalent to $\mathbb{Z}$. This is a stronger statement, because $\pi_1(\mathbb{S}^1) = \left\|\Omega(\mathbb{S}^1)\right\|_0$ by definition; so if $\Omega(\mathbb{S}^1) = \mathbb{Z}$, then $\left\|\Omega(\mathbb{S}^1)\right\|_0 = \|\mathbb{Z}\|_0$ by congruence, and $\mathbb{Z}$ is a set, so $\|\mathbb{Z}\|_0 = \mathbb{Z}$. That $\mathbb{Z}$ is a set follows from Remark 6.10.7; see also Remark 6.10.11. Moreover, knowing that $\Omega(\mathbb{S}^1)$ is a set will imply that $\pi_n(\mathbb{S}^1)$ is trivial for $n > 1$, so we will actually have calculated *all* the homotopy groups of $\mathbb{S}^1$.

8.1.1 Getting started

It is not too hard to define functions in both directions between $\Omega(\mathbb{S}^1)$ and $\mathbb{Z}$. By specializing Corollary 6.10.13 to $\mathsf{loop} : \mathsf{base} = \mathsf{base}$, we have a function $\mathsf{loop}^- : \mathbb{Z} \to (\mathsf{base} = \mathsf{base})$ defined (loosely speaking) by

$$\mathsf{loop}^n = \begin{cases} \underbrace{\mathsf{loop} \cdot \mathsf{loop} \cdot \cdots \cdot \mathsf{loop}}_{n} & \text{if } n > 0, \\ \underbrace{\mathsf{loop}^{-1} \cdot \mathsf{loop}^{-1} \cdot \cdots \cdot \mathsf{loop}^{-1}}_{-n} & \text{if } n < 0, \\ \mathsf{refl}_{\mathsf{base}} & \text{if } n = 0. \end{cases}$$

Defining a function $g : \Omega(\mathbb{S}^1) \to \mathbb{Z}$ in the other direction is a bit trickier. Note that the successor function $\mathsf{succ} : \mathbb{Z} \to \mathbb{Z}$ is an equivalence, and hence induces a path $\mathsf{ua}(\mathsf{succ}) : \mathbb{Z} = \mathbb{Z}$ in the universe $\mathcal{U}$. Thus, the recursion principle of $\mathbb{S}^1$ induces a map $c : \mathbb{S}^1 \to \mathcal{U}$ by $c(\mathsf{base}) :\equiv \mathbb{Z}$ and $\mathsf{ap}_c(\mathsf{loop}) :\equiv \mathsf{ua}(\mathsf{succ})$. Then we have $\mathsf{ap}_c : (\mathsf{base} = \mathsf{base}) \to (\mathbb{Z} = \mathbb{Z})$, and we can define $g(p) :\equiv \mathsf{ap}_c(p)(0)$.

With these definitions, we can even prove that $g(\mathsf{loop}^n) = n$ for any $n : \mathbb{Z}$, using the induction principle Lemma 6.10.12 for n. (We will prove something more general a little later on.) However, the other equality $\mathsf{loop}^{g(p)} = p$ is significantly harder. The obvious thing to try is path induction, but path induction does not apply to loops such as $p : (\mathsf{base} = \mathsf{base})$ that have *both* endpoints fixed! A new idea is required, one which can be explained both in terms of classical homotopy theory and in terms of type theory. We begin with the former.

8.1.2 The classical proof

In classical homotopy theory, there is a standard proof of $\pi_1(\mathbb{S}^1) = \mathbb{Z}$ using universal covering spaces. Our proof can be regarded as a type-theoretic version of this proof, with covering spaces appearing here as fibrations whose fibers are sets. Recall that *fibrations* over a space B in homotopy theory correspond to type families $B \to \mathcal{U}$ in type theory. In particular, for a point $x_0 : B$, the type family $(x \mapsto (x_0 = x))$ corresponds to the *path fibration* $P_{x_0}B \to B$, in which the points of $P_{x_0}B$ are paths in B starting at x_0, and the map to B selects the other endpoint of such a path. This total space $P_{x_0}B$ is contractible, since we can "retract" any path to its initial endpoint x_0 — we have seen the type-theoretic version of this as Lemma 3.11.8. Moreover, the fiber over x_0 is the loop space $\Omega(B, x_0)$ — in type theory this is obvious by definition of the loop space.

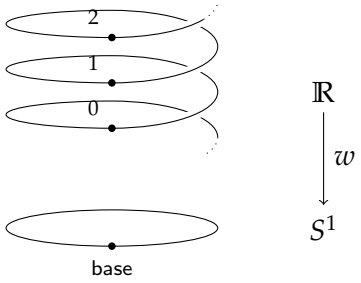


Figure 8.1: The winding map in classical topology

Now in classical homotopy theory, where $\mathbb{S}^1$ is regarded as a topological space, we may proceed as follows. Consider the "winding" map $w : \mathbb{R} \to \mathbb{S}^1$, which looks like a helix projecting down onto the circle (see Figure 8.1). This map w sends each point on the helix to the point on the circle that it is "sitting above". It is a fibration, and the fiber over each point is isomorphic to the integers. If we lift the path that goes counterclockwise around the loop on the bottom, we go up one level in the helix, incrementing the integer in the fiber. Similarly, going clockwise around the loop on the bottom corresponds to going down one level in the helix, decrementing this count. This fibration is called the *universal cover* of the circle.

Now a basic fact in classical homotopy theory is that a map $E_1 \to E_2$ of fibrations over B which is a homotopy equivalence between E_1 and E_2 induces a homotopy equivalence on all fibers. (We have already seen the type-theoretic version of this as well in Theorem 4.7.7.) Since $\mathbb{R}$ and $P_{\mathsf{base}}S^1$ are both contractible topological spaces, they are homotopy equivalent, and thus their fibers $\mathbb{Z}$ and $\Omega(\mathbb{S}^1)$ over the basepoint are also homotopy equivalent.

8.1.3 The universal cover in type theory

Let us consider how we might express the preceeding proof in type theory. We have already remarked that the path fibration of $\mathbb{S}^1$ is represented by the type family $(x \mapsto (\mathsf{base} = x))$. We have also already seen a good candidate for the universal cover of $\mathbb{S}^1$: it's none other than the type family $c : \mathbb{S}^1 \to \mathcal{U}$ which we defined in §8.1.1! By definition, the fiber of this family over base is $\mathbb{Z}$, while the effect of transporting around loop is to add one — thus it behaves just as we would expect from Figure 8.1.

However, since we don't know yet that this family behaves like a universal cover is supposed to (for instance, that its total space is contractible), we use a different name for it. For reference, therefore, we repeat the definition.

Definition 8.1.1 (Universal Cover of $\mathbb{S}^1$)**.** Define $\mathsf{code} : \mathbb{S}^1 \to \mathcal{U}$ by circle-recursion, with

$$\mathsf{code}(\mathsf{base}) :\equiv \mathbb{Z}$$
$$\mathsf{ap}_{\mathsf{code}}(\mathsf{loop}) := \mathsf{ua}(\mathsf{succ}).$$

We emphasize briefly the definition of this family, since it is so different from how one usually defines covering spaces in classical homotopy theory. To define a function by circle recursion, we

need to find a point and a loop in the target. In this case, the target is $\mathcal{U}$, and the point we choose is $\mathbb{Z}$, corresponding to our expectation that the fiber of the universal cover should be the integers. The loop we choose is the successor/predecessor isomorphism on $\mathbb{Z}$, which corresponds to the fact that going around the loop in the base goes up one level on the helix. Univalence is necessary for this part of the proof, because we need to convert *non-trivial* equivalence on $\mathbb{Z}$ into an identity.

We call this the fibration of "codes", because its elements are combinatorial data that act as codes for paths on the circle: the integer n codes for the path which loops around the circle n times.

From this definition, it is simple to calculate that transporting with code takes loop to the successor function, and loop^{-1} to the predecessor function:

Lemma 8.1.2. $\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}, x) = x + 1$ *and* $\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^{-1}, x) = x - 1$

Proof. For the first, we calculate as follows:

$$\begin{aligned}
\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}, x) &= \mathsf{transport}^{A\mapsto A}((\mathsf{code}(\mathsf{loop})), x) && \text{by associativity}\\
&= \mathsf{transport}^{A\mapsto A}(\mathsf{ua}(\mathsf{succ}), x) && \text{by computation for } \mathsf{rec}_{\mathbb{S}^1}\\
&= x + 1 && \text{computation for ua}
\end{aligned}$$

The second follows from the first, because $\mathsf{transport}^B(p, -)$ and $\mathsf{transport}^B(p^{-1}, -)$ are always inverses, so $\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^{-1}, -)$ must be the inverse of succ. □

We can now see what was wrong with our first approach: we defined f and g only on the fibers $\Omega(\mathbb{S}^1)$ and $\mathbb{Z}$, when we should have defined a whole morphism *of fibrations* over $\mathbb{S}^1$. In type theory, this means we should have defined functions having types

$$\prod_{x:\mathbb{S}^1} ((\mathsf{base} = x) \to \mathsf{code}(x)) \qquad \text{and/or} \tag{8.1.3}$$

$$\prod_{x:\mathbb{S}^1} (\mathsf{code}(x) \to (\mathsf{base} = x)) \tag{8.1.4}$$

instead of only the special cases of these when x is base. This is also an instance of a common observation in type theory: when attempting to prove something about particular inhabitants of some inductive type, it is often easier to generalize the statement so that it refers to *all* inhabitants of that type, which we can then prove by induction. Looked at in this way, the proof of $\Omega(\mathbb{S}^1) = \mathbb{Z}$ fits into the same pattern as the characterization of the identity types of coproducts and natural numbers in §§2.12 and 2.13.

At this point, there are two ways to finish the proof. We can continue mimicking the classical argument by constructing (8.1.3) or (8.1.4) (it doesn't matter which), proving that a homotopy equivalence between total spaces induces an equivalence on fibers, and then that the total space of the universal cover is contractible. The first type-theoretic proof of $\Omega(\mathbb{S}^1) = \mathbb{Z}$ followed this pattern; we call it the *homotopy-theoretic* proof.

Later, however, we discovered that there is an alternative proof, which has a more type-theoretic feel and more closely follows the proofs in §§2.12 and 2.13. In this proof, we directly construct both (8.1.3) and (8.1.4), and prove that they are mutually inverse by calculation. We will call this the *encode-decode* proof, because we call the functions (8.1.3) and (8.1.4) *encode* and

decode respectively. Both proofs use the same construction of the cover given above. Where the classical proof induces an equivalence on fibers from an equivalence between total spaces, the encode-decode proof constructs the inverse map (*decode*) explicitly as a map between fibers. And where the classical proof uses contractibility, the encode-decode proof uses path induction, circle induction, and integer induction. These are the same tools used to prove contractibility—indeed, path induction *is* essentially contractibility of the path fibration composed with $\mathsf{transport}$—but they are applied in a different way.

Since this is a book about homotopy type theory, we present the encode-decode proof first. A homotopy theorist who gets lost is encouraged to skip to the homotopy-theoretic proof (§8.1.5).

8.1.4 The encode-decode proof

We begin with the function (8.1.3) that maps paths to codes:

Definition 8.1.5. Define $\mathsf{encode} : \prod_{(x:\mathbb{S}^1)}(\mathsf{base} = x) \to \mathsf{code}(x)$ by

$$\mathsf{encode}\ p :\equiv \mathsf{transport}^{\mathsf{code}}(p, 0)$$

(we leave the argument x implicit).

Encode is defined by lifting a path into the universal cover, which determines an equivalence, and then applying the resulting equivalence to 0. The interesting thing about this function is that it computes a concrete number from a loop on the circle, when this loop is represented using the abstract groupoidal framework of homotopy type theory. To gain an intuition for how it does this, observe that by the above lemmas, $\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}, x)$ is the successor function succ and $\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^{-1}, x)$ is the predecessor function pred. Further, $\mathsf{transport}$ is functorial (Chapter 2), so

$$\mathsf{transport}^{\mathsf{code}}(\mathsf{loop} \cdot \mathsf{loop}, -) \quad \text{is} \quad (\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}, -)) \circ (\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}, -))$$

and so on. Thus, when p is a composition like

$$\mathsf{loop} \cdot \mathsf{loop}^{-1} \cdot \mathsf{loop} \cdot \cdots$$

$\mathsf{transport}^{\mathsf{code}}(p, -)$ will compute a composition of functions like

$$\mathsf{succ} \circ \mathsf{pred} \circ \mathsf{succ} \circ \cdots$$

Applying this composition of functions to 0 will compute the *winding number* of the path—how many times it goes around the circle, with orientation marked by whether it is positive or negative, after inverses have been canceled. Thus, the computational behavior of encode follows from the reduction rules for higher-inductive types and univalence, and the action of $\mathsf{transport}$ on compositions and inverses.

Note that the instance $\mathsf{encode}' :\equiv \mathsf{encode}_{\mathsf{base}}$ has type $(\mathsf{base} = \mathsf{base}) \to \mathbb{Z}$. This will be one half of our desired equivalence; indeed, it is exactly the function g defined in §8.1.1.

Similarly, the function (8.1.4) is a generalization of the function loop^- from §8.1.1.

Definition 8.1.6. Define $\mathsf{decode} : \prod_{(x:\mathbb{S}^1)} \mathsf{code}(x) \to (\mathsf{base} = x)$ by circle induction on x. It suffices to give a function $\mathsf{code}(\mathsf{base}) \to (\mathsf{base} = \mathsf{base})$, for which we use loop^-, and to show that loop^- respects the loop.

Proof. To show that loop^- respects the loop, it suffices to give a path from loop^- to itself that lies over loop. Formally, this means a path from $\mathsf{transport}^{(x' \mapsto \mathsf{code}(x') \to (\mathsf{base} = x'))}(\mathsf{loop}, \mathsf{loop}^-)$ to loop^-. We define such a path as follows:

$$\begin{aligned}
&\mathsf{transport}^{(x' \mapsto \mathsf{code}(x') \to (\mathsf{base} = x'))}(\mathsf{loop}, \mathsf{loop}^-) \\
&\quad = \mathsf{transport}^{x' \mapsto (\mathsf{base} = x')}(\mathsf{loop}) \circ \mathsf{loop}^- \circ \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^{-1}) \\
&\quad = (- \cdot \mathsf{loop}) \circ (\mathsf{loop}^-) \circ \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^{-1}) \\
&\quad = (- \cdot \mathsf{loop}) \circ (\mathsf{loop}^-) \circ \mathsf{pred} \\
&\quad = (n \mapsto \mathsf{loop}^{n-1} \cdot \mathsf{loop})
\end{aligned}$$

On the first line, we apply the characterization of transport when the outer connective of the fibration is $\to$, which reduces the transport to pre- and post-composition with transport at the domain and range types. On the second line, we apply the characterization of transport when the type family is $x \mapsto \mathsf{base} = x$, which is post-composition of paths. On the third line, we use the action of code on loop^{-1} from Lemma 8.1.2. And on the fourth line, we simply reduce the function composition. Thus, it suffices to show that for all n, $\mathsf{loop}^{n-1} \cdot \mathsf{loop} = \mathsf{loop}^n$, which is an easy induction, using the groupoid laws. □

We can now show that encode and decode are quasi-inverses. What used to be the difficult direction is now easy!

Lemma 8.1.7. *For all for all $x : \mathbb{S}^1$ and $p : \mathsf{base} = x$, $\mathsf{decode}_x(\mathsf{encode}_x(p)) = p$.*

Proof. By path induction, it suffices to show that $\mathsf{decode}_{\mathsf{base}}(\mathsf{encode}_{\mathsf{base}}(\mathsf{refl}_{\mathsf{base}})) = \mathsf{refl}_{\mathsf{base}}$. But $\mathsf{encode}_{\mathsf{base}}(\mathsf{refl}_{\mathsf{base}}) \equiv \mathsf{transport}^{\mathsf{code}}(\mathsf{refl}_{\mathsf{base}}, 0) \equiv 0$, and $\mathsf{decode}_{\mathsf{base}}(0) \equiv \mathsf{loop}^0 \equiv \mathsf{refl}_{\mathsf{base}}$. □

The other direction is not much harder.

Lemma 8.1.8. *For all $x : \mathbb{S}^1$ and $c : \mathsf{code}(x)$, we have $\mathsf{encode}_x(\mathsf{decode}_x(c)) = c$.*

Proof. The proof is by circle induction. It suffices to show the case for base, because the case for loop is a path between paths in $\mathbb{Z}$, which is immediate because $\mathbb{Z}$ is a set.

Thus, it suffices to show, for all $n : \mathbb{Z}$, that

$$\mathsf{encode}'(\mathsf{loop}^n) = n$$

The proof is by induction, using Corollary 6.10.13.

- In the case for 0, the result is true by definition.

- In the case for $n+1$,

$$\begin{aligned}
\mathsf{encode}'(\mathsf{loop}^{n+1}) &= \mathsf{encode}'(\mathsf{loop}^n \cdot \mathsf{loop}) && \text{by definition of } \mathsf{loop}^- \\
&= \mathsf{transport}^{\mathsf{code}}((\mathsf{loop}^n \cdot \mathsf{loop}), 0) && \text{by definition of } \mathsf{encode} \\
&= \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}, (\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^n, 0))) && \text{by functoriality} \\
&= (\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^n, 0)) + 1 && \text{by Lemma 8.1.2} \\
&= n+1 && \text{by the IH}
\end{aligned}$$

- The case for negatives is analogous. □

Finally, we conclude the theorem.

Theorem 8.1.9. *There is a family of equivalences* $\prod_{(x:\mathbb{S}^1)}((\mathsf{base} = x) \simeq \mathsf{code}(x))$.

Proof. The maps encode and decode are quasi-inverses by Lemmas 8.1.7 and 8.1.7. □

Instantiating at base gives

Corollary 8.1.10. $\Omega(\mathbb{S}^1, \mathsf{base}) \simeq \mathbb{Z}$.

A simple induction shows that this equivalence takes addition to composition, so that $\Omega(\mathbb{S}^1) = \mathbb{Z}$ as groups.

Corollary 8.1.11. $\pi_1(\mathbb{S}^1) = \mathbb{Z}$, *while* $\pi_n(\mathbb{S}^1) = 0$ *for* $n > 1$.

Proof. For $n = 1$, we sketched the proof from Corollary 8.1.10 above. For $n > 1$, we have $\left\|\Omega^n(\mathbb{S}^1)\right\|_0 = \left\|\Omega^{n-1}(\Omega\mathbb{S}^1)\right\|_0 = \left\|\Omega^{n-1}(\mathbb{Z})\right\|_0$. And since $\mathbb{Z}$ is a set, $\Omega^{n-1}(\mathbb{Z})$ is contractible, so this is trivial. □

8.1.5 The homotopy-theoretic proof

In §8.1.3, we defined the putative universal cover $\mathsf{code} : \mathbb{S}^1 \to \mathcal{U}$ in type theory, and in §8.1.5 we defined a map $\mathsf{encode} : \prod_{(x:\mathbb{S}^1)}(\mathsf{base} = x) \to \mathsf{code}(x)$ from the path fibration to the universal cover. What remains for the classical proof is to show that this map induces an equivalence on total spaces because both are contractible, and to deduce from this that it must be an equivalence on each fiber.

We have already seen in Lemma 3.11.8 that the total space $\sum_{(x:\mathbb{S}^1)}(\mathsf{base} = x)$ is contractible. For the other, we have:

Lemma 8.1.12. *The type* $\sum_{(x:\mathbb{S}^1)} \mathsf{code}(x)$ *is contractible.*

Proof. We apply the flattening lemma (Lemma 6.12.2) with the following values:

- $A :\equiv \mathbf{1}$ and $B :\equiv \mathbf{1}$, with f and g the obvious functions. Thus, the base higher inductive type W in the flattening lemma is equivalent to $\mathbb{S}^1$.
- $C : A \to \mathcal{U}$ is constant at $\mathbb{Z}$.

- $D : \prod_{(b:B)}(\mathbb{Z} \simeq \mathbb{Z})$ is constant at succ.

Then the type family $P : \mathbb{S}^1 \to \mathcal{U}$ defined in the flattening lemma is equivalent to $\mathsf{code} : \mathbb{S}^1 \to \mathcal{U}$. Thus, the flattening lemma tells us that $\sum_{(x:\mathbb{S}^1)} \mathsf{code}(x)$ is equivalent to higher inductive type with the following generators, which we denote R:

- A function $\mathsf{c} : \mathbb{Z} \to R$.
- For each $z : \mathbb{Z}$, a path $\mathsf{p}_z : \mathsf{c}(z) = \mathsf{c}(\mathsf{succ}(z))$.

We might call this type the **homotopical reals**; it plays the same role as the topological space $\mathbb{R}$ in the classical proof.

Thus, it remains to show that R is contractible. As center of contraction we choose $\mathsf{c}(0)$; we must now show that $x = \mathsf{c}(0)$ for all $x : R$. We do this by induction on R. Firstly, when x is $\mathsf{c}(z)$, we must give a path $q_z : \mathsf{c}(0) = \mathsf{c}(z)$, which we can do by induction on $z : \mathbb{Z}$:

$$\begin{aligned} q_0 &:\equiv \mathsf{refl}_{\mathsf{c}(0)} \\ q_{n+1} &:\equiv q_n \cdot \mathsf{p}_n && n \geq 0 \\ q_{n-1} &:\equiv q_n \cdot {\mathsf{p}_{n-1}}^{-1} && n \leq 0. \end{aligned}$$

Secondly, we must show that for any $z : \mathbb{Z}$, the path q_z is transported along p_z to q_{z+1}. By transport of paths, this means we want $q_z \cdot \mathsf{p}_z = q_{z+1}$. This is easy by induction on z, using the definition of q_z. This completes the proof that R is contractible, and thus so is $\sum_{(x:\mathbb{S}^1)} \mathsf{code}(x)$. □

Corollary 8.1.13. *The map induced by* encode*:*

$$\textstyle\sum_{(x:\mathbb{S}^1)}(\mathsf{base} = x) \to \sum_{(x:\mathbb{S}^1)} \mathsf{code}(x)$$

is an equivalence.

Proof. Both types are contractible. □

Theorem 8.1.14. $\Omega(\mathbb{S}^1, \mathsf{base}) \simeq \mathbb{Z}$.

Proof. Apply Theorem 4.7.7 to encode, using Corollary 8.1.13. □

In essence, the two proofs are not very different: the encode-decode one may be seen as a "reduction" or "unpackaging" of the homotopy-theoretic one. Each has its advantages; the interplay between the two points of view is part of the interest of the subject.

8.1.6 The universal cover as an identity system

Note that the fibration $\mathsf{code} : \mathbb{S}^1 \to \mathcal{U}$ together with $0 : \mathsf{code}(\mathsf{base})$ is a *pointed predicate* in the sense of Definition 5.8.1. From this point of view, we can see that the encode-decode proof in §8.1.4 consists of proving that code satisfies Theorem 5.8.2(iii), while the homotopy-theoretic proof in §8.1.5 consists of proving that it satisfies Theorem 5.8.2(iv). This suggests a third approach.

Theorem 8.1.15. *The pair* $(\mathsf{code}, 0)$ *is an identity system at* $\mathsf{base} : \mathbb{S}^1$ *in the sense of Definition 5.8.1.*

Proof. Let $D : \prod_{(x:\mathbb{S}^1)} \mathsf{code}(x) \to \mathcal{U}$ and $d : D(\mathsf{base}, 0)$ be given; we want to define a function $f : \prod_{(x:\mathbb{S}^1)} \prod_{(c:\mathsf{code}(x))} D(x, c)$. By circle induction, it suffices to specify $f(\mathsf{base}) : \prod_{(c:\mathsf{code}(\mathsf{base}))} D(\mathsf{base}, c)$ and verify that $\mathsf{loop}_*(f(\mathsf{base})) = f(\mathsf{base})$.

Of course, $\mathsf{code}(\mathsf{base}) \equiv \mathbb{Z}$. By Lemma 8.1.2 and induction on n, we may obtain a path $p_n : \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^n, 0) = n$ for any integer n. Therefore, by paths in Σ-types, we have a path $\mathsf{pair}^=(\mathsf{loop}^n, p_n) : (\mathsf{base}, 0) = (\mathsf{base}, n)$ in $\sum_{(x:\mathbb{S}^1)} \mathsf{code}(x)$. Transporting d along this path in the fibration $\hat{D} : (\sum_{(x:\mathbb{S}^1)} \mathsf{code}(x)) \to \mathcal{U}$ associated to D, we obtain an element of $D(\mathsf{base}, n)$ for any $n : \mathbb{Z}$. We define this element to be $f(\mathsf{base})(n)$:

$$f(\mathsf{base})(n) :\equiv \mathsf{transport}^{\hat{D}}(\mathsf{pair}^=(\mathsf{loop}^n, p_n), d).$$

Now we need $\mathsf{transport}^{\lambda x.\, \prod_{(c:\mathsf{code}(x))} D(x,c)}(\mathsf{loop}, f(\mathsf{base})) = f(\mathsf{base})$. By Lemma 2.9.7, this means we need to show that for any $n : \mathbb{Z}$,

$$\mathsf{transport}^{\hat{D}}(\mathsf{pair}^=(\mathsf{loop}, \mathsf{refl}_{\mathsf{loop}_*(n)}), f(\mathsf{base})(n)) =_{D(\mathsf{base}, \mathsf{loop}_*(n))} f(\mathsf{base})(\mathsf{loop}_*(n)).$$

Now we have a path $q : \mathsf{loop}_*(n) = n + 1$, so transporting along this, it suffices to show

$$\mathsf{transport}^{D(\mathsf{base})}(q, \mathsf{transport}^{\hat{D}}(\mathsf{pair}^=(\mathsf{loop}, \mathsf{refl}_{\mathsf{loop}_*(n)}), f(\mathsf{base})(n))) \\ =_{D(\mathsf{base}, n+1)} \mathsf{transport}^{D(\mathsf{base})}(q, f(\mathsf{base})(\mathsf{loop}_*(n))).$$

By a couple of lemmas about transport and dependent application, this is equivalent to

$$\mathsf{transport}^{\hat{D}}(\mathsf{pair}^=(\mathsf{loop}, q), f(\mathsf{base})(n)) =_{D(\mathsf{base}, n+1)} f(\mathsf{base})(n + 1).$$

However, expanding out the definition of $f(\mathsf{base})$, we have

$$\begin{aligned} \mathsf{transport}^{\hat{D}}(\mathsf{pair}^=(\mathsf{loop}, q), f(\mathsf{base})(n)) &= \mathsf{transport}^{\hat{D}}(\mathsf{pair}^=(\mathsf{loop}, q), \mathsf{transport}^{\hat{D}}(\mathsf{pair}^=(\mathsf{loop}^n, p_n), d)) \\ &= \mathsf{transport}^{\hat{D}}(\mathsf{pair}^=(\mathsf{loop}^n, p_n) \cdot \mathsf{pair}^=(\mathsf{loop}, q), d) \\ &= \mathsf{transport}^{\hat{D}}(\mathsf{pair}^=(\mathsf{loop}^{n+1}, p_{n+1}), d) \\ &= f(\mathsf{base})(n + 1). \end{aligned}$$

We have used the functoriality of transport, the characterization of composition in Σ-types (which was an exercise for the reader), and a lemma relating p_n and q to p_{n+1} which we leave it to the reader to state and prove.

This completes the construction of $f : \prod_{(x:\mathbb{S}^1)} \prod_{(c:\mathsf{code}(x))} D(x, c)$. Since

$$f(\mathsf{base}, 0) \equiv \mathsf{pair}^=(\mathsf{loop}^0, p_0)_*(d) = \mathsf{refl}_{\mathsf{base}*}(d) = d,$$

we have shown that $(\mathsf{code}, 0)$ is an identity system. □

Corollary 8.1.16. *For any $x : \mathbb{S}^1$, we have $(\mathsf{base} = x) \simeq \mathsf{code}(x)$.*

Proof. By Theorem 5.8.2. □

Of course, this proof also contains essentially the same elements as the previous two. Roughly, we can say that it unifies the proofs of Definition 8.1.6 and Lemma 8.1.8, performing the requisite inductive argument only once in a generic case.

8.2 Connectedness of suspensions

Recall from §7.5 that a type A is called **n-connected** if $\|A\|_n$ is contractible. The aim of this section is to prove that the operation of suspension increases connectedness.

Theorem 8.2.1. *If A is n-connected then the suspension of A is $(n+1)$-connected.*

Proof. By definition, the suspension of A is $\mathbf{1} \sqcup^A \mathbf{1}$, so we need to prove that the following type is contractible:

$$\left\| \mathbf{1} \sqcup^A \mathbf{1} \right\|_{n+1}.$$

By Theorem 7.4.12 we know that $\left\| \mathbf{1} \sqcup^A \mathbf{1} \right\|_{n+1}$ is a pushout in $(n+1)$-Type of the diagram

$$\begin{array}{ccc} \|A\|_{n+1} & \longrightarrow & \|\mathbf{1}\|_{n+1} \\ \downarrow & & \\ \|\mathbf{1}\|_{n+1} & & \end{array}.$$

Given that $\|\mathbf{1}\|_{n+1} = \mathbf{1}$, the type $\left\| \mathbf{1} \sqcup^A \mathbf{1} \right\|_{n+1}$ is also a pushout of the following diagram in $(n+1)$-Type (because both diagrams are equal)

$$\mathscr{D} = \begin{array}{ccc} \|A\|_{n+1} & \longrightarrow & \mathbf{1} \\ \downarrow & & \\ \mathbf{1} & & \end{array}.$$

We will now prove that $\mathbf{1}$ is also a pushout of $\mathscr{D}$ in $(n+1)$-Type. Let E be an $(n+1)$-truncated type; we need to prove that the following map is an equivalence

$$\begin{cases} (\mathbf{1} \to E) & \longrightarrow & \mathrm{cocone}_{\mathscr{D}}(E) \\ y & \longmapsto & (y, y, \lambda u.\, \mathrm{refl}_{y(\star)}) \end{cases}.$$

where we recall that $\mathrm{cocone}_{\mathscr{D}}(E)$ is the type

$$\sum_{(f:\mathbf{1}\to E)} \sum_{(g:\mathbf{1}\to E)} (\|A\|_{n+1} \to (f(\star) =_E g(\star))).$$

The map $\begin{cases} (\mathbf{1} \to E) & \longrightarrow & E \\ f & \longmapsto & f(\star) \end{cases}$ is an equivalence, hence we also have

$$\mathrm{cocone}_{\mathscr{D}}(E) = \sum_{(x:E)} \sum_{(y:E)} (\|A\|_{n+1} \to (x =_E y)).$$

Now A is n-connected hence so is $\|A\|_{n+1}$ because $\left\| \|A\|_{n+1} \right\|_n = \|A\|_n = \mathbf{1}$, and $(x =_E y)$ is n-truncated because E is $(n+1)$-connected. Hence by Corollary 7.5.9 the following map is an equivalence

$$\begin{cases} (x =_E y) & \longrightarrow & (\|A\|_{n+1} \to (x =_E y)) \\ p & \longmapsto & \lambda z.\, p \end{cases}$$

Hence we have

$$\mathrm{cocone}_{\mathscr{D}}(E) = \sum_{(x:E)} \sum_{(y:E)} (x =_E y).$$

But the following map is an equivalence

$$\begin{cases} E & \longrightarrow & \sum_{(x:E)} \sum_{(y:E)} (x =_E y) \\ x & \longmapsto & (x, x, \mathsf{refl}_x) \end{cases}.$$

Hence

$$\mathrm{cocone}_{\mathscr{D}}(E) = E.$$

Finally we get an equivalence

$$(\mathbf{1} \to E) \simeq \mathrm{cocone}_{\mathscr{D}}(E)$$

We can now unfold the definitions in order to get the explicit expression of this map, and we see easily that this is exactly the map we had at the beginning.

Hence we proved that $\mathbf{1}$ in a pushout of $\mathscr{D}$ in $(n+1)$-Type. Using uniqueness of pushouts we get that $\left\|\mathbf{1} \sqcup^A \mathbf{1}\right\|_{n+1} = \mathbf{1}$ which proves that the suspension of A is $(n+1)$-connected. □

Corollary 8.2.2. *For all $n : \mathbb{N}$, the sphere $\mathbb{S}^n$ is $(n-1)$-connected.*

Proof. We prove this by induction on n. For $n = 0$ we have to prove that $\mathbb{S}^0$ is inhabited which is clear. Let $n : \mathbb{N}$ be such that $\mathbb{S}^n$ is $(n-1)$-connected. By definition $\mathbb{S}^{n+1}$ is the suspension of $\mathbb{S}^n$, hence by the previous lemma $\mathbb{S}^{n+1}$ is n-connected. □

8.3 $\pi_{k\leq n}$ of an n-connected space and $\pi_{k<n}(\mathbb{S}^n)$

Let (A, a) be a pointed type and $n : \mathbb{N}$. Recall from Example 6.11.4 that if $n > 0$ the set $\pi_n(A, a)$ has a group structure, and if $n > 1$ the group is Abelian.

We can now say something about homotopy groups of n-truncated and n-connected types.

Lemma 8.3.1. *If A is n-truncated and $a : A$, then $\pi_k(A, a) = \mathbf{1}$ for all $k > n$.*

Proof. The loop space of an n-type is an $(n-1)$-type, hence $\Omega^k(A, a)$ is an $(n-k)$-type, and we have $(n-k) \leq -1$ so $\Omega^k(A, a)$ is a mere proposition. But $\Omega^k(A, a)$ is inhabited, so it is actually contractible and $\pi_k(A, a) = \left\|\Omega^k(A, a)\right\|_0 = \|\mathbf{1}\|_0 = \mathbf{1}$. □

Lemma 8.3.2. *If A is n-connected and $a : A$, then we have*

$$\forall (k \leq n).\, \pi_k(A, a) = \mathbf{1}$$

Proof. We have the following sequence of equalities:

$$\pi_k(A, a) \;=\; \left\|\Omega^k(A, a)\right\|_0 \;=\; \Omega^k(\|(A, a)\|_k) \;=\; \Omega^k(\|\|(A, a)\|_n\|_k) \;=\; \Omega^k(\|\mathbf{1}\|_k) \;=\; \Omega^k(\mathbf{1}) \;=\; \mathbf{1}.$$

The third line uses the fact that $k \leq n$ in order to use that $\| - \|_k \circ \| - \|_n = \| - \|_k$ and the fourth line uses the fact that A is n-connected. □

Corollary 8.3.3. $\pi_k(\mathbb{S}^n) = \mathbf{1}$ *for $k < n$.*

Proof. The sphere $\mathbb{S}^n$ is $(n-1)$-connected by Corollary 8.2.2, so Lemma 8.3.2 applies. □

8.4 Fiber sequences and the long exact sequence

Given a fibration $E \twoheadrightarrow B$ with fiber F, there is a relation between the homotopy groups of F, E and B in the form of a long exact sequence. We will start by defining the fiber sequence associated to a map between pointed spaces

Definition 8.4.1. A **pointed space** is a space X together with a point $x_0 : X$. A **pointed map** between two pointed spaces (X, x_0) and (Y, y_0) is a map f between X and Y together with a path $f_0 : f(x_0) = y_0$.

Definition 8.4.2. Given a pointed space X, the **loop space** of X is the pointed type $((x_0 = x_0), \mathsf{refl}_{x_0})$, where x_0 is the base point of X.

Definition 8.4.3. Given a pointed map between pointed spaces $f : X \to Y$, we define a pointed map $\Omega f : \Omega X \to \Omega Y$ by

$$(\Omega f)(p) :\equiv {f_0}^{-1} \cdot f(p) \cdot f_0.$$

The path $(\Omega f)_0$ is the obvious path of type

$${f_0}^{-1} \cdot f(\mathsf{refl}_{x_0}) \cdot f_0 = \mathsf{refl}_{y_0}.$$

Definition 8.4.4. A **fiber sequence** is a (possibly bounded) sequence of pointed types and pointed maps

$$\ldots \longrightarrow X_{n+1} \xrightarrow{f_{n+1}} X_n \xrightarrow{f_n} X_{n-1} \longrightarrow \ldots$$

equipped with homotopies $f_n \circ f_{n+1} = 0$ for every n (where $0 : X_{n+1} \to X_{n-1}$ is the constant map at the base point of X_{n-1}) and such that the unique map $X_{n+1} \to \mathsf{fib}_{f_n}(x)$ making the following diagram commute is an equivalence.

$$\begin{array}{ccccc} X_{n+1} & \xrightarrow{f_{n+1}} & X_n & \xrightarrow{f_n} & X_{n-1} \\ \downarrow \sim & \nearrow & & & \\ \mathsf{fib}_{f_n}(x) & & & & \end{array}$$

Given a map $f : X \to Y$ it is easy to extend it to a fiber sequence infinite to the left and ending by f by taking the iterated fiber. The following lemma shows that what we obtain is the sequence of iterated loop spaces of the base space, the total space and the fiber.

Lemma 8.4.5. *Let $f : X \to Y$ be a pointed map of pointed spaces. Then the fiber of the fiber of f is equivalent to ΩY, and the map from the fiber iterated three times of f (which is ΩX) to the fiber of the fiber of f (which is ΩY) is $\Omega f \circ -^{-1}$.*

Proof. The fiber of the fiber of f is

$$\left\{ z : \mathsf{fib}_f(y_0) \;\middle|\; \mathsf{pr}_1(z) = x_0 \right\} = \left\{ x : A, p : f(x) = y_0 \mid x = x_0 \right\}.$$

So a point of this space F is a triple consisting of a point $x : X$, a path p from $f(x)$ to y_0 and a path q from x to x_0. Let's prove that F and ΩY are equivalent.

We define the maps $g : F \to \Omega Y$ and $h : \Omega Y \to F$ by

$$\begin{aligned} g(x, p, q) &:\equiv p^{-1} \cdot f(q) \cdot f_0, \\ h(p) &:\equiv (x_0, f_0 \cdot p^{-1}, \mathsf{refl}_{x_0}). \end{aligned}$$

We can now check that they are inverse to each other:

$$\begin{aligned} g(h(p)) &= g(x_0, f_0 \cdot p^{-1}, \mathsf{refl}_{x_0}) \\ &= (f_0 \cdot p^{-1})^{-1} \cdot f(\mathsf{refl}_{x_0}) \cdot f_0 \\ &= (p^{-1})^{-1} \cdot {f_0}^{-1} \cdot f_0 \\ &= p \end{aligned}$$

and

$$\begin{aligned} h(g(x, p, q)) &= h(p^{-1} \cdot f(q) \cdot f_0) \\ &= (x_0, f_0 \cdot (p^{-1} \cdot f(q) \cdot f_0)^{-1}, \mathsf{refl}_{x_0}) \\ &= (x_0, f_0 \cdot {f_0}^{-1} \cdot f(q)^{-1} \cdot (p^{-1})^{-1}, \mathsf{refl}_{x_0}) \\ &= (x_0, f(q)^{-1} \cdot p, \mathsf{refl}_{x_0}). \end{aligned}$$

But we have $(x, p, q) = (x_0, f(q)^{-1} \cdot p, \mathsf{refl}_{x_0})$ because q is an equality between x and x_0, the paths p and $f(q)^{-1} \cdot p$ are equal over q, and the paths q and refl_{x_0} are equal over q.

We have the following fiber sequence

$$F' \xrightarrow{k} F \longrightarrow F_0 \longrightarrow X \longrightarrow Y$$

and we have constructed equivalences $e_X : \Omega X \simeq F'$ and $e_Y : \Omega Y \simeq F$. The map $\Omega X \to \Omega Y$ obtained in this way sends a loop p to the following path:

$$\begin{aligned} e_Y^{-1}(k(e_X(p))) &= e_Y^{-1}(k((x_0, f_0), \mathsf{refl}_{x_0} \cdot p^{-1}, \mathsf{refl}_{(x_0, f_0)}) \\ &= e_Y^{-1}((x_0, f_0), \mathsf{refl}_{x_0} \cdot p^{-1}) \\ &= {f_0}^{-1} \cdot f\left(\mathsf{refl}_{x_0} \cdot p^{-1}\right) \cdot f_0 \\ &= (\Omega f)(p^{-1}) \end{aligned}$$

□

From this fiber sequence we will deduce an exact sequence of pointed sets.

Definition 8.4.6. An **exact sequence** is a (possibly bounded) sequence of pointed sets and pointed maps:

$$\ldots \longrightarrow X_{n+1} \longrightarrow X_n \longrightarrow X_{n-1} \longrightarrow \ldots$$

such that for every n the image of the map $X_{n+1} \to X_n$ (cf Definition 7.6.3) is equal to the kernel of the map $X_n \to X_{n-1}$ (the kernel is the set of elements of X_n whose image is equal to the base point of X_{n-1}).

If the sets X_n are groups, we require moreover that the maps are morphisms of groups.

Lemma 8.4.7. *The 0-truncation of a fiber sequence is an exact sequence.*

Proof. Given a fiber sequence

$$\mathsf{fib}_f(z_0) \xrightarrow{g} Y \xrightarrow{f} Z$$

we need to prove that

$$\left\|\mathsf{fib}_f(z_0)\right\|_0 \xrightarrow{\|g\|_0} \|Y\|_0 \xrightarrow{\|f\|_0} \|Z\|_0$$

is exact. We need to prove that $\mathsf{im}(\|g\|_0) \subset \ker(\|f\|_0)$ and $\ker(\|f\|_0) \subset \mathsf{im}(\|g\|_0)$. The first inclusion is equivalent to $\|g\|_0 \circ \|f\|_0 = 0$, which is true by functoriality of 0-truncation and using the fact that $g \circ f = 0$.

For the second inclusion assume we have $y' : \|Y\|_0$ and $p' : \|f\|_0(y') = |z_0|_0$, we want to prove that there merely exists a $t : \mathsf{fib}_f(z_0)$ such that $g(t) = y'$.

We want to prove a mere proposition, hence we can assume that y' is of the form $|y|_0$ for some $y : Y$ and then p' is of type $|f(y)|_0 = |z_0|_0$. According to Theorem 7.3.10 this type is equivalent to $\|f(y) = z_0\|_{-1}$ hence we can assume that p' is of the form $|p|_{-1}$ for some $p : f(y) = z_0$. But now the pair (y, p) is an element of $\left|\mathsf{fib}_f(z_0)\right|_0$ whose image by $\|g\|_0$ is $|y|_0 = y'$ which is what we want. □

Theorem 8.4.8. *Let $f : X \to Y$ be a pointed map between pointed spaces with fiber F. Then we have a long exact sequence of homotopy groups:*

$$\begin{array}{ccccc}
\vdots & & \vdots & & \vdots \\
 & \swarrow & & \swarrow & \\
\pi_k(F) & \longrightarrow & \pi_k(X) & \longrightarrow & \pi_k(Y) \\
 & \swarrow & & \swarrow & \\
\vdots & & \vdots & & \vdots \\
 & \swarrow & & \swarrow & \\
\pi_2(F) & \longrightarrow & \pi_2(X) & \longrightarrow & \pi_2(Y) \\
 & \swarrow & & \swarrow & \\
\pi_1(F) & \longrightarrow & \pi_1(X) & \longrightarrow & \pi_1(Y) \\
 & \swarrow & & \swarrow & \\
\pi_0(F) & \longrightarrow & \pi_0(X) & \longrightarrow & \pi_0(Y)
\end{array}$$

Proof. This is the exact sequence obtained by taking the 0-truncation of the fiber sequence associated to f. Note that π_k is a group for $k \geq 1$ and the maps $\pi_k(F) \to \pi_k(X)$ and $\pi_k(X) \to \pi_k(Y)$ are morphisms of groups. The fact that the maps $\pi_{k+1}(Y) \to \pi_k(F)$ are also morphisms of groups is not immediately obvious but can be easily checked using the formulas above. □

The usual properties of exact sequences of abelian groups can be proved as usual. In particular we have:

Lemma 8.4.9. *Suppose given an exact sequence of abelian groups:*

$$K \longrightarrow G \xrightarrow{f} H \longrightarrow Q.$$

(i) If $K = 0$, then f is injective.

(ii) If $Q = 0$, then f is surjective.

(iii) If $K = Q = 0$, then f is an isomorphism.

Proof. Since the kernel of f is the image of $K \to G$, if $K = 0$ then the kernel of f is $\{0\}$; hence f is injective because it's a group morphism. Similarly, since the image of f is the kernel of $H \to Q$, if $Q = 0$ then the image of f is all of H, so f is surjective. Hence f is an isomorphism of groups by Theorem 4.6.3. □

As an immediate application, we can now quantify in what way n-connectedness of a map is stronger than inducing an equivalence on n-truncations.

Corollary 8.4.10. *Let $f : A \to B$ be n-connected and $a : A$, and define $b :\equiv f(a)$. Then:*

(i) If $k \leq n$, then $\pi_k(f) : \pi_k(A, a) \to \pi_k(B, b)$ is an isomorphism.

(ii) If $k = n + 1$, then $\pi_k(f) : \pi_k(A, a) \to \pi_k(B, b)$ is surjective.

Proof. As part of the long exact sequence, for each k we have an exact sequence

$$\pi_k(\mathsf{fib}_f(b)) \longrightarrow \pi_k(A, a) \xrightarrow{f} \pi_k(B, b) \longrightarrow \pi_{k-1}(\mathsf{fib}_f(b)).$$

Now since f is n-connected, $\left\|\mathsf{fib}_f(b)\right\|_n$ is contractible. Therefore, if $k \leq n$, then $\pi_k(\mathsf{fib}_f(b)) = \left\|\Omega^k(\mathsf{fib}_f(b))\right\|_0 = \Omega^k(\left\|\mathsf{fib}_f(b)\right\|_k)$ is also contractible. Thus, $\pi_k(f)$ is an isomorphism for $k \leq n$ by Lemma 8.4.9(iii), while for $k = n + 1$ it is surjective by Lemma 8.4.9(ii). □

In §8.8 we will see that the converse of Corollary 8.4.10 also holds.

8.5 The Hopf fibration

In this section we will define the **Hopf fibration**.

Theorem 8.5.1 (Hopf Fibration)**.** *There is a fibration H over $\mathbb{S}^2$ whose fibers are $\mathbb{S}^1$ and whose total space is $\mathbb{S}^3$.*

The Hopf fibration will allow us to compute several homotopy groups of spheres. Indeed, we have the following long exact sequence of homotopy groups associated to the Hopf fibration

(see §8.4):

$$\begin{array}{ccccc}
\pi_k(\mathbb{S}^1) & \longrightarrow & \pi_k(\mathbb{S}^3) & \longrightarrow & \pi_k(\mathbb{S}^2) \\
\vdots & & \vdots & & \vdots \\
\pi_2(\mathbb{S}^1) & \longrightarrow & \pi_2(\mathbb{S}^3) & \longrightarrow & \pi_2(\mathbb{S}^2) \\
\pi_1(\mathbb{S}^1) & \longrightarrow & \pi_1(\mathbb{S}^3) & \longrightarrow & \pi_1(\mathbb{S}^2)
\end{array}$$

We've already computed all $\pi_n(\mathbb{S}^1)$ and $\pi_k(\mathbb{S}^n)$ for $k < n$, so the long exact sequence becomes the following:

$$\begin{array}{ccccc}
0 & \longrightarrow & \pi_k(\mathbb{S}^3) & \longrightarrow & \pi_k(\mathbb{S}^2) \\
\vdots & & \vdots & & \vdots \\
0 & \longrightarrow & \pi_3(\mathbb{S}^3) & \longrightarrow & \pi_3(\mathbb{S}^2) \\
0 & \longrightarrow & 0 & \longrightarrow & \pi_2(\mathbb{S}^2) \\
\mathbb{Z} & \longrightarrow & 0 & \longrightarrow & 0
\end{array}$$

In particular we get the following result:

Corollary 8.5.2. *We have $\pi_2(\mathbb{S}^2) \simeq \mathbb{Z}$ and $\pi_k(\mathbb{S}^3) \simeq \pi_k(\mathbb{S}^2)$ for every $k \geq 3$ (where the map is induced by the Hopf fibration, seen as a map from the total space $\mathbb{S}^3$ to the base space $\mathbb{S}^2$).*

Given that the double loop space of the circle is contractible, we actually get the slightly more precise result that the Hopf fibration induces an equivalence $\Omega^2(\mathbb{S}^3) \simeq \Omega^2(\mathbb{S}^2)$. In classical homotopy theory this would be a consequence of Whitehead's theorem, but Whitehead's theorem is not necessarily valid in homotopy type theory. We will not use the more precise version here though.

We first start with a lemma explaining how to construct fibrations over pushouts.

Lemma 8.5.3. *Let $\mathscr{D} = Y \xleftarrow{j} X \xrightarrow{k} Z$ be a span and assume that we have*

- *Two fibrations $E_Y : Y \to \mathcal{U}$ and $E_Z : Z \to \mathcal{U}$*
- *An equivalence e_X between $E_Y \circ j : X \to \mathcal{U}$ and $E_Z \circ k : X \to \mathcal{U}$, ie*

$$e_X : \prod_{x:X} E_Y(j(x)) \simeq E_Z(k(x))$$

Then we can construct a fibration $E : Y \sqcup^X Z \to \mathcal{U}$ such that

- *For all $y : Y$, $E(\mathsf{inl}(y)) \equiv E_Y(y)$*
- *For all $z : Z$, $E(\mathsf{inr}(z)) \equiv E_Z(z)$*
- *For all $x : X$, $E(\mathsf{glue}(x)) = \mathsf{ua}(e_X(x))$ (note that both sides of the equation are paths in $\mathcal{U}$ from $E_Y(j(x))$ to $E_Z(k(x))$)*

Moreover the total space of this fibration fits in the following pushout square:

$$\begin{array}{ccc} \sum_{(x:X)} E_Y(j(x)) & \xrightarrow{\mathsf{id}\times e_X}_{\sim} & \sum_{(x:X)} E_Z(k(x)) \xrightarrow{k\times\mathsf{id}} \sum_{(z:Z)} E_Z(z) \\ {\scriptstyle j\times\mathsf{id}}\downarrow & & \downarrow{\scriptstyle \mathsf{inr}} \\ \sum_{(y:Y)} E_Y(y) & \xrightarrow[\mathsf{inl}]{} & \sum_{(t:Y\sqcup^X Z)} E(t) \end{array}$$

Proof. We define E by simple elimination on the pushout $Y \sqcup^X Z$. We need for that to specify the value of E on elements of the form $\mathsf{inl}(y)$, $\mathsf{inr}(z)$ and the action of E on paths $\mathsf{glue}(x)$, so we can just choose the following values:

$$\begin{aligned} E(\mathsf{inl}(y)) &:\equiv E_Y(y), \\ E(\mathsf{inr}(z)) &:\equiv E_Z(z), \\ E(\mathsf{glue}(x)) &:= \mathsf{ua}(e_X(x)). \end{aligned}$$

To see that the total space of this fibration is a pushout, we apply the flattening lemma (Lemma 6.12.2) with the following values:

- $A :\equiv Y + Z$, $B :\equiv X$ and $f, g : B \to A$ are defined by $f(x) :\equiv \mathsf{inl}(j(x))$, $g(x) :\equiv \mathsf{inr}(k(x))$,
- the type family $C : A \to \mathcal{U}$ is defined by

$$C(\mathsf{inl}(y)) :\equiv E_Y(y) \qquad \text{and} \qquad C(\mathsf{inr}(z)) :\equiv E_Z(z),$$

- the family of equivalences $D : \prod_{(b:B)} C(f(b)) \simeq C(g(b))$ is defined to be e_X.

The base higher inductive type W in the flattening lemma is equivalent to the pushout $Y \sqcup^X Z$ and the type family $P : Y \sqcup^X Z \to \mathcal{U}$ is equivalent to the E defined above.

Thus the flattening lemma tells us that $\sum_{(t:Y\sqcup^X Z)} E(t)$ is equivalent the higher inductive type $E^{\mathrm{tot}'}$ with the following generators:

- a function $\mathsf{z} : \sum_{(a:Y+Z)} C(a) \to E^{\mathrm{tot}'}$,
- for each $x : X$ and $t : E_Y(j(x))$, a path $\mathsf{z}(\mathsf{inl}(j(x)), t) = \mathsf{z}(\mathsf{inr}(k(x)), e_C(t))$.

Using the flattening lemma again or a direct computation, it is easy to see that $\sum_{(a:Y+Z)} C(a) \simeq \sum_{(y:Y)} E_Y(y) + \sum_{(z:Z)} E_Z(z)$, hence $E^{\mathrm{tot}'}$ is equivalent to the higher inductive type E^{tot} with the following generators:

- a function $\mathsf{inl} : \sum_{(y:Y)} E_Y(y) \to E^{\mathrm{tot}}$,
- a function $\mathsf{inr} : \sum_{(z:Z)} E_Z(z) \to E^{\mathrm{tot}}$,
- For each $(x, t) : \sum_{(x:X)} E_Y(j(x))$ a path $\mathsf{glue}(x, t) : \mathsf{inl}(j(x), t) = \mathsf{inr}(k(x), e_X(t))$.

Thus the total space of E is the pushout of the total spaces of E_Y and E_Z, as required. □

8.5.1 The Hopf construction

Definition 8.5.4. A **H-space** consists of

- a type A,
- a base point $e : A$,
- a binary operation $\mu : A \times A \to A$, and
- for every $a : A$, equalities $\mu(e, a) = a$ and $\mu(a, e) = a$.

Lemma 8.5.5. *Let A be a connected H-space. Then for every $a : A$, the maps $\mu(a, -) : A \to A$ and $\mu(-, a) : A \to A$ are equivalences.*

Proof. Let us prove that for every $a : A$ the map $\mu(a, -)$ is an equivalence. The other statement is symmetric. The statement that $\mu(a, -)$ is an equivalence corresponds to a type family $P : A \to$ Prop and proving it corresponds to finding a section of this type family.

The type Prop is a set (Theorem 7.1.11) hence we can define a new type family $P' : \|A\|_0 \to$ Prop by $P'(|a|_0) :\equiv P(a)$. But A is connected by assumption, hence $\|A\|_0$ is contractible. This implies that in order to find a section of P', it is enough to find a point in the fiber of P' over $|e|_0$. But we have $P'(|e|_0) = P(e)$ which is inhabited because $\mu(e, -)$ is equal to the identity map by definition of an H-space, hence is an equivalence.

We have proved that for every $x : \|A\|_0$ the proposition $P'(x)$ is true, hence in particular for every $a : A$ the proposition $P(a)$ is true because $P(a)$ is $P'(|a|_0)$. □

Definition 8.5.6. Let A be a connected H-space. We define a fibration over ΣA using Lemma 8.5.3.

Given that ΣA is the pushout $\mathbf{1} \sqcup^A \mathbf{1}$, we can define a fibration over ΣA by specifying

- two fibrations over $\mathbf{1}$ (i.e. two types F_1 and F_2), and
- a family $e : A \to (F_1 \simeq F_2)$ of equivalences between F_1 and F_2, one for every element of A.

We take A for F_1 and F_2, and for $a : A$ we take the equivalence $\mu(a, -)$ for $e(a)$.

According to Lemma 8.5.3, we have the following diagram:

$$\begin{array}{ccccc} A & \xleftarrow{\mathsf{pr}_2} & A \times A & \xrightarrow{\mu} & A \\ \downarrow & & \downarrow{\scriptstyle \mathsf{pr}_1} & & \downarrow \\ 1 & \longleftarrow & A & \longrightarrow & 1 \end{array}$$

and the fibration we just constructed is a fibration over ΣA whose total space is the pushout of the top line.

Moreover, with $f(x, y) :\equiv (\mu(x, y), y)$ we have the following diagram:

$$\begin{array}{ccccc} A & \xleftarrow{\mathsf{pr}_2} & A \times A & \xrightarrow{\mu} & A \\ {\scriptstyle \mathsf{id}}\downarrow & & \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle \mathsf{id}} \\ A & \xleftarrow[\mathsf{pr}_2]{} & A \times A & \xrightarrow[\mathsf{pr}_1]{} & A \end{array}$$

The diagram commutes and the three vertical maps are equivalences, the inverse of f being the function g defined by

$$g(u,v) :\equiv ((\mu(-,v))^{-1}u, v).$$

This shows that the two lines are equivalent (hence equal) spans, so the total space of the fibration we constructed is equivalent to the pushout of the bottom line which is by definition the join of A with itself. We just proved:

Lemma 8.5.7. *Given a connected H-space A, there is a fibration, called the **Hopf construction**, over ΣA with fiber A and total space $A * A$.*

8.5.2 The Hopf fibration

We will first construct a structure of H-space on the circle $\mathbb{S}^1$, hence by Lemma 8.5.7 we will get a fibration over $\mathbb{S}^2$ with fiber $\mathbb{S}^1$ and total space $\mathbb{S}^1 * \mathbb{S}^1$. We will then prove that this join is equivalent to $\mathbb{S}^3$.

Lemma 8.5.8. *There is an H-space structure on the circle $\mathbb{S}^1$.*

Proof. For the base point of the H-space structure we choose base. Now we need to define the multiplication operation $\mu : \mathbb{S}^1 \times \mathbb{S}^1 \to \mathbb{S}^1$. We will first define $\widetilde{\mu} : \mathbb{S}^1 \to (\mathbb{S}^1 \to \mathbb{S}^1)$ and then μ will be defined by

$$\mu(x,y) :\equiv (\widetilde{\mu}(x))(y).$$

We define $\widetilde{\mu}$ by simple elimination for $\mathbb{S}^1$:

$$\widetilde{\mu}(\mathsf{base}) :\equiv \mathsf{id}_{\mathbb{S}^1}, \qquad \text{and} \qquad \widetilde{\mu}(\mathsf{loop}) := \mathsf{funext}(h).$$

where h is of type $\prod_{(x:\mathbb{S}^1)}(x = x)$ and is defined by dependent elimination in the following way:

$$h(\mathsf{base}) :\equiv \mathsf{loop} \qquad \text{and} \qquad h(\mathsf{loop}) := \alpha.$$

Thanks to Theorem 2.11.5, in order to define α it is enough to inhabit the type $\mathsf{loop} \cdot \mathsf{loop} = \mathsf{loop} \cdot \mathsf{loop}$ which can be done by the reflexivity path.

Now we just have to prove that $\mu(x, \mathsf{base}) = \mu(\mathsf{base}, x) = x$ for every $x : \mathbb{S}^1$. By definition, if $x : \mathbb{S}^1$ we have $\mu(\mathsf{base}, x) = (\widetilde{\mu}(\mathsf{base}))(x) = \mathsf{id}_{\mathbb{S}^1}(x) = x$. For the equality $\mu(x, \mathsf{base}) = x$ we do it by induction on $x : \mathbb{S}^1$:

- If x is base then $\mu(\mathsf{base}, \mathsf{base}) = \mathsf{base}$ by definition, so we have $\mathsf{refl}_{\mathsf{base}} : \mu(\mathsf{base}, \mathsf{base}) = \mathsf{base}$.
- When x varies along loop we need to prove that

$$\mathsf{refl}_{\mathsf{base}} \cdot \mathsf{ap}_{\lambda x.\, x}\mathsf{loop} = \mathsf{ap}_{\lambda x.\, \mu(x,\mathsf{base})}\mathsf{loop} \cdot \mathsf{refl}_{\mathsf{base}}.$$

  The left-hand side is equal to loop and for the right-hand side we have:

$$\begin{aligned}
\mathsf{ap}_{\lambda x.\, \mu(x,\mathsf{base})}\mathsf{loop} \cdot \mathsf{refl}_{\mathsf{base}} &= \mathsf{ap}_{\lambda x.\, (\widetilde{\mu}(x))(\mathsf{base})}\mathsf{loop} \\
&= \mathsf{happly}(\mathsf{ap}_{\lambda x.\, (\widetilde{\mu}(x))}\mathsf{loop}, \mathsf{base}) \\
&= \mathsf{happly}(\mathsf{funext}(h), \mathsf{base}) \\
&= h(\mathsf{base}) \\
&= \mathsf{loop}.
\end{aligned}$$

□

Recall from §6.8 that if A and B are two spaces, the **join** $A * B$ of A and B is the pushout of the diagram

$$A \xleftarrow{\mathsf{pr}_1} A \times B \xrightarrow{\mathsf{pr}_2} B.$$

Lemma 8.5.9. *The operation of join is associative: if A, B and C are three spaces then we have an equivalence $(A * B) * C \simeq A * (B * C)$.*

Proof. We define a map $f : (A * B) * C \to A * (B * C)$ by induction. We first need to define $f \circ \mathsf{inl} : A * B \to A * (B * C)$ which will be done by induction, then $f \circ \mathsf{inr} : C \to A * (B * C)$, and then $\mathsf{ap}_f \circ \mathsf{glue} : \prod_{(t:(A*B)\times C)} f(\mathsf{inl}(\mathsf{pr}_1(t))) = f(\mathsf{inr}(\mathsf{pr}_2(t)))$ which will be done by induction on the first component of t:

$$\begin{aligned}
(f \circ \mathsf{inl})(\mathsf{inl}(a)) &:\equiv \mathsf{inl}(a),\\
(f \circ \mathsf{inl})(\mathsf{inr}(b)) &:\equiv \mathsf{inr}(\mathsf{inl}(b)),\\
\mathsf{ap}_{f\circ\mathsf{inl}}(\mathsf{glue}(a,b)) &:= \mathsf{glue}(a,\mathsf{inl}(b)),\\
f(\mathsf{inr}(c)) &:\equiv \mathsf{inr}(\mathsf{inr}(c)),\\
\mathsf{ap}_f(\mathsf{glue}(\mathsf{inl}(a),c)) &:= \mathsf{glue}(a,\mathsf{inr}(c)),\\
\mathsf{ap}_f(\mathsf{glue}(\mathsf{inr}(b),c)) &:= \mathsf{ap}_{\mathsf{inr}}(\mathsf{glue}(b,c)),\\
\mathsf{apd}_{\lambda x.\,\mathsf{ap}_f(\mathsf{glue}(x,c))}(\mathsf{glue}(a,b)) &:= \mathsf{apd}_{\lambda x.\,\mathsf{glue}(a,x)}(\mathsf{glue}(b,c)).
\end{aligned}$$

For the last equation, note that the right-hand side is of type

$$\mathsf{transport}^{\lambda x.\,\mathsf{inl}(a)=\mathsf{inr}(x)}(\mathsf{glue}(b,c),\mathsf{glue}(a,\mathsf{inl}(b))) = \mathsf{glue}(a,\mathsf{inr}(c))$$

whereas it is supposed to be of type

$$\mathsf{transport}^{\lambda x.\,f(\mathsf{inl}(x))=f(\mathsf{inr}(c))}(\mathsf{glue}(a,b),\mathsf{ap}_f(\mathsf{glue}(\mathsf{inl}(a),c))) = \mathsf{ap}_f(\mathsf{glue}(\mathsf{inr}(b),c)).$$

But both types are equivalent to the following type:

$$\mathsf{glue}(a,\mathsf{inr}(c)) = \mathsf{glue}(a,\mathsf{inl}(b)) \cdot \mathsf{ap}_{\mathsf{inr}}(\mathsf{glue}(b,c)).$$

Similarly we can define a map $g : A * (B * C) \to (A * B) * C$, and checking that f and g are inverse to each other is a long and tedious but essentially straightforward computation. □

A more conceptual proof sketch is as follows.

Proof. Let us consider the following diagram where the maps are the obvious projections:

$$\begin{array}{ccccc}
A & \longleftarrow & A \times C & \longrightarrow & A \times C \\
\uparrow & & \uparrow & & \uparrow \\
A \times B & \longleftarrow & A \times B \times C & \longrightarrow & A \times C \\
\downarrow & & \downarrow & & \downarrow \\
B & \longleftarrow & B \times C & \longrightarrow & C
\end{array}$$

Taking the colimit of the columns gives the following diagram, whose colimit is $(A * B) * C$:

$$A * B \xleftarrow{} (A * B) \times C \xrightarrow{} C$$

On the other hand, taking the colimit of the lines gives a diagram whose colimit is $A * (B * C)$.

Hence using a Fubini-like theorem for colimits (that we haven't proved) we have an equivalence $(A * B) * C \simeq A * (B * C)$. The proof of this Fubini theorem for colimits still requires the long and tedious computation, though. □

Lemma 8.5.10. *For any type A, there is an equivalence $\Sigma A \simeq \mathbf{2} * A$.*

Proof. It is easy to define the two maps back and forth and to prove that they are inverse to each other. The details are left as an exercise to the reader. □

We can now construct the Hopf fibration:

Theorem 8.5.11. *There is a fibration over $\mathbb{S}^2$ of fiber $\mathbb{S}^1$ and total space $\mathbb{S}^3$.*

Proof. We proved that $\mathbb{S}^1$ has a structure of H-space (cf Lemma 8.5.8) hence by Lemma 8.5.7 there is a fibration over $\mathbb{S}^2$ of fiber $\mathbb{S}^1$ and total space $\mathbb{S}^1 * \mathbb{S}^1$. But by the two previous results and Lemma 6.5.1 we have:

$$\mathbb{S}^1 * \mathbb{S}^1 = (\Sigma\mathbf{2}) * \mathbb{S}^1 = (\mathbf{2} * \mathbf{2}) * \mathbb{S}^1 = \mathbf{2} * (\mathbf{2} * \mathbb{S}^1) = \Sigma(\Sigma\mathbb{S}^1) = \mathbb{S}^3.$$ □

8.6 The Freudenthal suspension theorem

Before proving the Freudenthal suspension theorem, we need some auxiliary lemmas about connectedness. In Chapter 7 we proved a number of facts about n-connected maps and n-types for fixed n; here we are now interested in what happens when we vary n. For instance, in Lemma 7.5.7 we showed that n-connected maps are characterized by an "induction principle" relative to families of n-types. If we want to "induct along" an n-connected map into a family of k-types for $k > n$, we don't immediately know that there is a function by such an induction principle, but the following lemma says that at least our ignorance can be quantified.

Lemma 8.6.1. *If $f : A \to B$ is n-connected and $P : B \to k\text{-}\mathsf{Type}$ is a family of k-types for $k \geq n$, then the induced function*

$$(- \circ f) : \Big(\prod_{b:B} P(b)\Big) \to \Big(\prod_{a:A} P(f(a))\Big)$$

is $(k - n - 2)$-truncated.

Proof. We induct on the natural number $k - n$. When $k = n$, this is Lemma 7.5.7. For the inductive step, suppose f is n-connected and P is a family of $k+1$-types. To show that $(- \circ f)$ is $(k - n - 1)$-truncated, let $k : \prod_{(a:A)} P(a)$; then we have

$$\mathsf{fib}_{(-\circ f)}(k) \simeq \sum_{(g:\prod_{(b:B)} P(b))} \prod_{(a:A)} g(f(a)) = k(a).$$

Let (g, p) and (h, q) lie in this type, so $p : g \circ f \sim k$ and $q : h \circ f \sim k$; then we also have

$$\big((g, p) = (h, q)\big) \simeq \sum_{r:g\sim h} r \circ f = p \cdot q^{-1}.$$

However, here the right-hand side is a fiber of the map

$$(- \circ f) : \Big(\prod_{b:B} Q(b)\Big) \to \Big(\prod_{a:A} Q(f(a))\Big)$$

where $Q(b) :\equiv (g(b) = h(b))$. Since P is a family of $(k+1)$-types, Q is a family of k-types, so the the inductive hypothesis implies that this fiber is a $(k-n-2)$-type. Thus, all path spaces of $\mathsf{fib}_{(-\circ f)}(k)$ are $(k-n-2)$-types, so it is a $(k-n-1)$-type. □

Recall that if (A, a_0) and (B, b_0) are pointed types, then their **wedge** $A \vee B$ is defined to be the pushout of $A \xleftarrow{a_0} \mathbf{1} \xrightarrow{b_0} B$. There is a canonical map $i : A \vee B \to A \times B$ defined by the two maps $\lambda a.\, (a, b_0)$ and $\lambda b.\, (a_0, b)$; the following lemma essentially says that this map is highly connected if A and B are so. It is a bit more convenient both to prove and use, however, if we use the characterization of connectedness from Lemma 7.5.7 and substitute in the universal property of the wedge (generalized to type families).

Lemma 8.6.2 (Wedge connectivity lemma)**.** *Suppose that (A, a_0) and (B, b_0) are n- and m-connected pointed types, respectively, with $n, m \geq 0$, and let $P : A \to B \to (n+m)$-Type. Then for any $f : \prod_{(a:A)} P(a, b_0)$ and $g : \prod_{(b:B)} P(a_0, b)$ with $p : f(a_0) = g(b_0)$, there exists $h : \prod_{(a:A)} \prod_{(b:B)} P(a, b)$ with homotopies $q : \prod_{(a:A)} h(a, b_0) = f(a)$ and $r : \prod_{(b:B)} h(a_0, b) = g(b)$ such that $p = q(a_0)^{-1} \cdot r(b_0)$.*

Proof. Define $P : A \to \mathcal{U}$ by

$$P(a) :\equiv \sum_{k:\prod_{(b:B)} P(a,b)} (f(a) = k(b_0)).$$

Then we have $(g, p) : P(a_0)$. Since $a_0 : \mathbf{1} \to A$ is $(n-1)$-connected, if P is a family of $(n-1)$-types then we will have $\ell : \prod_{(a:A)} P(a)$ such that $\ell(a_0) = (g, p)$, in which case we can define $h(a, b) :\equiv \mathsf{pr}_1(\ell(a))(b)$. However, for fixed a, the type $P(a)$ is the fiber over $f(a)$ of the map

$$\Big(\prod_{b:B} P(a, b)\Big) \to P(a, b_0)$$

given by precomposition with $b_0 : \mathbf{1} \to B$. Since $b_0 : \mathbf{1} \to B$ is $(m-1)$-connected, for this fiber to be $(n-1)$-connected, by Lemma 8.6.1 it suffices for each type $P(a, b)$ to be an $(n+m)$-type, which we have assumed. □

Let (X, x_0) be a pointed type, and recall the definition of the suspension ΣX from §6.5, with constructors $\mathsf{N}, \mathsf{S} : \Sigma X$ and $\mathsf{merid} : X \to (\mathsf{N} = \mathsf{S})$. We regard ΣX as a pointed space with basepoint N, so that we have $\Omega\Sigma X :\equiv (\mathsf{N} =_{\Sigma X} \mathsf{N})$. Then there is a canonical map

$$\sigma : X \to \Omega\Sigma X$$
$$\sigma(x) :\equiv \mathsf{merid}(x) \cdot \mathsf{merid}(x_0)^{-1}.$$

Remark 8.6.3. In classical algebraic topology, one considers the *reduced suspension*, in which the path $\mathsf{merid}(x_0)$ is collapsed down to a point, identifying N and S. The reduced and unreduced suspensions are homotopy equivalent, so the distinction is invisible to our purely homotopy-theoretic eyes — and higher inductive types only allow us to "identify" points up to a higher path anyway, there is no purpose to considering reduced suspensions in homotopy type theory. However, the unreducedness of our suspension is the reason for the (possibly unexpected) appearance of $\mathsf{merid}(x_0)^{-1}$ in the definition of σ.

Our goal is now to prove the following.

Theorem 8.6.4 (The Freudenthal suspension theorem)**.** *Suppose that X is n-connected and pointed, with $n \geq 0$. Then the map $\sigma : X \to \Omega\Sigma(X)$ is $2n$-connected.*

We will use the encode-decode method, but applied in a slightly different way. Originally, we used it when we wanted to characterize the loop space $\Omega(A, a_0)$ of some type as equivalent to some other type B, which we did by constructing a family $\mathsf{code} : A \to \mathcal{U}$ with $\mathsf{code}(a_0) :\equiv B$ and a family of equivalences $\mathsf{decode} : \prod_{(x:A)} \mathsf{code}(x) \simeq (a_0 = x)$. We have also generalized it to characterize truncations of loop spaces by way of a family of equivalences $\prod_{(x:A)} \mathsf{code}(x) \simeq \|a_0 = x\|_n$.

In this case, however, we want to show that $\sigma : X \to \Omega\Sigma X$ is $2n$-connected. We could use the previous method to prove that $\|X\|_{2n} \to \|\Omega\Sigma X\|_{2n}$ is an equivalence, but this is a slightly weaker statement than the map being $2n$-connected (insert references, proof?). However, note that in the general case, to prove that $\mathsf{decode}(x)$ is an equivalence, we could equivalently be proving that its fibers are contractible, and we would still be able to use induction over the base type. This we can generalize to prove connectedness of a map into a loop space, i.e. that the *truncations* of its fibers are contractible. Moreover, instead of constructing code and decode separately, we can construct directly a family of *codes for the truncations of the fibers*.

Lemma 8.6.5. *If X is n-connected and pointed with $n \geq 0$, then there is a family*

$$\mathsf{code} : \prod_{y:\Sigma X} (\mathsf{N} = y) \to \mathcal{U} \tag{8.6.6}$$

such that

$$\mathsf{code}(\mathsf{N}, p) :\equiv \|\mathsf{fib}_\sigma(p)\|_{2n} \quad \equiv \quad \left\| \textstyle\sum_{(x:X)} (\mathsf{merid}(x) \cdot \mathsf{merid}(x_0)^{-1} = p) \right\|_{2n} \tag{8.6.7}$$

$$\mathsf{code}(\mathsf{S}, q) :\equiv \|\mathsf{fib}_{\mathsf{merid}}(q)\|_{2n} \quad \equiv \quad \left\| \textstyle\sum_{(x:X)} (\mathsf{merid}(x) = q) \right\|_{2n}. \tag{8.6.8}$$

Our eventual goal will be to prove that $\mathsf{code}(y, p)$ is contractible for all $y : \Sigma X$ and $p : \mathsf{N} = y$, then applying this with $y :\equiv \mathsf{N}$ will prove that all fibers of σ are $2n$-connected, and thus σ is $2n$-connected.

Proof of Lemma 8.6.5. We define $\mathsf{code}(y, p)$ by induction on $y : \Sigma X$, with (8.6.7) and (8.6.8) as the first two cases. It remains to construct, for each $x_1 : X$, a dependent path

$$\mathsf{code}(\mathsf{N}) =^{\lambda y.\, (\mathsf{N}=y)\to\mathcal{U}}_{\mathsf{merid}(x_1)} \mathsf{code}(\mathsf{S}).$$

By Lemma 2.9.6, this is equivalent to giving a family of paths

$$\prod_{q:\mathsf{N}=\mathsf{S}} \mathsf{code}(\mathsf{N})(\mathsf{transport}^{\lambda y.\,(\mathsf{N}=y)}(\mathsf{merid}(x_1), q)) = \mathsf{code}(\mathsf{S})(q).$$

And by univalence and transport in path types, this is equivalent to a family of equivalences

$$\prod_{q:\mathsf{N}=\mathsf{S}} \mathsf{code}(\mathsf{N}, q \cdot \mathsf{merid}(x_1)^{-1}) \simeq \mathsf{code}(\mathsf{S}, q).$$

We will define a family of maps

$$\prod_{q:\mathsf{N}=\mathsf{S}} \mathsf{code}(\mathsf{N}, q \cdot \mathsf{merid}(x_1)^{-1}) \to \mathsf{code}(\mathsf{S}, q). \tag{8.6.9}$$

and then show that they are all equivalences. Thus, let $q : \mathsf{N} = \mathsf{S}$; by the universal property of truncation and the definitions of $\mathsf{code}(\mathsf{N}, -)$ and $\mathsf{code}(\mathsf{S}, -)$, it will suffice to define for each $x_2 : X$, a map

$$\left(\mathsf{merid}(x_2) \cdot \mathsf{merid}(x_0)^{-1} = q \cdot \mathsf{merid}(x_1)^{-1}\right) \to \left\| \textstyle\sum_{(x:X)} (\mathsf{merid}(x) = q) \right\|_{2n}.$$

Now for each $x_1, x_2 : X$, this type is $2n$-truncated, while X is n-connected. Thus, by Lemma 8.6.2, it suffices to define this map when x_1 is x_0, when x_2 is x_0, and check that they agree when both are x_0.

When x_1 is x_0, the hypothesis is $r : \mathsf{merid}(x_2) \cdot \mathsf{merid}(x_0)^{-1} = q \cdot \mathsf{merid}(x_0)^{-1}$. Thus, by canceling $\mathsf{merid}(x_0)^{-1}$ from r to get $r' : \mathsf{merid}(x_2) = q$, so we can define the image to be $|(x_2, r')|_{2n}$.

When x_2 is x_0, the hypothesis is $r : \mathsf{merid}(x_0) \cdot \mathsf{merid}(x_0)^{-1} = q \cdot \mathsf{merid}(x_1)^{-1}$. Rearranging this, we obtain $r'' : \mathsf{merid}(x_1) = q$, and we can define the image to be $|(x_1, r'')|_{2n}$.

Finally, when both x_1 and x_2 are x_0, it suffices to show the resulting r' and r'' agree; this is an easy lemma about path composition. This completes the definition of (8.6.9). To show that it is a family of equivalences, since being an equivalence is a mere proposition and $x_0 : \mathbf{1} \to X$ is (at least) (-1)-connected, it suffices to assume x_1 is x_0. In this case, inspecting the above construction we see that it is essentially the $2n$-truncation of the function that cancels $\mathsf{merid}(x_0)^{-1}$, which is an equivalence. □

In addition to (8.6.7) and (8.6.8), we will need to extract from the construction of code some information about how it acts on paths. For this we use the following lemma.

Lemma 8.6.10. *Let $A : \mathcal{U}$, $B : A \to \mathcal{U}$, and $C : \prod_{(a:A)} B(a) \to \mathcal{U}$, and also $a_1, a_2 : A$ with $m : x = y$ and $b : B(a_2)$. Then the function*

$$\mathsf{transport}^{\hat{C}}(\mathsf{pair}^{=}(m, t), -) : C(a_1, \mathsf{transport}^{B}(m^{-1}, b)) \to C(a_2, b),$$

where $t : \mathsf{transport}^{B}(m, \mathsf{transport}^{B}(m^{-1}, b)) = b$ is the obvious coherence path and $\hat{C} : (\sum_{(a:A)} B(a)) \to \mathcal{U}$ is the uncurried form of C, is equal to the equivalence obtained by univalence from the composite

$$\begin{aligned} C(x, \mathsf{transport}^{B}(m^{-1}, b)) &= \mathsf{transport}^{\lambda a.\, B(a) \to \mathcal{U}}(m, C(a_1))(b) && \text{by (2.9.4)} \\ &= C(a_2, b) && \text{by } \mathsf{happly}(\mathsf{apd}_C(m), b). \end{aligned}$$

Proof. By path induction, we may assume y is x and m is refl_x, in which case both functions are the identity. □

We apply this lemma with $A :\equiv \Sigma X$ and $B :\equiv \lambda y.\, (\mathsf{N} = y)$ and $C :\equiv \mathsf{code}$, while $a_1 :\equiv \mathsf{N}$ and $a_2 :\equiv \mathsf{S}$ and $m :\equiv \mathsf{merid}(x_1)$ for some $x_1 : X$, and finally $b :\equiv q$ is some path $\mathsf{N} = \mathsf{S}$. The computation rule for induction over ΣX identifies $\mathsf{apd}_C(m)$ with a path constructed in a certain way out of univalence and function extensionality. The second function described in Lemma 8.6.10 essentially consists of undoing these applications of univalence and function extensionality, reducing back to the particular functions (8.6.9) that we defined using Lemma 8.6.2. Therefore, Lemma 8.6.10 says that transporting along $\mathsf{pair}^{=}(q, t)$ essentially recovers these functions.

Finally, by construction, when x_1 or x_2 coincides with x_0 and the input is in the image of $|-|_{2n}$, we know more explicitly what these functions are. Thus, for any $x_2 : X$, we have

$$\mathsf{transport}^{\hat{\mathsf{code}}}(\mathsf{pair}^{=}(\mathsf{merid}(x_0), t), |(x_2, r)|_{2n}) = \left|(x_1, r'')\right|_{2n} \tag{8.6.11}$$

where $r : \mathsf{merid}(x_2) \cdot \mathsf{merid}(x_0)^{-1} = q \cdot \mathsf{merid}(x_0)^{-1}$ as before, and $r' : \mathsf{merid}(x_2) = q$ is obtained by canceling $\mathsf{merid}(x_0)^{-1}$. Similarly, for any $x_1 : X$, we have

$$\mathsf{transport}^{\hat{\mathsf{code}}}(\mathsf{pair}^{=}(\mathsf{merid}(x_1), t), |(x_0, r)|_{2n}) = \left|(x_1, r'')\right|_{2n} \tag{8.6.12}$$

where $r : \mathsf{merid}(x_0) \cdot \mathsf{merid}(x_0)^{-1} = q \cdot \mathsf{merid}(x_1)^{-1}$, and $r'' : \mathsf{merid}(x_1) = q$ is obtained by rearrangement.

Proof of Theorem 8.6.4. It remains to show that $\mathsf{code}(y, p)$ is contractible for each $y : \Sigma X$ and $p : \mathsf{N} = y$. First we must choose a center of contraction, say $c(y, p) : \mathsf{code}(y, p)$. This corresponds to the definition of the function encode in our previous proofs, so we define it by transport. Note that in the special case when y is N and p is $\mathsf{refl}_{\mathsf{N}}$, we have

$$\mathsf{code}(\mathsf{N}, \mathsf{refl}_{\mathsf{N}}) \equiv \left\| \textstyle\sum_{(x:X)} (\mathsf{merid}(x) \cdot \mathsf{merid}(x_0)^{-1} = \mathsf{refl}_{\mathsf{N}}) \right\|_{2n}.$$

Thus, we can choose $c(\mathsf{N}, \mathsf{refl}_{\mathsf{N}}) :\equiv \left|(x_0, \mathsf{rinv}_{\mathsf{merid}(x_0)})\right|_{2n}$, where rinv_q is the obvious path $q \cdot q^{-1} = \mathsf{refl}$ for any q. We can now obtain $c : \prod_{(y:\Sigma X)} \prod_{(p:\mathsf{N}=y)} \mathsf{code}(y, p)$ by path-induction on p, but it will be important below that we can also give a concrete definition in terms of transport:

$$c(y, p) :\equiv \mathsf{transport}^{\hat{\mathsf{code}}}(\mathsf{pair}^{=}(p, \mathsf{tid}_p), c(\mathsf{N}, \mathsf{refl}_{\mathsf{N}}))$$

where $\hat{\mathsf{code}} : \left(\sum_{(y:\Sigma X)} (\mathsf{N} = y)\right) \to \mathcal{U}$ is the uncurried version of code, and $\mathsf{tid}_p : p_*(\mathsf{refl}) = p$ is a standard lemma.

Next, we must show that every element of $\mathsf{code}(y, p)$ is equal to $c(y, p)$. Again, by path induction, it suffices to assume y is N and p is $\mathsf{refl}_{\mathsf{N}}$. In fact, we will prove it more generally when y is N and p is arbitrary. That is, we will show that for any $p : \mathsf{N} = \mathsf{N}$ and $d : \mathsf{code}(\mathsf{N}, p)$ we have $d = c(\mathsf{N}, p)$. Since this equality is a $(2n-1)$-type, we may assume d is of the form $|(x_1, r)|_{2n}$ for some $x_1 : X$ and $r : \mathsf{merid}(x_1) \cdot \mathsf{merid}(x_0)^{-1} = p$.

Now by a further path induction, we may assume that r is reflexivity, and p is $\mathsf{merid}(x_1) \cdot \mathsf{merid}(x_0)^{-1}$. (This is why we generalized to arbitrary p above.) Thus, we have to prove that

$$\left|(x_1, \mathsf{refl}_{\mathsf{merid}(x_1)\cdot\mathsf{merid}(x_0)^{-1}})\right|_{2n} = c\left(\mathsf{N}, \mathsf{refl}_{\mathsf{merid}(x_1)\cdot\mathsf{merid}(x_0)^{-1}}\right). \tag{8.6.13}$$

By definition, the right-hand side of this equality is

$$\begin{aligned}
&\mathsf{transport}^{\hat{\mathsf{code}}}\left(\mathsf{pair}^{=}(\mathsf{merid}(x_1)\cdot\mathsf{merid}(x_0)^{-1}, _), |(x_0, _)|_{2n}\right) \\
&\quad = \mathsf{transport}^{\hat{\mathsf{code}}}\left(\mathsf{pair}^{=}(\mathsf{merid}(x_0)^{-1}, _), \mathsf{transport}^{\hat{\mathsf{code}}}\left(\mathsf{pair}^{=}(\mathsf{merid}(x_1), _), |(x_0, _)|_{2n}\right)\right) \\
&\quad\quad = \mathsf{transport}^{\hat{\mathsf{code}}}\left(\mathsf{pair}^{=}(\mathsf{merid}(x_0)^{-1}, _), |(x_1, _)|_{2n}\right) = |(x_1, _)|_{2n}
\end{aligned}$$

where the blanks _ denote coherence paths. Here the first step is functoriality of transport, the second invokes (8.6.12), and the third invokes (8.6.11) (with transport moved to the other side). Thus we have the same first component as the left-hand side of (8.6.13). We leave it to the reader to verify that the coherence paths all cancel, giving reflexivity in the second component. □

Corollary 8.6.14 (Freudenthal Equivalence)**.** *Suppose that X is n-connected and pointed, with $n \geq 0$. Then $\|X\|_{2n} \simeq \|\Omega\Sigma(X)\|_{2n}$.*

Proof. By Theorem 8.6.4, σ is $2n$-connected. By Lemma 7.5.14, it is therefore an equivalence on $2n$-truncations. □

One important corollary of the Freudenthal suspension theorem is that the homotopy groups of spheres are stable in a certain range (these are the northeast-to-southwest diagonals in Table 8.1):

Corollary 8.6.15 (Stability for Spheres)**.** *If $k \leq 2n-2$, then $\pi_{k+1}(S^{n+1}) = \pi_k(S^n)$.*

Proof. Assume $k \leq 2n-2$. By Corollary 8.2.2, $\mathbb{S}^n$ is $(n-1)$-connected. Therefore, by Corollary 8.6.14,

$$\|\Omega(\Sigma(\mathbb{S}^n))\|_{2(n-1)} = \|\mathbb{S}^n\|_{2(n-1)}$$

By Lemma 7.3.13, because $k \leq 2(n-1)$, applying $\|-\|_k$ to both sides shows that this equation holds for k:

$$\|\Omega(\Sigma(\mathbb{S}^n))\|_k = \|\mathbb{S}^n\|_k. \tag{8.6.16}$$

Then, the main idea of the proof is as follows; we omit checking that these equivalences act

appropriately on the base points of these spaces:

$$\begin{aligned}
\pi_{k+1}(\mathbb{S}^{n+1}) &\equiv \left\|\Omega^{k+1}(\mathbb{S}^{n+1})\right\|_0 \\
&\equiv \left\|\Omega^k(\Omega(\mathbb{S}^{n+1}))\right\|_0 \\
&\equiv \left\|\Omega^k(\Omega(\Sigma(\mathbb{S}^n)))\right\|_0 \\
&= \Omega^k(\|(\Omega(\Sigma(\mathbb{S}^n)))\|_k) && \text{(by Theorem 7.3.10)} \\
&= \Omega^k(\|\mathbb{S}^n\|_k) && \text{(by (8.6.16))} \\
&= \left\|\Omega^k(\mathbb{S}^n)\right\|_0 && \text{(by Theorem 7.3.10)} \\
&\equiv \pi_k\mathbb{S}^n. && \square
\end{aligned}$$

This means that once we have calculated one entry in one of these stable diagonals, we know all of them. For example:

Theorem 8.6.17. *$\pi_n(\mathbb{S}^n) = \mathbb{Z}$ for every $n \geq 1$.*

Proof. The proof is by induction on n. We already have $\pi_1(\mathbb{S}^1) = \mathbb{Z}$ (Corollary 8.1.11) and $\pi_2(\mathbb{S}^2) = \mathbb{Z}$ (Corollary 8.5.2). When $n \geq 2$, $n \leq (2n-2)$. Therefore, by Corollary 8.6.15, $\pi_{n+1}(S^{n+1}) = \pi_n(S^n)$, and this equivalence, combined with the inductive hypothesis, gives the result. □

Corollary 8.6.18. *$\mathbb{S}^{n+1}$ is not an n-type for any $n \geq -1$.*

8.7 The van Kampen theorem

The van Kampen theorem calculates the fundamental group π_1 of a (homotopy) pushout of spaces. It is traditionally stated for a topological space X which is the union of two open subspaces U and V, but in homotopy-theoretic terms this is just a convenient way of ensuring that X is the pushout of U and V over their intersection. Thus, we will prove a version of the van Kampen theorem for arbitrary pushouts.

In this section we will describe a proof of the van Kampen theorem which uses the same encode-decode method that we used for $\pi_1(\mathbb{S}^1)$ in §8.1. There is also a more homotopy-theoretic approach; see Exercise 9.11.

We need a more refined version of the encode-decode method. In §8.1 (as well as in §§2.12 and 2.13) we used it to characterize the path space of a (higher) inductive type W — deriving as a consequence a characterization of the loop space $\Omega(W)$, and thereby also of its 0-truncation $\pi_1(W)$. In the van Kampen theorem, our goal is only to characterize the fundamental group $\pi_1(W)$, and we do not have any explicit description of the loop spaces or the path spaces to use.

It turns out that we can use the same technique directly for a truncated version of the path fibration, thereby characterizing not only the fundamental *group* $\pi_1(W)$, but also the whole fundamental *groupoid*. Specifically, for a type X, write $\Pi_1 X : X \to X \to \mathcal{U}$ for the 0-truncation of its

identity type, i.e. $\Pi_1 X(x,y) :\equiv \|x = y\|_0$. Note that we have induced groupoid operations

$$(- \cdot -) : \Pi_1 X(x,y) \to \Pi_1 X(y,z) \to \Pi_1 X(x,z)$$
$$(-)^{-1} : \Pi_1 X(x,y) \to \Pi_1 X(y,x)$$
$$\mathsf{refl}_x : \Pi_1 X(x,x)$$
$$\mathsf{ap}_f : \Pi_1 X(x,y) \to \Pi_1 Y(fx,fy)$$

for which we use the same notation as the corresponding operations on paths.

8.7.1 Naive van Kampen

We begin with a "naive" version of the van Kampen theorem, which is useful but not quite as useful as the classical version. In §8.7.2 we will improve it to a more useful version.

Given types A, B, C and functions $f : A \to B$ and $g : A \to C$, let P be their pushout $B \sqcup^A C$. As we saw in §6.8, P is the higher inductive type generated by

- $i : B \to P$,
- $j : C \to P$, and
- for all $x : A$, a path $kx : ifx = jgx$.

Define $\mathsf{code} : P \to P \to \mathcal{U}$ by double induction on P as follows.

- $\mathsf{code}(ib, ib')$ is a set-quotient (see §6.10) of the type of sequences

  $$(b, p_0, x_1, q_1, y_1, p_1, x_2, q_2, y_2, p_2, \dots, y_n, p_n, b')$$

  where

  - $n : \mathbb{N}$
  - $x_k : A$ and $y_k : A$ for $0 < k \leq n$
  - $p_0 : \Pi_1 B(b, fx_1)$ and $p_n : \Pi_1 B(fy_n, b')$ for $n > 0$, and $p_0 : \Pi_1 B(b, b')$ for $n = 0$
  - $p_k : \Pi_1 B(fy_k, fx_{k+1})$ for $1 \leq k < n$
  - $q_k : \Pi_1 C(gx_k, gy_k)$ for $1 \leq k \leq n$

  The quotient is generated by the following equalities:

  $$(\dots, q_k, y_k, \mathsf{refl}_{fy_k}, y_k, q_{k+1}, \dots) = (\dots, q_k \cdot q_{k+1}, \dots)$$
  $$(\dots, p_k, x_k, \mathsf{refl}_{gx_k}, x_k, p_{k+1}, \dots) = (\dots, p_k \cdot p_{k+1}, \dots)$$

  (see Remark 8.7.3 below).
- $\mathsf{code}(jc, jc')$ is identical, with the roles of B and C reversed. We likewise notationally reverse the roles of x and y, and of p and q.
- $\mathsf{code}(ib, jc)$ and $\mathsf{code}(jc, ib)$ are similar, with the parity changed so that they start in one type and end in the other.

- For $a : A$ and $b : B$, we require an equivalence

$$\mathsf{code}(ib, ifa) \simeq \mathsf{code}(ib, jga). \tag{8.7.1}$$

  We define this to consist of the two functions defined on sequences by

$$\begin{aligned}(\dots, y_n, p_n, fa) &\mapsto (\dots, y_n, p_n, a, \mathsf{refl}_{ga}, ga)\\ (\dots, x_n, p_n, a, \mathsf{refl}_{fa}, fa) &\mapsto (\dots, x_n, p_n, ga)\end{aligned}$$

  Both of these functions are easily seen to respect the equivalence relations, and hence to define functions on the types of codes. The left-to-right-to-left composite is

$$(\dots, y_n, p_n, fa) \mapsto (\dots, y_n, p_n, a, \mathsf{refl}_{ga}, a, \mathsf{refl}_{fa}, fa)$$

  which is equal to the identity by a generating equality of the quotient. The other composite is analogous. Thus we have defined an equivalence (8.7.1).
- Similarly, we require equivalences

$$\begin{aligned}\mathsf{code}(jc, ifa) &\simeq \mathsf{code}(jc, jga)\\ \mathsf{code}(ifa, ib) &\simeq (jga, ib)\\ \mathsf{code}(ifa, jc) &\simeq (jga, jc)\end{aligned}$$

  all of which are defined in exactly the same way (the second two by adding reflexivity terms on the beginning rather than the end).
- Finally, we need to know that for $a, a' : A$, the following diagram commutes:

$$\begin{array}{ccc}\mathsf{code}(ifa, ifa') & \longrightarrow & \mathsf{code}(ifa, jga')\\ \downarrow & & \downarrow\\ \mathsf{code}(jga, ifa') & \longrightarrow & \mathsf{code}(jga, jga')\end{array}. \tag{8.7.2}$$

  This amounts to saying that if we add something to the beginning and then something to the end of a sequence, we might as well have done it in the other order.

Remark 8.7.3. One might expect to see in the definition of code some additional generating equations for the set-quotient, such as

$$\begin{aligned}(\dots, p_{k-1} \cdot fw, x'_k, q_k, \dots) &= (\dots, p_{k-1}, x_k, gw \cdot q_k, \dots) \qquad \text{for } w : \Pi_1 A(x_k, x'_k)\\ (\dots, q_k \cdot gw, y'_k, p_k, \dots) &= (\dots, q_k, y_k, fw \cdot p_k, \dots) \qquad \text{for } w : \Pi_1 A(y_k, y'_k).\end{aligned}$$

However, these are not necessary! In fact, they follow automatically by path-induction on w. This is the main difference between the "naive" van Kampen theorem and the more refined one we will consider in the next subsection.

Continuing on, we can characterize transporting in the fibration code:

- For $p : b =_B b'$ and $u : P$, we have

$$\mathsf{transport}^{b \mapsto \mathsf{code}(u,ib)}(p, (\dots, y_n, p_n, b)) = (\dots, y_n, p_n \cdot p, b').$$

- For $q : c =_C c'$ and $u : P$, we have

$$\mathsf{transport}^{c \mapsto \mathsf{code}(u,jc)}(q, (\dots, x_n, q_n, c)) = (\dots, x_n, q_n \cdot q, c').$$

Here we are abusing notation by using the same name for a path in X and its image in $\Pi_1 X$. Note that transport in $\Pi_1 X$ is also given by concatenation with (the image of) a path. From this we can prove the above statements by induction on u. We also have:

- For $a : A$ and $u : P$,

$$\mathsf{transport}^{v \mapsto \mathsf{code}(u,v)}(ha, (\dots, y_n, p_n, fa)) = (\dots, y_n, p_n, a, \mathsf{refl}_{ga}, ga).$$

This follows essentially from the definition of code.

We also construct a function

$$r : \prod_{u:P} \mathsf{code}(u, u)$$

by induction on u as follows:

$$rib :\equiv (b, \mathsf{refl}_b, b)$$
$$rjc :\equiv (c, \mathsf{refl}_c, c)$$

and for rka we take the composite equality

$$\begin{aligned}(ka, ka)_*(fa, \mathsf{refl}_{fa}, fa) &= (ga, \mathsf{refl}_{ga}, a, \mathsf{refl}_{fa}, a, \mathsf{refl}_{ga}, ga) \\ &= (ga, \mathsf{refl}_{ga}, ga)\end{aligned}$$

where the first equality is by the observation above about transporting in code, and the second is an instance of the set quotient relation used to define code.

We will now prove:

Theorem 8.7.4 (Naive van Kampen theorem)**.** *For all $u, v : P$ there is an equivalence*

$$\Pi_1 P(u, v) \simeq \mathsf{code}(u, v).$$

Proof. To define a function

$$\mathsf{encode} : \Pi_1 P(u, v) \to \mathsf{code}(u, v)$$

it suffices to define a function $(u =_P v) \to \mathsf{code}(u, v)$, since $\mathsf{code}(u, v)$ is a set. We do this by transport:

$$\mathsf{encode}(p) :\equiv \mathsf{transport}^{v \mapsto \mathsf{code}(u,v)}(p, r(u)).$$

Now to define

$$\mathsf{decode} : \mathsf{code}(u, v) \to \Pi_1 P(u, v)$$

we proceed as usual by induction on $u, v : P$. In each case for u and v, we apply i or j to all the equalities p_k and q_k as appropriate and concatenate the results in P, using h to identify the endpoints. For instance, when $u \equiv ib$ and $v \equiv ib'$, we define

$$\begin{aligned}&\mathsf{decode}(b, p_0, x_1, q_1, y_1, p_1, \dots, y_n, p_n, b') \\ &\qquad :\equiv i(p_0) \cdot h(x_1) \cdot j(q_1) \cdot h(y_1)^{-1} \cdot i(p_1) \cdot \dots \cdot h(y_n)^{-1} \cdot i(p_n). \quad (8.7.5)\end{aligned}$$

This respects the set-quotient equivalence relation and the equivalences such as (8.7.1), since $h : fi \sim gj$ is natural and f and g are functorial.

As usual, to show that the composite

$$\Pi_1 P(u, v) \xrightarrow{\mathsf{encode}} \mathsf{code}(u, v) \xrightarrow{\mathsf{decode}} \Pi_1 P(u, v)$$

is the identity, we first peel off the 0-truncation (since the target is a set) and then apply path-induction. The input refl_u goes to ru, which then goes back to refl_u (applying a further induction on u to decompose $\mathsf{decode}(ru)$).

Finally, consider the composite

$$\mathsf{code}(u, v) \xrightarrow{\mathsf{decode}} \Pi_1 P(u, v) \xrightarrow{\mathsf{encode}} \mathsf{code}(u, v).$$

We proceed by induction on $u, v : P$. When $u \equiv ib$ and $v \equiv ib'$, this composite is

$$\begin{aligned}(b, p_0, x_1, q_1, y_1, p_1, \dots, y_n, p_n, b') &\mapsto \left(ip_0 \cdot hx_1 \cdot jq_1 \cdot hy_1{}^{-1} \cdot ip_1 \cdot \dots \cdot hy_n{}^{-1} \cdot ip_n\right)_* (rib) \\ &= (ip_n)_* \cdots (jq_1)_* (hx_1)_* (ip_0)_* (b, \mathsf{refl}_b, b) \\ &= (ip_n)_* \cdots (jq_1)_* (hx_1)_* (b, p_0, if x_1) \\ &= (ip_n)_* \cdots (jq_1)_* (b, p_0, x_1, \mathsf{refl}_{gx_1}, jgx_1) \\ &= (ip_n)_* \cdots (b, p_0, x_1, q_1, jgy_1) \\ &= \quad \vdots \\ &= (b, p_0, x_1, q_1, y_1, p_1, \dots, y_n, p_n, b').\end{aligned}$$

i.e., the identity function. The other three point cases are analogous, and the path cases are trivial since all the types are sets. □

Theorem 8.7.4 allows us to calculate the fundamental groups of many spaces, provided A is a set, for in that case, each $\mathsf{code}(u, v)$ is, by definition, a set-quotient of a *set* by a relation.

Example 8.7.6. Let $A :\equiv \mathbf{2}$, $B :\equiv \mathbf{1}$, and $C :\equiv \mathbf{1}$. Then $P \simeq S^1$. Inspecting the definition of, say, $\mathsf{code}(i(\star), i(\star))$, we see that the paths all may as well be trivial, so the only information is in the sequence of elements $x_1, y_1, \dots, x_n, y_n : \mathbf{2}$. Moreover, if we have $x_k = y_k$ or $y_k = x_{k+1}$ for any k, then the set-quotient relations allow us to excise both of those elements. Thus, every such sequence is equal to a canonical *reduced* one in which no two adjacent elements are equal. Clearly such a reduced sequence is uniquely determined by its length (a natural number n) together with, if $n > 1$, the information of whether x_1 is $0_{\mathbf{2}}$ or $1_{\mathbf{2}}$, since that determines the rest of the sequence uniquely. And these data can, of course, be identified with an integer, where n is the absolute value and x_1 encodes the sign. Thus we recover $\pi_1(S^1) \cong \mathbb{Z}$.

Since Theorem 8.7.4 asserts only a bijection of families of sets, this isomorphism $\pi_1(S^1) \cong \mathbb{Z}$ is likewise only a bijection of sets. We could, however, define a concatenation operation on code (by concatenating sequences) and show that encode and decode form an isomorphism respecting this structure. (In the language of Chapter 9, these would be "pregroupoids".) We leave the details to the reader.

Example 8.7.7. More generally, let $B :\equiv \mathbf{1}$ and $C :\equiv \mathbf{1}$ but A be arbitrary, so that P is the suspension of A. Then once again the paths p_k and q_k are trivial, so that the only information in a path code is a sequence of elements $x_1, y_1, \dots, x_n, y_n : A$. The first two generating equalities say that adjacent equal elements can be canceled, so it makes sense to think of this sequence as a word of the form

$$x_1 y_1^{-1} x_2 y_2^{-1} \cdots x_n y_n^{-1}$$

in a group. Indeed, it looks similar to the free group on A (or equivalently on $\|A\|_0$; see Remark 6.11.8), but we are considering only words that start with a non-inverted element, alternate between inverted and non-inverted elements, and end with an inverted one. This effectively reduces the size of the generating set by one. For instance, if A has a point $a : A$, then we can identify $\pi_1(\Sigma A)$ with the group presented by $\|A\|_0$ as generators with the relation $|a|_0 = e$; see Exercises 8.1 and 8.2 for details.

Example 8.7.8. Let $A :\equiv \mathbf{1}$ and B and C be arbitrary, so that f and g simply equip B and C with basepoints b and c, say. Then P is the *wedge* $B \vee C$ of B and C (the coproduct in the category of based spaces). In this case, it is the elements x_k and y_k which are trivial, so that the only information is a sequence of loops $(p_0, q_1, p_1, \dots, p_n)$ with $p_k : \pi_1(B, b)$ and $q_k : \pi_1(C, c)$. Such sequences, modulo the equivalence relation we have imposed, are easily identified with the explicit description of the *free product* of the groups $\pi_1(B, b)$ and $\pi_1(C, c)$, as constructed in §6.11. Thus, we have $\pi_1(B \vee C) \cong \pi_1(B) * \pi_1(C)$.

However, Theorem 8.7.4 stops just short of being the full classical van Kampen theorem, which handles the case where A is not necessarily a set, and states that $\pi_1(B \sqcup^A C) \cong \pi_1(B) *_{\pi_1(A)} \pi_1(C)$ (with base point coming from A). Indeed, the conclusion of Theorem 8.7.4 says nothing at all about $\pi_1(A)$; the paths in A are "built into the quotienting" in a type-theoretic way that makes it hard to extract explicit information, in that $\mathsf{code}(u, v)$ is a set-quotient of a non-set by a relation. For this reason, in the next subsection we consider a better version of the van Kampen theorem.

8.7.2 The van Kampen theorem with a set of basepoints

The improvement of van Kampen we present now is closely analogous to a similar improvement in classical algebraic topology, where A is equipped with a *set S of base points*. In fact, it turns out to be unnecessary for our proof to assume that the "set of basepoints" is a *set* — it might just as well be an arbitrary type; the utility of assuming S is a set arises later, when applying the theorem to obtain computations. What is important is that S contains at least one point in each connected component of A. We formalize this in type theory by saying that we have a type S and a function $k : S \to A$ which is (-1)-*connected* in the "fiberwise" sense, i.e., left orthogonal to (-1)-truncated objects. (Apparently this is what a classical topologist would call a "0-connected" map, due to

thinking in terms of cofibers rather than fibers.) If $S \equiv A$ and k is the identity function, then we will recover the naive van Kampen theorem. Another example to keep in mind is when A is pointed and connected, with $k : \mathbf{1} \to A$ the point.

Let $A, B, C, f, g, P, i, j, h$ be as in the previous section. We now define, given our (-1)-connected map $k : S \to A$, an auxiliary type which improves the connectedness of k. Let T be the higher inductive type generated by

- A function $\ell : S \to T$, and
- For each $s, s' : S$, a function $m : (ks =_A ks') \to (\ell s =_T \ell s')$.

There is an obvious induced function $\bar{k} : T \to A$ such that $\bar{k}\ell = k$, and any $p : ks = ks'$ is equal to the composite $ks = \bar{k}\ell s \overset{\bar{k}mp}{=} \bar{k}\ell s' = ks'$.

Lemma 8.7.9. *$\bar{k}$ is (fiberwise) 0-connected.*

Proof. We must show that for all $a : A$, the 0-truncation of the type $\sum_{(t:T)}(\bar{k}t = a)$ is contractible. Since contractibility is a mere proposition and k is (-1)-connected, we may assume that $a = ks$ for some $s : S$. Now we can take the center of contraction to be $|(\ell s, q)|_0$ where q is the equality $\bar{k}\ell s = ks$.

It remains to show that for any $\phi : \left\| \sum_{(t:T)}(\bar{k}t = ks) \right\|_0$ we have $\phi = |(\ell s, q)|_0$. Since the latter is a mere proposition, and in particular a set, we may assume that $\phi = |(t, p)|_0$ for $t : T$ and $p : \bar{k}t = ks$.

Now we can do induction on $t : T$. If $t \equiv \ell s'$, then $ks' = \bar{k}\ell s' \overset{p}{=} ks$ yields via m an equality $\ell s = \ell s'$. Hence by definition of $\bar{k}$ and of equality in homotopy fibers, we obtain an equality $(ks', p) = (ks, q)$, and thus $|(ks', p)|_0 = |(ks, q)|_0$. Next we must show that as t varies along m these equalities agree. But they are equalities in a set (namely $\left\| \sum_{(t:T)}(\bar{k}t = ks) \right\|_0$), and hence this is automatic. □

Remark 8.7.10. T can be regarded as the (homotopy) coequalizer of the "kernel pair" of k. If S and A were sets, then the (-1)-connectivity of k would imply that A is the 0-truncation of this coequalizer (see Chapter 10). For general types, higher topos theory suggests that (-1)-connectivity of k will imply instead that A is the colimit (a.k.a. "geometric realization") of the "simplicial kernel" of k. The type T is the colimit of the "1-skeleton" of this simplicial kernel, so it makes sense that it improves the connectivity of k by 1. More generally, we might expect the colimit of the n-skeleton to improve connectivity by n.

Now we define $\mathsf{code} : P \to P \to \mathcal{U}$ by double induction as follows

- $\mathsf{code}(ib, ib')$ is now a set-quotient of the type of sequences

$$(b, p_0, x_1, q_1, y_1, p_1, x_2, q_2, y_2, p_2, \dots, y_n, p_n, b')$$

  where

  - $n : \mathbb{N}$
  - $x_k : S$ and $y_k : S$ for $0 < k \leq n$

- $p_0 : \Pi_1 B(b, fkx_1)$ and $p_n : \Pi_1 B(fky_n, b')$ for $n > 0$, and $p_0 : \Pi_1 B(b, b')$ for $n = 0$
- $p_k : \Pi_1 B(fky_k, fkx_{k+1})$ for $1 \leq k < n$
- $q_k : \Pi_1 C(gkx_k, gky_k)$ for $1 \leq k \leq n$

The quotient is generated by the following equalities (see Remark 8.7.3):

$$\begin{aligned}
(\dots, q_k, y_k, \mathsf{refl}_{fy_k}, y_k, q_{k+1}, \dots) &= (\dots, q_k \cdot q_{k+1}, \dots) \\
(\dots, p_k, x_k, \mathsf{refl}_{gx_k}, x_k, p_{k+1}, \dots) &= (\dots, p_k \cdot p_{k+1}, \dots) \\
(\dots, p_{k-1} \cdot fw, x'_k, q_k, \dots) &= (\dots, p_{k-1}, x_k, gw \cdot q_k, \dots) \qquad \text{for } w : \Pi_1 A(kx_k, kx'_k) \\
(\dots, q_k \cdot gw, y'_k, p_k, \dots) &= (\dots, q_k, y_k, fw \cdot p_k, \dots) \qquad \text{for } w : \Pi_1 A(ky_k, ky'_k).
\end{aligned}$$

We will need below the definition of the case of decode on such a sequence, which as before concatenates all the paths p_k and q_k together with instances of h to give an element of $\Pi_1 P(ifb, ifb')$, cf. (8.7.5). As before, the other three point cases are nearly identical.

- For $a : A$ and $b : B$, we require an equivalence

$$\mathsf{code}(ib, ifa) \simeq \mathsf{code}(ib, jga). \tag{8.7.11}$$

Since code is set-valued, by Lemma 8.7.9 we may assume that $a = \bar{k}t$ for some $t : T$. Next, we can do induction on t. If $t \equiv \ell s$ for $s : S$, then we define (8.7.11) as in §8.7.1:

$$\begin{aligned}
(\dots, y_n, p_n, fks) &\mapsto (\dots, y_n, p_n, s, \mathsf{refl}_{gks}, gks) \\
(\dots, x_n, p_n, s, \mathsf{refl}_{fks}, fks) &\mapsfrom (\dots, x_n, p_n, gks)
\end{aligned}$$

These respect the equivalence relations, and define quasi-inverses just as before. Now suppose t varies along $m_{s,s'}(w)$ for some $w : ks = ks'$; we must show that (8.7.11) respects transporting along $\bar{k}mw$. By definition of $\bar{k}$, this essentially boils down to transporting along w itself. By the characterization of transport in path types, what we need to show is that

$$w_*(\dots, y_n, p_n, fks) = (\dots, y_n, p_n \cdot fw, fks')$$

is mapped by (8.7.11) to

$$w_*(\dots, y_n, p_n, s, \mathsf{refl}_{gks}, gks) = (\dots, y_n, p_n, s, \mathsf{refl}_{gks} \cdot gw, gks')$$

But this follows directly from the new generators we have imposed on the set-quotient relation defining code.

- The other three requisite equivalences are defined similarly.
- Finally, since the commutativity (8.7.2) is a mere proposition, by (-1)-connectedness of k we may assume that $a = ks$ and $a' = ks'$, in which case it follows exactly as before.

Theorem 8.7.12 (van Kampen with a set of basepoints)**.** *For all $u, v : P$ there is an equivalence*

$$\Pi_1 P(u, v) \simeq \mathsf{code}(u, v).$$

with code *defined as in this section.*

Proof. Basically just like before. To show that decode respects the new generators of the quotient relation, we use the naturality of h. And to show that decode respects the equivalences such as (8.7.11), we need to induct on $\overline{k}$ and on T in order to decompose those equivalences into their definitions, but then it becomes again simply functoriality of f and g. The rest is easy. In particular, no additional argument is required for $\mathsf{encode} \circ \mathsf{decode}$, since the goal is to prove an equality in a set, and so the case of h is trivial. □

Theorem 8.7.12 allows us to calculate the fundamental groups of spaces, even when A is not a set, provided S is a set, for in that case, each $\mathsf{code}(u, v)$ is, by definition, a set-quotient of a *set* by a relation. In that respect, it is an improvement over Theorem 8.7.4.

Example 8.7.13. Suppose $S :\equiv \mathbf{1}$, so that A has a basepoint $a :\equiv k(\star)$ and is connected. Then code for loops in the pushout can be identified with alternating sequences of loops in $\pi_1(B, f(a))$ and $\pi_1(C, g(a))$, modulo an equivalence relation which allows us to slide elements of $\pi_1(A, a)$ between them (after applying f and g respectively). Thus, $\pi_1(P)$ can be identified with the *amalgamated free product* $\pi_1(B) *_{\pi_1(A)} \pi_1(C)$ (the pushout in the category of groups), as constructed in §6.11. This (in the case when B and C are open subspaces of P and A their intersection) is probably the most classical version of the van Kampen theorem.

Example 8.7.14. As a special case of Example 8.7.13, suppose additionally that $C :\equiv \mathbf{1}$, so that P is the cofiber B/A. Then every loop in C is equal to reflexivity, so the relations on path codes allow us to collapse all sequences to a single loop in B. The additional relations require that multiplying on the left, right, or in the middle by an element in the image of $\pi_1(A)$ is the identity. We can thus identify $\pi_1(B/A)$ with the quotient of the group $\pi_1(B)$ by the normal subgroup generated by the image of $\pi_1(A)$.

Example 8.7.15. As a further special case of Example 8.7.14, let $B :\equiv S^1 \vee S^1$, let $A :\equiv S^1$, and let $f : A \to B$ pick out the composite loop $p \cdot q \cdot p^{-1} \cdot q^{-1}$, where p and q are the generating loops in the two copies of S^1 comprising B. Then P is a presentation of the torus T^2. Indeed, it is not hard to identify P with the presentation of T^2 as described in §6.7, using the cone on a particular loop. Thus, $\pi_1(T^2)$ is the quotient of the free group on two generators (i.e., $\pi_1(B)$) by the relation $p \cdot q \cdot p^{-1} \cdot q^{-1} = 1$. This clearly yields the free *abelian* group on two generators, which is $\mathbb{Z} \times \mathbb{Z}$.

Example 8.7.16. More generally, any CW complex can be obtained by repeatedly "coning off" spheres, as described in §6.7. That is, we start with a set X_0 of points ("0-cells"), which is the "0-skeleton" of the CW complex. We take the pushout

$$\begin{array}{ccc} S_1 \times \mathbb{S}^0 & \xrightarrow{f_1} & X_0 \\ \downarrow & & \downarrow \\ \mathbf{1} & \longrightarrow & X_1 \end{array}$$

for some set S_1 of 1-cells and some family f_1 of "attaching maps", obtaining the "1-skeleton" X_1. Then we take the pushout

$$\begin{array}{ccc} S_2 \times \mathbb{S}^1 & \xrightarrow{f_2} & X_1 \\ \downarrow & & \downarrow \\ \mathbf{1} & \longrightarrow & X_2 \end{array}$$

for some set S_2 of 2-cells and some family f_2 of attaching maps, obtaining the 2-skeleton X_2, and so on. The fundamental group of each pushout can be calculated from the van Kampen theorem: we obtain the group presented by generators derived from the 1-skeleton, and relations derived from S_2 and f_2. The pushouts after this stage do not alter the fundamental group, since $\pi_1(\mathbb{S}^n)$ is trivial for $n > 1$ (we will prove this in §8.3).

Example 8.7.17. In particular, suppose given any presentation of a (set-)group $G = \langle X \mid R \rangle$, with X a set of generators and R a set of words in these generators. Let $B :\equiv \bigvee_X S^1$ and $A :\equiv \bigvee_R S^1$, with $f : A \to B$ sending each copy of S^1 to the corresponding word in the generating loops of B. It follows that $\pi_1(P) \cong G$; thus we have constructed a connected type whose fundamental group is G. Since any group has a presentation, any group is the fundamental group of some type. If we 1-truncate such a type, we obtain a type whose only nontrivial homotopy group is G; this is called an **Eilenberg–Mac Lane space** $K(G, 1)$.

8.8 Whitehead's theorem and Whitehead's principle

In classical homotopy theory, a map $f : A \to B$ which induces an isomorphism $\pi_n(A, a) \cong \pi_n(B, f(a))$ for all points a in A (and also an isomorphism $\pi_0(A) \cong \pi_0(B)$) is necessarily a homotopy equivalence, as long as the spaces A and B are well-behaved (e.g. have the homotopy types of CW-complexes). This is known as *Whitehead's theorem*. In fact, the "ill-behaved" spaces for which Whitehead's theorem fails are invisible to type theory. Roughly, the well-behaved topological spaces suffice to present ∞-groupoids, and homotopy type theory deals with ∞-groupoids directly rather than actual topological spaces. Thus, one might expect that Whitehead's theorem would be true in univalent foundations.

However, this is *not* the case: Whitehead's theorem is not provable. In fact, there are known models of type theory in which it fails to be true, although for entirely different reasons than its failure for ill-behaved topological spaces. These models are "non-hypercomplete ∞-toposes" (see [Lur09]); roughly speaking, they consist of sheaves of ∞-groupoids over ∞-dimensional base spaces.

From a foundational point of view, therefore, we may speak of *Whitehead's principle* as a "classicality axiom", akin to LEM and AC. It may consistently be assumed, but it is not part of the computationally motivated type theory, nor does it hold in all natural models. But when working from set-theoretic foundations, this principle is invisible: it cannot fail to be true in a world where ∞-groupoids are built up out of sets (using topological spaces, simplicial sets, or any other such model).

This may seem odd, but actually it should not be surprising. Homotopy type theory is the *abstract* theory of homotopy types, whereas the homotopy theory of topological spaces or simplicial sets in set theory is a *concrete* model of this theory, in the same way that the integers are a concrete model of the abstract theory of rings. It is to be expected that any concrete model will have special properties which are not intrinsic to the corresponding abstract theory, but which we might sometimes want to assume as additional axioms (e.g. the integers are a Principal Ideal Domain, but not all rings are).

It is beyond the scope of this book to describe any models of type theory, so we will not explain how Whitehead's principle might fail in some of them. However, we can prove that it

holds whenever the types involved are n-truncated for some finite n, by "downward" induction on n. In addition to being of interest in its own right (for instance, it implies the essential uniqueness of Eilenberg–Mac Lane spaces), the proof of this result will hopefully provide some intuitive explanation for why we cannot hope to prove an analogous theorem without truncation hypotheses.

We begin with the following modification of Theorem 4.6.3, which will eventually supply the induction step in the proof of the truncated Whitehead's principle. It may be regarded as a type-theoretic, ∞-groupoidal version of the classical statement that a fully faithful and essentially surjective functor is an equivalence of categories.

Theorem 8.8.1. *Suppose $f : A \to B$ is a function such that*

(i) $\|f\|_0 : \|A\|_0 \to \|B\|_0$ is surjective, and
(ii) for any $x, y : A$, the function $\mathsf{ap}_f : (x =_A y) \to (f(x) =_B f(y))$ is an equivalence.

Then f is an equivalence.

Proof. Note that (ii) is precisely the statement that f is an embedding, c.f. §4.6. Thus, by Theorem 4.6.3, it suffices to show that f is surjective, i.e. that for any $b : B$ we have $\left\|\mathsf{fib}_f(b)\right\|_{-1}$. Suppose given b; then since $\|f\|_0$ is surjective, there merely exists an $a : A$ such that $\|f\|_0(|a|_0) = |b|_0$. And since our goal is a mere proposition, we may assume given such an a. Then we have $|f(a)|_0 = \|f\|_0(|a|_0) = |b|_0$, hence $\|f(a) = b\|_{-1}$. Again, since our goal is still a mere proposition, we may assume $f(a) = b$. Hence $\mathsf{fib}_f(b)$ is inhabited, and thus merely inhabited. □

Since homotopy groups are truncations of loop spaces, rather than path spaces, we need to modify this theorem to speak about these instead.

Corollary 8.8.2. *Suppose $f : A \to B$ is a function such that*

(i) $\|f\|_0 : \|A\|_0 \to \|B\|_0$ is a bijection, and
(ii) for any $x : A$, the function $\mathsf{ap}_f : \Omega(A, x) \to \Omega(B, f(x))$ is an equivalence.

Then f is an equivalence.

Proof. By Theorem 8.8.1, it suffices to show that $\mathsf{ap}_f : (x =_A y) \to (f(x) =_B f(y))$ is an equivalence for any $x, y : A$. And by Corollary 4.4.7, we may assume $f(x) =_B f(y)$. In particular, $|f(x)|_0 = |f(y)|_0$, so since $\|f\|_0$ is an equivalence, we have $|x|_0 = |y|_0$, hence $|x = y|_{-1}$. Since we are trying to prove a mere proposition (f being an equivalence), we may assume given $p : x = y$. But now the following square commutes up to homotopy:

$$\begin{array}{ccc} \Omega(A, x) & \xrightarrow{- \cdot p} & x =_A y \\ {\scriptstyle \mathsf{ap}_f}\downarrow & & \downarrow{\scriptstyle \mathsf{ap}_f} \\ \Omega(B, f(x)) & \xrightarrow[- \cdot f(p)]{} & f(x) =_B f(y). \end{array}$$

The top and bottom maps are equivalences, and the left-hand map is so by assumption. Hence, by the 2-out-of-3 property, so is the right-hand map. □

Now we can prove the truncated Whitehead's principle.

Theorem 8.8.3. *Suppose A and B are n-types and $f : A \to B$ is such that*

(i) $\|f\|_0 : \|A\|_0 \to \|B\|_0$ *is an isomorphism, and*
(ii) $\pi_k(f) : \pi_k(A, x) \to \pi_k(B, f(x))$ *is a bijection for all* $k \geq 1$ *and all* $x : A$.

Then f is an equivalence.

Condition (i) is almost the case of (ii) when $k = 0$, except that it makes no reference to any basepoint $x : A$.

Proof. We proceed by induction on n. When $n = -2$, the statement is trivial. Thus, suppose it to be true for all functions between n-types, and let A and B be $(n+1)$-types and $f : A \to B$ as above. The first condition in Corollary 8.8.2 holds by assumption, so it will suffice to show that for any $x : A$, the function $\mathsf{ap}_f : \Omega(A, x) \to \Omega(B, f(x))$ is an equivalence. However, $\Omega(A, x)$ and $\Omega(B, f(x))$ are n-types, and $\pi_k(\mathsf{ap}_f) = \pi_{k+1}(f)$, so this follows from the inductive hypothesis. □

Note that if A and B are not n-types for any finite n, then there is no way for the induction to get started.

Corollary 8.8.4. *If A is a 0-connected n-type and $\pi_k(A, a) = 0$ for all k and $a : A$, then A is contractible.*

Proof. Apply Theorem 8.8.3 to the map $A \to \mathbf{1}$. □

As an application, we can deduce the converse of Corollary 8.4.10.

Corollary 8.8.5. *For $n \geq 0$, a map $f : A \to B$ is n-connected if and only if the following all hold:*

(i) $\|f\|_0 : \|A\|_0 \to \|B\|_0$ *is an isomorphism.*
(ii) For any $a : A$ and $k \leq n$, the map $\pi_k(f) : \pi_k(A, a) \to \pi_k(B, f(a))$ is an isomorphism.
(iii) For any $a : A$, the map $\pi_{n+1}(f) : \pi_{n+1}(A, a) \to \pi_{n+1}(B, f(a))$ is surjective.

Proof. The "only if" direction is Corollary 8.4.10. Conversely, by the long exact sequence of a fibration (Theorem 8.4.8), the hypotheses imply that $\pi_k(\mathsf{fib}_f(f(a))) = 0$ for all $k \leq n$ and $a : A$, and that $\left\|\mathsf{fib}_f(f(a))\right\|_0$ is contractible. Since $\pi_k(\mathsf{fib}_f(f(a))) = \pi_k(\left\|\mathsf{fib}_f(f(a))\right\|_n)$ for $k \leq n$, and $\left\|\mathsf{fib}_f(f(a))\right\|_n$ is n-connected, by Corollary 8.8.4 it is contractible for any a.

It remains to show that $\left\|\mathsf{fib}_f(b)\right\|_n$ is contractible for $b : B$ not neccesarily of the form $f(a)$. However, by assumption, there is $x : \|A\|_0$ with $|b|_0 = \|f\|_0(x)$. Since contractibility is a mere proposition, we may assume x is of the form $|a|_0$ for $a : A$, in which case $|b|_0 = \|f\|_0(|a|_0) = |f(a)|_0$, and therefore $\|b = f(a)\|_{-1}$. Again since contractibility is a mere proposition, we may assume $b = f(a)$, and the result follows. □

A map f such that $\pi_k(f)$ is a bijection for all k is called **∞-connected** or a **weak equivalence**. A type Z is called **∞-truncated** or **hypercomplete** if the induced map

$$(- \circ f) : (B \to Z) \to (A \to Z)$$

is an equivalence whenever f is ∞-connected — that is, if Z thinks every ∞-connected map is an equivalence. Then if we want to assume Whitehead's principle as an axiom, we may use either of the following equivalent forms.

- Every ∞-connected function is an equivalence.
- Every type is ∞-truncated.

In higher topos models, the ∞-truncated types form a reflective subuniverse in the sense of §7.7 (the "hypercompletion" of an ∞-topos), but we do not know whether this is true in general.

It may not be obvious that there *are* any types which are not n-types for any n, but in fact there are. Indeed, in classical homotopy theory, $\mathbb{S}^n$ has this property for any $n \geq 2$. We have not proven this fact in homotopy type theory yet, but there are other types which we can prove to have "infinite truncation level".

Example 8.8.6. Suppose we have $B : \mathbb{N} \to \mathcal{U}$ such that for each n, the type $B(n)$ contains an n-loop which is not equal to n-fold reflexivity, say $p_n : \Omega^n(B(n), b_n)$ with $p_n \neq \mathsf{refl}^n_{b_n}$. (For instance, we could define $B(n) :\equiv \mathbb{S}^n$, once we have shown that $\pi_n(\mathbb{S}^n) = \mathbb{Z}$.) Consider $C :\equiv \prod_{(n:\mathbb{N})} B(n)$, with the point $c : C$ defined by $c(n) :\equiv b_n$. Since loop spaces commute with products, for any m we have

$$\Omega^m(C, c) \;\simeq\; \prod_{n:\mathbb{N}} \Omega^m(B(n), b_n).$$

Under this equivalence, refl^m_c corresponds to the function $(n \mapsto \mathsf{refl}^m_{b_n})$. Now define q_m in the right-hand type by

$$q_m(n) :\equiv \begin{cases} p_n & m = n \\ \mathsf{refl}^m_{b_n} & m \neq n. \end{cases}$$

If we had $q_m = (n \mapsto \mathsf{refl}^m_{b_n})$, then we would have $p_n = \mathsf{refl}^n_{b_n}$, which is not the case. Thus, $q_m \neq (n \mapsto \mathsf{refl}^m_{b_n})$, and so there is a point of $\Omega^m(C, c)$ which is unequal to refl^m_c. Hence C is not an m-type, for any $m : \mathbb{N}$.

We expect it should also be possible to show that a universe $\mathcal{U}$ itself is not an n-type for any n, using the fact that it contains higher inductive types such as $\mathbb{S}^n$ for all n. However, this has not yet been done.

8.9 A general statement of the encode-decode method

We have used the encode-decode method to characterize the path spaces of various types, including coproducts (Theorem 2.12.5), natural numbers (Theorem 2.13.1), truncations (Theorem 7.3.10), the circle (Corollary 8.1.10), and pushouts (Theorem 8.7.12). Variants of this technique are used in the proofs of many of the other theorems mentioned in the introduction to this section, such as $\pi_n(\mathbb{S}^n)$, the Freudenthal suspension theorem, the Blakers–Massey theorem, and $K(G, n)$. While it is tempting to try to abstract this method into a lemma, this is difficult because slightly different variants are needed for different problems. For example, different variations on the same method can be used to characterize a loop space (as in Theorem 2.12.5 and Corollary 8.1.10) or a whole path space (as in Theorem 2.13.1), to give a complete characterization of a loop space

($\Omega_1(\mathbb{S}^1)$) or only to characterize some truncation of it ($\pi_k(\mathbb{S})$), and to calculate homotopy groups or to prove that a map is n-connected (Freudenthal and Blakers-Massey).

However, we can state lemmas for specific variants of the method. The proofs of these lemmas are almost trivial; the main point is to clarify the method by stating them in generality. The simplest case is using an encode-decode method to characterize the loop space of a type, as in Theorem 2.12.5 and Corollary 8.1.10.

Lemma 8.9.1 (Encode-decode for Loop Spaces)**.** *Given a pointed type* (A, a_0) *and a fibration* $\mathsf{code} : A \to \mathcal{U}$*, if*

(i) $c_0 : \mathsf{code}(a_0)$,
(ii) $\mathsf{decode} : \prod_{(x:A)} \mathsf{code}(x) \to a_0 = x$,
(iii) for all $c : \mathsf{code}(a_0)$, $\mathsf{transport}^{\mathsf{code}}(\mathsf{decode}(c), c_0) = c$*, and*
(iv) $\mathsf{decode}(c_0) = \mathsf{refl}$,

then $(a_0 = a_0)$ *is equivalent to* $\mathsf{code}(a_0)$.

Proof. Define $\mathsf{encode} : \prod_{(x:A)} a_0 = x \to \mathsf{code}(x)$ by

$$\mathsf{encode}_x(\alpha) = \mathsf{transport}^{\mathsf{code}}(\alpha, c_0).$$

We show that encode_{a_0} and decode_{a_0} are quasi-inverses. The compostion $\mathsf{encode}_{a_0} \circ \mathsf{decode}_{a_0}$ is immediate by assumption (iii). For the other composition, we show

$$\prod_{(x:A)} \prod_{(p:a_0=x)} \mathsf{decode}_x(\mathsf{encode}_x p) = p.$$

By path induction, it suffices to show $\mathsf{decode}_{a_0}(\mathsf{encode}_{a_0}\mathsf{refl}) = \mathsf{refl}$. After reducing the transport, it suffices to show $\mathsf{decode}_{a_0}(c_0) = \mathsf{refl}$, which is assumption (iv). □

If a fiberwise equivalence between $a_0 = -$ and code is desired, it suffices to strengthen condition (iii) to

$$\prod_{(x:A)} \prod_{(c:\mathsf{code}(x))} \mathsf{encode}_x(\mathsf{decode}_x(c)) = c.a$$

However, to calculate a loop space (e.g. $\Omega(\mathbb{S}^1)$), this stronger assumption is not necessary.

Another variation, which comes up often when calculating homotopy groups, characterizes the truncation of a loop space:

Lemma 8.9.2 (Encode-decode for Truncations of Loop Spaces)**.** *Assume a pointed type* (A, a_0) *and a fibration* $\mathsf{code} : A \to \mathcal{U}$*, where for every* x, $\mathsf{code}(x)$ *is a k-type. Define*

$$\mathsf{encode} : \prod_{x:A} \|a_0 = x\|_k \to \mathsf{code}(x).$$

by truncation recursion (using the fact that $\mathsf{code}(x)$ *is a k-type), mapping* $\alpha : a_0 = x$ *to* $\mathsf{transport}^{\mathsf{code}}(\alpha, c_0)$*. Suppose:*

(i) $c_0 : \mathsf{code}(a_0)$,

(ii) $\mathsf{decode} : \prod_{(x:A)} \mathsf{code}(x) \to \|a_0 = x\|_k$,
(iii) $\mathsf{encode}_{a_0}(\mathsf{decode}_{a0}(c)) = c$ *for all* $c : \mathsf{code}(a_0)$, *and*
(iv) $\mathsf{decode}(c_0) = |\mathsf{refl}|$.

Then $\|a_0 = a_0\|_k$ *is equivalent to* $\mathsf{code}(a_0)$.

Proof. That $\mathsf{decode} \circ \mathsf{encode}$ is identity is immediate by (iii). To prove $\mathsf{encode} \circ \mathsf{decode}$, we first do a truncation induction, by which it suffices to show

$$\prod_{(x:A)} \prod_{(p:a_0=x)} \mathsf{decode}_x(\mathsf{encode}_x(|p|)) = |p|.$$

The truncation induction is allowed because paths in a k-type are a k-type. To show this type, we do a path induction, and after reducing the encode, use assumption (iv). □

8.10 Additional Results

Though we do not present the proofs in this chapter, we have also proved the following results in homotopy type theory.

Theorem 8.10.1. *There exists a k such that for all $n \geq 3$, $\pi_{n+1}(\mathbb{S}^n) = \mathbb{Z}_k$.*

Notes on the proof. The proof consists of a calculation of $\pi_4(\mathbb{S}^3)$, together with an appeal to stability (Corollary 8.6.15). In the classical statement of this result, k is 2. While we have not yet checked that k is in fact 2, our proof of $\pi_4(\mathbb{S}^3)$ is constructive, assuming univalence and higher inductive types are constructive. Thus, given a computational interpretation of homotopy type theory, we could run the proof on a computer to verify that k is 2. This example is quite intriguing, because it is the first calculation of a homotopy group for which we have not needed to know the answer in advance. □

Recall from §6.8 that $X \sqcup^C Y$ denotes the (homotopy) pushout of X and Y along C.

Theorem 8.10.2 (Blakers-Massey theorem)**.** *Suppose we are given maps $f : C \to X$, and $g : C \to Y$. Taking first the pushout $X \sqcup^C Y$ of f and g and then the pullback of its inclusions $\mathsf{inl} : X \to X \sqcup^C Y \leftarrow Y : \mathsf{inr}$, we have an induced map $C \to X \times_{(X \sqcup^C Y)} Y$.*

If f is i-connected and g is j-connected, then this induced map is $(i + j)$-connected. In other words, for any points $x : X$, $y : Y$, the corresponding fiber $C_{x,y}$ of $(f, g) : C \to X \times Y$ gives an approximation to the path space $\mathsf{inl}(x) =_{X \sqcup^C Y} \mathsf{inr}(y)$ in the pushout.

It should be noted that in classical algebraic topology, the Blakers–Massey theorem is often stated in a somewhat different form, where the maps f and g are replaced by inclusions of subcomplexes of CW complexes, and the homotopy pushout and homotopy pullback by a union and intersection, respectively. In order to express the theorem in homotopy type theory, we have to replace notions of this sort with ones that are homotopy-invariant. We have seen another example of this in the van Kampen theorem (§8.7), where we had to replace a union of open subsets by a homotopy pushout.

Theorem 8.10.3 (Eilenberg–Mac Lane Spaces)**.** *For any abelian group G and positive number n, there is a type $K(G,n)$ such that $\pi_n(K(G,n)) = G$, and $\pi_k(K(G,n)) = 0$ for $k \neq n$.*

Theorem 8.10.4 (Covering spaces)**.** *For a connected space A, there is an equivalence between covering spaces over A and sets with an action of $\pi_1(A)$.*

Notes

Theorem	Status
$\pi_1(\mathbb{S}^1)$	✔✔
$\pi_{k<n}(\mathbb{S}^n)$	✔✔
long-exact-sequence of homotopy groups	✔✔
total space of Hopf fibration is $\mathbb{S}^3$	✔
$\pi_2(\mathbb{S}^2)$	✔✔
$\pi_3(\mathbb{S}^2)$	✔
$\pi_n(\mathbb{S}^n)$	✔✔
$\pi_4(\mathbb{S}^3)$	✔
Freudenthal suspension theorem	✔✔
Blakers-Massey theorem	✔✔
Eilenberg–Mac Lane spaces $K(G,n)$	✔✔
van Kampen theorem	✔✔
covering spaces	✔✔
Whitehead's principle for n-types	✔✔

Table 8.2: Theorems from homotopy theory proved by hand (✔) and by computer (✔✔).

The theorems described in this chapter are standard results in classical homotopy theory; many are described by [Hat02]. In these notes, we review the development of the new synthetic proofs of them in homotopy type theory. Table 8.2 lists the homotopy-theoretic theorems that have been proven in homotopy type theory, and whether they have been computer-checked. Almost all of these results were developed during the spring term at IAS in 2013, as part of a significant collaborative effort. Many people contributed to these results, for example by being the principal author of a proof, by suggesting problems to work on, by participating in many discussions and seminars about these problems, or by giving feedback on results. The following people were the principal authors of the first homotopy type theory proofs of the above theorems. Unless indicated otherwise, for the theorems that have been computer-checked, the principal authors were also the first ones to formalize the proof using a computer proof assistant.

- Shulman gave the homotopy-theoretic calculation of $\pi_1(\mathbb{S}^1)$. Licata later discovered the encode-decode proof and the encode-decode method.
- Brunerie calculated $\pi_{k<n}(\mathbb{S}^n)$. Licata later gave an encode-decode version.
- Voevodsky constructed the long exact sequence of homotopy groups.

- Lumsdaine constructed the Hopf fibration. Brunerie proved that its total space is $\mathbb{S}^3$, thereby calculating $\pi_2(\mathbb{S}^2)$ and $\pi_3(\mathbb{S}^3)$.
- Licata and Brunerie gave a direct calculation of $\pi_n(\mathbb{S}^n)$.
- Lumsdaine proved the Freudenthal suspension theorem; Licata and Lumsdaine formalized this proof.
- Lumsdaine, Finster, and Licata proved the Blakers-Massey theorem; Lumsdaine, Brunerie, Licata, and Hou formalized it.
- Licata gave an encode-decode calculation of $\pi_2(\mathbb{S}^2)$, and a calculation of $\pi_n(\mathbb{S}^n)$ using the Freudenthal suspension theorem; using similar techniques, he constructed $K(G, n)$.
- Shulman proved the van Kampen theorem; Hou formalized this proof.
- Licata proved Whitehead's theorem for n-types.
- Brunerie calculated $\pi_4(\mathbb{S}^3)$.
- Hou established the theory of covering spaces and formalized it.

The interplay between homotopy theory and type theory was crucial to the development of these results. For example, the first proof that $\pi_1(\mathbb{S}^1) = \mathbb{Z}$ followed a classical homotopy theoretic one. A type-theoretic analysis of this proof resulted in the development of the encode-decode method. The first calculation of $\pi_2(\mathbb{S}^2)$ also followed classical methods, but this led quickly to an encode-decode proof of the result. The encode-decode calculation generalized to $\pi_n(\mathbb{S}^n)$, which in turn led to the proof of the Freudenthal suspension theorem, by combining an encode-decode argument with classical homotopy-theoretic reasoning about connectedness, which in turn led to the Blakers-Massey theorem and Eilenberg-Mac Lane spaces. The rapid development of this series of results illustrates the promise of our new understanding of the connections between these two subjects.

Exercises

Exercise 8.1. Continuing from Example 8.7.7, prove that if A has a point $a : A$, then we can identify $\pi_1(\Sigma A)$ with the group presented by $\|A\|_0$ as generators with the relation $|a|_0 = e$. Then show that if we assume excluded middle, this is also the free group on $\|A\|_0 \setminus \{|a|_0\}$.

Exercise 8.2. Again continuing from Example 8.7.7, but this time without assuming A to be pointed, show that we can identify $\pi_1(\Sigma A)$ with the group presented by generators $\|A\|_0 \times \|A\|_0$ and relations

$$(a, b) = (b, a)^{-1}, \qquad (a, c) = (a, b) \cdot (b, c), \qquad \text{and} \qquad (a, a) = e.$$

Chapter 9

Category theory

Of the branches of mathematics, category theory is one which perhaps fits the least comfortably in set theoretic foundations. One problem is that most of category theory is invariant under weaker notions of "sameness" than equality, such as isomorphism in a category or equivalence of categories, in a way which set theory fails to capture. But this is the same sort of problem that the univalence axiom solves for types, by identifying equality with equivalence. Thus, in univalent foundations it makes sense to consider a notion of "category" in which equality of objects is identified with isomorphism in a similar way.

Ignoring size issues, in set-based mathematics a category consists of a *set* A_0 of objects and, for each $x, y \in A_0$, a *set* $\hom_A(x, y)$ of morphisms. Under Univalent Foundations, a "naive" definition of category would simply mimic this with a *type* of objects and *types* of morphisms. If we allowed these types to contain arbitrary higher homotopy, then we ought to impose higher coherence conditions, leading to some notion of $(\infty, 1)$-category, but at present our goal is more modest. We consider only 1-categories, and therefore we restrict the types $\hom_A(x, y)$ to be sets, i.e. 0-types. If we impose no further conditions, we will call this notion a *precategory*.

If we add the requirement that the type A_0 of objects is a set, then we end up with a definition that behaves much like the traditional set-theoretic one. Following Toby Bartels, we call this notion a *strict category*. But we can also require a generalized version of the univalence axiom, identifying $(x =_{A_0} y)$ with the type $\mathsf{iso}(x, y)$ of isomorphisms from x to y. Since we regard this as usually the "correct" definition, we will call it simply a *category*.

A good example of the difference between the three notions of category is provided by the statement "every fully faithful and essentially surjective functor is an equivalence of categories", which in classical set-based category theory is equivalent to the axiom of choice.

(i) For strict categories, this is still equivalent to to the axiom of choice.
(ii) For precategories, there is no consistent axiom of choice which can make it true.
(iii) For categories, it is provable *without* any axiom of choice.

We will prove the latter statement in this chapter, as well as other pleasant properties of categories, e.g. that equivalent categories are equal (as elements of the type of categories). We will also describe a universal way of "saturating" a precategory A into a category $\widehat{A}$, which we call its *Rezk completion*, although it could also reasonably be called the *stack completion* (see the Notes).

The Rezk completion also sheds further light on the notion of equivalence of categories. For instance, the functor $A \to \widehat{A}$ is always fully faithful and essentially surjective, hence a "weak equivalence". It follows that a precategory is a category exactly when it "sees" all fully faithful and essentially surjective functors as equivalences; thus our notion of "category" is already inherent in the notion of "fully faithful and essentially surjective functor".

We assume the reader has some basic familiarity with classical category theory. Recall that whenever we write $\mathcal{U}$ it denotes some universe of types, but perhaps a different one at different times; everything we say remains true for any consistent choice of universe levels. We will use the basic notions of homotopy type theory from Chapters 1 and 2 and the propositional truncation from Chapter 3, but not much else from Part I, except that our second construction of the Rezk completion will use a higher inductive type.

9.1 Categories and precategories

In classical mathematics, there are many equivalent definitions of a category. In our case, since we have dependent types, it is natural to choose the arrows to be a type family indexed by the objects. This matches the way hom-types are always used in category theory: we never even consider comparing two arrows unless we know their sources and targets agree. Furthermore, it seems clear that for a theory of 1-categories, the hom-types should all be sets. This leads us to the following definition.

Definition 9.1.1. A **precategory** A consists of the following.

(i) A type A_0 of **objects**. We write $a : A$ for $a : A_0$.
(ii) For each $a, b : A$, a set $\hom_A(a, b)$ of **arrows** or **morphisms**.
(iii) For each $a : A$, a morphism $1_a : \hom_A(a, a)$.
(iv) For each $a, b, c : A$, a function

$$\hom_A(b, c) \to \hom_A(a, b) \to \hom_A(a, c)$$

denoted infix by $g \mapsto f \mapsto g \circ f$, or sometimes simply by gf.
(v) For each $a, b : A$ and $f : \hom_A(a, b)$, we have $f = 1_b \circ f$ and $f = f \circ 1_a$.
(vi) For each $a, b, c, d : A$ and $f : \hom_A(a, b)$, $g : \hom_A(b, c)$, $h : \hom_A(c, d)$, we have $h \circ (g \circ f) = (h \circ g) \circ f$.

The problem with the notion of precategory is that for objects $a, b : A$, we have two possibly-different notions of "sameness". On the one hand, we have $a =_{A_0} b$. But on the other hand, there is the standard categorical notion of *isomorphism*.

Definition 9.1.2. A morphism $f : \hom_A(a, b)$ is an **isomorphism** if there is a morphism $g : \hom_A(b, a)$ such that $g \circ f = 1_a$ and $f \circ g = 1_b$. We write $a \cong b$ for the type of such isomorphisms.

Lemma 9.1.3. *For any $f : \hom_A(a, b)$, the type "f is an isomorphism" is a mere proposition. Therefore, for any $a, b : A$ the type $a \cong b$ is a set.*

Proof. Suppose given $g : \hom_A(b,a)$ and $\eta : (1_a = g \circ f)$ and $\epsilon : (f \circ g = 1_b)$, and similarly g', η', and ϵ'. We must show $(g,\eta,\epsilon) = (g',\eta',\epsilon')$. But since all hom-sets are sets, their identity types are mere propositions, so it suffices to show $g = g'$. For this we have

$$g' = 1_a \circ g' = (g \circ f) \circ g' = g \circ (f \circ g') = g \circ 1_b = g$$

using η and ϵ'. □

If $f : a \cong b$, then we write f^{-1} for its inverse, which by Lemma 9.1.3 is uniquely determined.

The only relationship between these two notions of sameness that we have in a precategory is the following.

Lemma 9.1.4 (idtoiso)**.** *If A is a precategory and $a, b : A$, then*

$$(a = b) \to (a \cong b).$$

Proof. By induction on identity, we may assume a and b are the same. But then we have $1_a : \hom_A(a,a)$, which is clearly an isomorphism. □

Evidently, this situation is analogous to the issue that motivated us to introduce the univalence axiom. In fact, we have the following:

Example 9.1.5. There is a precategory $\mathcal{S}et$, whose type of objects is Set, and with $\hom_{\mathcal{S}et}(A,B) :\equiv (A \to B)$. The identity morphisms are identity functions and the composition is function composition. For this precategory, Lemma 9.1.4 is equal to (the restriction to sets of) the identity-to-equivalence map from the chapter on univalence.

Thus, it is natural to make the following definition.

Definition 9.1.6. A **category** is a precategory such that for all $a, b : A$, the function $\mathsf{idtoiso}_{a,b}$ from Lemma 9.1.4 is an equivalence.

In particular, in a category, if $a \cong b$, then $a = b$.

Example 9.1.7. The univalence axiom implies immediately that $\mathcal{S}et$ is a category. One can also show, using univalence, that any precategory of set-level structures such as groups, rings, topological spaces, etc. is a category.

We also note the following.

Lemma 9.1.8. *In a category, the type of objects is a 1-type.*

Proof. It suffices to show that for any $a, b : A$, the type $a = b$ is a set. But $a = b$ is equivalent to $a \cong b$, which is a set. □

We write isotoid for the inverse $(a \cong b) \to (a = b)$ of the map idtoiso from Lemma 9.1.4. The following relationship between the two is important.

Lemma 9.1.9. *For $p : a = a'$ and $q : b = b'$ and $f : \hom_A(a,b)$, we have*

$$(p,q)_*(f) = \mathsf{idtoiso}(q) \circ f \circ \mathsf{idtoiso}(p)^{-1} \tag{9.1.10}$$

Proof. By induction, we may assume p and q are refl_a and refl_b respectively. Then the left-hand side of (9.1.10) is simply f. But by definition, $\mathsf{idtoiso}(\mathsf{refl}_a)$ is 1_a, and $\mathsf{idtoiso}(\mathsf{refl}_b)$ is 1_b, so the right-hand side of (9.1.10) is $1_b \circ f \circ 1_a$, which is equal to f. □

Similarly, we can show

$$\mathsf{idtoiso}(p^{-1}) = (\mathsf{idtoiso}(p))^{-1} \tag{9.1.11}$$

$$\mathsf{idtoiso}(p \cdot q) = \mathsf{idtoiso}(q) \circ \mathsf{idtoiso}(p) \tag{9.1.12}$$

$$\mathsf{isotoid}(f \circ e) = \mathsf{isotoid}(e) \cdot \mathsf{isotoid}(f) \tag{9.1.13}$$

and so on.

Example 9.1.14. A precategory in which each set $\hom_A(a, b)$ is a mere proposition is equivalently a type A_0 equipped with a mere relation "$\leq$" that is reflexive ($a \leq a$) and transitive (if $a \leq b$ and $b \leq c$, then $a \leq c$). We call this a **preorder**.

In a preorder, a proof $f : a \leq b$ is an isomorphism just when there exists some proof $g : b \leq a$. Thus, $a \cong b$ is the mere proposition that $a \leq b$ and $b \leq a$. Therefore, a preorder A is a category just when (1) each type $a = b$ is a mere proposition, and (2) for any $a, b : A_0$ there exists a function $(a \cong b) \to (a = b)$. In other words, A_0 must be a set, and $\leq$ must be antisymmetric (if $a \leq b$ and $b \leq a$, then $a = b$). We call this a **(partial) order** or a **poset**.

Example 9.1.15. If A is a category, then A_0 is a set if and only if for any $a, b : A_0$, the type $a \cong b$ is a mere proposition. This is equivalent to saying that every isomorphism in A is an identity; thus it is rather stronger than the classical notion of "skeletal" category. Categories of this sort are sometimes called **gaunt** [BSP11]. There is not really any notion of "skeletality" for our categories, unless one considers Definition 9.1.6 itself to be such.

Example 9.1.16. For any 1-type X, there is a category with X as its type of objects and with $\hom(x, y) :\equiv (x = y)$. If X is a set, we call this the **discrete** category on X. In general, we call it a **groupoid** (see Exercise 9.6).

Example 9.1.17. For *any* type X, there is a precategory with X as its type of objects and with $\hom(x, y) :\equiv \|x = y\|_0$. The composition operation

$$\|y = z\|_0 \to \|x = y\|_0 \to \|x = z\|_0$$

is defined by induction on truncation from concatenation $(y = z) \to (x = y) \to (x = z)$. We call this the **fundamental pregroupoid** of X. (In fact, we have met it already in §8.7; see also Exercise 9.11.)

Example 9.1.18. There is a precategory whose type of objects is $\mathcal{U}$ and with $\hom(X, Y) :\equiv \|X \to Y\|_0$, and composition defined by induction on truncation from ordinary composition $(Y \to Z) \to (X \to Y) \to (X \to Z)$. We call this the **homotopy precategory of types**.

Example 9.1.19. Let $\mathcal{R}el$ be the following precategory:

- Its objects are sets.
- $\hom_{\mathcal{R}el}(X, Y) = X \to Y \to \mathsf{Prop}$.
- For a set X, we have $1_X(x, x') :\equiv (x = x')$.

- For $R : \hom_{\mathcal{R}el}(X, Y)$ and $S : \hom_{\mathcal{R}el}(Y, Z)$, their composite is defined by

$$(S \circ R)(x, z) :\equiv \left\| \sum_{y:Y} R(x, y) \times S(y, z) \right\|.$$

Suppose $R : \hom_{\mathcal{R}el}(X, Y)$ is an isomorphism, with inverse S. We observe the following.

(i) If $R(x, y)$ and $S(y', x)$, then $(R \circ S)(y', y)$, and hence $y' = y$. Similarly, if $R(x, y)$ and $S(y, x')$, then $x = x'$.

(ii) For any x, we have $x = x$, hence $(S \circ R)(x, x)$. Thus, there merely exists a $y : Y$ such that $R(x, y)$ and $S(y, x)$.

(iii) Suppose $R(x, y)$. By (ii), there merely exists a y' with $R(x, y')$ and $S(y', x)$. But then by (i), merely $y' = y$, and hence $y' = y$ since Y is a set. Therefore, by transporting $S(y', x)$ along this equality, we have $S(y, x)$. In conclusion, $R(x, y) \to S(y, x)$. Similarly, $S(y, x) \to R(x, y)$.

(iv) If $R(x, y)$ and $R(x, y')$, then by (iii), $S(y', x)$, so that by (i), $y = y'$. Thus, for any x there is at most one y such that $R(x, y)$. And by (ii), there merely exists such a y, hence there exists such a y.

In conclusion, if $R : \hom_{\mathcal{R}el}(X, Y)$ is an isomorphism, then for each $x : X$ there is exactly one $y : Y$ such that $R(x, y)$, and dually. Thus, there is a function $f : X \to Y$ sending each x to this y, which is an equivalence; hence $X = Y$. With a little more work, we conclude that $\mathcal{R}el$ is a category.

In a textbook on category theory written for readers who had grown up with informal homotopy type theory, we would probably say very little about precategories from now on, restricting ourselves to the case of categories. However, since this book has a somewhat different purpose and audience, we will develop many concepts for precategories as well as categories, in order to emphasize how much better-behaved categories are, as compared both to precategories and to ordinary categories in classical mathematics.

We will also see in §§9.6–9.7 that in slightly more exotic contexts, there are uses for certain kinds of precategories other than categories, each of which "fixes" the equality of objects in different ways. This emphasizes the "pre"-ness of precategories: they are the raw material out of which multiple important categorical structures can be defined.

9.2 Functors and transformations

The following definitions are fairly obvious, and need no modification.

Definition 9.2.1. Let A and B be precategories. A **functor** $F : A \to B$ consists of

(i) A function $F_0 : A_0 \to B_0$, generally also denoted F.

(ii) For each $a, b : A$, a function $F_{a,b} : \hom_A(a, b) \to \hom_B(Fa, Fb)$, generally also denoted F.

(iii) For each $a : A$, we have $F(1_a) = 1_{Fa}$.

(iv) For each $a, b, c : A$ and $f : \hom_A(a, b)$ and $g : \hom_B(b, c)$, we have

$$F(g \circ f) = Fg \circ Ff.$$

Note that by induction on identity, a functor also preserves idtoiso.

Definition 9.2.2. For functors $F, G : A \to B$, a **natural transformation** $\gamma : F \to G$ consists of

(i) For each $a : A$, a morphism $\gamma_a : \hom_B(Fa, Ga)$ (the "components").
(ii) For each $a, b : A$ and $f : \hom_A(a, b)$, we have $Gf \circ \gamma_a = \gamma_b \circ Ff$ (the "naturality axiom").

Since each type $\hom_B(Fa, Gb)$ is a set, its identity type is a mere proposition. Thus, the naturality axiom is a mere proposition, so identity of natural transformations is determined by identity of their components. In particular, for any F and G, the type of natural transformations from F to G is again a set.

Similarly, identity of functors is determined by identity of the functions $A_0 \to B_0$ and (transported along this) of the corresponding functions on hom-sets.

Definition 9.2.3. For precategories A, B, there is a precategory B^A defined by

- $(B^A)_0$ is the type of functors from A to B.
- $\hom_{B^A}(F, G)$ is the type of natural transformations from F to G.

Proof. We define $(1_F)_a :\equiv 1_{Fa}$. Naturality follows by the unit axioms of a precategory. For $\gamma : F \to G$ and $\delta : G \to H$, we define $(\delta \circ \gamma)_a :\equiv \delta_a \circ \gamma_a$. Naturality follows by associativity. Similarly, the unit and associativity laws for B^A follow from those for B. □

Lemma 9.2.4. *A natural transformation $\gamma : F \to G$ is an isomorphism in B^A if and only if each γ_a is an isomorphism in B.*

Proof. If γ is an isomorphism, then we have $\delta : G \to F$ that is its inverse. By definition of composition in B^A, $(\delta\gamma)_a \equiv \delta_a\gamma_a$ and similarly. Thus, $\delta\gamma = 1_F$ and $\gamma\delta = 1_G$ imply $\delta_a\gamma_a = 1_{Fa}$ and $\gamma_a\delta_a = 1_{Ga}$, so γ_a is an isomorphism.

Conversely, suppose each γ_a is an isomorphism, with inverse called δ_a, say. We define a natural transformation $\delta : G \to F$ with components δ_a; for the naturality axiom we have

$$Ff \circ \delta_a = \delta_b \circ \gamma_b \circ Ff \circ \delta_a = \delta_b \circ Gf \circ \gamma_a \circ \delta_a = \delta_b \circ Gf.$$

Now since composition and identity of natural transformations is determined on their components, we have $\gamma\delta = 1_G$ and $\delta\gamma = 1_F$. □

The following result is fundamental.

Theorem 9.2.5. *If A is a precategory and B is a category, then B^A is a category.*

Proof. Let $F, G : A \to B$; we must show that $\mathsf{idtoiso} : (F = G) \to (F \cong G)$ is an equivalence.

To give an inverse to it, suppose $\gamma : F \cong G$ is a natural isomorphism. Then for any $a : A$, we have an isomorphism $\gamma_a : Fa \cong Ga$, hence an identity $\mathsf{isotoid}(\gamma_a) : Fa = Ga$. By function extensionality, we have an identity $\bar{\gamma} : F_0 =_{(A_0 \to B_0)} G_0$.

Now since the last two axioms of a functor are mere propositions, to show that $F = G$ it will suffice to show that for any $a, b : A$, the functions

$$F_{a,b} : \hom_A(a,b) \to \hom_B(Fa, Fb) \qquad \text{and}$$
$$G_{a,b} : \hom_A(a,b) \to \hom_B(Ga, Gb)$$

become equal when transported along $\bar{\gamma}$. By computation for function extensionality, when applied to a, $\bar{\gamma}$ becomes equal to $\mathsf{isotoid}(\gamma_a)$. But by Lemma 9.1.9, transporting $Ff : \hom_B(Fa, Fb)$ along $\mathsf{isotoid}(\gamma_a)$ and $\mathsf{isotoid}(\gamma_b)$ is equal to the composite $\gamma_b \circ Ff \circ (\gamma_a)^{-1}$, which by naturality of γ is equal to Gf.

This completes the definition of a function $(F \cong G) \to (F = G)$. Now consider the composite

$$(F = G) \to (F \cong G) \to (F = G).$$

Since hom-sets are sets, their identity types are mere propositions, so to show that two identities $p, q : F = G$ are equal, it suffices to show that $p =_{F_0 = G_0} q$. But in the definition of $\bar{\gamma}$, if γ were of the form $\mathsf{idtoiso}(p)$, then γ_a would be equal to $\mathsf{idtoiso}(p_a)$ (this can easily be proved by induction on p). Thus, $\mathsf{isotoid}(\gamma_a)$ would be equal to p_a, and so by function extensionality we would have $\bar{\gamma} = p$, which is what we need.

Finally, consider the composite

$$(F \cong G) \to (F = G) \to (F \cong G).$$

Since identity of natural transformations can be tested componentwise, it suffices to show that for each a we have $\mathsf{idtoiso}(\bar{\gamma})_a = \gamma_a$. But as observed above, we have $\mathsf{idtoiso}(\bar{\gamma})_a = \mathsf{idtoiso}((\bar{\gamma})_a)$, while $(\bar{\gamma})_a = \mathsf{isotoid}(\gamma_a)$ by computation for function extensionality. Since isotoid and idtoiso are inverses, we have $\mathsf{idtoiso}(\bar{\gamma})_a = \gamma_a$ as desired. □

In particular, naturally isomorphic functors between categories (as opposed to precategories) are equal.

We now define all the usual ways to compose functors and natural transformations.

Definition 9.2.6. For functors $F : A \to B$ and $G : B \to C$, their composite $G \circ F : A \to C$ is given by

- The composite $(G_0 \circ F_0) : A_0 \to C_0$
- For each $a, b : A$, the composite

$$(G_{Fa,Fb} \circ F_{a,b}) : \hom_A(a,b) \to \hom_C(GFa, GFb).$$

It is easy to check the axioms.

Definition 9.2.7. For functors $F : A \to B$ and $G, H : B \to C$ and a natural transformation $\gamma : G \to H$, the composite $(\gamma F) : GF \to HF$ is given by

- For each $a : A$, the component γ_{Fa}.

Naturality is easy to check. Similarly, for γ as above and $K : C \to D$, the composite $(K\gamma) : KG \to KH$ is given by

- For each $b : B$, the component $K(\gamma_b)$.

Lemma 9.2.8. *For functors $F, G : A \to B$ and $H, K : B \to C$ and natural transformations $\gamma : F \to G$ and $\delta : H \to K$, we have*

$$(\delta G)(H\gamma) = (K\gamma)(\delta F).$$

Proof. It suffices to check componentwise: at $a : A$ we have

$$\begin{aligned}((\delta G)(H\gamma))_a &\equiv (\delta G)_a (H\gamma)_a \\ &\equiv \delta_{Ga} \circ H(\gamma_a) \\ &= K(\gamma_a) \circ \delta_{Fa} \qquad \text{(by naturality of } \delta) \\ &\equiv (K\gamma)_a \circ (\delta F)_a \\ &\equiv ((K\gamma)(\delta F))_a.\end{aligned}$$

□

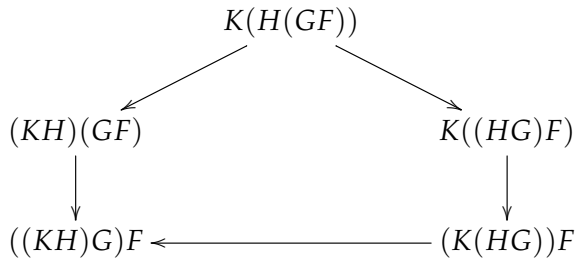
Classically, one defines the "horizontal composite" of $\gamma : F \to G$ and $\delta : H \to K$ to be the common value of $(\delta G)(H\gamma)$ and $(K\gamma)(\delta F)$. We will refrain from doing this, because while equal, these two transformations are not *definitionally* equal. This also has the consequence that we can use the symbol $\circ$ (or juxtaposition) for all kinds of composition unambiguously: there is only one way to compose two natural transformations (as opposed to composing a natural transformation with a functor on either side).

Lemma 9.2.9. *Composition of functors is associative: $H(GF) = (HG)F$.*

Proof. Since composition of functions is associative, this follows immediately for the actions on objects and on homs. And since hom-sets are sets, the rest of the data is automatic. □

The equality in Lemma 9.2.9 is likewise not definitional. (Composition of functions is definitionally associative, but the axioms that go into a functor must also be composed, and this breaks definitional associativity.) For this reason, we need also to know about *coherence* for associativity.

Lemma 9.2.10. *Lemma 9.2.9 is coherent, i.e. the following pentagon of equalities commutes:*



Proof. As in Lemma 9.2.9, this is evident for the actions on objects, and the rest is automatic. □

We will henceforth abuse notation by writing $H \circ G \circ F$ or HGF for either $H(GF)$ or $(HG)F$, transporting along Lemma 9.2.9 whenever necessary. We have a similar coherence result for units.

Lemma 9.2.11. *For a functor $F : A \to B$, we have equalities $(1_B \circ F) = F$ and $(F \circ 1_A) = F$, such that given also $G : B \to C$, the following triangle of equalities commutes.*

$$\begin{array}{ccc} G \circ (1_B \circ F) & \longrightarrow & (G \circ 1_B) \circ F \\ & \searrow \quad \swarrow & \\ & G \circ F. & \end{array}$$

See Exercises 9.4 and 9.5 for further development of these ideas.

9.3 Adjunctions

Definition 9.3.1. A functor $F : A \to B$ is a **left adjoint** if there exists

- A functor $G : B \to A$.
- A natural transformation $\eta : 1_A \to GF$.
- A natural transformation $\epsilon : FG \to 1_B$.
- $(\epsilon F)(F\eta) = 1_F$.
- $(G\epsilon)(\eta G) = 1_G$.

Lemma 9.3.2. *If A is a category (but B may be only a precategory), then the type "F is a left adjoint" is a mere proposition.*

Proof. Suppose given (G, η, ϵ) with the triangle identities and also (G', η', ϵ'). Define $\gamma : G \to G'$ to be $(G'\epsilon)(\eta' G)$, and $\delta : G' \to G$ to be $(G\epsilon')(\eta G')$. Then

$$\begin{aligned} \delta\gamma &= (G\epsilon')(\eta G')(G'\epsilon)(\eta' G) \\ &= (G\epsilon')(GFG'\epsilon)(\eta G'FG)(\eta' G) \\ &= (G\epsilon)(G\epsilon' FG)(GF\eta' G)(\eta G) \\ &= (G\epsilon)(\eta G) \\ &= 1_G \end{aligned}$$

using Lemma 9.2.8 and the triangle identities. Similarly, we show $\gamma\delta = 1_{G'}$, so γ is a natural isomorphism $G \cong G'$. By Theorem 9.2.5, we have an identity $G = G'$.

Now we need to know that when η and ϵ are transported along this identity, they become equal to η' and ϵ'. By Lemma 9.1.9, this transport is given by composing with γ or δ as appropriate. For η, this yields

$$(G'\epsilon F)(\eta' GF)\eta = (G'\epsilon F)(G'F\eta)\eta' = \eta'$$

using Lemma 9.2.8 and the triangle identity. The case of ϵ is similar. Finally, the triangle identities transport correctly automatically, since hom-sets are sets. □

In §9.5 we will give another proof of Lemma 9.3.2.

9.4 Equivalences

Definition 9.4.1. A functor $F : A \to B$ is an **equivalence of (pre)categories** if it is a left adjoint for which η and ϵ are isomorphisms. We write $A \simeq B$ for the type of equivalences of categories from A to B.

By Lemmas 9.1.3 and 9.3.2, if A is a category, then the type "F is an equivalence of precategories" is a mere proposition.

Lemma 9.4.2. *If for $F : A \to B$ there exists $G : B \to A$ and isomorphisms $GF \cong 1_A$ and $FG \cong 1_B$, then F is an equivalence of precategories.*

Proof. Just like the proof of Theorem 4.2.3 for equivalences of types. □

Definition 9.4.3. We say a functor $F : A \to B$ is **faithful** if for all $a, b : A$, the function

$$F_{a,b} : \hom_A(a,b) \to \hom_B(Fa, Fb)$$

is injective, and **full** if for all $a, b : A$ this function is surjective. If it is both (hence each $F_{a,b}$ is an equivalence) we say F is **fully faithful**.

Definition 9.4.4. We say a functor $F : A \to B$ is **split essentially surjective** if for all $b : B$ there exists an $a : A$ such that $Fa \cong b$.

Lemma 9.4.5. *For any precategories A and B and functor $F : A \to B$, the following types are equivalent.*

(i) F is an equivalence of precategories.
(ii) F is fully faithful and split essentially surjective.

Proof. Suppose F is an equivalence of precategories, with G, η, ϵ specified. Then we have the function

$$\begin{aligned} \hom_B(Fa, Fb) &\to \hom_A(a,b), \\ g &\mapsto {\eta_b}^{-1} \circ G(g) \circ \eta_a. \end{aligned}$$

For $f : \hom_A(a,b)$, we have

$${\eta_b}^{-1} \circ G(F(f)) \circ \eta_a = {\eta_b}^{-1} \circ \eta_b \circ f = f$$

while for $g : \hom_B(Fa, Fb)$ we have

$$\begin{aligned} F({\eta_b}^{-1} \circ G(g) \circ \eta_a) &= F({\eta_b}^{-1}) \circ F(G(g)) \circ F(\eta_a) \\ &= \epsilon_{Fb} \circ F(G(g)) \circ F(\eta_a) \\ &= g \circ \epsilon_{Fa} \circ F(\eta_a) \\ &= g \end{aligned}$$

using naturality of ϵ, and the triangle identities twice. Thus, $F_{a,b}$ is an equivalence, so F is fully faithful. Finally, for any $b : B$, we have $Gb : A$ and $\epsilon_b : FGb \cong b$.

On the other hand, suppose F is fully faithful and split essentially surjective. Define $G_0 : B_0 \to A_0$ by sending $b : B$ to the $a : A$ given by the specified essential splitting, and write ϵ_b for the likewise specified isomorphism $FGb \cong b$.

Now for any $g : \hom_B(b, b')$, define $G(g) : \hom_A(Gb, Gb')$ to be the unique morphism such that $F(G(g)) = (\epsilon_{b'})^{-1} \circ g \circ \epsilon_b$ (which exists since F is fully faithful). Finally, for $a : A$ define $\eta_a : \hom_A(a, GFa)$ to be the unique morphism such that $F\eta_a = {\epsilon_{Fa}}^{-1}$. It is easy to verify that G is a functor and that (G, η, ϵ) exhibit F as an equivalence of precategories.

Now consider the composite (i)→(ii)→(i). We clearly recover the same function $G_0 : B_0 \to A_0$. For the action of G on hom-sets, we must show that for $g : \hom_B(b, b')$, $G(g)$ is the (necessarily unique) morphism such that $F(G(g)) = (\epsilon_{b'})^{-1} \circ g \circ \epsilon_b$. But this equation holds by the assumed naturality of ϵ. We also clearly recover ϵ, while η is uniquely characterized by $F\eta_a = {\epsilon_{Fa}}^{-1}$ (which is one of the triangle identities assumed to hold in the structure of an equivalence of precategories). Thus, this composite is equal to the identity.

Finally, consider the other composite (ii)→(i)→(ii). Since being fully faithful is a mere proposition, it suffices to observe that we recover, for each $b : B$, the same $a : A$ and isomorphism $Fa \cong b$. But this is clear, since we used this function and isomorphism to define G_0 and ϵ in (i), which in turn are precisely what we used to recover (ii) again. Thus, the composites in both directions are equal to identities, hence we have an equivalence $(i) \simeq (ii)$. □

However, if B is not a category, then neither type in Lemma 9.4.5 may necessarily be a mere proposition. This suggests considering as well the following notions.

Definition 9.4.6. A functor $F : A \to B$ is **essentially surjective** if for all $b : B$, there *merely* exists an $a : A$ such that $Fa \cong b$. We say F is a **weak equivalence** if it is fully faithful and essentially surjective.

Being a weak equivalence is *always* a mere proposition. For categories, however, there is no difference between equivalences and weak ones.

Lemma 9.4.7. *If $F : A \to B$ is fully faithful and A is a category, then for any $b : B$ the type $\sum_{(a:A)}(Fa \cong b)$ is a mere proposition. Hence a functor between categories is an equivalence if and only if it is a weak equivalence.*

Proof. Suppose given (a, f) and (a', f') in $\sum_{(a:A)}(Fa \cong b)$. Then ${f'}^{-1} \circ f$ is an isomorphism $Fa \cong Fa'$. Since F is fully faithful, we have $g : a \cong a'$ with $Fg = {f'}^{-1} \circ f$. And since A is a category, we have $p : a = a'$ with $\mathsf{idtoiso}(p) = g$. Now $Fg = {f'}^{-1} \circ f$ implies $((F_0)(p))_*(f) = f'$, hence (by the characterization of equalities in dependent sums) $(a, f) = (a', f')$.

Thus, for fully faithful functors whose domain is a category, essential surjectivity is equivalent to split essential surjectivity, and so being a weak equivalence is equivalent to being an equivalence. □

This is an important advantage of our category theory over set-based approaches. With a purely set-based definition of category, the statement "every fully faithful and essentially surjective functor is an equivalence of categories" is equivalent to the mere axiom of choice (that is, the strong one, not the trivial one). Here we have it for free, as a category-theoretic version

of the function comprehension principle. (In fact, this property characterizes categories among precategories; see §9.9.)

On the other hand, the following characterization of equivalences of categories is perhaps even more useful.

Definition 9.4.8. A functor $F : A \to B$ is an **isomorphism of (pre)categories** if F is fully faithful and $F_0 : A_0 \to B_0$ is an equivalence of types.

Note that being an isomorphism of precategories is always a mere property. Let $A \cong B$ denote the type of isomorphisms of (pre)categories from A to B.

Lemma 9.4.9. *For precategories A and B and $F : A \to B$, the following are equivalent.*

(i) F is an isomorphism of precategories.

(ii) There exist $G : B \to A$ and $\eta : 1_A = GF$ and $\epsilon : FG = 1_B$ such that

$$(\lambda H.\, FH)(\eta) = (\lambda K.\, KF)\Big(\epsilon^{-1}\Big). \tag{9.4.10}$$

(iii) There merely exist $G : B \to A$ and $\eta : 1_A = GF$ and $\epsilon : FG = 1_B$.

Note that if B_0 is not a 1-type, then (9.4.10) may not be a mere proposition.

Proof. First note that since hom-sets are sets, equalities between equalities of functors are uniquely determined by their object-parts. Thus, by function extensionality, (9.4.10) is equivalent to

$$(F_0)(\eta_0)_a = (\epsilon_0)^{-1}{}_{F_0 a}. \tag{9.4.11}$$

for all $a : A_0$. Note that this is precisely the triangle identity for G_0, η_0, and ϵ_0 to be a proof that F_0 is an equivalence of types.

Now suppose (i). Let $G_0 : B_0 \to A_0$ be the inverse of F_0, with $\eta_0 : \mathsf{id}_{A_0} = G_0 F_0$ and $\epsilon_0 : F_0 G_0 = \mathsf{id}_{B_0}$ satisfying the triangle identity, which is precisely (9.4.11). Now define $G_{b,b'} : \hom_B(b, b') \to \hom_A(G_0 b, G_0 b')$ by

$$G_{b,b'}(g) :\equiv (F_{G_0 b, G_0 b'})^{-1}\Big(\mathsf{idtoiso}((\epsilon_0)^{-1}{}_{b'}) \circ g \circ \mathsf{idtoiso}((\epsilon_0)_b)\Big)$$

(using the assumption that F is fully faithful). Since idtoiso takes opposites to inverses and concatenation to composition, and F is a functor, it follows that G is a functor.

By definition, we have $(GF)_0 \equiv G_0 F_0$, which is equal to id_{A_0} by η_0. To obtain $1_A = GF$, we need to show that when transported along η_0, the identity function of $\hom_A(a, a')$ becomes equal to the composite $G_{Fa,Fa'} \circ F_{a,a'}$. In other words, for any $f : \hom_A(a, a')$ we must have

$$\mathsf{idtoiso}((\eta_0)_{a'}) \circ f \circ \mathsf{idtoiso}((\eta_0)^{-1}{}_a) = (F_{GFa,GFa'})^{-1}\Big(\mathsf{idtoiso}((\epsilon_0)^{-1}{}_{Fa'}) \circ F_{a,a'}(f) \circ \mathsf{idtoiso}((\epsilon_0)_{Fa})\Big).$$

But this is equivalent to

$$(F_{GFa,GFa'})\Big(\mathsf{idtoiso}((\eta_0)_{a'}) \circ f \circ \mathsf{idtoiso}({(\eta_0)^{-1}}_a)\Big) \\ = \mathsf{idtoiso}({(\epsilon_0)^{-1}}_{Fa'}) \circ F_{a,a'}(f) \circ \mathsf{idtoiso}((\epsilon_0)_{Fa}).$$

which follows from functoriality of F, the fact that F preserves idtoiso, and (9.4.11). Thus we have $\eta : 1_A = GF$.

On the other side, we have $(FG)_0 \equiv F_0 G_0$, which is equal to id_{B_0} by ϵ_0. To obtain $FG = 1_B$, we need to show that when transported along ϵ_0, the identity function of $\hom_B(b, b')$ becomes equal to the composite $F_{Gb,Gb'} \circ G_{b,b'}$. That is, for any $g : \hom_B(b, b')$ we must have

$$F_{Gb,Gb'}\Big((F_{Gb,Gb'})^{-1}\Big(\mathsf{idtoiso}({(\epsilon_0)^{-1}}_{b'}) \circ g \circ \mathsf{idtoiso}((\epsilon_0)_b)\Big)\Big) \\ = \mathsf{idtoiso}((\epsilon_0{}^{-1})_{b'}) \circ g \circ \mathsf{idtoiso}((\epsilon_0)_b).$$

But this is just the fact that $(F_{Gb,Gb'})^{-1}$ is the inverse of $F_{Gb,Gb'}$. And we have remarked that (9.4.10) is equivalent to (9.4.11), so (ii) holds.

Conversely, suppose given (ii); then the object-parts of G, η, and ϵ together with (9.4.11) show that F_0 is an equivalence of types. And for $a, a' : A_0$, we define $\overline{G}_{a,a'} : \hom_B(Fa, Fa') \to \hom_A(a, a')$ by

$$\overline{G}_{a,a'}(g) :\equiv \mathsf{idtoiso}(\eta^{-1})_{a'} \circ G(g) \circ \mathsf{idtoiso}(\eta)_a. \tag{9.4.12}$$

By naturality of $\mathsf{idtoiso}(\eta)$, for any $f : \hom_A(a, a')$ we have

$$\begin{aligned} \overline{G}_{a,a'}(F_{a,a'}(f)) &= \mathsf{idtoiso}(\eta^{-1})_{a'} \circ G(F(f)) \circ \mathsf{idtoiso}(\eta)_a \\ &= \mathsf{idtoiso}(\eta^{-1})_{a'} \circ \mathsf{idtoiso}(\eta)_{a'} \circ f \\ &= f. \end{aligned}$$

On the other hand, for $g : \hom_B(Fa, Fa')$ we have

$$\begin{aligned} F_{a,a'}(\overline{G}_{a,a'}(g)) &= F(\mathsf{idtoiso}(\eta^{-1})_{a'}) \circ F(G(g)) \circ F(\mathsf{idtoiso}(\eta)_a) \\ &= \mathsf{idtoiso}(\epsilon)_{Fa'} \circ F(G(g)) \circ \mathsf{idtoiso}(\epsilon^{-1})_{Fa} \\ &= \mathsf{idtoiso}(\epsilon)_{Fa'} \circ \mathsf{idtoiso}(\epsilon^{-1})_{Fa'} \circ g \\ &= g. \end{aligned}$$

(There are lemmas needed here regarding the compatibility between idtoiso and whiskering, which we leave it to the reader to state and prove.) Thus, $F_{a,a'}$ is an equivalence, so F is fully faithful; i.e. (i) holds.

Now the composite (i)→(ii)→(i) is equal to the identity since (i) is a mere proposition. On the other side, tracing through the above constructions we see that the composite (ii)→(i)→(ii) essentially preserves the object-parts G_0, η_0, ϵ_0, and the object-part of (9.4.10). And in the latter three cases, the object-part is all there is, since hom-sets are sets.

Thus, it suffices to show that we recover the action of G on hom-sets. In other words, we must show that if $g : \hom_B(b, b')$, then

$$G_{b,b'}(g) = \overline{G}_{G_0 b, G_0 b'}\Big(\mathsf{idtoiso}({(\epsilon_0)^{-1}}_{b'}) \circ g \circ \mathsf{idtoiso}((\epsilon_0)_b)\Big)$$

where $\overline{G}$ is defined by (9.4.12). However, this follows from functoriality of G and the *other* triangle identity, which we have seen in Chapter 4 is equivalent to (9.4.11).

Now since (i) is a mere proposition, so is (ii), so it suffices to show they are co-inhabited with (iii). Of course, (ii)→(iii), so let us assume (iii). Since (i) is a mere proposition, we may assume given G, η, and ϵ. Then G_0 along with η and ϵ imply that F_0 is an equivalence. Moreover, we also have natural isomorphisms $\mathsf{idtoiso}(\eta) : 1_A \cong GF$ and $\mathsf{idtoiso}(\epsilon) : FG \cong 1_B$, so by Lemma 9.4.2, F is an equivalence of precategories, and in particular fully faithful. □

From Lemma 9.4.9(ii) and idtoiso in functor categories, we conclude immediately that any isomorphism of precategories is an equivalence. For precategories, the converse can fail.

Example 9.4.13. Let X be a type and $x_0 : X$ an element, and let X_{ch} denote the *chaotic* or *indiscrete* precategory on X. By definition, we have $(X_{\mathrm{ch}})_0 :\equiv X$, and $\hom_{X_{\mathrm{ch}}}(x, x') = 1$ for all x, x'. Then the unique functor $X_{\mathrm{ch}} \to 1$ is an equivalence of precategories, but not an isomorphism unless X is contractible.

This example also shows that a precategory can be equivalent to a category without itself being a category. Of course, if a precategory is *isomorphic* to a category, then it must itself be a category.

However, for categories, the two notions coincide.

Lemma 9.4.14. *For categories A and B, a functor $F : A \to B$ is an equivalence of categories if and only if it is an isomorphism of categories.*

Proof. Since both are mere properties, it suffices to show they are co-inhabited. So first suppose F is an equivalence of categories, with (G, η, ϵ) given. We have already seen that F is fully faithful. By Theorem 9.2.5, the natural isomorphisms η and ϵ yield identities $1_A = GF$ and $FG = 1_B$, hence in particular identities $\mathsf{id}_A = G_0 \circ F_0$ and $F_0 \circ G_0 = \mathsf{id}_B$. Thus, F_0 is an equivalence of types.

Conversely, suppose F is fully faithful and F_0 is an equivalence of types, with inverse G_0, say. Then for each $b : B$ we have $G_0 b : A$ and an identity $FGb = b$, hence an isomorphism $FGb \cong b$. Thus, by Lemma 9.4.5, F is an equivalence of categories. □

Of course, there is yet a third notion of sameness for (pre)categories: equality. However, the univalence axiom implies that it coincides with isomorphism.

Lemma 9.4.15. *If A and B are precategories, then the function*

$$(A = B) \to (A \cong B)$$

(defined by induction from the identity functor) is an equivalence of types.

Proof. As usual for dependent sum types, to give an element of $A = B$ is equivalent to giving

- an identity $P_0 : A_0 = B_0$,
- for each $a, b : A_0$, an identity

$$P_{a,b} : \hom_A(a, b) = \hom_B(P_{0*}(a), P_{0*}(b)),$$

- identities $(P_{a,a})_*(1_a) = 1_{P_{0*}(a)}$ and $(P_{a,c})_*(gf) = (P_{b,c})_*(g) \circ (P_{a,b})_*(f)$.

(Again, we use the fact that the identity types of hom-sets are mere propositions.) However, by univalence, this is equivalent to giving

- an equivalence of types $F_0 : A_0 \simeq B_0$,
- for each $a, b : A_0$, an equivalence of types

$$F_{a,b} : \hom_A(a,b) \simeq \hom_B(F_0(a), F_0(b)),$$

- and identities $F_{a,a}(1_a) = 1_{F_0(a)}$ and $F_{a,c}(gf) = F_{b,c}(g) \circ F_{a,b}(f)$.

But this consists exactly of a functor $F : A \to B$ that is an isomorphism of categories. And by induction on identity, this equivalence $(A = B) \simeq (A \cong B)$ is equal to the one obtained by induction. □

Thus, for categories, equality also coincides with equivalence. We can interpret this as saying that categories, functors, and natural transformations form, not just a pre-2-category, but a 2-category.

Theorem 9.4.16. *If A and B are categories, then the function*

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

(defined by induction from the identity functor) is an equivalence of types.

Proof. By Lemmas 9.4.14 and 9.4.15. □

As a consequence, the type of categories is a 2-type. For since $A \simeq B$ is a subtype of the type of functors from A to B, which are the objects of a category, it is a 1-type; hence the identity types $A = B$ are also 1-types.

9.5 The Yoneda lemma

Recall that we have a category $\mathcal{S}et$ whose objects are sets and whose morphisms are functions. We now show that every precategory has a $\mathcal{S}et$-valued hom-functor. First we need to define opposites and products of (pre)categories.

Definition 9.5.1. For a precategory A, its **opposite** A^{op} is a precategory with the same type of objects, with $\hom_{A^{\mathrm{op}}}(a,b) :\equiv \hom_A(b,a)$, and with identities and composition inherited from A.

Definition 9.5.2. For precategories A and B, their **product** $A \times B$ is a precategory with $(A \times B)_0 :\equiv A_0 \times B_0$ and

$$\hom_{A\times B}((a,b),(a',b')) :\equiv \hom_A(a,a') \times \hom_B(b,b').$$

Identities are defined by $1_{(a,b)} :\equiv (1_a, 1_b)$ and composition by $(g,g')(f,f') :\equiv ((gf),(g'f'))$.

Lemma 9.5.3. *For precategories A, B, C, the following types are equivalent.*

(i) Functors $A \times B \to C$.

(ii) Functors $A \to C^B$.

Proof. Given $F : A \times B \to C$, for any $a : A$ we obviously have a functor $F_a : B \to C$. This gives a function $A_0 \to (C^B)_0$. Next, for any $f : \hom_A(a, a')$, we have for any $b : B$ the morphism $F_{(a,b),(a',b)}(f, 1_b) : F_a(b) \to F_{a'}(b)$. These are the components of a natural transformation $F_a \to F_{a'}$. Functoriality in a is easy to check, so we have a functor $\hat{F} : A \to C^B$.

Conversely, suppose given $G : A \to C^B$. Then for any $a : A$ and $b : B$ we have the object $G(a)(b) : C$, giving a function $A_0 \times B_0 \to C_0$. And for $f : \hom_A(a, a')$ and $g : \hom_B(b, b')$, we have the morphism

$$G(a')_{b,b'}(g) \circ G_{a,a'}(f)_b = G_{a,a'}(f)_{b'} \circ G(a)_{b,b'}(g)$$

in $\hom_C(G(a)(b), G(a')(b'))$. Functoriality is again easy to check, so we have a functor $\check{F} : A \times B \to C$.

Finally, it is also clear that these operations are inverses. □

Now for any precategory A, we have a hom-functor

$$\hom_A : A^{\mathrm{op}} \times A \to \mathcal{S}et.$$

It takes a pair $(a, b) : (A^{\mathrm{op}})_0 \times A_0 \equiv A_0 \times A_0$ to the set $\hom_A(a, b)$. For a morphism $(f, f') : \hom_{A^{\mathrm{op}} \times A}((a, b), (a', b'))$, by definition we have $f : \hom_A(a', a)$ and $f' : \hom_A(b, b')$, so we can define

$$\begin{aligned}(\hom_A)_{(a,b),(a',b')}(f, f') &:\equiv (g \mapsto (f' g f)) \\ &: \hom_A(a, b) \to \hom_A(a', b').\end{aligned}$$

Functoriality is easy to check.

By Lemma 9.5.3, therefore, we have an induced functor $\mathbf{y} : A \to \mathcal{S}et^{A^{\mathrm{op}}}$, which we call the **Yoneda embedding**.

Theorem 9.5.4 (The Yoneda lemma)**.** *For any precategory A, any $a : A$, and any functor $F : \mathcal{S}et^{A^{op}}$, we have an isomorphism*

$$\hom_{\mathcal{S}et^{A^{op}}}(\mathbf{y}a, F) \cong Fa. \tag{9.5.5}$$

Moreover, this is natural in both a and F.

Proof. Given a natural transformation $\alpha : \mathbf{y}a \to F$, we can consider the component $\alpha_a : \mathbf{y}a(a) \to Fa$. Since $\mathbf{y}a(a) \equiv \hom_A(a, a)$, we have $1_a : \mathbf{y}a(a)$, so that $\alpha_a(1_a) : Fa$. This gives a function $(\alpha \mapsto \alpha_a(1_a))$ from left to right in (9.5.5).

In the other direction, given $x : Fa$, we define $\alpha : \mathbf{y}a \to F$ by

$$\alpha_{a'}(f) :\equiv F_{a',a}(f)(x).$$

Naturality is easy to check, so this gives a function from right to left in (9.5.5).

To show that these are inverses, first suppose given $x : Fa$. Then with α defined as above, we have $\alpha_a(1_a) = F_{a,a}(1_a)(x) = 1_{Fa}(x) = x$. On the other hand, if we suppose given $\alpha : \mathbf{y}a \to F$ and

define x as above, then for any $f : \hom_A(a', a)$ we have

$$\begin{aligned}\alpha_{a'}(f) &= \alpha_{a'}(\mathbf{y}a_{a',a}(f)) \\ &= (\alpha_{a'} \circ \mathbf{y}a_{a',a}(f))(1_a) \\ &= (F_{a',a}(f) \circ \alpha_a)(1_a) \\ &= F_{a',a}(f)(\alpha_a(1_a)) \\ &= F_{a',a}(f)(x).\end{aligned}$$

Thus, both composites are equal to identities. We leave the proof of naturality to the reader. □

Corollary 9.5.6. *The Yoneda embedding $\mathbf{y} : A \to \mathcal{S}et^{A^{op}}$ is fully faithful.*

Proof. By Theorem 9.5.4, we have

$$\hom_{\mathcal{S}et^{A^{\mathrm{op}}}}(\mathbf{y}a, \mathbf{y}b) \cong \mathbf{y}b(a) \equiv \hom_A(a, b).$$

It is easy to check that this isomorphism is in fact the action of $\mathbf{y}$ on hom-sets. □

Corollary 9.5.7. *If A is a category, then $\mathbf{y}_0 : A_0 \to (\mathcal{S}et^{A^{op}})_0$ is an embedding. In particular, if $\mathbf{y}a = \mathbf{y}b$, then $a = b$.*

Proof. By Corollary 9.5.6, $\mathbf{y}$ induces an isomorphism on sets of isomorphisms. But as A and $\mathcal{S}et^{A^{\mathrm{op}}}$ are categories and $\mathbf{y}$ is a functor, this is equivalently an isomorphism on identity types, which is the definition of being an embedding. □

Definition 9.5.8. A functor $F : \mathcal{S}et^{A^{\mathrm{op}}}$ is said to be **representable** if there exists $a : A$ and an isomorphism $\mathbf{y}a \cong F$.

Theorem 9.5.9. *If A is a category, then the type "F is representable" is a mere proposition.*

Proof. By definition "F is representable" is just the fiber of $\mathbf{y}_0$ over F. Since $\mathbf{y}_0$ is an embedding by Corollary 9.5.7, this fiber is a mere proposition. □

In particular, in a category, any two representations of the same functor are equal. We can use this to give a different proof of Lemma 9.3.2. First we give a characterization of adjunctions in terms of representability.

Lemma 9.5.10. *For any precategories A and B and a functor $F : A \to B$, the following types are equivalent.*

(i) F is a left adjoint.
(ii) For each $b : B$, the functor $(a \mapsto \hom_B(Fa, b))$ from A^{op} to $\mathcal{S}et$ is representable.

Proof. An element of the type (ii) consists of a function $G_0 : B_0 \to A_0$ together with, for every $a : A$ and $b : B$ an isomorphism

$$\gamma_{a,b} : \hom_B(Fa, b) \cong \hom_A(a, G_0 b)$$

such that $\gamma_{a,b}(g \circ Ff) = \gamma_{a',b}(g) \circ f$ for $f : \hom_A(a, a')$.

Given this, for $a : A$ we define $\eta_a :\equiv \gamma_{a,Fa}(1_{Fa})$, and for $b : B$ we define $\epsilon_b :\equiv (\gamma_{Gb,b})^{-1}(1_{Gb})$. Now for $g : \hom_B(b, b')$ we define

$$G_{b,b'}(g) :\equiv \gamma_{Gb,b'}(g \circ \epsilon_b)$$

The verifications that G is a functor and η and ϵ are natural transformations satisfying the triangle identities are exactly as in the classical case, and as they are all mere propositions we will not care about their values. Thus, we have a function (ii)→(i).

In the other direction, if F is a left adjoint, we of course have G_0 specified, and we can take $\gamma_{a,b}$ to be the composite

$$\hom_B(Fa, b) \xrightarrow{G_{Fa,b}} \hom_A(GFa, Gb) \xrightarrow{(-\circ\eta_a)} \hom_A(a, Gb).$$

This is clearly natural since η is, and it has an inverse given by

$$\hom_A(a, Gb) \xrightarrow{F_{a,Gb}} \hom_B(Fa, FGb) \xrightarrow{(\epsilon_b\circ-)} \hom_A(Fa, b)$$

(by the triangle identities). Thus we also have (i)→ (ii).

For the composite (ii)→(i)→ (ii), clearly the function G_0 is preserved, so it suffices to check that we get back γ. But the new γ is defined to take $f : \hom_B(Fa, b)$ to

$$\begin{aligned} G(f) \circ \eta_a &\equiv \gamma_{GFa,b}(f \circ \epsilon_{Fa}) \circ \eta_a \\ &= \gamma_{GFa,b}(f \circ \epsilon_{Fa} \circ F\eta_a) \\ &= \gamma_{GFa,b}(f) \end{aligned}$$

so it agrees with the old one.

Finally, for (i)→(ii)→ (i), we certainly get back the functor G on objects. The new $G_{b,b'} : \hom_B(b, b') \to \hom_A(Gb, Gb')$ is defined to take g to

$$\begin{aligned} \gamma_{Gb,b'}(g \circ \epsilon_b) &\equiv G(g \circ \epsilon_b) \circ \eta_{Gb} \\ &= G(g) \circ G\epsilon_b \circ \eta_{Gb} \\ &= G(g) \end{aligned}$$

so it agrees with the old one. The new η_a is defined to be $\gamma_{a,Fa}(1_{Fa}) \equiv G(1_{Fa}) \circ \eta_a$, so it equals the old η_a. And finally, the new ϵ_b is defined to be $(\gamma_{Gb,b})^{-1}(1_{Gb}) \equiv \epsilon_b \circ F(1_{Gb})$, which also equals the old ϵ_b. □

Corollary 9.5.11. *[Lemma 9.3.2] If A is a category and $F : A \to B$, then the type "F is a left adjoint" is a mere proposition.*

Proof. By Theorem 9.5.9, if A is a category then the type in Lemma 9.5.10(ii) is a mere proposition. □

9.6 Strict categories

Definition 9.6.1. A **strict category** is a precategory whose type of objects is a set.

In accordance with the mathematical red herring principle, a strict category is not necessarily a category. In fact, a category is a strict category precisely when it is gaunt (Example 9.1.15). Most of the time, category theory is about categories, not strict ones, but sometimes one wants to consider strict categories. The main advantage of this is that strict categories have a stricter notion of "sameness" than equivalence, namely isomorphism (or equivalently, by Lemma 9.4.15, equality).

Here is one origin of strict categories.

Example 9.6.2. Let A be a precategory and $x : A$ an object. Then there is a precategory $\mathsf{mono}(A, x)$ as follows:

- Its objects consist of an object $y : A$ and a monomorphism $m : \hom_A(y, x)$. (As usual, $m : \hom_A(y, x)$ is a monomorphism if $(m \circ f = m \circ g) \Rightarrow (f = g)$.)
- Its morphisms from (y, m) to (z, n) are arbitrary morphisms from y to z in A (not necessarily respecting m and n).

An equality $(y, m) = (z, n)$ of objects in $\mathsf{mono}(A, x)$ consists of an equality $p : y = z$ and an equality $p_*(m) = n$, which by Lemma 9.1.9 is equivalently an equality $m = n \circ \mathsf{idtoiso}(p)$. Since hom-sets are sets, the type of such equalities is a mere proposition. But since m and n are monomorphisms, the type of morphisms f such that $m = n \circ f$ is also a mere proposition. Thus, if A is a category, then $(y, m) = (z, n)$ is a mere proposition, and hence $\mathsf{mono}(A, x)$ is a strict category.

This example can be dualized, and generalized in various ways. Here is an interesting application of strict categories.

Example 9.6.3. Let E/F be a finite Galois extension of fields, and G its Galois group. Then there is a strict category whose objects are intermediate fields $F \subseteq K \subseteq E$, and whose morphisms are field homomorphisms which fix F pointwise (but need not commute with the inclusions into E). There is another strict category whose objects are subgroups $H \subseteq G$, and whose morphisms are morphisms of G-sets $G/H \to G/K$. The fundamental theorem of Galois theory says that these two precategories are isomorphic (not merely equivalent).

9.7 †-categories

It is also worth mentioning a useful kind of precategory whose type of objects is not a set, but which is not a category either.

Definition 9.7.1. A **†-precategory** is a precategory A together with the following.

(i) For each $x, y : A$, a function $(-)^\dagger : \hom_A(x, y) \to \hom_A(y, x)$.
(ii) For all $x : A$, we have $(1_x)^\dagger = 1_x$.
(iii) For all f, g we have $(g \circ f)^\dagger = f^\dagger \circ g^\dagger$.

(iv) For all f we have $(f^\dagger)^\dagger = f$.

Definition 9.7.2. A morphism $f : \hom_A(x, y)$ in a †-precategory is **unitary** if $f^\dagger \circ f = 1_x$ and $f \circ f^\dagger = 1_y$.

Of course, every unitary morphism is an isomorphism, and being unitary is a mere proposition. Thus for each $x, y : A$ we have a set of unitary isomorphisms from x to y, which we denote $(x \cong^\dagger y)$.

Lemma 9.7.3. *If $p : (x = y)$, then* $\mathsf{idtoiso}(p)$ *is unitary.*

Proof. By induction, we may assume p is refl_x. But then $(1_x)^\dagger \circ 1_x = 1_x \circ 1_x = 1_x$ and similarly. □

Definition 9.7.4. A **†-category** is a †-precategory such that for all $x, y : A$, the function

$$(x = y) \to (x \cong^\dagger y)$$

from Lemma 9.7.3 is an equivalence.

Example 9.7.5. The category $\mathcal{R}el$ from Example 9.1.19 becomes a †-precategory if we define $(R^\dagger)(y, x) :\equiv R(x, y)$. The proof that $\mathcal{R}el$ is a category actually shows that every isomorphism is unitary; hence $\mathcal{R}el$ is also a †-category.

Example 9.7.6. Any groupoid becomes a †-category if we define $f^\dagger :\equiv f^{-1}$.

Example 9.7.7. Let $\mathcal{H}ilb$ be the following precategory.

- Its objects are finite-dimensional vector spaces equipped with an inner product $\langle -, - \rangle$.
- Its morphisms are arbitrary linear maps.

By standard linear algebra, any linear map $f : V \to W$ between finite dimensional inner product spaces has a uniquely defined adjoint $f^\dagger : W \to V$, characterized by $\langle fv, w \rangle = \langle v, f^\dagger w \rangle$. In this way, $\mathcal{H}ilb$ becomes a †-precategory. Moreover, a linear isomorphism is unitary precisely when it is an *isometry*, i.e. $\langle fv, fw \rangle = \langle v, w \rangle$. It follows from this that $\mathcal{H}ilb$ is a †-category, though it is not a category (not every linear isomorphism is unitary).

There has been a good deal of general theory developed for †-categories under classical foundations. It was observed early on that the unitary isomorphisms, not arbitrary isomorphisms, are the correct notion of "sameness" for objects of a †-category, which has caused some consternation among category theorists. Homotopy type theory resolves this issue by identifying †-categories, like strict categories, as simply a different kind of precategory.

9.8 The Structure identity principle

The *structure identity principle* is an informal principle that expresses that isomorphic structures are identical. We aim to prove a general abstract result which can be applied to a wide family of notions of structure, where structures may be many-sorted or even dependently-sorted, infinitary, or even higher order.

The simplest kind of single-sorted structure consists of a type with no additional structure. The univalence axiom expresses the structure identity principle for that notion of structure in a strong form: for types A, B, the canonical function $(A = B) \to (A \simeq B)$ is an equivalence.

We start with a precategory X. In our application to single-sorted first order structures, X will be the category of $\mathcal{U}$-small sets, where $\mathcal{U}$ is a univalent type universe.

Definition 9.8.1. A **notion of structure** (P, H) over X consists of the following.

(i) A function $P : X_0 \to \mathcal{U}$. For each $x : X_0$ the elements of Px are called (P, H)**-structures** on x.

(ii) For $x, y : X_0$ and $\alpha : Px$, $\beta : Py$, to each $f : \hom_X(x, y)$ a mere proposition

$$H_{\alpha\beta}(f).$$

If $H_{\alpha\beta}(f)$ is true, we say that f is a (P, H)**-homomorphism** from α to β.

(iii) For $x : X_0$ and $\alpha : Px$, we have $H_{\alpha\alpha}(1_x)$.

(iv) For $x, y, z : X_0$ and $\alpha : Px$, $\beta : Py$, $\gamma : Pz$, if $f : \hom_X(x, y)$, we have

$$H_{\alpha\beta}(f) \to H_{\beta\gamma}(g) \to H_{\alpha\gamma}(g \circ f).$$

When (P, H) is a notion of structure, for $\alpha, \beta : Px$ we define

$$\alpha \leq_x \beta :\equiv H_{\alpha\beta}(1_x).$$

By (iii) and (iv), this is a preorder (Example 9.1.14) with Px its type of objects. We say that (P, H) is a **standard notion of structure** if this preorder is in fact a partial order, for all $x : X$.

Note that for a standard notion of structure, each type Px must actually be a set. We now define, for any notion of structure (P, H), a precategory of (P, H)**-structures**, $A = \mathsf{Str}_{(P,H)}(X)$.

- The type of objects of A is the type $A_0 :\equiv \sum_{(x:X)} Px$. If $a \equiv (x, \alpha) : A_0$, we may write $|a| :\equiv x$.
- For $(x, \alpha) : A_0$ and $(y, \beta) : A_0$, we define

$$\hom_A((x, \alpha), (y, \beta)) :\equiv \{ f : x \to y \mid H_{\alpha\beta}(f) \}.$$

The composition and identities are inherited from X; conditions (iii) and (iv) ensure that these lift to A.

Theorem 9.8.2 (Structure identity principle)**.** *If X is a category and (P, H) is a standard notion of structure over X, then the precategory $\mathsf{Str}_{(P,H)}(X)$ is a category.*

Proof. By the definition of equality in sum types, to give an equality $(x, \alpha) = (y, \beta)$ consists of

- An equality $p : x = y$, and
- An equality $p_*(\alpha) = \beta$.

Since P is set-valued, the latter is a mere proposition. On the other hand, it is easy to see that an isomorphism $(x, \alpha) \cong (y, \beta)$ in $\mathsf{Str}_{(P,H)}(X)$ consists of

- An isomorphism $f : x \cong y$ in X, such that
- $H_{\alpha\beta}(f)$ and $H_{\beta\alpha}(f^{-1})$.

Of course, the second of these is also a mere proposition. And since X is a category, the function $(x = y) \to (x \cong y)$ is an equivalence. Thus, it will suffice to show that for any $p : x = y$ and for any $(\alpha : Px)$, $(\beta : Py)$, we have $p_*(\alpha) = \beta$ if and only if both $H_{\alpha\beta}(\mathsf{idtoiso}(p))$ and $H_{\beta\alpha}(\mathsf{idtoiso}(p)^{-1})$.

The "only if" direction is just the existence of the function idtoiso for the category $\mathsf{Str}_{(P,H)}(X)$. For the "if" direction, by induction on p we may assume that $y \equiv x$ and $p \equiv \mathsf{refl}_x$. However, in this case $\mathsf{idtoiso}(p) \equiv 1_x$ and therefore $\mathsf{idtoiso}(p)^{-1} = 1_x$. Thus, $\alpha \leq_x \beta$ and $\beta \leq_x \alpha$, which implies $\alpha = \beta$ since (P, H) is a standard notion of structure. □

As an example, this methodology gives an alternative way to express the proof of Theorem 9.2.5.

Example 9.8.3. Let A be a precategory and B a category. There is a precategory B^{A_0} whose objects are functions $A_0 \to B_0$, and whose set of morphisms from $F_0 : A_0 \to B_0$ to $G_0 : A_0 \to B_0$ is $\prod_{(a:A_0)} \hom_B(F_0 a, G_0 a)$. Composition and identities are inherited directly from those in B. It is easy to show that $\gamma : \hom_{B^{A_0}}(F_0, G_0)$ is an isomorphism exactly when each component γ_a is an isomorphism, so that we have $(F_0 \cong G_0) \simeq \prod_{(a:A_0)}(F_0 a \cong G_0 a)$. Moreover, the map $\mathsf{idtoiso} : (F_0 = G_0) \to (F_0 \cong G_0)$ of B^{A_0} is equal to the composite

$$(F_0 = G_0) \to \prod_{a:A_0} (F_0 a = G_0 a) \to \prod_{a:A_0} (F_0 a \cong G_0 a) \to (F_0 \cong G_0)$$

in which the first map is an equivalence by function extensionality, the second because it is a dependent product of equivalences (since B is a category), and the third as remarked above. Thus, B^{A_0} is a category.

Now we define a notion of structure on B^{A_0} for which $P(F_0)$ is the type of operations $F : \prod_{(a,a':A_0)} \hom_A(a, a') \to \hom_B(F_0 a, F_0 a')$ which extend F_0 to a functor (i.e. preserve composition and identities). This is a set since each $\hom_B(-,-)$ is so. Given such F and G, we define $\gamma : \hom_{B^{A_0}}(F_0, G_0)$ to be a homomorphism if it forms a natural transformation. In Definition 9.2.3 we essentially verified that this is a notion of structure. Moreover, if F and F' are both structures on F_0 and the identity is a natural transformation from F to F', then for any $f : \hom_A(a, a')$ we have $F'f = F'f \circ 1_{F_0 a} = 1_{F_0 a} \circ Ff = Ff$. Applying function extensionality, we conclude $F = F'$. Thus, we have a *standard* notion of structure, and so by Theorem 9.8.2, the precategory B^A is a category.

As another example, we consider categories of structures first-order for a first-order (FO) signature. We define an **FO-signature**, Ω, to consist of sets Ω_0 and Ω_1 of function symbols, $\omega : \Omega_0$, and relation symbols, $\omega : \Omega_1$, each having an arity $|\omega|$ that is a set. An **Ω-structure** a consists of a set $|a|$ together with an assignment of an $|\omega|$-ary function $\omega^a : |a|^{|\omega|} \to |a|$ on $|a|$ to each function symbol, ω, and an assignment of an $|\omega|$-ary relation ω^a on $|a|$, assigning a mere proposition $\omega^a x$ to each $x : |a|^{|\omega|}$, to each relation symbol. And given Ω-structures a, b, a function

$f : |a| \to |b|$ is a **homomorphism** $a \to b$ if it preserves the structure; i.e. if for each symbol ω of the signature and each $x : |a|^{|\omega|}$,

(i) $f(\omega^a x) = \omega^b(f \circ x)$ if $\omega : \Omega_0$, and
(ii) $\omega^a x \to \omega^b(f \circ x)$ if $\omega : \Omega_1$.

Note that each $x : |a|^{|\omega|}$ is a function $x : |\omega| \to |a|$ so that $f \circ x : b^\omega$.

Now we assume given a univalent universe $\mathcal{U}$ and a $\mathcal{U}$-small signature Ω; i.e. $|\Omega|$ is a $\mathcal{U}$-small set and, for each $\omega : |\Omega|$, the set $|\omega|$ is $\mathcal{U}$-small. We form the precategory $\mathcal{S}et_\mathcal{U}$ of $\mathcal{U}$-small sets, which is a category because $\mathcal{U}$ is univalent. We want to define the precategory of $\mathcal{U}$-small Ω-structures over $\mathcal{S}et_\mathcal{U}$ and use Theorem 9.8.2 to show that it is a category.

We use the FO-signature Ω to give us a standard notion of structure (P, H) over $\mathcal{S}et_\mathcal{U}$.

Definition 9.8.4.

(i) For each $\mathcal{U}$-small set x define

$$Px :\equiv P_0 x \times P_1 x.$$

Here

$$P_0 x :\equiv \Pi_{\omega:\Omega_0}\ x^{|\omega|} \to x, \text{ and}$$
$$P_1 x :\equiv \Pi_{\omega:\Omega_1}\ x^{|\omega|} \to \mathbb{P}_\mathcal{U},$$

where $\mathbb{P}_\mathcal{U}$ is the set of $\mathcal{U}$-small propositions.

(ii) For $\mathcal{U}$-small sets x, y and $\alpha : P^\omega x,\ \beta : P^\omega y,\ f : x \to y$, define

$$H_{\alpha\beta}(f) :\equiv H_{0,\alpha\beta}(f) \wedge H_{1,\alpha\beta}(f).$$

Here

$$H_{0,\alpha\beta}(f) :\equiv \forall_{\omega:\Omega_0} \forall_{u:x^{|\omega|}}\ \ f(\alpha u) = \beta(f \circ u), \text{ and}$$
$$H_{1,\alpha\beta}(f) :\equiv \forall_{\omega:\Omega_1} \forall_{u:x^{|\omega|}}\ \ \alpha u \to \beta(f \circ u).$$

It is now routine to check that (P, H) is a standard notion of structure over $\mathcal{S}et_\mathcal{U}$ and hence we may use Theorem 9.8.2 to get that the precategory $Str_{(P,H)}(\mathcal{S}et_\mathcal{U})$ is a category. It only remains to observe that this is essentially the same as the precategory of $\mathcal{U}$-small Ω-structures over $\mathcal{S}et_\mathcal{U}$.

9.9 The Rezk completion

In this section we will give a universal way to replace a precategory by a category. In fact, we will give two. Both rely on the fact that "categories see weak equivalences as equivalences".

To prove this, we begin with a couple of lemmas which are completely standard category theory, phrased carefully so as to make sure we are using the eliminator for $\| - \|_{-1}$ correctly. One would have to be similarly careful in classical category theory if one wanted to avoid the axiom of choice: any time we want to define a function, we need to characterize its values uniquely somehow.

Lemma 9.9.1. *If A, B, C are precategories and $H : A \to B$ is an essentially surjective functor, then $(- \circ H) : C^B \to C^A$ is faithful.*

Proof. Let $F, G : B \to C$, and $\gamma, \delta : F \to G$ be such that $\gamma H = \delta H$; we must show $\gamma = \delta$. Thus let $b : B$; we want to show $\gamma_b = \delta_b$. This is a mere proposition, so since H is essentially surjective, we may assume given an $a : A$ and an isomorphism $f : Ha \cong b$. But now we have

$$\gamma_b = G(f) \circ \gamma_{Ha} \circ F(f^{-1}) = G(f) \circ \delta_{Ha} \circ F(f^{-1}) = \delta_b. \qquad \square$$

Lemma 9.9.2. *If A, B, C are precategories and $H : A \to B$ is essentially surjective and full, then $(- \circ H) : C^B \to C^A$ is fully faithful.*

Proof. It remains to show fullness. Thus, let $F, G : B \to C$ and $\gamma : FH \to GH$. We claim that for any $b : B$, the type

$$\sum_{(g : \hom_C(Fb, Gb))} \prod_{(a:A)} \prod_{(f : Ha \cong b)} (\gamma_a = Gf^{-1} \circ g \circ Ff) \tag{9.9.3}$$

is contractible. Since contractibility is a mere property, and H is essentially surjective, we may assume given $a_0 : A$ and $h : Ha_0 \cong b$.

Now take $g :\equiv Gh \circ \gamma_{a_0} \circ Fh^{-1}$. Then given any other $a : A$ and $f : Ha \cong b$, we must show $\gamma_a = Gf^{-1} \circ g \circ Ff$. Since H is full, there merely exists a morphism $k : \hom_A(a, a_0)$ such that $Hk = h^{-1} \circ f$. And since our goal is a mere proposition, we may assume given some such k. Then we have

$$\begin{aligned} \gamma_a &= GHk^{-1} \circ \gamma_{a_0} \circ FHk \\ &= Gf^{-1} \circ Gh \circ \gamma_{a_0} \circ Fh^{-1} \circ Ff \\ &= Gf^{-1} \circ g \circ Ff. \end{aligned}$$

Thus, (9.9.3) is inhabited. It remains to show it is a mere proposition. Let $g, g' : \hom_C(Fb, Gb)$ be such that for all $a : A$ and $f : Ha \cong b$, we have both $(\gamma_a = Gf^{-1} \circ g \circ Ff)$ and $(\gamma_a = Gf^{-1} \circ g' \circ Ff)$. The dependent product types are mere propositions, so all we have to prove is $g = g'$. But this is a mere proposition, so we may assume $a_0 : A$ and $h : Ha_0 \cong b$, in which case we have

$$g = Gh \circ \gamma_{a_0} \circ Fh^{-1} = g'.$$

This proves that (9.9.3) is contractible for all $b : B$. Now we define $\delta : F \to G$ by taking δ_b to be the unique g in (9.9.3) for that b. To see that this is natural, suppose given $f : \hom_B(b, b')$; we must show $Gf \circ \delta_b = \delta_{b'} \circ Ff$. As before, we may assume $a : A$ and $h : Ha \cong b$, and likewise $a' : A$ and $h' : Ha' \cong b'$. Since H is full as well as essentially surjective, we may also assume $k : \hom_A(a, a')$ with $Hk = {h'}^{-1} \circ f \circ h$.

Since γ is natural, $GHk \circ \gamma_a = \gamma_{a'} \circ FHk$. Using the definition of δ, we have

$$\begin{aligned} Gf \circ \delta_b &= Gf \circ Gh \circ \gamma_a \circ Fh^{-1} \\ &= Gh' \circ GHk \circ \gamma_a \circ Fh^{-1} \\ &= Gh' \circ \gamma_{a'} \circ FHk \circ Fh^{-1} \\ &= Gh' \circ \gamma_{a'} \circ {Fh'}^{-1} \circ Ff \\ &= \delta_{b'} \circ Ff. \end{aligned}$$

Thus, δ is natural. Finally, for any $a : A$, applying the definition of δ_{Ha} to a and 1_a, we obtain $\gamma_a = \delta_{Ha}$. Hence, $\delta \circ H = \gamma$. □

The rest of the theorem follows almost exactly the same lines, with the category-ness of C inserted in one crucial step, which we have bolded below for emphasis. This is the point at which we are trying to define a function into *objects* without using choice, and so we must be careful about what it means for an object to be "uniquely specified". In classical category theory, all one can say is that this object is specified up to unique isomorphism, but in set-theoretic foundations this is not a sufficient amount of uniqueness to give us a function without invoking AC. In univalent foundations, however, if C is a category, then isomorphism is equality, and we have the appropriate sort of uniqueness (namely, living in a contractible space).

Theorem 9.9.4. *If A, B are precategories, C is a category, and $H : A \to B$ is a weak equivalence, then $(- \circ H) : C^B \to C^A$ is an isomorphism.*

Proof. By Theorem 9.2.5, C^B and C^A are categories. Thus, by Lemma 9.4.14 it will suffice to show that $(- \circ H)$ is an equivalence. But since we know from the preceeding two lemmas that it is fully faithful, by Lemma 9.4.7 it will suffice to show that it is essentially surjective. Thus, suppose $F : A \to C$; we want there to merely exist a $G : B \to C$ such that $GH \cong F$.

For each $b : B$, let X_b be the type whose elements consist of:

(i) An element $c : C$; and

(ii) For each $a : A$ and $h : Ha \cong b$, an isomorphism $k_{a,h} : Fa \cong c$; such that

(iii) For each (a, h) and (a', h') as in (ii) and each $f : \hom_A(a, a')$ such that $h' \circ Hf = h$, we have $k_{a',h'} \circ Ff = k_{a,h}$.

We claim that for any $b : B$, the type X_b is contractible. As this is a mere proposition, we may assume given $a_0 : A$ and $h_0 : Ha_0 \cong b$. Let $c^0 :\equiv Fa_0$. Next, given $a : A$ and $h : Ha \cong b$, since H is fully faithful there is a unique isomorphism $g_{a,h} : a \to a_0$ with $Hg_{a,h} = {h_0}^{-1} \circ h$; define $k^0_{a,h} :\equiv Fg_{a,h}$. Finally, if $h' \circ Hf = h$, then ${h_0}^{-1} \circ h' \circ Hf = {h_0}^{-1} \circ h$, hence $g_{a',h'} \circ f = g_{a,h}$ and thus $k^0_{a',h'} \circ Ff = k^0_{a,h}$. Therefore, X_b is inhabited.

Now suppose given another $(c^1, k^1) : X_b$. Then $k^1_{a_0,h_0} : c^0 \equiv Fa_0 \cong c^1$. *Since C is a category, we have $p : c^0 = c^1$ with* $\mathsf{idtoiso}(p) = k^1_{a_0,h_0}$. And for any $a : A$ and $h : Ha \cong b$, by (iii) for (c^1, k^1) with $f :\equiv g_{a,h}$, we have

$$k^1_{a,h} = k^1_{a_0,h_0} \circ k^0_{a,h} = p_*(k^0_{a,h})$$

This gives the requisite data for an equality $(c^0, k^0) = (c^1, k^1)$, completing the proof that X_b is contractible.

Now since X_b is contractible for each b, the type $\prod_{(b:B)} X_b$ is also contractible. In particular, it is inhabited, so we have a function assigning to each $b : B$ a c and a k. Define $G_0(b)$ to be this c; this gives a function $G_0 : B_0 \to C_0$.

Next we need to define the action of G on morphisms. For each $b, b' : B$ and $f : \hom_B(b, b')$, let Y_f be the type whose elements consist of:

(iv) A morphism $g : \hom_C(Gb, Gb')$, such that

(v) For each $a : A$ and $h : Ha \cong b$, and each $a' : A$ and $h' : Ha' \cong b'$, and any $\ell : \hom_A(a, a')$, we have
$$(h' \circ H\ell = f \circ h) \to (k_{a',h'} \circ F\ell = g \circ k_{a,h}).$$

We claim that for any b, b' and f, the type Y_f is contractible. As this is a mere proposition, we may assume given $a_0 : A$ and $h_0 : Ha_0 \cong b$, and each $a_0' : A$ and $h_0' : Ha_0' \cong b'$. Then since H is fully faithful, there is a unique $\ell_0 : \hom_A(a_0, a_0')$ such that $h_0' \circ H\ell_0 = f \circ h_0$. Define $g_0 :\equiv k_{a_0',h_0'} \circ F\ell_0 \circ (k_{a_0,h_0})^{-1}$.

Now for any a, h, a', h', and ℓ such that $(h' \circ H\ell = f \circ h)$, we have $h^{-1} \circ h_0 : Ha_0 \cong Ha$, hence there is a unique $m : a_0 \cong a$ with $Hm = h^{-1} \circ h_0$ and hence $h \circ Hm = h_0$. Similarly, we have a unique $m' : a_0' \cong a'$ with $h' \circ Hm' = h_0'$. Now by (iii), we have $k_{a,h} \circ Fm = k_{a_0,h_0}$ and $k_{a',h'} \circ Fm' = k_{a_0',h_0'}$. We also have

$$\begin{aligned} Hm' \circ H\ell_0 &= (h')^{-1} \circ h_0' \circ H\ell_0 \\ &= (h')^{-1} \circ f \circ h_0 \\ &= (h')^{-1} \circ f \circ h \circ h^{-1} \circ h_0 \\ &= H\ell \circ Hm \end{aligned}$$

and hence $m' \circ \ell_0 = \ell \circ m$ since H is fully faithful. Finally, we can compute

$$\begin{aligned} g_0 \circ k_{a,h} &= k_{a_0',h_0'} \circ F\ell_0 \circ (k_{a_0,h_0})^{-1} \circ k_{a,h} \\ &= k_{a_0',h_0'} \circ F\ell_0 \circ Fm^{-1} \\ &= k_{a_0',h_0'} \circ (Fm')^{-1} \circ F\ell \\ &= k_{a',h'} \circ F\ell. \end{aligned}$$

Whew! We've shown that Y_f is inhabited. To show it is contractible, since hom-sets are sets, it thankfully suffices to take another $g_1 : \hom_C(Gb, Gb')$ satisfying (v) and show $g_0 = g_1$. However, we still have our specified $a_0, h_0, a_0', h_0', \ell_0$ around, and (v) implies both g_0 and g_1 must be equal to $k_{a_0',h_0'} \circ F\ell_0 \circ (k_{a_0,h_0})^{-1}$.

This completes the proof that Y_f is contractible for each $b, b' : B$ and $f : \hom_B(b, b')$. Therefore, there is a function assigning to each such f its unique inhabitant; denote this function $G_{b,b'} : \hom_B(b, b') \to \hom_C(Gb, Gb')$. The proof that G is a functor is straightforward; in each case we can choose a, h and apply (v).

Finally, for any $a_0 : A$, defining $c :\equiv Fa_0$ and $k_{a,h} :\equiv Fg$, where $g : \hom_A(a, a_0)$ is the unique isomorphism with $Hg = h$, gives an element of X_{Ha_0}. Thus, it is equal to the specified one; hence $GHa = Fa$. Similarly, for $f : \hom_A(a_0, a_0')$ we can define an element of Y_{Hf} by transporting along these equalities, which must therefore be equal to the specified one. Hence, we have $GH = F$, and thus $GH \cong F$ as desired. □

Therefore, if a precategory A admits a weak equivalence functor $A \to \hat{A}$, then that is its "reflection" into categories: any functor from A into a category will factor essentially uniquely through $\widehat{A}$. We now give two constructions of such a weak equivalence.

Theorem 9.9.5. *For any precategory A, there is a category $\widehat{A}$ and a weak equivalence $A \to \widehat{A}$.*

First proof. Let $\widehat{A}_0 :\equiv \left\{ F : \mathcal{S}et^{A^{\mathrm{op}}} \;\middle|\; \left\| \textstyle\sum_{(a:A)} (\mathbf{y}a \cong F) \right\| \right\}$, with hom-sets inherited from $\mathcal{S}et^{A^{\mathrm{op}}}$. Then the inclusion $\widehat{A} \to \mathcal{S}et^{A^{\mathrm{op}}}$ is fully faithful and an embedding on objects. Since $\mathcal{S}et^{A^{\mathrm{op}}}$ is a category (by Theorem 9.2.5, since $\mathcal{S}et$ is so by univalence), $\widehat{A}$ is also a category.

Let $A \to \widehat{A}$ be the Yoneda embedding. This is fully faithful by Corollary 9.5.6, and essentially surjective by definition of $\widehat{A}_0$. Thus it is a weak equivalence. □

This proof is very slick, but it has the drawback that it increases universe level. If A is a category in a universe $\mathcal{U}$, then in this proof $\mathcal{S}et$ must be at least as large as $\mathcal{S}et_{\mathcal{U}}$. Then $\mathcal{S}et_{\mathcal{U}}$ and $(\mathcal{S}et_{\mathcal{U}})^{A^{\mathrm{op}}}$ are not themselves categories in $\mathcal{U}$, but only in a higher universe, and *a priori* the same is true of $\widehat{A}$. One could imagine a resizing axiom that could deal with this, but it is also possible to give a direct construction using higher inductive types.

Second proof. We define a higher inductive 1-type $\widehat{A}_0$ with the following constructors:

- A function $i : A_0 \to \widehat{A}_0$.
- For each $a, b : A$ and $e : a \cong b$, an equality $je : ia = ib$.
- For each $a, b : A$ and $p : a = b$, an equality $j(\mathsf{idtoiso}(p)) = i(p)$.
- For each $a : A$, an equality $j(1_a) = \mathsf{refl}_{ia}$.
- For each $(a, b, c : A)$, $(f : a \cong b)$, and $(g : b \cong c)$, an equality $j(g \circ f) = j(g) \cdot j(f)$.

This will be the type of objects of $\widehat{A}$; we now build all the rest of the structure. (The following proof is of the sort that could benefit a lot from the help of a proof assistant: it is wide and shallow with many short cases to consider, and a large part of the work consists of writing down what needs to be checked.)

Step 1: We define a family $\hom_{\widehat{A}} : \widehat{A}_0 \to \widehat{A}_0 \to \mathsf{Set}$ by double induction on $\widehat{A}_0$, which is possible since Set is a 1-type. When x and y are of the form ia and ib, we take $\hom_{\widehat{A}}(ia, ib) :\equiv \hom_A(a, b)$. It remains to consider all the other possible pairs of constructors.

Let us keep $x = ia$ fixed at first. If y varies along the identity $je : ib = ib'$, for some $e : b \cong b'$, we require an identity $\hom_A(a, b) = \hom_A(a, b')$. By univalence, it suffices to give an equivalence $\hom_A(a, b) \simeq \hom_A(a, b')$. We take this to be the function $(e \circ -) : \hom_A(a, b) \to \hom_A(a, b')$. To see that this is an equivalence, we give its inverse as $(e^{-1} \circ -)$, with witnesses to inversion coming from the fact that e^{-1} is the inverse of e in A.

Next, as y varies along the identity $j(\mathsf{idtoiso}(p)) = i(p)$, for $p : b = b'$, we require an identity $(\mathsf{idtoiso}(p) \circ -) = \hom_A(a, -)(p)$. This is immediate by induction on p.

As y varies along the identity $j(1_b) = \mathsf{refl}_{ib}$, we require an identity $(1_b \circ -) = \mathsf{refl}_{\hom_A(a,b)}$; this follows from the identity axiom $1_b \circ g = g$ of a precategory. Similarly, as y varies along the identity $j(g \circ f) = j(g) \cdot j(f)$, we require an identity $((g \circ f) \circ -) = (g \circ (f \circ -))$, which follows from associativity.

Now we consider the other constructors for x. Say that x varies along the identity $j(e) : ia = ia'$, for some $e : a \cong a'$; we again must deal with all the constructors for y. If y is ib, then we require an identity $\hom_A(a, b) = \hom_A(a', b)$. By univalence, this may come from an equivalence, and for this we can use $(- \circ e^{-1})$, with inverse $(- \circ e)$.

Still with x varying along $j(e)$, suppose now that y also varies along $j(f)$ for some $f : b \cong b'$. Then we need to know that the two concatenated identities

$$\begin{aligned} &\hom_A(a,b) = \hom_A(a',b) = \hom_A(a',b') \qquad \text{and} \\ &\hom_A(a,b) = \hom_A(a,b') = \hom_A(a',b') \end{aligned}$$

are identical. This follows from associativity: $(f \circ -) \circ e^{-1} = f \circ (- \circ e^{-1})$. The rest of the constructors for y are trivial, since they are 2-fold equalities in sets.

For the last three constructors of x, all but the first constructor for y is likewise trivial. When x varies along the equality $j(\mathsf{idtoiso}(p)) = i(p)$ for $p : a = a'$ and y is ib, we require $(- \circ \mathsf{idtoiso}(p) = \hom_A(-,b)(p)$, which follows by induction on p. Finally, when x varies along $j(1_a) = \mathsf{refl}_{ia}$, we use the identity axiom again, and when x varies along $j(g \circ f) = j(g) \cdot j(f)$, we use associativity again. This completes the construction of $\hom_{\widehat{A}} : \widehat{A}_0 \to \widehat{A}_0 \to \mathsf{Set}$.

Step 2: We give the precategory structure on $\widehat{A}$, always by induction on $\widehat{A}_0$. We are now eliminating into sets (the hom-sets of $\widehat{A}$), so all but the first two constructors are trivial to deal with.

For identities, if x is ia then we have $\hom_{\widehat{A}}(x,x) \equiv \hom_A(a,a)$ and we define $1_x :\equiv 1_{ia}$. If x varies along je for $e : a \cong a'$, we must show that $je_*(1_{ia}) = 1_{ia'}$. Here the transport is with respect to the type family $x \mapsto \hom_{\widehat{A}}(x,x)$. But by definition of $\hom_{\widehat{A}}$, transporting along je is given by composing with e and e^{-1}, and we have $e \circ 1_{ia} \circ e^{-1} = 1_{ia'}$.

For composition, if x, y, z are ia, ib, ic respectively, then $\hom_{\widehat{A}}$ reduces to $\hom_A$ and we can define composition in $\widehat{A}$ to be composition in A. And when x, y, or z varies along je, then we verify the following equalities:

$$\begin{aligned} e \circ (g \circ f) &= (e \circ g) \circ f \\ g \circ f &= (g \circ e^{-1}) \circ (e \circ f) \\ (g \circ f) \circ e^{-1} &= g \circ (f \circ e^{-1}) \end{aligned}$$

Finally, the associativity and unitality axioms are mere propositions, so all constructors except the first are trivial. But in that case, we have the corresponding axioms in A.

Step 3: We show that $\widehat{A}$ is a category. That is, we must show that for all $x, y : \widehat{A}$, the function $\mathsf{idtoiso} : (x = y) \to (x \cong y)$ is an equivalence. First we define, for all $x, y : \widehat{A}$, a function $k_{x,y} : (x \cong y) \to (x = y)$ by induction. As before, since our goal is a set, it suffices to deal with the first two constructors.

When x and y are ia and ib respectively, we have $\hom_{\widehat{A}}(ia, ib) \equiv \hom_A(a,b)$, with composition and identities inherited as well, so that $(ia \cong ib)$ is equivalent to $(a \cong b)$. But now we have the constructor $j : (a \cong b) \to (ia = ib)$.

Next, if y varies along $j(e)$ for some $e : b \cong b'$, we must show that for $f : a \cong b$ we have $j(j(e)_*(f)) = j(e) \cdot j(f)$. But by definition of $\hom_{\widehat{A}}$ on equalities, transporting along $j(e)$ is equivalent to postcomposing with e, so this equality follows from the last constructor of $\widehat{A}_0$. The remaining case when x varies along $j(e)$ for $e : a \cong a'$ is similar. This completes the definition of $k : \prod_{(x,y:\widehat{A}_0)} (x \cong y) \to (x = y)$.

Now one thing we must show is that if $p : x = y$, then $k(\mathsf{idtoiso}(p)) = p$. By induction on p, we may assume it is refl_x, and hence $\mathsf{idtoiso}(p) \equiv 1_x$. Now we argue by induction on $x : \widehat{A}_0$, and since our goal is a mere proposition (since $\widehat{A}_0$ is a 1-type), all constructors except the first are trivial. But if x is ia, then $k(1_{ia}) \equiv j(1_a)$, which is equal to refl_{ia} by the penultimate constructor of $\widehat{A}_0$.

To complete the proof that $\widehat{A}$ is a category, we must show that if $f : x \cong y$, then $\mathsf{idtoiso}(k(f)) = f$. By induction we may assume that x and y are ia and ib respectively, in which case f must arise from an isomorphism $g : a \cong b$ and we have $k(f) \equiv j(g)$. However, for any p we have $\mathsf{idtoiso}(p) = p_*(1)$, so in particular $\mathsf{idtoiso}(j(g)) = j(g)_*(1_{ia})$. And by definition of $\hom_{\widehat{A}}$ on equalities, this is given by composing 1_{ia} with the equivalence g, hence is equal to g.

Note the similarity of this step to the encode/decode arguments used in §§2.12 and 2.13 and Chapter 8. Once again we are characterizing the identity types of a higher inductive type (here, $\widehat{A}_0$) by defining recursively a family of codes (here, $(x, y) \mapsto (x \cong y)$) and encoding and decoding functions by induction on $\widehat{A}_0$ and on paths.

Step 4: We define a weak equivalence $I : A \to \widehat{A}$. We take $I_0 :\equiv i : A_0 \to \widehat{A}_0$, and by construction of $\hom_{\widehat{A}}$ we have functions $I_{a,b} : \hom_A(a, b) \to \hom_{\widehat{A}}(Ia, Ib)$ forming a functor $I : A \to \widehat{A}$. This functor is fully faithful by construction, so it remains to show it is essentially surjective. That is, for all $x : \widehat{A}$ we want there to merely exist an $a : A$ such that $Ia \cong x$. As always, we argue by induction on x, and since the goal is a mere proposition, all but the first constructor are trivial. But if x is ia, then of course we have $a : A$ and $Ia \equiv ia$, hence $Ia \cong ia$. (Note that if we were trying to prove I to be *split* essentially surjective, we would be stuck, because we know nothing about equalities in A_0 and thus have no way to deal with any further constructors.) □

We call the construction $A \mapsto \widehat{A}$ the **Rezk completion**, although there is also an argument (coming from higher topos semantics) for calling it the **stack completion**.

We have seen that most precategories arising in practice are categories, since they are constructed from $\mathcal{S}et$, which is a category by the univalence axiom. However, there are a few cases in which the Rezk completion is necessary to obtain a category.

Example 9.9.6. Recall from Example 9.1.17 that for any type X there is a pregroupoid with X as its type of objects and $\hom(x, y) :\equiv \|x = y\|_0$. Its Rezk completion is the *fundamental groupoid* of X. Recalling that groupoids are equivalent to 1-types, it is not hard to identify this groupoid with $\|X\|_1$.

Example 9.9.7. Recall from Example 9.1.18 that there is a precategory whose type of objects is $\mathcal{U}$ and with $\hom(X, Y) :\equiv \|X \to Y\|_0$. Its Rezk completion may be called the **homotopy category of types**. Its type of objects can be identified with $\|\mathcal{U}\|_1$ (see Exercise 9.9).

The Rezk completion also allows us to show that the notion of "category" is determined by the notion of "weak equivalence of precategories". Thus, insofar as the latter is inevitable, so is the former.

Theorem 9.9.8. *A precategory C is a category if and only if for every weak equivalence of precategories $H : A \to B$, the induced functor $(- \circ H) : C^B \to C^A$ is an isomorphism of precategories.*

Proof. "Only if" is Theorem 9.9.4. In the other direction, let H be $I : A \to \widehat{A}$. Then since $(- \circ I)_0$ is an equivalence, there exists $R : \widehat{A} \to A$ such that $RI = 1_A$. Hence $IRI = I$, but again since

$(- \circ I)_0$ is an equivalence, this implies $IR = 1_{\widehat{A}}$. By Lemma 9.4.9(iii), I is an isomorphism of precategories. But then since $\widehat{A}$ is a category, so is A. □

Notes

The original definition of categories, of course, was within set theoretic-foundations, so that the collection of objects of a category formed a set (or, for large categories, a class). Over time, it became clear that all "category-theoretic" properties of objects were invariant under isomorphism, and that equality of objects in a category was not usually a very useful notion. Numerous authors [Bla79, Fre76, Mak95, Mak01] discovered that a dependently typed logic enabled formulating the definition of category without invoking any notion of equality for objects, and that the statements provable in this logic are precisely the "category-theoretic" ones that are invariant under isomorphism.

Although most of category theory appears to be invariant under isomorphism of objects and under equivalence of categories, there are some interesting exceptions, which have led to philosophical discussions about what it means to be "category-theoretic". For instance, Example 9.6.3 was brought up by Peter May on the categories mailing list in May 2010, as a case where it matters that two categories (defined as usual in set theory) are isomorphic rather than only equivalent. The case of †-categories was also somewhat confounding to those advocating an isomorphism-invariant version of category theory, since the "correct" notion of sameness between objects of a †-category is not ordinary isomorphism but *unitary* isomorphism.

The fact that categories satisfying the "saturation" or "univalence" principle as in Definition 9.1.6 are a good notion of category in univalent foundations occurred independently to Voevodsky, Shulman, and perhaps others around the same time, and was formalized by Ahrens and Kapulkin [AKS13]. This framework puts all the above examples in a unified context: some precategories are categories, others are strict categories, and so on. A general theorem that "isomorphism implies equality" for a large class of algebraic structures (assuming the univalence axiom) was proven by Coquand and Danielsson; the formulation of the structure identity principle in §9.8 is due to Aczel.

Independently of philosophical considerations about category theory, Rezk [Rez01] discovered that when defining a notion of $(\infty,1)$-category, it was very convenient to use not merely a *set* of objects with spaces of morphisms between them, but a *space* of objects incorporating all the equivalences and homotopies between them. This yields a very well-behaved sort of model for $(\infty,1)$-categories as particular simplicial spaces, which Rezk called *complete Segal spaces*. One especially good aspect of this model is the analogue of Lemma 9.4.14: a map of complete Segal spaces is an equivalence just when it is a levelwise equivalence of simplicial spaces.

When interpreted in Voevodsky's simplicial set model of univalent foundations, our precategories are similar to a truncated analogue of Rezk's "Segal spaces", while our categories correspond to his "complete Segal spaces". Strict categories correspond instead to (a weakened and truncated version of) what are called "Segal categories". It is known that Segal categories and complete Segal spaces are equivalent models for $(\infty,1)$-categories (see e.g. [Ber09]), so that in the simplicial set model, categories and strict categories yield "equivalent" category theories—although as we have seen, the former still have many advantages. However, in the more general categorical semantics of a higher topos, a strict category corresponds to an internal category (in the traditional sense) in the corresponding 1-topos of sheaves, while a category corresponds to a *stack*. The latter are generally a more appropriate sort of "category" relative to a topos.

In Rezk's context, what we have called the "Rezk completion" corresponds to fibrant replacement in the model category for complete Segal spaces. Since this is built using a transfinite induction argument, it most closely matches our second construction as a higher inductive type. However, in higher topos models of homotopy type theory, the Rezk completion corresponds to *stack completion*, which can be constructed either with a transfinite induction [JT91] or using a Yoneda embedding [Bun79].

Exercises

Exercise 9.1. For a precategory A and $a : A$, define the **slice precategory** A/a. Show that if A is a category, so is A/a.

Exercise 9.2. For any set X, prove that the slice category $\mathcal{S}et/X$ is equivalent to the functor category $\mathcal{S}et^X$, where in the latter case we regard X as a discrete category.

Exercise 9.3. Prove that a functor is an equivalence of categories if and only if it is a *right* adjoint whose unit and counit are isomorphisms.

Exercise 9.4. Define a **pre-2-category** to consist of the structure formed by precategories, functors, and natural transformations in §9.2. Similarly, define a **pre-bicategory** by replacing the equalities in Lemmas 9.2.9 and 9.2.11 with natural isomorphisms satisfying analogous coherence conditions. Define a function from pre-2-categories to pre-bicategories, and show that it becomes an equivalence when restricted and corestricted to those whose hom-precategories are categories.

Exercise 9.5. Define a **2-category** to be a pre-2-category satisfying a condition analogous to that of Definition 9.1.6. Verify that the pre-2-category of categories $\mathcal{C}at$ is a 2-category. How much of this chapter can be done internally to an arbitrary 2-category?

Exercise 9.6. Define a 2-category whose objects are 1-types, whose morphisms are functions, and whose 2-morphisms are homotopies. Prove that it is equivalent, in an appropriate sense, to the full sub-2-category of $\mathcal{C}at$ spanned by the *groupoids* (categories in which every arrow is an isomorphism).

Exercise 9.7. Recall that a **strict category** is a precategory whose type of objects is a set. Prove that the pre-2-category of strict categories is equivalent to the following pre-2-category.

- Its objects are categories A equipped with a surjection $p_A : A'_0 \to A_0$, where A'_0 is a set.
- Its morphisms are functors $F : A \to B$ equipped with a function $F'_0 : A'_0 \to B'_0$ such that $p_B \circ F'_0 = F_0 \circ p_A$.
- Its 2-morphisms are simply natural transformations.

Exercise 9.8. Define the pre-2-category of †-categories, which has †-structures on its hom-precategories. Show that two †-categories are equal precisely when they are "unitarily equivalent" in a suitable sense.

Exercise 9.9. Prove that a function $X \to Y$ is an equivalence if and only if its image in the homotopy category of Example 9.9.7 is an isomorphism. Show that the type of objects of this category is $\|\mathcal{U}\|_1$.

Exercise 9.10. Construct the †-Rezk completion of a †-precategory into a †-category, and give it an appropriate universal property.

Exercise 9.11. Using fundamental pregroupoids (Examples 9.1.17 and 9.9.6) and the Rezk completion (§9.9), give a different proof of van Kampen's theorem (§8.7).

Chapter 10

Set theory

Our conception of sets as types with particularly simple homotopical character, cf. §3.1, is quite different from the sets of Zermelo–Fraenkel set theory, which form a cumulative hierarchy with an intricate nested membership structure. For many mathematical purposes, the homotopy-theoretic sets are just as good as the Zermelo–Fraenkel ones, but there are important differences.

We begin this chapter in §10.1 by showing that the category $\mathcal{S}et$ has (most of) the usual properties of the category of sets. In constructive, predicative, univalent foundations, it is a "ΠW-pretopos"; whereas if we assume propositional resizing (§3.5) it is an elementary topos, and if we assume LEM and AC then it is a model of Lawvere's Elementary Theory of the Category of Sets. This is sufficient to ensure that the sets in homotopy type theory behave mostly like sets as used by most mathematicians outside of set theory.

In the rest of the chapter, we investigate some subjects that traditionally belong to "set theory". In §§10.2–10.4 we study cardinal and ordinal numbers. These are traditionally defined in set theory using the global membership relation, but we will see that the univalence axiom enables an equally convenient, more "structural" approach.

Finally, in §10.5 we consider the possibility of constructing *inside* of homotopy type theory a cumulative hierarchy of sets, equipped with a binary membership relation akin to that of Zermelo–Fraenkel set theory. This combines higher inductive types with ideas from the field of Algebraic Set Theory.

In this chapter we will often use the traditional logical notation described in §3.7.

10.1 The category of sets

Recall that in Chapter 9 we defined the category $\mathcal{S}et$ to consist of all 0-types (in some universe $\mathcal{U}$) and maps between them, and observed that it is a category (not just a precategory). We consider successively the levels of structure which $\mathcal{S}et$ possesses.

10.1.1 Limits and colimits

Since sets are closed under products, the universal property of products in Theorem 2.15.2 shows immediately that $\mathcal{S}et$ has finite products. In fact, infinite products follow just as easily from the

equivalence

$$\left(X \to \prod_{a:A} B(a)\right) \simeq \left(\prod_{a:A} (X \to B)\right).$$

And we remarked already that the pullback of $f : A \to C$ and $g : B \to C$ can be defined as $\sum_{(a:A)} \sum_{(b:B)} f(a) = g(b)$; this is a set if A, B, C are and inherits the correct universal property. Thus, $\mathcal{S}et$ is a *complete* category in the obvious sense.

Since sets are closed under $+$ and contain $\mathbf{0}$, $\mathcal{S}et$ has finite coproducts. Similarly, since $\sum_{(a:A)} B(a)$ is a set whenever A and each $B(a)$ are, it yields a coproduct of the family B in $\mathcal{S}et$. Finally, we showed in §7.4 that pushouts exist in n-types, which includes $\mathcal{S}et$ in particular. Thus, $\mathcal{S}et$ is also *cocomplete*.

10.1.2 Images

Next, we show that $\mathcal{S}et$ is a **regular category**, i.e.:

(i) $\mathcal{S}et$ is finitely complete.

(ii) The kernel pair $\mathsf{pr}_1, \mathsf{pr}_2 : (\sum_{(x,y:A)} f(x) = f(y)) \to A$ of any function $f : A \to B$ has a coequalizer.

(iii) Pullbacks of regular epimorphisms are again regular epimorphisms.

Recall that a **regular epimorphism** is a morphism that is the coequalizer of *some* pair of maps. Thus in (iii) the pullback of a coequalizer is required to again be a coequalizer, but not necessarily of the pulled-back pair.

The obvious candidate for the coequalizer of the kernel pair of $f : A \to B$ is its *image*, as defined in §7.6. Recall that we defined $\mathsf{im}(f) :\equiv \sum_{(b:B)} \left\| \mathsf{fib}_f(b) \right\|$, with functions $\tilde{f} : A \to \mathsf{im}(f)$ and $i_f : \mathsf{im}(f) \to B$ defined by

$$\begin{aligned} \tilde{f} &:\equiv \lambda a. \left(f(a), \left|(a, \mathsf{refl}_{f(a)})\right|\right) \\ i_f &:\equiv \mathsf{pr}_1 \end{aligned}$$

fitting into a diagram:

$$\sum_{(x,y:A)} f(x) = f(y) \underset{\mathsf{pr}_2}{\overset{\mathsf{pr}_1}{\rightrightarrows}} A \xrightarrow{\tilde{f}} \mathsf{im}(f), \quad A \xrightarrow{f} B, \quad \mathsf{im}(f) \overset{i_f}{\hookrightarrow} B$$

Recall that a function $f : A \to B$ is called *surjective* if $\forall(b : B). \left\| \mathsf{fib}_f(b) \right\|$, or equivalently $\forall(b : B). \exists(a : A). f(a) = b$. A function $f : A \to B$ between sets is called *injective* if $\forall(a, a' : A). f(a) = f(a') \Rightarrow a = a'$, or equivalently if each of its fibers is a mere proposition. Since these are the (-1)-connected and (-1)-truncated maps in the sense of Chapter 7, the general theory there implies that $\tilde{f}$ above is surjective and i_f is injective, and that this factorization is stable under pullback.

We now identify surjectivity and injectivity with the appropriate category-theoretic notions. First we observe that categorical monomorphisms and epimorphisms have a slightly stronger equivalent formulation.

Lemma 10.1.1. *For a morphism $f : \hom_A(a,b)$ in a category A, the following are equivalent.*

*(i) f is a **monomorphism**: for all $x : A$ and $g,h : \hom_A(x,a)$, if $f \circ g = f \circ h$ then $g = h$.*
(ii) (If A has products) the diagonal map $A \to A \times A$ is an isomorphism.
(iii) For all $x : A$ and $k : \hom_A(x,b)$, the type $\sum_{(h:\hom_A(x,a))} k = f \circ h$ is a mere proposition.
(iv) For all $x : A$ and $g : \hom_A(x,a)$, the type $\sum_{(h:\hom_A(x,a))} f \circ g = f \circ h$ is contractible.

Proof. The equivalence of conditions (i) and (ii) is standard category theory. Now consider the function $(f \circ -) : \hom_A(x,a) \to \hom_A(x,b)$ between sets. Condition (i) says that it is injective, while (iii) says that its fibers are mere propositions; hence they are equivalent. And (iii) implies (iv) by taking $k :\equiv f \circ g$ and recalling that an inhabited mere proposition is contractible. Finally, (iv) implies (i) since if $p : f \circ g = f \circ h$, then (g, refl) and (h,p) both inhabit the type in (iv), hence are equal and so $g = h$. □

Lemma 10.1.2. *A function $f : A \to B$ between sets is injective if and only if it is a monomorphism in $\mathcal{S}et$.*

Proof. Left to the reader. □

Of course, an **epimorphism** is a monomorphism in the opposite category. We now show that in $\mathcal{S}et$, the epimorphisms are precisely the surjections, and also precisely the coequalizers (regular epimorphisms).

The coequalizer of a pair of maps $f,g : A \to B$ in $\mathcal{S}et$ is defined as the 0-truncation of a general (homotopy) coequalizer. For clarity, we may call this the **set-coequalizer**. It is convenient to express its universal property as follows.

Lemma 10.1.3. *Let $f,g : A \to B$ be functions between sets A and B. The set-coequalizer $c_{f,g} : B \to Q$ has the property that, for any set C and any $h : B \to C$ with $h \circ f = h \circ g$, the type*

$$\sum_{k:Q\to C} k \circ c_{f,g} = h$$

is contractible.

Lemma 10.1.4. *For any function $f : A \to B$ between sets, the following are equivalent:*

(i) f is an epimorphism.
(ii) Consider the pushout diagram

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ \downarrow & & \downarrow{\scriptstyle \iota} \\ \mathbf{1} & \xrightarrow[t]{} & C_f \end{array}$$

in $\mathcal{S}et$ defining the mapping cone. Then the type C_f is contractible.

(iii) f is surjective.

Proof. Let $f : A \to B$ be a function between sets, and suppose it to be an epimorphism; we show C_f is contractible. The basic constructor of C_f corresponding to $\mathbf{1} \to C_f$ gives us an element $t : C_f$. We have to show that

$$\prod_{x:C_f} x = t.$$

Note that $x = t$ is a mere proposition, hence we can use induction on C_f: it suffices to find

$$I_0 : \prod_{b:B} \iota(b) = t$$
$$I_1 : \prod_{a:A} \alpha_1(a)^{-1} \cdot I_0(f(a)) = \mathsf{refl}_t.$$

where $\alpha_1 : \prod_{(a:A)} \iota(f(a)) = t$ is a basic constructor of C_f. Note that α_1 is a homotopy from $\iota \circ f$ to $\mathsf{const}_t \circ f$, so we find the elements

$$(\iota, \mathsf{refl}_{\iota\circ f}), (\mathsf{const}_t, \alpha_1) : \sum_{h:B\to C_f} \iota \circ f = h \circ f.$$

By the dual of Lemma 10.1.1(iv), there is a path

$$\gamma : (\iota, \mathsf{refl}_{\iota\circ f}) = (\mathsf{const}_t, \alpha_1)$$

Hence, we may define $I_0(b) :\equiv \mathsf{happly}(\mathsf{ap}_{\mathsf{pr}_1}(\gamma), b) : \iota(b) = t$. We also have

$$\mathsf{ap}_{\mathsf{pr}_2}(\gamma) : \mathsf{ap}_{\mathsf{pr}_1}(\gamma)_*\big(\mathsf{refl}_{\iota\circ f}\big) = \alpha_1.$$

This transport involves precomposition with f, which commutes with happly. Thus, from transport in path types we obtain $I_0(f(a)) = \alpha_1(a)$ for any $a : A$, which gives us I_1.

Now suppose C_f is contractible; we show f is surjective. We first construct a type family $P : C_f \to \mathsf{Prop}$ by recursion on C_f, which is valid since Prop is a set. On the point constructors, we define

$$P(t) :\equiv \mathbf{1}$$
$$P(\iota(b)) :\equiv \big\|\mathsf{fib}_f(b)\big\|.$$

To complete the construction of P, it remains to give a path $\big\|\mathsf{fib}_f(f(a))\big\| =_{\mathsf{Prop}} \mathbf{1}$ for all $a : A$. However, $\big\|\mathsf{fib}_f(f(a))\big\|$ is inhabited by $(f(a), \mathsf{refl}_{f(a)})$. Since it is a mere proposition, this means it is contractible — and thus equivalent, hence equal, to $\mathbf{1}$. This completes the definition of P. Now, since C_f is assumed to be contractible, it follows that $P(x)$ is equivalent to $P(t)$ for any $x : C_f$. In particular, $P(\iota(b)) \equiv \big\|\mathsf{fib}_f(b)\big\|$ is equivalent to $P(t) \equiv \mathbf{1}$ for each $b : B$, and hence contractible. Thus, f is surjective.

Finally, suppose $f : A \to B$ to be surjective, and consider a set C and two functions $g, h : B \to C$ with the property that $g \circ f = h \circ f$. Since f is assumed to be surjective, we have an

equivalence $B \simeq \mathsf{im}(f)$. We have the following equivalences

$$\begin{aligned}\prod_{b:B}(g(b) = h(b)) &\simeq \prod_{w:\mathsf{im}(f)} g(\mathsf{pr}_1 w) = h(\mathsf{pr}_1(w)) \\ &\simeq \prod_{(b:B)} \prod_{(a:A)} \prod_{(p:f(a)=b)} g(b) = h(b) \\ &\simeq \prod_{a:A} g(f(a)) = h(f(a)).\end{aligned}$$

By assumption, there is an element of the latter type. □

Theorem 10.1.5. *The category $\mathcal{S}et$ is regular. Moreover, surjective functions between sets are regular epimorphisms.*

Proof. It is a standard lemma in category theory that a category is regular as soon as it admits finite limits and a pullback-stable orthogonal factorization system $(\mathcal{E}, \mathcal{M})$ with $\mathcal{M}$ the monomorphisms, in which case $\mathcal{E}$ conists automatically of the regular epimorphisms. (See e.g. [Joh02, A1.3.4].) The existence of the factorization system was proved in Theorem 7.6.6. □

Lemma 10.1.6. *Pullbacks of regular epis in $\mathcal{S}et$ are regular epis.*

Proof. We showed in Theorem 7.6.9 that pullbacks of n-connected functions are n-connected. By Theorem 10.1.5, it suffices to apply this when $n = -1$. □

10.1.3 Quotients

Now that we know that $\mathcal{S}et$ is regular, to show that $\mathcal{S}et$ is exact, we need to show that every equivalence relation is effective. In other words, given an equivalence relation $R : A \to A \to \mathsf{Prop}$, there is a coequalizer c_R of the pair $\mathsf{pr}_1, \mathsf{pr}_2 : \sum_{(x,y:A)} R(x,y) \to A$ and, moreover, the pr_1 and pr_2 form the kernel pair of c_R.

We have already seen, in §6.10, two general ways to construct the quotient of a set by an equivalence relation $R : A \to A \to \mathsf{Prop}$. The first can be described as the set-coequalizer of the two projections

$$\mathsf{pr}_1, \mathsf{pr}_2 : \sum_{x,y:A} R(x,y) \to A.$$

The important property of such a quotient is the following.

Definition 10.1.7. A relation $R : A \to A \to \mathsf{Prop}$ is said to be **effective** if the square

$$\begin{array}{ccc} \sum_{(x,y:A)} R(x,y) & \xrightarrow{\mathsf{pr}_1} & A \\ {\scriptstyle \mathsf{pr}_2}\downarrow & & \downarrow{\scriptstyle c_R} \\ A & \xrightarrow[c_R]{} & A/R \end{array}$$

is a pullback.

Lemma 10.1.8. *Suppose (A, R) is an equivalence relation. Then there is an equivalence*

$$(c_R(x) = c_R(y)) \simeq R(x, y)$$

for any $x, y : A$. In other words, equivalence relations are effective.

Proof. We begin by extending R to a relation $\tilde{R} : A/R \to A/R \to \mathsf{Prop}$. After the construction of $\tilde{R}$ we will show that $\tilde{R}$ is equivalent to the identity type on A/R. We define $\tilde{R}$ by double induction on A/R (note that Prop is a set by univalence for mere propositions). We define $\tilde{R}(c_R(x), c_R(y)) :\equiv R(x, y)$. For $r : R(x, x')$ and $s : R(y, y')$, the transitivity and symmetry of R gives an equivalence from $R(x, y)$ to $R(x', y')$. This completes the definition of $\tilde{R}$. To finish the proof of the lemma, we need to show that $\tilde{R}(w, w') \simeq w = w'$ for every $w, w' : A/R$. We can do this by showing that the type $\sum_{(w':A/R)} \tilde{R}(w, w')$ is contractible for each $w : A/R$. We do this by induction. Let $x : A$. We have the element $\mathsf{pair} c_R(x), \rho(x) : \sum_{(w':A/R)} \tilde{R}(c_R(x), w')$, where ρ is the reflexivity proof of R, hence we only have to show that

$$\prod_{(w':A/R)} \prod_{(r:\tilde{R}(c_R(x),w'))} (w', r) = (c_R(x), \rho(x)),$$

which we do by induction on w'. Let $y : A$ and let $r : R(x, y)$. Then we have the path $p_R(r)^{-1} : c_R(y) = c_R(x)$. We automatically get a path from $p_R(r)^{-1} \cdot r = \rho(x)$, finishing the proof. □

The second construction of quotients, which uses impredicativity of mere propositions, is as the set of equivalence classes of R (a subset of its power set):

$$A /\!\!/ R :\equiv \{\, P : A \to \mathsf{Prop} \mid P \text{ is an equivalence class of } R \,\}$$

Note that if we regard R as a function from A to $A \to \mathsf{Prop}$, then $A /\!\!/ R$ is equivalent to $\mathsf{im}(R)$, as constructed in §10.1.2. Now in Theorem 10.1.5 we have shown that images are coequalizers. In particular, we immediately get the coequalizer diagram

$$\textstyle\sum_{(x,y:A)} R(x) = R(y) \;\underset{\mathsf{pr}_2}{\overset{\mathsf{pr}_1}{\rightrightarrows}}\; A \longrightarrow A /\!\!/ R.$$

We can use this to give an alternative proof that any equivalence relation is effective and that the two definitions of quotients agree.

Theorem 10.1.9. *For any function $f : A \to B$ between any two sets, the relation* $\ker(f) : A \to A \to \mathcal{U}$ *given by* $\ker(f, x, y) :\equiv f(x) = f(y)$ *is effective.*

Proof. We will use that $\mathsf{im}(f)$ is the coequalizer of $\mathsf{pr}_1, \mathsf{pr}_2 : \sum_{(x,y:A)} f(x) = f(y) \to A$. Note that the canonical kernel pair of the function $c_f :\equiv \lambda a.\, (f(a), \tau_1((a, \mathsf{refl}_{f(a)})))$ consists of the two projections

$$\mathsf{pr}_1, \mathsf{pr}_2 : \Big(\sum_{x,y:A} c_f(x) = c_f(y) \Big) \to A.$$

For any $x, y : A$, we have equivalences

$$\begin{aligned} c_f(x) = c_f(y) &\simeq \sum_{p:f(x)=f(y)} p \cdot \tau_1((x, \mathsf{refl}_{f(x)})) = \tau_1((y, \mathsf{refl}_{f(x)})) \\ &\simeq f(x) = f(y), \end{aligned}$$

where the last equivalence holds because $\left\|\mathsf{fib}_f(b)\right\|$ is a mere for any $b : B$. Therefore, we get that

$$\sum_{x,y:A} c_f(x) = c_f(y) \simeq \sum_{x,y:A} f(x) = f(y)$$

and hence we may conclude that $\ker f$ is an effective relation for any function f. □

Theorem 10.1.10. *Equivalence relations are effective and there is an equivalence $A/R \simeq A /\!\!/ R$.*

Proof. We need to analyze the coequalizer diagram

$$\textstyle\sum_{(x,y:A)} R(x) = R(y) \rightrightarrows A \longrightarrow A /\!\!/ R.$$

By the univalence axiom, the space $R(x) = R(y)$ is equivalent to the space of homotopies from $R(x)$ to $R(y)$, which is equivalent to $\prod_{(z:A)} R(x,z) \simeq R(y,z)$. Since R is an equivalence relation, the latter space is equivalent to $R(x,y)$. To summarize, we get that $(R(x) = R(y)) \simeq R(x,y)$, so R is effective since it is equivalent to an effective relation. Also, the diagram

$$\textstyle\sum_{(x,y:A)} R(x,y) \rightrightarrows A \longrightarrow A /\!\!/ R.$$

is a coequalizer diagram. Since coequalizers are unique up to equivalence, it follows that $A/R \simeq A /\!\!/ R$. □

We finish this section by mentioning a possible third construction of the quotient of a set A by an equivalence relation R. One considers the precategory with objects A and hom-sets R. The type of objects of the Rezk completion (see §9.9) of this precategory will be the quotient. The reader is invited to check the details of this construction.

10.1.4 $\mathcal{S}et$ is a ΠW-pretopos

The notion of a ΠW-*pretopos* — that is, a locally cartesian closed category with finite coproducts, effective equivalence relations, and W-types — is intended as a "predicative" notion of topos, i.e. a category of "predicative sets", which can serve the purpose for constructive mathematics that the usual category of sets does for classical mathematics.

Typically, in constructive type theory, one resorts to an external construction of "setoids" — an exact completion — to obtain a category with such closure properties. In particular, the well-behaved quotients are required for many constructions in mathematics that usually involve (non-constructive) power sets. It is noteworthy that univalent foundations provides these constructions *internally* (via higher inductive types), without requiring such external constructions. This represents a powerful advantage of our approach, as we shall see in subsequent examples.

Theorem 10.1.11. *The category $\mathcal{S}et$ is a* ΠW*-pretopos.*

Proof. We have an initial object 0 and finite, disjoint sums $A + B$. These are stable under pullback, simply because pullback has a right adjoint. Indeed, $\mathcal{S}et$ is locally cartesian closed, since for any map $f : A \to B$ between sets, the "fibrant replacement" $\sum_{(a:A)} f(a) = b$ is equivalent to

A (over B), and we have $\prod$-types for the replacement. We've just shown that $\mathcal{S}et$ is regular (Theorem 10.1.5) and that quotients are effective (Lemma 10.1.8). We thus have an LCC pretopos. Finally, since the n-types are closed under the formation of W-types (§5.3), we see that $\mathcal{S}et$ is a ΠW-pretopos. □

One naturally wonders what, if anything, prevents $\mathcal{S}et$ from being an (elementary) topos? In addition to the structure already mentioned, a topos has a *subobject classifier*: a pointed object classifying (equivalence classes of) monomorphisms. (In fact, in the presence of a subobject classifier, things become somewhat simpler: one merely needs cartesian closure in order to get the colimits). In homotopy type theory, univalence implies that the type $\mathsf{Prop} :\equiv \sum_{(X:\mathcal{U})} \mathsf{isProp}(X)$ does classify monos (by an argument similar to §4.8), but in general it is as large as the ambient universe $\mathcal{U}$. Thus, it is a "set" in the sense of being a 0-type, but it is not "small" in the sense of being an object of $\mathcal{U}$, hence not an object of the category $\mathcal{S}et$. However, if we assume an appropriate form of propositional resizing (see §3.5), then we may find a small version of Prop, so that $\mathcal{S}et$ becomes an elementary topos.

Theorem 10.1.12. *If there is a type $\Omega : \mathcal{U}$ of all mere propositions, then the category $\mathcal{S}et_{\mathcal{U}}$ is then an elementary topos.*

A sufficient condition for this is the law of excluded middle, in the "mere-propositional" form that we have called LEM; for then we have $\mathsf{Prop} = \mathbf{2}$, which *is* small, and which then also classifies all mere propositions. Moreover, in topos theory a well-known sufficient condition for LEM is the axiom of choice, which is of course often assumed as an axiom in classical set theory. In the next section, we briefly investigate the relation between these conditions in our setting.

10.1.5 The axiom of choice implies excluded middle

We begin with the following lemma.

Lemma 10.1.13. *If A is a mere proposition then its supension $\Sigma(A)$ is a set, and A is equivalent to $\mathsf{N} =_{\Sigma(A)} \mathsf{S}$.*

Proof. To show that $\Sigma(A)$ is a set, we define a family $P : \Sigma(A) \to \Sigma(A) \to \mathcal{U}$ with the property that $P(x, y)$ is a mere proposition for each $x, y : \Sigma(A)$, and which is equivalent to $\mathsf{Id}_{\Sigma(A)}$. We make the following definitions:

$$P(\mathsf{N},\mathsf{N}) :\equiv \mathbf{1} \qquad P(\mathsf{S},\mathsf{N}) :\equiv A$$
$$P(\mathsf{N},\mathsf{S}) :\equiv A \qquad P(\mathsf{S},\mathsf{S}) :\equiv \mathbf{1}.$$

We have to check that the definition preserves paths. Given any $a : A$, there is a meridian $\mathsf{merid}(a) : \mathsf{N} = \mathsf{S}$, so we should also have

$$P(\mathsf{N},\mathsf{N}) = P(\mathsf{N},\mathsf{S}) = P(\mathsf{S},\mathsf{N}) = P(\mathsf{S},\mathsf{S}).$$

But since A is inhabited by a, it is equivalent to $\mathbf{1}$, so we have

$$P(\mathsf{N},\mathsf{N}) \simeq P(\mathsf{N},\mathsf{S}) \simeq P(\mathsf{S},\mathsf{N}) \simeq P(\mathsf{S},\mathsf{S}).$$

The univalence axiom turns these into the desired equalities. Also, $P(x,y)$ is a mere proposition for all $x, y : \Sigma(A)$, which is proved by induction on x and y, and using the fact that being a mere proposition is a mere proposition.

Note that P is a reflexive relation. Therefore we may apply Theorem 7.2.2, so it suffices to construct $\tau : \prod_{(x,y:\Sigma(A))} P(x,y) \to (x = y)$. We do this by a double induction. When x is N, we define $\tau(\mathsf{N})$ by

$$\tau(\mathsf{N},\mathsf{N},\_) :\equiv \mathsf{refl}_{\mathsf{N}} \qquad \text{and} \qquad \tau(\mathsf{N},\mathsf{S},a) :\equiv \mathsf{merid}(a).$$

If A is inhabited by a then $\mathsf{merid}(a) : \mathsf{N} = \mathsf{S}$ so we also need $\mathsf{merid}(a)_*(\tau(\mathsf{N},\mathsf{N})) = \tau(\mathsf{N},\mathsf{S})$. This we get by function extensionality using the fact that, for all $x : A$,

$$\mathsf{merid}(a)_*(\tau(\mathsf{N},\mathsf{N},x)) = \tau(\mathsf{N},\mathsf{N},x) \cdot \mathsf{merid}(a)^{-1} \equiv$$
$$\mathsf{refl}_{\mathsf{N}} \cdot \mathsf{merid}(a) = \mathsf{merid}(a) = \mathsf{merid}(x) \equiv \tau(\mathsf{N},\mathsf{S},x).$$

In a symmetric fashion we may define $\tau(\mathsf{S})$ by

$$\tau(\mathsf{S},\mathsf{N},a) :\equiv \mathsf{merid}(a)^{-1} \qquad \text{and} \qquad \tau(\mathsf{S},\mathsf{S},\_) :\equiv \mathsf{refl}_{\mathsf{S}}.$$

To complete the construction of τ, we need to check $\mathsf{merid}(a)_*(\tau(\mathsf{N})) = \tau(\mathsf{S})$, given any $a : A$. The verification proceeds much along the same lines by induction on the second argument of τ.

It remains to show that the mere propositions A and $\mathsf{N} =_{\Sigma(A)} \mathsf{S}$ are equivalent, but this is very easy so we leave it as exercise. □

Theorem 10.1.14 (Diaconescu)**.** *The axiom of choice implies the law of excluded middle.*

Proof. We use the equivalent form of the axiom of choice given in Lemma 3.8.2. Consider a mere proposition A. The function $f : \mathbf{2} \to \Sigma(A)$ defined by $f(1_{\mathbf{2}}) :\equiv \mathsf{S}$ and $f(0_{\mathbf{2}}) :\equiv \mathsf{N}$ is surjective. Indeed, we have $(0_{\mathbf{2}}, \mathsf{refl}_{\mathsf{N}}) : \mathsf{fib}_f(\mathsf{N})$ and $(1_{\mathbf{2}}, \mathsf{refl}_{\mathsf{S}}) : \mathsf{fib}_f(\mathsf{S})$. Since $\|\mathsf{fib}_f(x)\|$ is a mere proposition, by induction the claimed surjectivity follows.

By Lemma 10.1.13 the supension $\Sigma(A)$ is a set, so by the axiom of choice there merely exists a section $g : \Sigma(A) \to \mathbf{2}$ of f. As equality on $\mathbf{2}$ is decidable we get

$$(g(f(0_{\mathbf{2}})) = g(f(1_{\mathbf{2}}))) + \neg(g(f(0_{\mathbf{2}})) = g(f(1_{\mathbf{2}}))),$$

and, since g is a section of f,

$$(f(0_{\mathbf{2}}) = f(1_{\mathbf{2}})) + \neg(f(0_{\mathbf{2}}) = f(1_{\mathbf{2}})).$$

Now we see that it suffices for A to be equivalent to $f(0_{\mathbf{2}}) = f(1_{\mathbf{2}})$, which is the case by Lemma 10.1.13. □

Theorem 10.1.15. *If the axiom of choice holds then the category $\mathcal{S}et$ is a well-pointed boolean elementary topos with choice.*

Proof. Since AC implies LEM, we have a Boolean elementary topos with choice by Theorem 10.1.12 and the remark following it. We leave the proof of well-pointedness as an exercise for the reader. □

Remark 10.1.16. The conditions on a category mentioned in the theorem are known as Lawvere's axioms for the *Elementary Theory of the Category of Sets* [Law05].

10.2 Cardinal numbers

Definition 10.2.1. The **type of cardinal numbers** is the 0-truncation of Set:

$$\mathsf{Card} :\equiv \|\mathsf{Set}\|_0$$

Thus, a **cardinal number**, or **cardinal**, is an inhabitant of $\mathsf{Card} \equiv \|\mathsf{Set}\|_0$.

Remark 10.2.2. Technically, there is a separate type "Card" associated to each universe "$\mathcal{U}$", but with these conventions we can state theorems beginning with "for all cardinal numbers…" and give them exactly the same sort of meaning as those beginning "for all types…".

If A is a set, we write $|A|$ for its image under the canonical projection $\mathsf{Set} \to \mathsf{Card}$. Of course, by definition, Card is a set. It also inherits the structure of a semiring from Set.

Definition 10.2.3. The operation of **cardinal addition**

$$\_ + \_ : \mathsf{Card} \to \mathsf{Card} \to \mathsf{Card}$$

is defined by induction on truncation:

$$|A| + |B| :\equiv |A + B|\,.$$

Proof. Since $\mathsf{Card} \to \mathsf{Card}$ is a set, to define $\alpha + \_ : \mathsf{Card} \to \mathsf{Card}$ for all $\alpha : \mathsf{Card}$, by induction it suffices to assume that α is $|A|$ for some $A : \mathsf{Set}$. Now we want to define $|A| + \_ : \mathsf{Card} \to \mathsf{Card}$, i.e. we want to define $|A| + \beta : \mathsf{Card}$ for all $\beta : \mathsf{Card}$. However, since Card is a set, by induction it suffices to assume that β is $|B|$ for some $B : \mathsf{Set}$. But now we can define $|A| + |B|$ to be $|A + B|$. □

Definition 10.2.4. Similarly, the operation of **cardinal multiplication**

$$\_ \cdot \_ : \mathsf{Card} \to \mathsf{Card} \to \mathsf{Card}$$

is defined by induction on truncation:

$$|A| \cdot |B| :\equiv |A \times B|$$

Lemma 10.2.5. Card *is a commutative semiring, i.e. for* $\alpha, \beta, \gamma : \mathsf{Card}$ *we have the following.*

$$\begin{gathered}
(\alpha + \beta) + \gamma = \alpha + (\beta + \gamma) \\
\alpha + 0 = \alpha \\
\alpha + \beta = \beta + \alpha \\
(\alpha \cdot \beta) \cdot \gamma = \alpha \cdot (\beta \cdot \gamma) \\
\alpha \cdot 1 = \alpha \\
\alpha \cdot \beta = \beta \cdot \alpha \\
\alpha \cdot (\beta + \gamma) = \alpha \cdot \beta + \alpha \cdot \gamma
\end{gathered}$$

where $0 :\equiv |\emptyset|$ *and* $1 :\equiv |\mathbf{1}|$.

Proof. We prove the commutativity of multiplication, $\alpha \cdot \beta = \beta \cdot \alpha$; the others are exactly analogous. Since Card is a set, the type $\alpha \cdot \beta = \beta \cdot \alpha$ is a mere proposition, and in particular a set. Thus, by induction it suffices to assume α and β are of the form $|A|$ and $|B|$ respectively, for some $A, B : \mathsf{Set}$. Now $|A| \cdot |B| \equiv |A \times B|$ and $|B| \times |A| \equiv |B \times A|$, so it suffices to show $A \times B = B \times A$. Finally, by univalence, it suffices to give an equivalence $A \times B \simeq B \times A$. But this is easy: take $(a, b) \mapsto (b, a)$ and its obvious inverse. □

Definition 10.2.6. The operation of **cardinal exponentiation** is also defined by induction on truncation:

$$|A|^{|B|} :\equiv |B \to A|\,.$$

Lemma 10.2.7. *For $\alpha, \beta, \gamma : \mathsf{Card}$ we have*

$$\begin{aligned}
\alpha^0 &= 1 \\
1^\alpha &= 1 \\
\alpha^1 &= \alpha \\
\alpha^{\beta+\gamma} &= \alpha^\beta \cdot \alpha^\gamma \\
\alpha^{\beta\cdot\gamma} &= (\alpha^\beta)^\gamma \\
(\alpha \cdot \beta)^\gamma &= \alpha^\gamma \cdot \beta^\gamma
\end{aligned}$$

Proof. Exactly like Lemma 10.2.5. □

Definition 10.2.8. The relation of **cardinal inequality**

$$\_ \leq \_ : \mathsf{Card} \to \mathsf{Card} \to \mathsf{Prop}$$

is defined by induction on truncation:

$$|A| \leq |B| :\equiv \|\mathsf{inj}(A, B)\|$$

where $\mathsf{inj}(A, B)$ is the type of injections from A to B. In other words, $|A| \leq |B|$ means that there merely exists an injection from A to B.

Lemma 10.2.9. *Cardinal inequality is a preorder, i.e. for $\alpha, \beta : \mathsf{Card}$ we have*

$$\begin{aligned}
&\alpha \leq \alpha \\
&(\alpha \leq \beta) \to (\beta \leq \gamma) \to (\alpha \leq \gamma)
\end{aligned}$$

Proof. As before, by induction on truncation. For instance, since $(\alpha \leq \beta) \to (\beta \leq \gamma) \to (\alpha \leq \gamma)$ is a mere proposition, by induction on 0-truncation we may assume α, β, and γ are $|A|$, $|B|$, and $|C|$ respectively. Now since $|A| \leq |C|$ is a mere proposition, by induction on (-1)-truncation we may assume given injections $f : A \to B$ and $g : B \to C$. But then $g \circ f$ is an injection from A to C, so $|A| \leq |C|$ holds. Reflexivity is even easier. □

We may likewise show that cardinal inequality is compatible with the semiring operations.

Lemma 10.2.10. *Consider the following statements:*

(i) There is an injection $A \to B$.
(ii) There is a surjection $B \to A$.

Then, assuming excluded middle:

- *Given $a_0 : A$, we have (i)→(ii).*
- *Therefore, if A is merely inhabited, we have (i) → merely (ii).*
- *Assuming the axiom of choice, we have (ii) → merely (i).*

Proof. If $f : A \to B$ is an injection, define $g : B \to A$ at $b : B$ as follows. Since f is injective, the fiber of f at b is a mere proposition. Therefore, by excluded middle, either there is an $a : A$ with $f(a) = b$, or not. In the first case, define $g(b) :\equiv a$; otherwise set $g(b) :\equiv a_0$. Then for any $a : A$, we have $a = g(f(a))$, so g is surjective.

The second statement follows from this by induction on truncation. For the third, if $g : B \to A$ is surjective, then by the axiom of choice, there merely exists a function $f : A \to B$ with $g(f(a)) = a$ for all a. But then f must be injective. □

Theorem 10.2.11 (Schroeder-Bernstein)**.** *Assuming excluded middle, for sets A and B we have*

$$\mathsf{inj}(A, B) \to \mathsf{inj}(B, A) \to (A \cong B)$$

Proof. The usual "back-and-forth" argument applies without significant changes. Note that it actually constructs an isomorphism $A \cong B$ (assuming excluded middle so that we can decide whether a given element belongs to a cycle, an infinite chain, a chain beginning in A, or a chain beginning in B). □

Corollary 10.2.12. *Assuming excluded middle, cardinal inequality is a partial order, i.e. for $\alpha, \beta : \mathsf{Card}$ we have*

$$(\alpha \leq \beta) \to (\beta \leq \alpha) \to (\alpha = \beta).$$

Proof. Since $\alpha = \beta$ is a mere proposition, by induction on truncation we may assume α and β are $|A|$ and $|B|$, respectively, and that we have injections $f : A \to B$ and $g : B \to A$. But then the Schroeder–Bernstein theorem gives an isomorphism $A \simeq B$, hence an equality $|A| = |B|$. □

Finally, we can reproduce Cantor's theorem, showing that for every cardinal there is a greater one.

Theorem 10.2.13 (Cantor)**.** *For $A : \mathsf{Set}$, there is no surjection $A \to (A \to \mathbf{2})$.*

Proof. Suppose $f : A \to (A \to \mathbf{2})$ is any function, and define $g : A \to \mathbf{2}$ by $g(a) :\equiv \neg f(a)(a)$. If $g = f(a_0)$, then $g(a_0) = f(a_0)(a_0)$ but $g(a_0) = \neg f(a_0)(a_0)$, a contradiction. Thus, f is not surjective. □

Corollary 10.2.14. *Assuming excluded middle, for any $\alpha : \mathsf{Card}$, there is a cardinal β such that $\alpha \leq \beta$ and $\alpha \neq \beta$.*

Proof. Let $\beta = 2^\alpha$. Now we want to show a mere propositon, so by induction we may assume α is $|A|$, so that $\beta \equiv |A \to \mathbf{2}|$. Using excluded middle, we have a function $f : A \to (A \to \mathbf{2})$ defined by

$$f(a)(a') :\equiv \begin{cases} \top & a = a' \\ \bot & a \neq a'. \end{cases}$$

And if $f(a) = f(a')$, then $f(a')(a) = f(a)(a) = \top$, so $a = a'$; hence f is injective. Thus, $\alpha \equiv |A| \leq |A \to \mathbf{2}| \equiv 2^\alpha$.

On the other hand, if $2^\alpha \leq \alpha$, then we would have an injection $(A \to \mathbf{2}) \to A$. By Lemma 10.2.10, since we have $(\lambda x.\, \bot) : A \to \mathbf{2}$ and excluded middle, there would then be a surjection $A \to (A \to \mathbf{2})$, contradicting Cantor's theorem. □

10.3 Ordinal numbers

Definition 10.3.1. Let A be a set and

$$\_ < \_ : A \to A \to \mathsf{Prop}$$

a binary relation on A. We define by induction what it means for an element $a : A$ to be **accessible** by $<$:

- If b is accessible for every $b < a$, then a is accessible.

We write $\mathsf{acc}(a)$ to mean that a is accessible.

It may seem that such an inductive definition can never get off the ground, but of course if a has the property that there are *no* b such that $b < a$, then a is vacuously accessible.

Lemma 10.3.2. *Accessibility is a mere property.*

Proof. We claim that for any $a_1, a_2 : A$, any $p : a_1 = a_2$, and any $s_1 : \mathsf{acc}(a_1)$ and $s_2 : \mathsf{acc}(a_2)$, we have $p_*(s_1) = s_2$. By induction, we may assume that s_1 is given by a function assigning to each $b_1 < a_1$ a proof $s_1(b_1) : \mathsf{acc}(b_1)$, and moreover that for any $b_2 : A$, any $q : b_1 = b_2$, and $t_2 : \mathsf{acc}(b_2)$, we have $q_*(s_1(b_1)) = t_2$. Similarly, we may assume that s_2 is given by a function assigning to each $b_2 < a_2$ a proof $s_2(b_2) : \mathsf{acc}(b_2)$, and that for any $b_1 : A$, any $q : b_1 = b_2$, and $t_1 : \mathsf{acc}(b_1)$, we have $q_*(t_1) = s_2(b_2)$.

Now by function extensionality, to show $p_*(s_1) = s_2$ it suffices to show that for any $b_2 < a_2$ we have $p_*(s_1)(b_2) = s_2(b_2)$. However, we can obtain this from the induction hypothesis for s_2 with $q :\equiv \mathsf{refl}_{b_2}$ and $t_1 :\equiv p_*(s_1)(b_2)$. This proves the claim.

Finally, we instantiate the claim with $a_2 :\equiv a_1 :\equiv a$ and $p :\equiv \mathsf{refl}_a$. Thus, for any $a : A$ and $s_1, s_2 : \mathsf{acc}(a)$, we have $s_1 = s_2$, as desired. □

Definition 10.3.3. A binary relation $<$ on a set A is **well-founded** if every element of A is accessible.

Lemma 10.3.4. *Well-foundedness is a mere property.*

Proof. Well-foundedness of $<$ is the type $\prod_{(a:A)} \mathsf{acc}(a)$, which is a mere proposition since each $\mathsf{acc}(a)$ is. □

Example 10.3.5. Perhaps the most familiar well-founded relation is the usual strict ordering on $\mathbb{N}$. To show that this is well-founded, we must show that n is accessible for each $n : \mathbb{N}$. This is just the usual proof of "strong induction" from ordinary induction on $\mathbb{N}$.

Specifically, we prove by induction on $n : \mathbb{N}$ that k is accessible for all $k \leq n$. The base case is just that 0 is accessible, which is vacuously true since nothing is strictly less than 0. For the inductive step, we assume that k is accessible for all $k \leq n$, which is to say for all $k < n + 1$; hence by definition $n + 1$ is also accessible.

A different relation on $\mathbb{N}$ which is also well-founded is obtained by setting only $n < \mathsf{succ}(n)$ for all $n : \mathbb{N}$. Well-foundedness of this relation is almost exactly the ordinary induction principle of $\mathbb{N}$.

Example 10.3.6. Let $A : \mathsf{Set}$ and $B : A \to \mathsf{Set}$ be a family of sets. Recall from §5.3 that the W-type $\mathsf{W}_{(a:A)} B(a)$ is inductively generated by the single constructor

- $\mathsf{sup} : \prod_{(a:A)} (B(a) \to \mathsf{W}_{(x:A)} B(x)) \to \mathsf{W}_{(x:A)} B(x)$

We define the relation $<$ on $\mathsf{W}_{(x:A)} B(x)$ by recursion on its second argument:

- For any $a : A$ and $f : B(a) \to \mathsf{W}_{(x:A)} B(x)$, we define $w < \mathsf{sup}(a, f)$ to mean that there merely exists a $b : B(a)$ such that $w = f(b)$.

Now we prove that every $w : \mathsf{W}_{(x:A)} B(x)$ is accessible for this relation, using the usual induction principle for $\mathsf{W}_{(x:A)} B(x)$. This means we assume given $a : A$ and $f : B(a) \to \mathsf{W}_{(x:A)} B(x)$, and also a lifting $f' : \prod_{(b:B(a))} \mathsf{acc}(f(b))$. But then by definition of $<$, we have $\mathsf{acc}(w)$ for all $w < \mathsf{sup}(a, f)$; hence $\mathsf{sup}(a, f)$ is accessible.

Well-foundedness allows us to define functions by recursion and prove statements by induction, such as for instance the following.

Lemma 10.3.7. *Suppose B is a set and we have a function*

$$g : \mathcal{P} B \to B$$

Then if $<$ is a well-founded relation on A, there is a function $f : A \to B$ such that for all $a : A$ we have

$$f(a) = g\Big(\{\, f(a') \mid a' < a \,\} \Big).$$

Proof. We first define, for every $a : A$ and $s : \mathsf{acc}(a)$, an element $\bar{f}(a, s) : B$. By induction, it suffices to assume that s is a function assigning to each $a' < a$ a proof $s(a') : \mathsf{acc}(a')$, and that moreover for each such a' we have an element $\bar{f}(a', s(a')) : B$. In this case, we define

$$\bar{f}(a, s) :\equiv g\Big(\{\, \bar{f}(a', s(a')) \mid a' < a \,\} \Big).$$

Now since $<$ is well-founded, we have a function $w : \prod_{(a:A)} \mathsf{acc}(a)$. Thus, we can define $f(a) :\equiv \bar{f}(a, w(a))$. □

In classical logic, well-foundedness has a more well-known reformulation.

Lemma 10.3.8. *Assuming excluded middle, $<$ is well-founded if and only if every nonempty subset $B \subseteq A$ merely has a minimal element.*

Proof. Suppose first $<$ is well-founded, and suppose $B \subseteq A$ is a subset with no minimal element. That is, for any $a : A$ with $a \in B$, there merely exists a $b : A$ with $b < a$ and $b \in B$.

We claim that for any $a : A$ and $s : \mathsf{acc}(a)$, we have $a \notin B$. By induction, we may assume s is a function assigning to each $a' < a$ a proof $s(a') : \mathsf{acc}(a)$, and that moreover for each such a' we have $a' \notin B$. If $a \in B$, then by assumption, there would merely exist a $b < a$ with $b \in B$, which contradicts this assumption. Thus, $a \notin B$; this completes the induction. Since $<$ is well-founded, we have $a \notin B$ for all $a : A$, i.e. B is empty.

Now suppose each nonempty subset merely has a minimal element. Let $B = \{\, a : A \mid \neg\mathsf{acc}(a) \,\}$. Then if B is nonempty, it merely has a minimal element. Thus there merely exists an $a : A$ with $a \in B$ such that for all $b < a$, we have $\mathsf{acc}(b)$. But then by definition (and induction on truncation), a is merely accessible, and hence accessible, contradicting $a \in B$. Thus, B is empty, so $<$ is well-founded. □

Definition 10.3.9. A well-founded relation $<$ on a set A is **extensional** if for any $a, b : A$, we have

$$\Big(\prod_{c:A} (c < a) \leftrightarrow (c < b)\Big) \to (a = b).$$

Note that since A is a set, extensionality is a mere proposition. This notion of "extensionality" is unrelated to function extensionality, and also unrelated to the extensionality of identity types. Rather, it is a "local" counterpart of the axiom of extensionality in classical set theory.

Theorem 10.3.10. *The type of extensional well-founded relations is a set.*

Proof. By the univalence axiom, it suffices to show that if $(A, <)$ is extensional and well-founded and $f : (A, <) \cong (A, <)$, then $f = \mathsf{id}_A$. We prove by induction on $<$ that $f(a) = a$ for all $a : A$. The inductive hypothesis is that for all $a' < a$, we have $f(a') = a'$.

Now since A is extensional, to conclude $f(a) = a$ it is sufficient to show

$$\prod_{c:A} (c < f(a)) \leftrightarrow (c < a).$$

However, since f is an automorphism, we have $(c < a) \leftrightarrow (f(c) < f(a))$. But $c < a$ implies $f(c) = c$ by the induction hypothesis, so $(c < a) \to (c < f(a))$. On the other hand, if $c < f(a)$, then $f^{-1}(c) < a$, and so $c = f(f^{-1}(c)) = f^{-1}(c)$ by the induction hypothesis again; thus $c < a$. Therefore, we have $(c < a) \leftrightarrow (c < f(a))$ for any $c : A$, so $f(a) = a$. □

Definition 10.3.11. If $(A, <)$ and $(B, <)$ are extensional and well-founded, a **simulation** is a function $f : A \to B$ such that

(i) if $a < a'$, then $f(a) < f(a')$, and
(ii) for all $a : A$ and $b : B$, if $b < f(a)$, then there merely exists an $a' < a$ with $f(a') = b$.

Lemma 10.3.12. *Any simulation is injective.*

Proof. We prove by double well-founded induction that for any $a, b : A$, if $f(a) = f(b)$ then $a = b$. The induction hypothesis for $a : A$ says that for any $a' < a$, and any $b : B$, if $f(a') = f(b)$ then $a = b$. The inner induction hypothesis for $b : A$ says that for any $b' < b$, if $f(a') = f(b')$ then $a' = b'$.

Suppose $f(a) = f(b)$; we must show $a = b$. By extensionality, it suffices to show that for any $c : A$ we have $(c < a) \leftrightarrow (c < b)$. If $c < a$, then $f(c) < f(a)$ by Definition 10.3.11(i). Hence $f(c) < f(b)$, so by Definition 10.3.11(ii) there merely exists $c' : A$ with $c' < b$ and $f(c) = f(c')$. By the induction hypothesis for a, we have $c = c'$, hence $c < b$. The dual argument is symmetrical. □

In particular, this implies that the word "merely" in Definition 10.3.11(ii) could be omitted without change of sense.

Corollary 10.3.13. *If $f : A \to B$ is a simulation, then for all $a : A$ and $b : B$, if $b < f(a)$, there* purely *exists an $a' < a$ with $f(a') = b$.*

Proof. Since f is injective, $\sum_{(a:A)}(f(a) = b)$ is a mere proposition. □

We say that a subset $C : \mathcal{P}B$ is an **initial segment** if $c \in C$ and $b < c$ imply $b \in C$. The image of a simulation must be an initial segment, while the inclusion of any initial segment is a simulation. Thus, by univalence, every simulation $A \to B$ is *equal* to the inclusion of some initial segment of B.

Theorem 10.3.14. *For a set A, let $P(A)$ be the type of extensional well-founded relations on A. If $<_A : P(A)$ and $<_B : P(B)$ and $f : A \to B$, let $H_{<_A <_B}(f)$ be the mere proposition that f is a simulation. Then (P, H) is a standard notion of structure over $\mathcal{S}et$ in the sense of §9.8.*

Proof. We leave it to the reader to verify that identities are simulations, and that composites of simulations are simulations. Thus, we have a notion of structure. For standardness, we must show that if $<$ and $\prec$ are two extensional well-founded relations on A, and id_A is a simulation in both directions, then $<$ and $\prec$ are equal. Since extensionality and well-foundedness are mere propositions, for this it suffices to have $\prod_{(a,b:A)}(a < b) \leftrightarrow (a \prec b)$. But this follows from Definition 10.3.11(i) for id_A. □

Corollary 10.3.15. *There is a category whose objects are sets equipped with extensional well-founded relations, and whose morphisms are simulations.*

In fact, this category is a poset.

Lemma 10.3.16. *For extensional and well-founded $(A, <)$ and $(B, <)$, there is at most one simulation $f : A \to B$.*

Proof. Suppose $f, g : A \to B$ are simulations. Since being a simulation is a mere property, it suffices to show $\prod_{(a:A)}(f(a) = g(a))$. By induction on $<$, we may suppose $f(a') = g(a')$ for all $a' < a$. And by extensionality of B, to have $f(a) = g(a)$ it suffices to have $\prod_{(b:B)}(b < f(a)) \leftrightarrow (b < g(a))$.

But since f is a simulation, if $b < f(a)$, then we have $a' < a$ with $f(a') = b$. By the inductive hypothesis, we have also $g(a') = b$, hence $b < g(a)$. The dual argument is symmetrical. □

Thus, if A and B are equipped with extensional and well-founded relations, we may write $A \leq B$ to mean there exists a simulation $f : A \to B$. Corollary 10.3.15 implies that if $A \leq B$ and $B \leq A$, then $A = B$.

Definition 10.3.17. An **ordinal** is a set A with an extensional well-founded relation which is *transitive*, i.e. $\prod_{(a,b,c:A)}(a < b) \to (b < c) \to (a < c)$.

Example 10.3.18. Of course, the usual strict order on $\mathbb{N}$ is transitive. It is easily seen to be extensional as well; thus it is an ordinal. As usual, we denote this ordinal by ω.

Let Ord denote the type of ordinals. By the previous results, Ord is a set and has a natural partial order. We now show that Ord also admits a well-founded relation.

If A is an ordinal and $a : A$, let $A_{/a}$ denote the initial segment $\{ b : A \mid b < a \}$. Note that if $A_{/a} = A_{/b}$ as ordinals, then that isomorphisms must respect their inclusions into A (since simulations form a poset), and hence they are equal as subsets of A. Therefore, since A is extensional, $a = b$. Thus the function $a \mapsto A_{/a}$ is an injection $A \to \mathsf{Ord}$.

Definition 10.3.19. For ordinals A and B, a simulation $f : A \to B$ is said to be **bounded** if there exists $b : B$ such that $A = B_{/b}$.

The remarks above imply that such a b is unique when it exists, so that boundedness is a mere property.

We write $A < B$ if there exists a bounded simulation from A to B. Since simulations are unique, $A < B$ is also a mere proposition.

Theorem 10.3.20. $(\mathsf{Ord}, <)$ *is an ordinal.*

Remark 10.3.21. If universe levels were made explicit, this theorem would state that the set of ordinals in one universe is an ordinal in the next higher universe.

Proof. Let A be an ordinal; we first show that $A_{/a}$ is accessible (in Ord) for all $a : A$. By induction, suppose $A_{/b}$ is accessible for all $b : A$. By definition of accessibility, we must show that B is accessible in Ord for all $B < A_{/a}$. However, if $B < A_{/a}$ then there is some $b < a$ such that $B = (A_{/a})_{/b} = A_{/b}$, which is accessible by the inductive hypothesis. Thus, $A_{/a}$ is accessible for all $a : A$.

Now to show that A is accessible in Ord, by definition we must show B is accessible for all $B < A$. But as before, $B < A$ means $B = A_{/a}$ for some $a : A$, which is accessible as we just proved. Thus, Ord is well-founded.

For extensionality, suppose A and B are ordinals such that $\prod_{(C:\mathsf{Ord})}(C < A) \leftrightarrow (C < B)$. Then for every $a : A$, since $A_{/a} < A$, we have $A_{/a} < B$, hence there is $b : B$ with $A_{/a} = B_{/b}$. Define $f : A \to B$ to take each a to the corresponding b; it is straightforward to verify that f is an isomorphism. Thus $A \cong B$, hence $A = B$ by univalence.

Finally, it is easy to see that $<$ is transitive. □

Treating Ord as an ordinal is often very convenient, but it has its pitfalls as well. For instance, consider the following lemma, where we pay attention to how universes are used.

Lemma 10.3.22. *Let $\mathcal{U}$ be a universe. For any $A : \mathsf{Ord}_{\mathcal{U}}$, there is a $B : \mathsf{Ord}_{\mathcal{U}}$ such that $A < B$.*

Proof. Let $B = A + \mathbf{1}$, with the element $\star : \mathbf{1}$ being greater than all elements of A. Then B is an ordinal and it is easy to see that $A \cong B_{/\star}$. □

This lemma illustrates a potential pitfall of the "typically ambiguous" style of using $\mathcal{U}$ to denote an arbitrary, unspecified universe. Consider the following alternative proof of it.

Another putative proof of Lemma 10.3.22. Note that $C < A$ if and only if $C = A_{/a}$ for some $a : A$. This gives an isomorphism $A \cong \mathsf{Ord}_{/A}$, so that $A < \mathsf{Ord}$. Thus we may take $B :\equiv \mathsf{Ord}$. □

The second proof would be valid if we had stated Lemma 10.3.22 in a typically ambiguous style. But the resulting lemma would be less useful, because the second proof would constrain the second "Ord" in the lemma statement to refer to a higher universe level than the first one. The first proof allows both universes to be the same.

Similar remarks apply to the next lemma, which could be proved in a less useful way by observing that $A \leq \mathsf{Ord}$ for any $A : \mathsf{Ord}$.

Lemma 10.3.23. *Let $\mathcal{U}$ be a universe. For any $X : \mathcal{U}_{\mathcal{U}}$ and $F : X \to \mathsf{Ord}_{\mathcal{U}}$, there exists $B : \mathsf{Ord}_{\mathcal{U}}$ such that $Fx \leq B$ for all $x : X$.*

Proof. Let B be the quotient of the equivalence relation on $\sum_{(x:X)} Fx$ defined as follows:

$$(x, y) = (x', y') \;:\equiv\; \Big((Fx)_{/y} \cong (Fx')_{/y'}\Big).$$

Define $(x, y) < (x', y')$ if $(Fx)_{/y} < (Fx')_{/y'}$. This clearly descends to the quotient, and can be seen to make B into an ordinal. Moreover, for each $x : X$ the induced map $Fx \to B$ is a simulation. □

10.4 Classical well-orderings

We now show the equivalence of our ordinals with the more familiar classical well-orderings.

Lemma 10.4.1. *Assuming excluded middle, every ordinal is trichotomous:*

$$\prod_{a,b:A} (a < b) \vee (a = b) \vee (b < a).$$

Proof. By induction on a, we may assume that for every $a' < a$ and every $b' : A$, we have $(a' < b') \vee (a' = b') \vee (b' < a')$. Now by induction on b, we may assume that for every $b' < b$, we have $(a < b') \vee (a = b') \vee (b' < a)$.

By excluded middle, either there merely exists a $b' < b$ such that $a < b'$, or there merely exists a $b' < b$ such that $a = b'$, or for every $b' < b$ we have $b' < a$. In the first case, merely $a < b$ by transitivity, hence $a < b$ as it is a mere proposition. Similarly, in the second case, $a < b$ by transport. Thus, suppose $\prod_{(b':A)} (b' < b) \to (b' < a)$.

Now analogously, either there merely exists $a' < a$ such that $b < a'$, or there merely exists $a' < a$ such that $a' = b$, or for every $a' < a$ we have $a' < b$. In the first and second cases, $b < a$, so we may suppose $\prod_{(a':A)} (a' < a) \to (a' < b)$. However, by extensionality, our two suppositions now imply $a = b$. □

Lemma 10.4.2. *A well-founded relation contains no cycles, i.e.*

$$\prod_{(n:\mathbb{N})} \prod_{(a:\mathbb{N}_n \to A)} \neg\Big((a_0 < a_1) \wedge \cdots \wedge (a_{n-1} < a_n) \wedge (a_n < a_0)\Big).$$

Proof. We prove by induction on $a : A$ that there is no cycle containing a. Thus, suppose by induction that for all $a' < a$, there is no cycle containing a'. But in any cycle containing a, there is some element less than a and contained in the same cycle. □

Theorem 10.4.3. *Assuming excluded middle, $(A, <)$ is an ordinal if and only if every nonempty subset $B \subseteq A$ has a least element.*

Proof. If A is an ordinal, then by Lemma 10.3.8 every nonempty subset merely has a minimal element. But trichotomy implies that any minimal element is a least element. Moreover, least elements are unique when they exist, so merely having one is as good as having one.

Conversely, if every nonempty subset has a least element, then A is well-founded by Lemma 10.3.8. We also have trichotomy, since for any a, b the set $\{ a, b \}$ merely has a least element, which must be either a or b. This implies transitivity, since if $a < b$ and $a < c$, then either $a = c$ or $c < a$ would produce a cycle. Similarly, it implies extensionality, for if $\prod_{(c:A)}(c < a) \leftrightarrow (c < b)$, then $a < b$ implies (letting c be a) that $a < a$, which is a cycle, and similarly if $b < a$; hence $a = b$. □

In classical mathematics, the characterization of Theorem 10.4.3 is taken as the definition of a *well-ordering*, with the *ordinals* being a canonical set of representatives of isomorphism classes for well-orderings. In our context, the structure identity principle means that there is no need to look for such representatives: any well-ordering is as good as any other.

We now move on to consider consequences of the axiom of choice.

Theorem 10.4.4. *Assuming excluded middle, the following are equivalent.*

(i) For every set X, there merely exists a function $f : \mathcal{P}_+X \to X$ such that $f(Y) \in Y$ for all $Y : \mathcal{P}X$.
(ii) Every set merely admits the structure of an ordinal.

Of course, (i) is a standard classical version of the axiom of choice.

Proof. One direction is easy: suppose (ii). Since we aim to prove the mere proposition (i), we may assume A is an ordinal. But then we can define $f(B)$ to be the least element of B.

Now suppose (i). As before, since (ii) is a mere proposition, we may assume given such an f. We extend f to a function

$$\bar{f} : \mathcal{P}X \cong (\mathcal{P}_+X) + \mathbf{1} \longrightarrow X + \mathbf{1}$$

in the obvious way. Now for any ordinal A, we can define $g_A : A \to X + \mathbf{1}$ by well-founded recursion:

$$g_A(a) :\equiv \bar{f}\Big(X \setminus \Big\{ g_A(b) \;\Big|\; (b < a) \wedge (g_A(b) \in X) \Big\}\Big)$$

(regarding X as a subset of $X + \mathbf{1}$ in the obvious way).

Let A' be the preimage of X; then we claim the restriction $g'_A : A' \to X$ is injective. For if $a, a' : A$ with $a \neq a'$, then by trichotomy and without loss of generality, we may assume $a' < a$. Thus $g_A(a') \in \{ g_A(b) \mid b < a \}$, so since $f(Y) \in Y$ for all Y we have $g_A(a) \neq g_A(a')$.

Moreover, A' is an initial segment of A. For $g_A(a)$ lies in $\mathbf{1}$ if and only if $\{\, g_A(b) \mid b < a \,\} = X$, and if this holds then it also holds for any $a' > a$. Thus, A' is itself an ordinal.

Finally, since Ord is an ordinal, we can take $A :\equiv \mathsf{Ord}$. Let X' be the image of $g'_{\mathsf{Ord}} : \mathsf{Ord}' \to X$; then the inverse of g'_{Ord} yields an injection $H : X' \to \mathsf{Ord}$. By Lemma 10.3.23, there is an ordinal C such that $Hx \leq C$ for all $x : X'$. Then by Lemma 10.3.22, there is a further ordinal D such that $C < D$, hence $Hx < D$ for all $x : X'$. Now we have

$$\begin{aligned} g_{\mathsf{Ord}}(D) &= \bar{f}\left(X \setminus \left\{ g_{\mathsf{Ord}}(B) \;\middle|\; B < D \wedge (g_{\mathsf{Ord}}(B) \in X) \right\}\right) \\ &= \bar{f}\left(X \setminus \left\{ g_{\mathsf{Ord}}(B) \;\middle|\; B : \mathsf{Ord} \wedge (g_{\mathsf{Ord}}(B) \in X) \right\}\right) \end{aligned}$$

since if $B : \mathsf{Ord}$ and $(g_{\mathsf{Ord}}(B) \in X)$, then $B = Hx$ for some $x : X'$, hence $B < D$. Now if

$$\left\{ g_{\mathsf{Ord}}(B) \;\middle|\; B : \mathsf{Ord} \wedge (g_{\mathsf{Ord}}(B) \in X) \right\}$$

is not all of X, then $g_{\mathsf{Ord}}(D)$ would lie in X but not in this subset, which would be a contradiction since D is itself a potential value for B. So this set must be all of X, and hence g'_{Ord} is surjective as well as injective. Thus, we can transport the ordinal structure on Ord' to X. □

Remark 10.4.5. If we had given the wrong proof of Lemma 10.3.22 or Lemma 10.3.23, then the resulting proof of Theorem 10.4.4 would be invalid: there would be no way to consistently assign universe levels.

Corollary 10.4.6. *Assuming the axiom of choice, the function* $\mathsf{Ord} \to \mathsf{Set}$ *(which forgets the order structure) is a surjection.*

Note that Ord is a set, while Set is a 1-type. In general, there is no reason for a 1-type to admit any surjective function from a set. Even the axiom of choice does not appear to imply that *every* 1-type does so, but it readily implies that this is so for 1-types constructed out of Set, such as the types of objects of categories of structures as in §9.8. The following corollary also applies to such categories.

Corollary 10.4.7. *Assuming AC,* $\mathcal{S}et$ *admits a weak equivalence functor from a strict category.*

Proof. Let $X_0 :\equiv \mathsf{Ord}$, and for $A, B : X_0$ let $\hom_X(A, B) :\equiv (A \to B)$. Then X is a strict category, since Ord is a set, and the above surjection $X_0 \to \mathsf{Set}$ extends to a weak equivalence functor $X \to \mathcal{S}et$. □

Now recall from §10.2 that we have a further surjection $|-| : \mathsf{Set} \to \mathsf{Card}$, and hence a composite surjection $\mathsf{Ord} \to \mathsf{Card}$ which sends each ordinal to its cardinality.

Theorem 10.4.8. *Assuming AC, the surjection* $\mathsf{Ord} \to \mathsf{Card}$ *has a section.*

Proof. There is an easy and wrong proof of this: since Ord and Card are both sets, AC implies that any surjection between them *merely* has a section. However, we actually have a canonical *specified* section: because Ord is an ordinal, every nonempty subset of it has a uniquely specified least element. Thus, we can map each cardinal to the least element in the corresponding fiber. □

It is traditional in set theory to identify cardinals with their image in Ord: the least ordinal having that cardinality.

It follows that Card also canonically admits the structure of an ordinal: in fact, one isomorphic to Ord. Specifically, we define by well-founded recursion a function $\aleph : \mathsf{Ord} \to \mathsf{Ord}$, such that $\aleph(A)$ is the least ordinal having cardinality greater than $\aleph(A_{/a})$ for all $a : A$. Then (assuming AC) the image of $\aleph$ is exactly the image of Card.

10.5 The cumulative hierarchy

We can define a cumulative hierarchy V of all sets in a given universe $\mathcal{U}$ as a higher inductive type, in such a way that V is again a set (in a larger universe $\mathcal{U}'$), equipped with a binary "membership" relation $x \in y$ which satisfies the usual laws of set theory.

Definition 10.5.1. The **cumulative hierarchy** V relative to a type universe $\mathcal{U}$ is the higher inductive type generated by the following constructors.

(i) For every $A : \mathcal{U}$ and $f : A \to V$, there is an element $\mathsf{set}(A, f) : \mathrm{V}$.

(ii) For all $A, B : \mathcal{U}$, $f : A \to V$ and $g : B \to V$ such that

$$\big(\forall(a : A).\, \exists(b : B).\, f(a) =_V g(b)\big) \wedge \big(\forall(b : B).\, \exists(a : A).\, f(a) =_V g(b)\big) \tag{10.5.2}$$

there is a path $\mathsf{set}(A, f) =_V \mathsf{set}(B, g)$.

(iii) The 0-truncation constructor: for all $x, y : V$ and $p, q : x = y$, we have $p = q$.

In set-theoretic language, $\mathsf{set}(A, f)$ can be understood as the set that is the image of A under f, i.e. $\{f(a) \mid a \in A\}$, but of course we cannot write this in type theory. The hierarchy V is bootstrapped from the empty map $! : \mathbf{0} \to V$, which gives the empty set as $\emptyset = \mathsf{set}(\mathbf{0}, !)$. Then the singleton $\{\emptyset\}$ enters V through $\mathbf{1} \to V$, defined as $\star \mapsto \emptyset$, and so on. The type V lives in the same universe as the base universe $\mathcal{U}$.

The second constructor of V has a form unlike any we have seen before: it involves not only paths in V (which in §6.9 we claimed were slightly fishy) but truncations of sums of them. It certainly does not fit the general scheme described in §6.13, and thus it may not be obvious what its induction principle should be. Fortunately, like our first definition of the 0-truncation in §6.9, it can be re-expressed using auxiliary higher inductive types.

We leave it to the reader to work out the details (see Exercise 10.10). At the end of the day, the induction principle for V (written in pattern-matching language) says that given $P : V \to \mathsf{Set}$, in order to construct $h : \prod_{(x:V)} P(x)$, it suffices to give the following.

(i) For any $f : A \to V$, construct $h(\mathsf{set}(A, f))$, assuming as given $h(f(a))$ for all $a : A$.

(ii) Verify that if $f : A \to V$ and $g : B \to V$ satisfy (10.5.2), then $h(\mathsf{set}(A, f)) = h(\mathsf{set}(B, g))$, assuming inductively that $h(f(a)) = h(g(b))$ whenever $f(a) = g(b)$.

The second clause checks that the map being defined must respect the paths introduced in (10.5.2). As usual when we state higher induction principles using pattern-matching, it may seem tautologous, but is not. The point is that "$h(f(a))$" is essentially a formal symbol which

we cannot peek inside of, which $h(\mathsf{set}(A, f))$ must be defined in terms of. Thus, in the second clause, we assume equality of these formal symbols when appropriate, and verify that the elements resulting from the construction of the first clause are also equal.

Observe that, by induction, for each $v : V$ there merely exist $A : \mathcal{U}$ and $f : A \to V$ such that $v = \mathsf{set}(A, f)$. Thus, it is reasonable to try to define the **membership relation** $x \in v$ on V by setting:

$$(x \in \mathsf{set}(A, f)) :\equiv \exists(a : A).\, x = f(a).$$

To see that the definition is valid, we must use the recursion principle of V. Thus, suppose we have a path $\mathsf{set}(A, f) = \mathsf{set}(B, g)$ constructed through (10.5.2). If $x \in \mathsf{set}(A, f)$ then there merely is $a : A$ such that $x = f(a)$, but by (10.5.2) there merely is $b : B$ such that $f(a) = g(b)$, hence $x = g(b)$ and $x \in \mathsf{set}(B, f)$. The converse is symmetric.

The **subset relation** $x \subseteq y$ is defined on V as usual by

$$(x \subseteq y) :\equiv \forall(z : V).\, z \in x \Rightarrow z \in y.$$

A **class** may be taken to be a mere predicate on V. We can say that a class $C : V \to \mathsf{Prop}$ is a V**-set** if there merely exists $v \in V$ such that

$$\forall(x : V).\, C(x) \Leftrightarrow x \in v.$$

We may also use the conventional notation for classes:

$$\{\, x \mid C(x) \,\} :\equiv \lambda x.\, C(x)$$

A class $C : V \to \mathsf{Prop}$ will be called $\mathcal{U}$**-small** if all of its values $C(x)$ lie in $\mathcal{U}$, specifically $C : V \to \mathsf{Prop}_{\mathcal{U}}$. Since V lives in the same universe $\mathcal{U}'$ as does the base universe $\mathcal{U}$ from which it is built, the same is true for the identity types $v =_V w$ for any $v, w : V$. For a well-behaved theory in the absence of propositional resizing, therefore, it will be convenient to have a $\mathcal{U}$-small "resizing" of the identity relation, which we can define by induction as follows.

Definition 10.5.3. Define the **bisimulation** relation

$$\sim : V \times V \longrightarrow \mathsf{Prop}_{\mathcal{U}}$$

by double induction over V, where for $\mathsf{set}(A, f)$ and $\mathsf{set}(B, g)$ we let:

$$\mathsf{set}(A, f) = \mathsf{set}(B, g) :\equiv \\ \big(\forall(a : A).\, \exists(b : B).\, f(a) = g(b)\big) \wedge \big(\forall(b : B).\, \exists(a : A).\, f(a) = g(b)\big)$$

To verify that the definition is correct, we just need to check that it respects paths $\mathsf{set}(A, f) = \mathsf{set}(B, g)$ constructed through (10.5.2), but this is obvious, and that $\mathsf{Prop}_{\mathcal{U}}$ is a set, which it is. Note that $u = v$ is in $\mathsf{Prop}_{\mathcal{U}}$ by construction.

Lemma 10.5.4. *For any $u, v : V$ we have $(u =_V v) = (u = v)$.*

Proof. It suffices to show that $(u = v) \to (u =_V v)$. By induction on u and v, we may assume they are $\mathsf{set}(A, f)$ and $\mathsf{set}(B, g)$ respectively. Then by definition, $\mathsf{set}(A, f) \sim \mathsf{set}(B, g)$ implies $(\forall(a : A).\, \exists(b : B).\, f(a) = g(b))$ and conversely. But the induction hypothesis then tells us that $(\forall(a : A).\, \exists(b : B).\, f(a) = g(b))$ and conversely. So by the path-constructor for V we have $\mathsf{set}(A, f) =_V \mathsf{set}(B, g)$. □

Now we can use the resized identity relation to get the following useful principle.

Lemma 10.5.5. *For every $u : V$ there is a given $A_u : \mathcal{U}$ and monic $m_u : A \rightarrowtail V$ such that $u = \mathsf{set}(A_u, m_u)$.*

Proof. Take any presentation $u = \mathsf{set}(A, f)$ and factor $f : A \to V$ as

$$f = m_u \circ e_u : A \twoheadrightarrow A_u \rightarrowtail V.$$

Clearly $u = \mathsf{set}(A_u, m_u)$ if only A_u is still in $\mathcal{U}$, which holds if the kernel of $e_u : A \twoheadrightarrow A_u$ is in $\mathcal{U}$. But the kernel of $e_u : A \twoheadrightarrow A_u$ is the pullback along $f : A \to V$ of the identity on V, which we just showed to be $\mathcal{U}$-small, up to equivalence. Now, this construction of the pair (A_u, m_u) with $m_u : A_u \rightarrowtail V$ and $u = \mathsf{set}(A_u, m_u)$ from $u : V$ is unique up to equivalence over V, and hence up to identity by univalence. Thus by the principle of unique choice (3.9.2) there is a map $c : V \to \sum_{(A:\mathcal{U})}(A \to V)$ such that $c(u) = (A_u, m_u)$, with $m_u : A_u \rightarrowtail V$ and $u = \mathsf{set}(c(u))$, as claimed. □

Definition 10.5.6. For $u : V$, the just constructed monic presentation $m_u : A_u \rightarrowtail V$ such that $u = \mathsf{set}(A_u, m_u)$ may be called the **type of members** of u and denoted $m_u : [u] \rightarrowtail V$, or even $[u] \rightarrowtail V$. We can think of $[u]$ as the "subclass of V consisting of members of u".

Theorem 10.5.7. *The following hold for $(V, \in)$:*

(i) extensionality:
$$\forall(x, y : V).\, x \subseteq y \wedge y \subseteq x \Leftrightarrow x = y.$$

(ii) empty set: *for all $x : V$, we have $\neg(x \in \emptyset)$.*

(iii) pairing: *for all $u, v : V$, the class $\{\, x \mid x = u \vee x = v \,\}$ is a V-set.*

(iv) infinity: *there is a $v : V$ with $\emptyset \in v$ and $x \in v$ implies $x \cup \{x\} \in v$.*

(v) union: *for all $v : V$, the class $\{\, x \mid \exists(u : V).\, x \in u \in v \,\}$ is a V-set.*

(vi) function set: *for all $u, v : V$, the class $\{\, x \mid x : u \to v \,\}$ is a V-set.*

(vii) $\in$-induction: *if $C : V \to \mathsf{Prop}$ is a class such that $C(a)$ holds whenever $C(x)$ for all $x \in a$, then $C(v)$ for all $v : V$.*

(viii) replacement: *given any $r : V \to V$ and $a : V$, the class*

$$\{\, x \mid \exists(y : V).\, y \in a \wedge x = r(y) \,\}$$

is a V-set.

(ix) separation: *given any $a : V$ and $\mathcal{U}$-small $C : V \to \mathsf{Prop}_{\mathcal{U}}$, the class*

$$\{\, x \mid x \in a \wedge C(x) \,\}$$

is a V-set.

Sketch of proof.

(i) Extensionality: if $\mathsf{set}(A, f) \subseteq \mathsf{set}(B, g)$ then $f(a) \in \mathsf{set}(B, g)$ for every $a : A$, therefore for every $a : A$ there merely exists $b : B$ such that $f(a) = g(b)$. The assumption $\mathsf{set}(B, g) \subseteq \mathsf{set}(A, f)$ gives the other half of (10.5.2), therefore $\mathsf{set}(A, f) = \mathsf{set}(B, g)$.

(ii) Empty set: suppose $x \in \emptyset = \mathsf{set}(\mathbf{0}, !)$. Then $\exists(a : \mathbf{0}).\, x = !a$, which is absurd.

(iii) Pairing: given u and v, let $w = \mathsf{set}(1 + 1, [\lambda x.\, u, \lambda x.\, v])$.

(iv) Infinity: take $w = \mathsf{set}(\mathbb{N}, I)$, where $I : \mathbb{N} \to V$ is given by the recursion $I(0) = \emptyset$ and $I(n + 1) = I(n) \cup \{I(n)\}$.

(v) Union: Take any $v : V$ and any presentation $f : A \to V$ with $v = \mathsf{set}(A, f)$. Then let $\tilde{A} :\equiv \sum_{(a:A)} [fa]$, where $[fa] \rightarrowtail V$ is the type of members from Definition 10.5.6. $\tilde{A}$ is plainly $\mathcal{U}$-small and has a projection map $\pi : \tilde{A} \to A$. We then let $w :\equiv \mathsf{set}(\tilde{A}, f \circ \pi)$.

(vi) Function set: given $u, v : V$, take the types of elements $[u] \rightarrowtail V$ and $[u] \rightarrowtail V$, and the function type $[u] \to [v]$. We want to define a map

$$r : ([u] \to [v]) \longrightarrow V$$

with "$r(f) = \{(x, f(x)) \mid x : [u]\}$", but in order for this to make sense we must first define the ordered pair (x, y), and then we take the map $r' : x \mapsto (x, f(x))$, and then we can put $r(f) :\equiv \mathsf{set}([u], r')$. But the ordered pair can be defined in terms of unordered pairing as usual.

(vii) $\in$-induction: let $C : V \to \mathsf{Prop}$ be a class such that $C(a)$ holds whenever $C(x)$ for all $x \in a$, and take any $v = \mathsf{set}(B, g)$. To show that $C(v)$ by induction, assume that $C(g(b))$ for all $b : B$. For every $x \in v$ there merely exists some $b : B$ with $x = g(b)$, and so $C(x)$. Thus $C(v)$.

(viii) Replacement: the statement "C is a V-set" is a mere proposition, so we may proceed by induction as follows. Supposing x is $\mathsf{set}(A, f)$, we claim that $w :\equiv \mathsf{set}(A, r \circ f)$ is the set we are looking for. If $C(y)$ then there merely exists $z : V$ and $a : A$ such that $z = f(a)$ and $y = r(z)$, therefore $y \in w$. Conversely, if $y \in w$ then there merely exists $a : A$ such that $y = r(f(a))$, so if we take $z :\equiv f(a)$ we see that $C(y)$ holds.

(ix) Let us say that a class $C : V \to \mathsf{Prop}$ is **separable** if for any $a : V$ the class

$$a \cap C :\equiv \{\, x \mid x \in a \wedge C(x) \,\}$$

is a V-set. We need to show that any $\mathcal{U}$-small $C : V \to \mathsf{Prop}_{\mathcal{U}}$ is separable. Indeed, given $a = \mathsf{set}(A, f)$, let $A' = \exists(x : A).\, C(fx)$, and take $f' = f \circ i$, where $i : A' \to A$ is the obvious inclusion. Then we can take $a' = \mathsf{set}(A', f')$ and we have $x \in a \wedge C(x) \Leftrightarrow x \in a'$ as claimed. We needed the assumption that C lands in $\mathcal{U}$ in order for $A' = \exists(x : A).\, C(fx)$ to be in $\mathcal{U}$. □

It is also convenient to have a strictly syntactic criterion of separability, so that one can read off from the expression for a class that it produces a V-set. One such familiar condition is being "Δ_0", which means that the expression is built up from equality $x =_V y$ and membership $x \in y$, using only mere-propositional connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and quantifers $\forall$, $\exists$ over particular sets, i.e. of the form $\exists(x \in a)$ and $\forall(y \in b)$.

Corollary 10.5.8. *If the class $C : V \to \mathsf{Prop}$ is Δ_0 in the above sense, then it is separable.*

Proof. Recall that we have a $\mathcal{U}$-small resizing $x = y$ of identity $x = y$. Since $x \in y$ is defined in terms of $x = y$, we also have a $\mathcal{U}$-small resizing of membership

$$x \mathrel{\widetilde{\in}} \mathsf{set}(A, f) :\equiv \exists(a : A).\, x = f(a).$$

Now, let Φ be a Δ_0 expression for C, so that as classes $\Phi = C$ (strictly speaking, we should distinguish expressions from their meanings, but we will blur the difference). Let $\widetilde{\Phi}$ be the result of replacing all occurrences of $=$ and $\in$ by their resized equivalents $\sim$ and $\mathrel{\widetilde{\in}}$. Clearly then $\widetilde{\Phi}$ also expresses C, in the sense that for all $x : V$, $\widetilde{\Phi}(x) \Leftrightarrow C(x)$, and hence $\widetilde{\Phi} = C$ by univalence. It now suffices to show that $\widetilde{\Phi}$ is $\mathcal{U}$-small, for then it will be separable by the theorem.

We show that $\widetilde{\Phi}$ is $\mathcal{U}$-small by induction on the construction of the expression. The base cases are $x = y$ and $x \mathrel{\widetilde{\in}} y$, which have already been resized into $\mathcal{U}$. It is also clear that $\mathcal{U}$ is closed under the mere-propositional operations (and (-1)-truncation), so it just remains to check the bounded quantifiers $\exists(x \in a)$ and $\forall(y \in b)$. By definition,

$$\begin{aligned} \exists(x \in a)P(x) &:\equiv \left\| \sum_{x:V} (x \mathrel{\widetilde{\in}} a \wedge P(x)) \right\| \\ \forall(y \in b)P(x) &:\equiv \prod_{x:V} (x \mathrel{\widetilde{\in}} a \to P(x)) \end{aligned}$$

Let us consider $\left\| \sum_{(x:V)} (x \mathrel{\widetilde{\in}} a \wedge P(x)) \right\|$. Although the body $(x \mathrel{\widetilde{\in}} a \wedge P(x))$ is $\mathcal{U}$-small since $P(x)$ is so by the induction hypothesis, the quantification over V need not stay inside $\mathcal{U}$. However, in the present case we can replace this with a quantification over the type $[a] \rightarrowtail V$ of members of a, and easily show that

$$\sum_{x:V} (x \mathrel{\widetilde{\in}} a \wedge P(x)) = \sum_{x:[a]} P(x).$$

The righthand side does remain in $\mathcal{U}$, since both $[a]$ and $P(x)$ are in $\mathcal{U}$. The case of $\prod_{(x:V)}(x \mathrel{\widetilde{\in}} a \to P(x))$ is analogous, using $\prod_{(x:V)}(x \mathrel{\widetilde{\in}} a \to P(x)) = \prod_{(x:[a])} P(x)$. □

We have shown that in type theory with a universe $\mathcal{U}$, the cumulative hierarchy V is a model of a "constructive set theory" with many of the standard axioms. However, as far as we know, it lacks the *strong collection* and *subset collection* axioms which are included in Constructive Zermelo–Fraenkel Set Theory [Acz78]. We do not expect these to hold, since they seem to be related to the interaction between the type-theoretic axiom of choice AC_∞ and the (setoid-like) definition of equality in the usual interpretation of this set theory into type theory. Indeed, our model $(V, \in)$ is based on a cumulative hierarchy V which is a higher inductive type *inside* the system, rather than being an *external* construction, and so it is not surprising that it differs in some ways from prior interpretations.

Finally, consider the result of adding the axiom of choice for sets to our type theory, in the form AC from §10.1.5 above. This has the consequence that LEM then also holds, by Theorem 10.1.14, and so Set is a topos with subobject classifier **2**, by Theorem 10.1.12. In this case, we have $\mathsf{Prop} = \mathbf{2} : \mathcal{U}$, and so *all classes are separable*. Thus we have shown:

Lemma 10.5.9. *In type theory with* AC*, the law of* ***(full) separation*** *holds for V: given* any *class $C : V \to \mathsf{Prop}$ and $a : V$, the class $a \cap C$ is a V-set.*

Theorem 10.5.10. *In type theory with* AC *and a universe $\mathcal{U}$, the cumulative hierarchy V is a model of Zermelo–Fraenkel set theory with choice, ZFC.*

Proof. We have all the axioms listed in Theorem 10.5.7, plus full separation, so we just need to show that there are power sets $\mathcal{P}(a) : V$ for all $a : V$. But since we have LEM these are simply function types $\mathcal{P}(a) = (a \to \mathbf{2})$. □

Notes

The basic properties one expects of the category of sets date back to the early days of elementary topos theory. The *Elementary theory of the category of sets* referred to in §10.1.5 was introduced by Lawvere in [Law05], as a category-theoretic axiomatization of set theory. The notion of ΠW-pretopos, regarded as a predicative version of an elementary topos, was introduced in [MP02]; see also [Pal09].

The treatment of the category of sets in §10.1 roughly follows that in [RS13]. The fact that epimorphisms are surjective (Lemma 10.1.4) is well known in classical mathematics, but is not as trivial as it may seem to prove *predicatively*. The proof in [MRR88] uses the power set operation (which is impredicative), although it can also be seen as a predicative proof of the weaker statement that a map in a universe $\mathcal{U}_i$ is surjective if it is an epimorphism in the next universe $\mathcal{U}_{i+1}$. A predicative proof for setoids was given by Wilander [Wil10]. Our proof is similar to Wilander's, but avoids setoids by using pushouts and univalence.

The implication in Theorem 10.1.14 from AC to LEM is an adaptation to homotopy type theory of a theorem from topos theory due to Diaconescu [Dia75]; it was posed as a problem already by Bishop [Bis67, Problem 2,p58].

For the intuitionistic theory of ordinal numbers, see [Tay96] and also [JM95]. Definitions of well-foundedness in type theory by an induction principle, including the inductive predicate of accessibility, were studied in [Hue80, Pau86, Nor88], although the idea dates back to Getzen's proof of the consistency of arithmetic [Gen36].

The idea of algebraic set theory, which informs our development in §10.5 of the cumulative hierarchy, is due to [JM95], but it derives from earlier work by [Acz78].

Exercises

Exercise 10.1. Following the pattern of $\mathcal{S}et$, we would like to make a category $\mathcal{T}ype$ of all types and maps between them (in a given universe $\mathcal{U}$). In order for this to be a category in the sense of §9.1, however, we must first declare $\hom(X, Y) :\equiv \|X \to Y\|_0$, with composition defined by

induction on truncation from ordinary composition $(Y \to Z) \to (X \to Y) \to (X \to Z)$. This was defined as the *homotopy precategory of types* in Example 9.1.18. It is still not a category, however, but only a precategory (its type of objects $\mathcal{U}$ is not even a 0-type). It becomes a category by Rezk completion (see Example 9.9.7), and its type of objects can be identified with $\|\mathcal{U}\|_1$ by Exercise 9.9. Show that the resulting category $\mathcal{T}ype$, unlike $\mathcal{S}et$, is not a pretopos.

Exercise 10.2. Show that if every surjection has a section in the category $\mathcal{S}et$, then the axiom of choice holds.

Exercise 10.3. Show that with AC, the category $\mathcal{S}et$ is well-pointed, in the sense that the following statement holds: for any $f, g : A \to B$, if $f \neq g$ then there is a function $a : 1 \to A$ such that $f(a) \neq g(a)$. Show that the "slice category" $\mathcal{S}et/\mathbf{2}$ consisting of functions $A \to \mathbf{2}$ and commutative triangles does not have this property. (Hint: the terminal object in $\mathcal{S}et/\mathbf{2}$ is the identity function $\mathbf{2} \to \mathbf{2}$, so in this category, there are objects X that have no elements $1 \to X$.)

Exercise 10.4. Prove that if $(A, <_A)$ and $(B, <_B)$ are well-founded, extensional, or ordinals, then so is $A + B$, with $<$ defined by

$$\begin{aligned}
(a < a') &:\equiv (a <_A a') && \text{for } a, a' : A \\
(b < b') &:\equiv (b <_B b') && \text{for } b, b' : B \\
(a < b) &:\equiv \mathbf{1} && \text{for } (a : A), (b : B) \\
(b < a) &:\equiv \emptyset && \text{for } (a : A), (b : B)
\end{aligned}$$

Exercise 10.5. Prove that if $(A, <_A)$ and $(B, <_B)$ are well-founded, extensional, or ordinals, then so is $A \times B$, with $<$ defined by

$$((a, b) < (a', b')) :\equiv (a <_A a') \vee ((a = a') \wedge (b <_B b')).$$

Exercise 10.6. Define the usual algebraic operations on ordinals, and prove that they satisfy the usual properties.

Exercise 10.7. Note that $\mathbf{2}$ is an ordinal, under the obvious relation $<$ such that $0_{\mathbf{2}} < 1_{\mathbf{2}}$ only.

(i) Define a relation $<$ on Prop which makes it into an ordinal.
(ii) Show that $\mathbf{2} =_{\mathsf{Ord}} \mathsf{Prop}$ if and only if LEM holds.

Exercise 10.8. Recall that we denote $\mathbb{N}$ by ω when regarding it as an ordinal; thus we have also the ordinal $\omega + 1$. On the other hand, let us define

$$\mathbb{N}_\infty :\equiv \left\{ a : \mathbb{N} \to \mathbf{2} \;\middle|\; \forall (n : \mathbb{N}).\, (a_n \leq a_{\mathsf{succ}(n)}) \right\}$$

where $\leq$ denotes the obvious partial order on $\mathbf{2}$, with $0_{\mathbf{2}} \leq 1_{\mathbf{2}}$.

(i) Define a relation $<$ on $\mathbb{N}_\infty$ which makes it into an ordinal.
(ii) Show that $\omega + 1 =_{\mathsf{Ord}} \mathbb{N}_\infty$ if and only if the limited principle of omniscience (11.5.8) holds.

Exercise 10.9. Show that if $(A, <)$ is well-founded and extensional and $A : \mathcal{U}$, then there is a simulation $A \to V$, where $(V, \in)$ is the cumulative hierarchy from §10.5 built from the universe $\mathcal{U}$.

Exercise 10.10. Given types A and B, define a **bitotal relation** to be $R : A \to B \to \mathsf{Prop}$ such that

$$\Big(\forall(a : A).\, \exists(b : B).\, R(a,b)\Big) \wedge \Big(\forall(b : B).\, \exists(a : A).\, R(a,b)\Big).$$

For such A, B, R, let $A \sqcup^R B$ be the higher inductive type generated by

- $i : A \to A \sqcup^R B$
- $j : B \to A \sqcup^R B$
- For each $a : A$ and $b : B$ such that $R(a,b)$, a path $i(a) = j(b)$.

Show that the cumulative hierarchy V can be defined by the following more straightforward list of constructors, and that the resulting induction principle is the one given in §10.5.

- For every $A : \mathcal{U}$ and $f : A \to V$, there is an element $\mathsf{set}(A, f) : V$.
- For any $A, B : \mathcal{U}$ and bitotal relation $R : A \to B \to \mathsf{Prop}$, and any map $h : A \sqcup^R B \to V$, there is a path $\mathsf{set}(A, h \circ i) = \mathsf{set}(B, h \circ j)$.
- The 0-truncation constructor.

Exercise 10.11. In Constructive Zermelo–Fraenkel Set Theory, the **axiom of strong collection** has the form:

$$\forall(x \in v).\, \exists(y).\, R(x,y) \Rightarrow \\ \exists(w).\, \big[\big(\forall(x \in v).\, \exists(y \in w).\, R(x,y)\big) \wedge \big(\forall(y \in w).\, \exists(x \in v).\, R(x,y)\big)\big]$$

Does it hold in the cumulative hierarchy V? (We do not know the answer to this.)

Chapter 11

Real numbers

Any foundation of mathematics worthy of its name must eventually address the construction of real numbers as understood by mathematical analysis, namely as a complete archimedean ordered field. Completeness can be understood either in the sense of Cauchy or Dedekind. In this chapter we shall look at both, Dedekind's construction of reals as cuts, and Cauchy reals as completion of rational numbers by limits of Cauchy sequences. The Dedekind reals include the Cauchy real numbers. Without any extra assumptions we cannot hope to show the converse as we will discuss in §11.4.

After completing some basic algebraic operations on the real numbers we consider three abstract notion of compactness. To show that the unit interval is sequentially compact requires classical logic. However, we can readily prove that the interval is totally bounded and complete. The most interesting property is Heine-Borel compactness. It can be show to hold for so-called pointfree coverings.

We close the section by constructing Conway's surreals as a higher inductive type in §11.6. It is worth pointing out that this construction is much more natural in univalent type theory than in classical set theory.

It is worth pointing out that the total space of the universal cover of the circle, which in §8.1.5 played a role similar to "the real numbers" in classical algebraic topology, is *not* the type of reals we are looking for. That type is contractible, and thus equivalent to the singleton type, so it cannot be equipped with a non-trivial algebraic structure.

11.1 The field of rational numbers

We first construct the rational numbers $\mathbb{Q}$, as the reals can then be seen as a completion of $\mathbb{Q}$. An expert will point out that $\mathbb{Q}$ could be replaced by any approximate field, i.e., a subring of $\mathbb{Q}$ in which arbitrarily precise approximate inverses exist. An example is the ring of dyadic rationals, which are those of the form $n/2^k$. If we were implementing constructive mathematics on a computer, an approximate field would be more suitable, but we leave such finesse for those who care about the digits of π.

We constructed the integers $\mathbb{Z}$ in §6.10 as a quotient of $\mathbb{N} \times \mathbb{N}$, and observed that this quotient

is generated by an idempotent. In §6.11 we saw that $\mathbb{Z}$ is the free group on $\mathbf{1}$; we could similarly show that it is the free commutative ring on $\mathbf{0}$. The field of rationals $\mathbb{Q}$ is constructed along the same lines as well, namely as the quotient

$$\mathbb{Q} :\equiv (\mathbb{Z} \times \mathbb{N})/\approx$$

where

$$(u, a) \approx (v, b) :\equiv (u(b+1) = v(a+1)).$$

In other words, a pair (u, a) represents the rational number $u/(1+a)$. There can be no division by zero because we cunningly added one to the denominator a. Here too we have a canonical choice of representatives, namely fractions in lowest terms. Thus we may apply Lemma 6.10.8 to obtain a set $\mathbb{Q}$, which again has a decidable equality.

We do not bother to write down the arithmetical operations on $\mathbb{Q}$ as we trust our readers know how to compute with fractions even in the case when one is added to the denominator. Let us just record the conclusion that there is an entirely unproblematic construction of the ordered field of rational numbers $\mathbb{Q}$, with a decidable equality and decidable order. It can also be characterized as the initial ordered field.

Let $\mathbb{Q}_+ = \{\, q : \mathbb{Q} \mid q > 0 \,\}$ be the type of positive rational numbers.

11.2 Dedekind reals

Let us first recall the basic idea of Dedekind's construction. We use two-sided Dedekind cuts, as opposed to an often used one-sided version, because the symmetry makes constructions more elegant, and it works constructively as well as classically. A *Dedekind cut* consists of a pair (L, U) of subsets $L, U \subseteq \mathbb{Q}$, called the *lower* and *upper cut* respectively, which are:

(i) *inhabited*: there are $q \in L$ and $r \in U$,
(ii) *rounded:* $q \in L \Leftrightarrow \exists(r \in \mathbb{Q}).\, q < r \in L$ and $r \in U \Leftrightarrow \exists(q \in \mathbb{Q}).\, U \ni q < r$,
(iii) *disjoint:* $\neg(q \in L \wedge q \in U)$, and
(iv) *located:* $q < r \Rightarrow q \in L \vee r \in U$.

Reading the roundedness condition from left to right tells us that cuts are *open*, and from right to left that they are *lower*, respectively *upper*, sets. The locatedness condition states that there is no large gap between L and U. Because cuts are always open, they never include the "point in between", even when it is rational. A typical Dedekind cut looks like this:

We might naively translate the informal definition into type theory by saying that a cut is a pair of maps $L, U : \mathbb{Q} \to \mathsf{Prop}$. But we saw in §3.5 that Prop is an ambiguous notation for $\mathsf{Prop}_{\mathcal{U}_i}$ where $\mathcal{U}_i$ is a universe. Once we use a particular $\mathcal{U}_i$ to define cuts, the type of reals will reside in the next universe $\mathcal{U}_{i+1}$, a property of reals two levels higher in $\mathcal{U}_{i+2}$, a property of subsets of reals in $\mathcal{U}_{i+3}$, etc. In principle we should be able to keep track of the universe levels, especially with the help of a proof assistant, but doing so here would just burden us with bureaucracy

that we prefer to avoid. We shall therefore make a simplifying assumption that a single type of propositions Ω is sufficient for all our purposes.

In fact, the construction of the Dedekind reals is quite resilient to logical manipulations. There are several ways in which we can make sense of using a single type Ω:

(i) We could identify Ω with the ambiguous Prop and track all the universes that appear in definitions and constructions.

(ii) We could assume impredicativity of mere propositions, cf. §3.5, which essentially collapses the $\mathsf{Prop}_{\mathcal{U}_i}$'s to the lowest level, which we call Ω.

(iii) A classical mathematician who is not interested in the intricacies of type-theoretic universes or computation may simply assume the law of excluded middle (3.4.1) for mere propositions so that $\Omega \equiv \mathbf{2}$. This not only eradicates questions about levels of Prop, but also turns everything we do into the standard classical construction of real numbers. We demonstrate this point further in §11.5.

(iv) On the other end of the spectrum one might ask for a minimal requirement that makes the constructions work. The condition that a mere predicate be a Dedekind cut is expressible using only conjunctions, disjunctions, and existential quantifiers over $\mathbb{Q}$, which is a countable set. Thus we could take Ω to be the initial *σ-frame*, i.e., a lattice with countable joins in which binary meets distribute over countable joins. (The initial σ-frame cannot be the two-point lattice $\mathbf{2}$ because $\mathbf{2}$ is not closed under countable joins, unless we assume excluded middle.) This would lead to a construction of Ω as a higher inductive-inductive type, but one experiment of this kind in §11.3 is enough.

In all of the above cases Ω is a set. Without further ado, we translate the informal definition into type theory. Throughout this chapter, we use the logical notation from Definition 3.7.1.

Definition 11.2.1. A **Dedekind cut** is a pair (L, U) of mere predicates $L : \mathbb{Q} \to \Omega$ and $U : \mathbb{Q} \to \Omega$ which is:

(i) *inhabited*: $\exists(q : \mathbb{Q}).\, L(q)$ and $\exists(r : Q).\, U(r)$,

(ii) *rounded:* for all $q, r : \mathbb{Q}$,

$$L(q) \Leftrightarrow \exists(r : \mathbb{Q}).\, (q < r) \wedge L(r) \quad \text{and} \quad U(r) \Leftrightarrow \exists(q : \mathbb{Q}).\, (q < r) \wedge U(q),$$

(iii) *disjoint:* $\neg(L(q) \wedge U(q))$ for all $q : \mathbb{Q}$,

(iv) *located:* $(q < r) \Rightarrow L(q) \vee U(r)$ for all $q, r : \mathbb{Q}$.

We let $\mathsf{isCut}(L, U)$ denote the conjunction of these conditions. The type of **Dedekind reals** is

$$\mathbb{R}_{\mathsf{d}} :\equiv \{\, (L, U) : (\mathbb{Q} \to \Omega) \times (\mathbb{Q} \to \Omega) \mid \mathsf{isCut}(L, U) \,\}.$$

It is apparent that $\mathsf{isCut}(L, U)$ is a mere proposition, and since $\mathbb{Q} \to \Omega$ is a set the Dedekind reals form a set too. See Exercises 11.2 to 11.4 for variants of Dedekind cuts which lead to extended reals, lower and upper reals, and the interval domain.

There is an embedding $\mathbb{Q} \to \mathbb{R}_{\mathsf{d}}$ which associates with each rational $q : \mathbb{Q}$ the cut (L_q, U_q) where

$$L_q(r) :\equiv (r < q) \qquad \text{and} \qquad U_q(r) :\equiv (q < r).$$

We shall simply write q for the cut (L_q, U_q) associated with a rational number.

11.2.1 The algebraic structure of Dedekind reals

The construction of the algebraic and order-theoretic structure of Dedekind reals proceeds as usual in intuitionistic logic. Rather than dwelling on details we point out the differences between the classical and intuitionistic setup. Writing L_x and U_x for the lower and upper cut of a real number $x : \mathbb{R}_{\mathsf{d}}$, we define addition as

$$\begin{aligned} L_{x+y}(q) &:\equiv \exists(r,s:\mathbb{Q}).\, L_x(r) \wedge L_y(s) \wedge q = r+s, \\ U_{x+y}(q) &:\equiv \exists(r,s:\mathbb{Q}).\, U_x(r) \wedge U_y(s) \wedge q = r+s \end{aligned}$$

and the additive inverse by

$$\begin{aligned} L_{-x}(q) &:\equiv \exists(r:\mathbb{Q}).\, U_x(r) \wedge q = -r, \\ U_{-x}(q) &:\equiv \exists(r:\mathbb{Q}).\, L_x(r) \wedge q = -r. \end{aligned}$$

With these operations $(\mathbb{R}_{\mathsf{d}}, 0, +, -)$ is an abelian group. Multiplication is a bit more cumbersome:

$$\begin{aligned} L_{x\cdot y}(q) :\equiv\ & \exists(a,b,c,d:\mathbb{Q}).\, L_x(a) \wedge U_x(b) \wedge L_y(c) \wedge U_y(d) \wedge \\ & \qquad q < \min(a\cdot c, a\cdot d, b\cdot c, b\cdot d), \\ U_{x\cdot y}(q) :\equiv\ & \exists(a,b,c,d:\mathbb{Q}).\, L_x(a) \wedge U_x(b) \wedge L_y(c) \wedge U_y(d) \wedge \\ & \qquad \max(a\cdot c, a\cdot d, b\cdot c, b\cdot d) < q. \end{aligned}$$

These formulas are related to multiplication of intervals in interval arithmetic, where intervals $[a,b]$ and $[c,d]$ with rational endpoints multiply to the interval

$$[a,b]\cdot[c,d] = [\min(ac,ad,bc,bd), \max(ac,ad,bc,bd)].$$

For instance, the formula for the lower cut can be read as saying that $q < x\cdot y$ when there are intervals $[a,b]$ and $[c,d]$ containing x and y, respectively, such that q is to the left of $[a,b]\cdot[c,d]$. It is generally useful to think of an interval $[a,b]$ such that $L_x(a)$ and $U_x(b)$ as an approximation of x, see Exercise 11.4.

We now have a commutative ring with unit $(\mathbb{R}_{\mathsf{d}}, 0, 1, +, -, \cdot)$. To treat multiplicative inverses, we must first introduce order. Define $\leq$ and $<$ as

$$\begin{aligned} (x \leq y) &:\equiv \forall(q:\mathbb{Q}).\, L_x(q) \Rightarrow L_y(q), \\ (x < y) &:\equiv \exists(q:\mathbb{Q}).\, U_x(q) \wedge L_y(q). \end{aligned}$$

Lemma 11.2.2. *For all $x : \mathbb{R}_{\mathsf{d}}$ and $q, r : \mathbb{Q}$, $L_x(q) \Leftrightarrow (q < x)$ and $U_x(r) \Leftrightarrow (x < r)$.*

Proof. If $L_x(q)$ then by roundedness there merely is $r > q$ such that $L_x(r)$, and since $U_q(r)$ it follows that $q < x$. Conversely, if $q < x$ then there is $r : \mathbb{Q}$ such that $U_q(r)$ and $L_x(r)$, hence $L_x(q)$ because L_x is a lower set. The other half of the proof is symmetric. □

The relation $\leq$ is a partial order, and $<$ is transitive and irreflexive. The trichotomy law

$$(x < y) \vee (x = y) \vee (y < x)$$

is valid if we assume excluded middle, but without it we get the intuitionistic version

$$(x < y) \Rightarrow (x < z) \vee (z < y). \tag{11.2.3}$$

To see this, suppose $x < y$. Then there merely exists $q : \mathbb{Q}$ such that $U_x(q)$ and $L_y(q)$. By roundedness there merely exist $r, s : \mathbb{Q}$ such that $r < q < s$, $U_x(r)$ and $L_y(s)$. Then, by locatedness $L_z(r)$ or $U_z(s)$. In the first case we get $x < z$ and in the second $z < y$. At first sight it might not be clear what (11.2.3) has to do with the linear order. But if we take $x \equiv u - \epsilon$ and $y \equiv u + \epsilon$ for $\epsilon > 0$, then we get

$$(u - \epsilon < z) \vee (z < u + \epsilon).$$

This is linear order "up to a small numerical error", i.e., since it is unreasonable to expect that we can actually compute with infinite precision, we should not be surprised that we can decide $<$ only up to whatever finite precision we have computed.

Classically, multiplicative inverses exist for all numbers which are different from zero. However, without excluded middle, a stronger condition is required. Say that $x, y : \mathbb{R}_d$ are **apart** from each other, written $x \# y$, when $(x < y) \vee (y < x)$:

$$(x \# y) :\equiv (x < y) \vee (y < x).$$

If $x \# y$, then $\neg(x = y)$. The converse is true if we assume excluded middle, but is not provable constructively. Indeed, if $\neg(x = y)$ implies $x \# y$, then a little bit of excluded middle follows; see Exercise 11.9.

Theorem 11.2.4. *A real is invertible if, and only if, it is apart from* 0*.*

Remark 11.2.5. We observe that a real is invertible if, and only if, it is merely invertible. Indeed, the same is true in any ring, since a ring is a set, and multiplicative inverses are unique if they exist. See the discussion following Corollary 3.9.2.

Proof. Suppose $x \cdot y = 1$. Then there merely exist $a, b, c, d : \mathbb{Q}$ such that $a < x < b$, $c < y < d$ and $0 < \min(ac, ad, bc, bd)$. From $0 < ac$ it follows that $0 < a < x < c$ or $a < x < c < 0$, hence $x \# 0$.

Conversely, if $x \# 0$ then

$$\begin{aligned} L_{x^{-1}}(q) &:\equiv \exists(r : \mathbb{Q}).\, U_x(r) \wedge ((0 < r \wedge qr < 1) \vee (r < 0 \wedge 1 < qr)) \\ U_{x^{-1}}(q) &:\equiv \exists(r : \mathbb{Q}).\, L_x(r) \wedge ((0 < r \wedge qr > 1) \vee (r < 0 \wedge 1 > qr)) \end{aligned}$$

defines the desired inverse. Indeed, $L_{x^{-1}}$ and $U_{x^{-1}}$ are bounded because $x \# 0$. □

The archimedean principle can be stated in several ways. We find it most illuminating in the form which says that $\mathbb{Q}$ is dense in $\mathbb{R}_d$.

Theorem 11.2.6 (Archimedean principle for $\mathbb{R}_d$)**.** *For all $x, y : \mathbb{R}_d$ if $x < y$ then there merely exists $q : \mathbb{Q}$ such that $x < q < y$.*

Proof. By definition of $<$. □

Before tackling completeness of Dedekind reals, let us state precisely what algebraic structure they possess. In the following definition we are not aiming at a minimal axiomatization, but rather at a useful amount of structure and properties.

Definition 11.2.7. An **ordered field** is a set F together with constants 0, 1, operations $+$, $-$, $\cdot$, min, max, and mere relations $\leq$, $<$, $\#$ such that:

(i) $(F, 0, 1, +, -, \cdot)$ is a commutative ring with unit;
(ii) $x : F$ is invertible if, and only if $x \# 0$;
(iii) $(F, \leq, \min, \max)$ is a lattice;
(iv) the strict order $<$ is transitive, irreflexive, and linear ($x < y \Rightarrow x < z \vee z < y$);
(v) apartness $\#$ is reflexive, symmetric and cotransitive ($x \# y \Rightarrow x \# z \vee y \# z$);
(vi) for all $x, y, z : F$:

$$\begin{aligned} x \leq y &\Leftrightarrow \neg(y < x), & x < y \leq z &\Rightarrow x < z, \\ x \# y &\Leftrightarrow (x < y) \vee (y < x), & x \leq y < z &\Rightarrow x < z, \\ x \leq y &\Leftrightarrow x + z \leq y + z, & x \leq y \wedge 0 \leq z &\Rightarrow xz \leq yz, \\ x < y &\Leftrightarrow x + z < y + z, & 0 < z &\Rightarrow (x < y \Leftrightarrow xz < yz), \\ 0 < x + y &\Rightarrow 0 < x \vee 0 < y, & 0 &< 1. \end{aligned}$$

Every such field has a canonical embedding $\mathbb{Q} \to F$. An ordered field is **archimedean** when for all $x, y : F$, if $x < y$ then there merely exists $q : \mathbb{Q}$ such that $x < q < y$.

Theorem 11.2.8. *The Dedekind reals form an ordered archimedean field.*

Proof. We omit the proof in the hope that what we have demonstrated so far makes the theorem plausible. □

11.2.2 Dedekind reals are Cauchy complete

Recall that $x : \mathbb{N} \to \mathbb{Q}$ is a *Cauchy sequence* when it satisfies

$$\prod_{(\epsilon:\mathbb{Q}_+)} \sum_{(n:\mathbb{N})} \prod_{m,k \geq n} |x_m - x_k| < \epsilon. \tag{11.2.9}$$

Note that we did *not* truncate the inner existential because we actually want to compute rates of convergence—an approximation without an error estimate carries little useful information. By Theorem 2.15.7, (11.2.9) yields a function $M : \mathbb{Q}_+ \to \mathbb{N}$, called the *modulus of convergence*, such that $m, k \geq M(\epsilon)$ implies $|x_m - x_k| < \epsilon$. From this we get $|x_{M(\delta/2)} - x_{M(\epsilon/2)}| < \delta + \epsilon$ for all $\epsilon : \mathbb{Q}_+$. In fact, the map $(\epsilon \mapsto x_{M(\epsilon/2)}) : \mathbb{Q}_+ \to \mathbb{Q}$ carries the same information about the limit as the original Cauchy condition (11.2.9). We shall work with these approximation functions rather than with Cauchy sequences.

Definition 11.2.10. A **Cauchy approximation** is a map $x : \mathbb{Q}_+ \to \mathbb{R}_{\mathsf{d}}$ which satisfies

$$\forall(\delta, \epsilon : \mathbb{Q}_+).\, |x_\delta - x_\epsilon| < \delta + \epsilon. \tag{11.2.11}$$

The **limit** of a Cauchy approximation $x : \mathbb{Q}_+ \to \mathbb{R}_\mathsf{d}$ is a number $\ell : \mathbb{R}_\mathsf{d}$ such that

$$\forall(\epsilon, \theta : \mathbb{Q}_+). |x_\epsilon - \ell| < \epsilon + \theta.$$

Theorem 11.2.12. *Every Cauchy approximation in $\mathbb{R}_\mathsf{d}$ has a limit.*

Proof. Note that we are showing existence, not mere existence, of the limit. Given a Cauchy approximation $x : \mathbb{Q}_+ \to \mathbb{R}_\mathsf{d}$, define

$$L_y(q) :\equiv \exists(\epsilon, \theta : \mathbb{Q}_+). L_{x_\epsilon}(q + \epsilon + \theta),$$
$$U_y(q) :\equiv \exists(\epsilon, \theta : \mathbb{Q}_+). U_{x_\epsilon}(q - \epsilon - \theta).$$

It is clear that L_y and U_y are bounded, rounded, and disjoint. To establish locatedness, consider any $q, r : \mathbb{Q}$ such that $q < r$. There is $\epsilon : \mathbb{Q}_+$ such that $5\epsilon < r - q$. Since $q + 2\epsilon < r - 2\epsilon$ merely $L_{x_\epsilon}(q + 2\epsilon)$ or $U_{x_\epsilon}(r - 2\epsilon)$. In the first case we have $L_y(q)$ and in the second $U_y(r)$.

To show that y is the limit of x, consider any $\epsilon, \theta : \mathbb{Q}_+$. Because $\mathbb{Q}$ is dense in $\mathbb{R}_\mathsf{d}$ there merely exist $q, r : \mathbb{Q}$ such that

$$x_\epsilon - \epsilon - \theta/2 < q < x_\epsilon - \epsilon - \theta/4 < x_\epsilon < x_\epsilon + \epsilon + \theta/4 < r < x_\epsilon + \epsilon + \theta/2,$$

and thus $q < y < r$. Now either $y < x_\epsilon + \theta/2$ or $x_\epsilon - \theta/2 < y$. In the first case we have

$$x_\epsilon - \epsilon - \theta/2 < q < y < x_\epsilon + \theta/2,$$

and in the second

$$x_\epsilon - \theta/2 < y < r < x_\epsilon + \epsilon + \theta/2.$$

In either case it follows that $|y - x_\epsilon| < \epsilon + \theta$. □

For sake of completeness we record the classic formulation as well.

Corollary 11.2.13. *Suppose $x : \mathbb{N} \to \mathbb{R}_\mathsf{d}$ satisfies the Cauchy condition (11.2.9). Then there exists $y : \mathbb{R}_\mathsf{d}$ such that*

$$\prod_{(\epsilon:\mathbb{Q}_+)} \sum_{(n:\mathbb{N})} \prod_{m \geq n} |x_m - y| < \epsilon.$$

Proof. By Theorem 2.15.7 there is $M : \mathbb{Q}_+ \to \mathbb{N}$ such that $\bar{x}(\epsilon) :\equiv x_{M(\epsilon/2)}$ is a Cauchy approximation. Let y be its limit, which exists by Theorem 11.2.12. Given any $\epsilon : \mathbb{Q}_+$, let $n :\equiv M(\epsilon/4)$ and observe that, for any $m \geq n$,

$$|x_m - y| \leq |x_m - x_n| + |x_n - y| = |x_m - x_n| + |\bar{x}(\epsilon/2) - y| < \epsilon/4 + \epsilon/2 + \epsilon/4 = \epsilon.$$ □

11.2.3 Dedekind reals are Dedekind complete

We obtained $\mathbb{R}_\mathsf{d}$ as the type of Dedekind cuts on $\mathbb{Q}$. But we could have instead started with any archimedean ordered field F and constructed Dedekind cuts on F. These would again form an archimedean ordered field $\bar{F}$, the **Dedekind completion of** F, with F contained as a subfield. What happens if we apply this construction to $\mathbb{R}_\mathsf{d}$, do we get even more real numbers? The answer is negative. In fact, we shall prove a stronger result: $\mathbb{R}_\mathsf{d}$ is final.

Say that an ordered field F is **admissible for** Ω when the strict order $<$ on F is a map $< : F \to F \to \Omega$.

Theorem 11.2.14. *Every archimedean ordered field which is admissible for Ω is a subfield of $\mathbb{R}_\mathsf{d}$.*

Proof. Let F be an archimedean ordered field. For every $x : F$ define $L, U : \mathbb{Q} \to \Omega$ by

$$L_x(q) :\equiv (q < x) \qquad \text{and} \qquad U_x(q) :\equiv (x < q).$$

(We have just used the assumption that F is admissible for Ω.) Then (L_x, U_x) is a Dedekind cut. Indeed, the cuts are inhabited and rounded because F is archimedean and $<$ is transitive, disjoint because $<$ is irreflexive, and located because $<$ is a linear order. Let $e : F \to \mathbb{R}_\mathsf{d}$ be the map $e(x) :\equiv (L_x, U_x)$.

We claim that e is a field embedding which preserves and reflects the order. First of all, notice that $e(q) = q$ for a rational number q. Next we have the equivalences, for all $x, y : F$,

$$x < y \Leftrightarrow (\exists (q : \mathbb{Q}).\, x < q < y) \Leftrightarrow (\exists (q : \mathbb{Q}).\, U_x(q) \wedge L_y(q)) \Leftrightarrow e(x) < e(y),$$

so e indeed preserves and reflects the order. That $e(x + y) = e(x) + e(y)$ holds because, for all $q : \mathbb{Q}$,

$$q < x + y \Leftrightarrow \exists (r, s : \mathbb{Q}).\, r < x \wedge s < y \wedge q = r + s.$$

The implication from right to left is obvious. For the other direction, if $q < x + y$ then there merely exists $r : \mathbb{Q}$ such that $q - y < r < x$, and by taking $s :\equiv q - r$ we get the desired r and s. We leave preservation of multiplication by e as an exercise. □

To establish that the Dedekind cuts on $\mathbb{R}_\mathsf{d}$ do not give us anything new, we need just one more lemma.

Lemma 11.2.15. *If F is admissible for Ω then so is its Dedekind completion.*

Proof. Let $\bar{F}$ be the Dedekind completion of F. The strict order on $\bar{F}$ is defined by

$$((L, U) < (L', U')) :\equiv \exists (q : \mathbb{Q}).\, U(q) \wedge L'(q).$$

Since $U(q)$ and $L'(q)$ are elements of Ω, the lemma holds as long as Ω is closed under conjunctions and countable existentials, which we assumed from the outset. □

Corollary 11.2.16. *The Dedekind reals are Dedekind complete: for every real-valued Dedekind cut (L, U) there is a unique $x : \mathbb{R}_\mathsf{d}$ such that $L(y) = (y < x)$ and $U(y) = (x < y)$ for all $y : \mathbb{R}_\mathsf{d}$.*

Proof. By Lemma 11.2.15 the Dedekind completion $\bar{\mathbb{R}}_\mathsf{d}$ of $\mathbb{R}_\mathsf{d}$ is admissible for Ω, so by Theorem 11.2.14 we have an embedding $\bar{\mathbb{R}}_\mathsf{d} \to \mathbb{R}_\mathsf{d}$, as well as an embedding $\mathbb{R}_\mathsf{d} \to \bar{\mathbb{R}}_\mathsf{d}$. But these embeddings must be isomorphisms, because their compositions are order-preserving field homomorphisms which fix the dense subfield $\mathbb{Q}$, which means that they are the identity. The corollary now follows immediately from the fact that $\bar{\mathbb{R}}_\mathsf{d} \to \mathbb{R}_\mathsf{d}$ is an isomorphism. □

11.3 Cauchy reals

The Cauchy reals are, by intent, the completion of $\mathbb{Q}$ under limits of Cauchy sequences. In the classical construction of the Cauchy reals, we consider the set $\mathcal{C}$ of all Cauchy sequences in $\mathbb{Q}$ and then form a suitable quotient $\mathcal{C}/\approx$. Then, to show that $\mathcal{C}/\approx$ is Cauchy complete, we consider a Cauchy sequence $x : \mathbb{N} \to \mathcal{C}/\approx$, lift it to a sequence of sequences $\bar{x} : \mathbb{N} \to \mathcal{C}$, and construct the limit of x using $\bar{x}$. However, the lifting of x to $\bar{x}$ uses the axiom of countable choice (the instance of (3.8.1) where $X = \mathbb{N}$) or the law of excluded middle, which we may wish to avoid. Every construction of reals whose last step is a quotient suffers from this deficiency. There are three common ways out of the conundrum in constructive mathematics:

(i) Pretend that the reals are a setoid $(\mathcal{C}, \approx)$, i.e., the type of Cauchy approximations $\mathcal{C}$ with a coincidence relation attached to it by administrative decree. A sequence of reals then simply *is* a sequence of Cauchy approximations representing them.

(ii) Give in to temptation and accept the Axiom of Countable Choice. After all, the axiom is valid in most models of constructive mathematics based on a computational viewpoint, such as realizability models.

(iii) Declare the Cauchy reals unworthy and construct the Dedekind reals instead. Such a verdict is perfectly valid in certain contexts, such as in sheaf-theoretic models of constructive mathematics. However, as we saw in §11.2, the constructive Dedekind reals have their own problems.

Using higher inductive types, however, there is a fourth solution, which we believe to be preferable to any of the above, and interesting even to a classical mathematician. The idea is that the Cauchy real numbers should be the *free complete metric space* generated by $\mathbb{Q}$. In general, the construction of a free gadget of any sort requires applying the gadget operations repeatedly many times to the generators. For instance, the elements of the free group on a set X are not just binary products and inverses of elements of X, but words obtained by iterating the product and inverse constructions. Thus, we might naturally expect the same to be true for Cauchy completion, with the relevant "operation" being "take the limit of a Cauchy sequence". (In this case, the iteration would have to take place transfinitely, since even after infinitely many steps there will be new Cauchy sequences to take the limit of.)

The argument referred to above shows that if excluded middle or Countable Choice hold, then Cauchy completion is very special: when building the completion of a space, it suffices to stop applying the operation after *one step*. This may be regarded as analogous to the fact that free monoids and free groups can be given explicit descriptions in terms of (reduced) words. However, we saw in §6.11 that higher inductive types allow us to construct free gadgets *directly*, whether or not there is also an explicit description available. In this section we show that the same is true for the Cauchy reals (a similar technique would construct the Cauchy completion of any metric space). Specifically, higher inductive types allow us to *simultaneously* add limits of Cauchy sequences and quotient by the coincidence relation, so that we can avoid the problem of lifting a sequence of reals to a sequence of representatives.

11.3.1 Construction of Cauchy reals

The construction of the Cauchy reals $\mathbb{R}_c$ as a higher inductive type is a bit more subtle than that of the free algebraic structures considered in §6.11. We intend to include a "take the limit" constructor whose input is a Cauchy sequence of reals, but the notion of "Cauchy sequence of reals" depends on having some way to measure the "distance" between real numbers. In general, of course, the distance between two real numbers will be another real number, leading to a potentially problematic circularity.

However, what we actually need for the notion of Cauchy sequence (or Cauchy approximation) of reals is not the general notion of "distance", but a way to say that "the distance between two real numbers is less than ϵ" for any $\epsilon : \mathbb{Q}_+$. This can be represented by a family of binary relations, which we will denote $\sim_\epsilon : \mathbb{R}_c \to \mathbb{R}_c \to \mathsf{Prop}$. The intended meaning of $x \sim_\epsilon y$ is $|x - y| < \epsilon$, but since we do not have notions of subtraction, absolute value, or inequality available yet (we are only just defining $\mathbb{R}_c$, after all), we will have to define these relations $\sim_\epsilon$ at the same time as we define $\mathbb{R}_c$ itself. And since $\sim_\epsilon$ is a type family indexed by two copies of $\mathbb{R}_c$, we cannot do this with an ordinary mutual (higher) inductive definition; instead we have to use a *higher inductive-inductive definition*.

Recall from §5.7 that the ordinary notion of inductive-inductive definition allows us to define a type and a type family indexed by it by simultaneous induction. Of course, the "higher" version of this allows both the type and the family to have path-constructors as well as point-constructors. We will not attempt to formulate any general theory of higher inductive-inductive definitions, but hopefully the description we will give of $\mathbb{R}_c$ and $\sim_\epsilon$ will make the idea transparent.

Remark 11.3.1. We might also consider a *higher inductive-recursive definition*, in which $\sim_\epsilon$ is defined using the *recursion* principle of $\mathbb{R}_c$, simultaneously with the *inductive* definition of $\mathbb{R}_c$. We choose the inductive-inductive route instead for two reasons. Firstly, higher inductive-recursive definitions seem to be more difficult to justify in homotopical semantics. Secondly, and more importantly, the inductive-inductive definition yields a more powerful induction principle, which we will need in order to develop even the basic theory of Cauchy reals.

Definition 11.3.2. Let $\mathbb{R}_c$ and the relation $\sim : \mathbb{Q}_+ \times \mathbb{R}_c \times \mathbb{R}_c \to \mathcal{U}$ be the following higher inductive-inductive type family. The type $\mathbb{R}_c$ of **Cauchy reals** is generated by the following constructors:

- *rational points:* for any $q : \mathbb{Q}$ there is a real $\mathsf{rat}(q)$.
- *limit points*: for any $x : \mathbb{Q}_+ \to \mathbb{R}_c$ such that

$$\forall(\delta, \epsilon : \mathbb{Q}_+).\, x_\delta \sim_{\delta+\epsilon} x_\epsilon \tag{11.3.3}$$

  there is a point $\mathsf{lim}(x) : \mathbb{R}_c$. We call x a **Cauchy approximation**.
- *paths:* for $u, v : \mathbb{R}_c$ such that

$$\forall(\epsilon : \mathbb{Q}_+).\, u \sim_\epsilon v \tag{11.3.4}$$

  then there is a path $\mathsf{eq}_{\mathbb{R}_c}(u, v) : u =_{\mathbb{R}_c} v$.

Simultaneously, the type family $\sim : \mathbb{R}_c \to \mathbb{R}_c \to \mathbb{Q}_+ \to \mathcal{U}$ is generated by the following constructors. Here q and r denote rational numbers; δ, ϵ, and η denote positive rationals; u and v denote Cauchy reals; and x and y denote Cauchy approximations:

- if $-\epsilon < q - r < \epsilon$, then $\mathsf{rat}(q) \sim_\epsilon \mathsf{rat}(r)$,
- if $\mathsf{rat}(q) \sim_{\epsilon-\delta} y_\delta$, then $\mathsf{rat}(q) \sim_\epsilon \lim(y)$,
- if $x_\delta \sim_{\epsilon-\delta} \mathsf{rat}(r)$, then $\lim(x) \sim_\epsilon \mathsf{rat}(r)$,
- if $x_\delta \sim_{\epsilon-\delta-\eta} y_\eta$, then $\lim(x) \sim_\epsilon \lim(y)$,
- if $\xi, \zeta : u \sim_\epsilon v$ then $\xi = \zeta$ (propositional truncation).

The first constructor of $\mathbb{R}_c$ says that any rational number can be regarded as a real number. The second says that from any Cauchy approximation to a real number, we can obtain a new real number called its "limit". And the third expresses the idea that if two Cauchy approximations coincide, then their limits are equal. (Note that the notion of "coincidence" is *simpler* than that of Definition 11.2.10, since we are free to use the relation $\sim$ between Cauchy reals themselves. Moreover, it also handles equalities between limits and rational points.)

The first four constructors of $\sim$ specify when two rational numbers are close, when a rational is close to a limit, and when two limits are close. In the case of two rational numbers, this is just the usual notion of ϵ-closeness for rational numbers, whereas the other cases can be derived by noting that each approximant x_δ is supposed to be within δ of the limit $\lim(x)$.

We remind ourselves of proof-relevance: a real number obtained from $\lim$ is represented not just by a Cauchy approximation x, but also a proof p of (11.3.3), so we should technically have written $\lim(x, p)$ instead of just $\lim(x)$. A similar observation also applies to $\mathsf{eq}_{\mathbb{R}_c}$ and (11.3.4), but we shall write just $\mathsf{eq}_{\mathbb{R}_c} : u = v$ instead of $\mathsf{eq}_{\mathbb{R}_c}(u, v, p) : u = v$. These abuses of notation are mitigated by the fact that we are omitting mere propositions and information that is readily guessed. Likewise, the last constructor of $\sim_\epsilon$ justifies our leaving the other four nameless.

We are immediately able to populate $\mathbb{R}_c$ with many real numbers. For suppose $x : \mathbb{N} \to \mathbb{Q}$ is a traditional Cauchy sequence of rational numbers, and let $M : \mathbb{Q}_+ \to \mathbb{N}$ be its modulus of convergence. Then $\mathsf{rat} \circ x \circ M : \mathbb{Q}_+ \to \mathbb{R}_c$ is a Cauchy approximation, using the first constructor of $\sim$ to produce the necessary witness. Thus, $\lim(\mathsf{rat} \circ x \circ m)$ is a real number. Various famous real numbers $\sqrt{2}$, π, e, ... are all limits of such Cauchy sequences of rationals.

11.3.2 Induction and recursion on Cauchy reals

In order to do anything useful with $\mathbb{R}_c$, of course, we need to give its induction principle. As is the case whenever we inductively define two or more objects at once, the basic induction principle for $\mathbb{R}_c$ and $\sim$ requires a simultaneous induction over both at once. Thus, we should expect it to say that assuming two type families over $\mathbb{R}_c$ and $\sim$, respectively, together with data corresponding to each constructor, there exist sections of both of these families. However, since $\sim$ is indexed on two copies of $\mathbb{R}_c$, the precise dependencies of these families is a bit subtle. The induction principle will apply to any pair of type families:

$$A : \mathbb{R}_c \to \mathcal{U}$$
$$B : \prod_{x,y:\mathbb{R}_c} A(x) \to A(y) \to \prod_{\epsilon:\mathbb{Q}_+} (x \sim_\epsilon y) \to \mathcal{U}.$$

The type of A is obvious, but the type of B requires a little thought. Since B must depend on $\sim$, but $\sim$ in turn depends on two copies of $\mathbb{R}_c$ and one copy of $\mathbb{Q}_+$, it is fairly obvious that B must also depend on the variables $x, y : \mathbb{R}_c$ and $\epsilon : \mathbb{Q}_+$ as well as an element of $(x \sim_\epsilon y)$. What is slightly less obvious is that B must also depend on $A(x)$ and $A(y)$.

This may be more evident if we consider the non-dependent case (the recursion principle), where A is a simple type. In this case we would expect B not to depend on $x, y : \mathbb{R}_c$ or $x \sim_\epsilon y$. But the recursion principle (along with its associated uniqueness principle) is supposed to say that $\mathbb{R}_c$ with $\sim_\epsilon$ is an "initial object" in some category, so in this case the dependency structure of A and B should mirror that of $\mathbb{R}_c$ and $\sim_\epsilon$: that is, we should have $B : A \to A \to \mathbb{Q}_+ \to \mathcal{U}$. Combining this observation with the fact that, in the dependent case, B must also depend on $x, y : \mathbb{R}_c$ and $x \sim_\epsilon y$, leads inevitably to the type given above for B.

Now, given A and B as above, the hypotheses of the induction principle consist of the following data, one for each constructor of $\mathbb{R}_c$ or $\sim$:

- For any $q : \mathbb{Q}$, an element $f_q : A(\mathsf{rat}(q))$.
- For any Cauchy approximation x, and any $a : \prod_{(\epsilon:\mathbb{Q}_+)} A(x_\epsilon)$ such that
$$\forall(\delta, \epsilon : \mathbb{Q}_+).\, B(x_\delta, x_\epsilon, a_\delta, a_\epsilon, \delta + \epsilon, \_), \tag{11.3.5}$$
an element $f_{x,a} : A(\mathsf{lim}(x))$. We call such a a **dependent Cauchy approximation** over x.
- For $u, v : \mathbb{R}_c$ and $h : \forall(\epsilon : \mathbb{Q}_+).\, u \sim_\epsilon v$, and $a : A(u)$ and $b : A(v)$ such that $\forall(\epsilon : \mathbb{Q}_+).\, B(u, v, a, b, \epsilon, h(\epsilon))$, a dependent path $a =^A_{\mathsf{eq}_{\mathbb{R}_c}(u,v)} b$.
- For $q, r : \mathbb{Q}$ and $\epsilon : \mathbb{Q}_+$, if $-\epsilon < q - r < \epsilon$, an element of $B(\mathsf{rat}(q), \mathsf{rat}(r), f_q, f_r, \epsilon, \_)$.
- For $q : \mathbb{Q}$ and $\delta, \epsilon : \mathbb{Q}_+$ and y a Cauchy approximation, and b a dependent Cauchy approximation over y, if $\mathsf{rat}(q) \sim_{\epsilon-\delta} y_\delta$, then
$$B(\mathsf{rat}(q), y_\delta, f_q, b_\delta, \epsilon - \delta, \_) \to B(\mathsf{rat}(q), \mathsf{lim}(y), f_q, f_{y,b}, \epsilon, \_).$$
- Similarly, for $r : \mathbb{Q}$ and $\delta, \epsilon : \mathbb{Q}_+$ and x a Cauchy approximation, and a a dependent Cauchy approximation over x, if $x_\delta \sim_{\epsilon-\delta} \mathsf{rat}(r)$, then
$$B(x_\delta, \mathsf{rat}(r), a_\delta, f_r, \epsilon - \delta, \_) \to B(\mathsf{lim}(y), \mathsf{rat}(q), f_{x,a}, f_r, \epsilon, \_).$$
- For $\epsilon, \delta, \eta : \mathbb{Q}_+$ and x, y Cauchy approximations, and a and b dependent Cauchy approximations over x and y respectively, if $x_\delta \sim_{\epsilon-\delta-\eta} y_\eta$, then
$$B(x_\delta, y_\eta, a_\delta, b_\eta, \epsilon - \delta - \eta, \_) \to B(\mathsf{lim}(x), \mathsf{lim}(y), f_{x,a}, f_{y,b}, \epsilon, \_).$$
- For $\epsilon : \mathbb{Q}_+$ and $x, y : \mathbb{R}_c$ and $\xi, \zeta : x \sim_\epsilon y$, and $a : A(x)$ and $b : A(y)$, any two elements of $B(x, y, a, b, \epsilon, \xi)$ and $B(x, y, a, b, \epsilon, \zeta)$ are dependently equal over $\xi = \zeta$. Note that as usual, this is essentially equivalent to asking that B is a mere relation.

Under these hypotheses, we deduce functions

$$f : \prod_{x:\mathbb{R}_c} A(x)$$
$$g : \prod_{(x,y:\mathbb{R}_c)} \prod_{(\epsilon:\mathbb{Q}_+)} \prod_{(\xi:x\sim_\epsilon y)} B(x, y, f(x), f(y), \epsilon, \xi)$$

which compute as expected:

$$f(\mathsf{rat}(q)) :\equiv f_q$$
$$f(\mathsf{lim}(x)) :\equiv f_{x,(f,g)[x]}$$

Here $(f,g)[x]$ denotes the result of applying f and g to a Cauchy approximation x to obtain a dependent Cauchy approximation over x. That is, we define $(f,g)[x]_\epsilon :\equiv f(x_\epsilon) : A(x_\epsilon)$, and then for any $\epsilon, \delta : \mathbb{Q}_+$ we have $g(x_\epsilon, x_\delta, \epsilon+\delta, \_)$ to witness the fact that $(f,g)[x]$ is a dependent Cauchy approximation.

This induction principle is admittedly quite a mouthful. To help make sense of it, we observe that it contains as special cases two separate induction principles for $\mathbb{R}_\mathsf{c}$ and for $\sim$. Firstly, suppose given only a type family $A : \mathbb{R}_\mathsf{c} \to \mathcal{U}$, and define B to be constant at $\mathbf{1}$. Then much of the required data becomes trivial, and we are left with:

- for any $q : \mathbb{Q}$, an element $f_q : A(\mathsf{rat}(q))$,
- for any Cauchy approximation x, and any $a : \prod_{(\epsilon:\mathbb{Q}_+)} A(x_\epsilon)$, an element $f_{x,a} : A(\mathsf{lim}(x))$,
- for $u, v : \mathbb{R}_\mathsf{c}$ and $h : \forall(\epsilon : \mathbb{Q}_+).\, u \sim_\epsilon v$, and $a : A(u)$ and $b : A(v)$, a dependent path $a =^A_{\mathsf{eq}_{\mathbb{R}_\mathsf{c}}(u,v)} b$.

Given these data, the induction principle yields a function $f : \prod_{(x:\mathbb{R}_\mathsf{c})} A(x)$ such that

$$f(\mathsf{rat}(q)) :\equiv f_q$$
$$f(\mathsf{lim}(x)) :\equiv f_{x,f(x)}.$$

We call this principle **$\mathbb{R}_\mathsf{c}$-induction**; it says essentially that if we take $\sim_\epsilon$ as given, then $\mathbb{R}_\mathsf{c}$ is inductively generated by its constructors.

In particular, if A is a mere property, the third hypothesis in $\mathbb{R}_\mathsf{c}$-induction is trivial. Thus, we may prove mere properties of real numbers by simply proving them for rationals and for limits of Cauchy approximations. Here is an example.

Lemma 11.3.6. *For any $u : \mathbb{R}_\mathsf{c}$ and $\epsilon : \mathbb{Q}_+$, we have $u \sim_\epsilon u$.*

Proof. Define $A(u) :\equiv \forall(\epsilon : \mathbb{Q}_+).\, (u \sim_\epsilon u)$. Since this is a mere proposition (by the last constructor of $\sim$), by $\mathbb{R}_\mathsf{c}$-induction, it suffices to prove it when u is $\mathsf{rat}(q)$ and when u is $\mathsf{lim}(x)$. In the first case, we obviously have $|q-q| < \epsilon$ for any ϵ, hence $\mathsf{rat}(q) \sim_\epsilon \mathsf{rat}(q)$ by the first constructor of $\sim$. And in the second case, we may assume inductively that $x_\delta \sim_\epsilon x_\delta$ for all $\delta, \epsilon : \mathbb{Q}_+$. Then in particular, we have $x_{\epsilon/3} \sim_{\epsilon/3} x_{\epsilon/3}$, whence $\mathsf{lim}(x) \sim_\epsilon \mathsf{lim}(x)$ by the fourth constructor of $\sim$. □

Theorem 11.3.7. *$\mathbb{R}_\mathsf{c}$ is a set.*

Proof. We have just shown that the mere relation $P(u,v) :\equiv \forall(\epsilon : \mathbb{Q}_+).\, (u \sim_\epsilon v)$ is reflexive. Since it implies identity, by the path-constructor of $\mathbb{R}_\mathsf{c}$, the result follows from Theorem 7.2.2. □

We can also show that although $\mathbb{R}_\mathsf{c}$ may not be a quotient of the set of Cauchy sequences of *rationals*, it is nevertheless a quotient of the set of Cauchy sequences of *reals*. (Of course, this is not

a valid *definition* of $\mathbb{R}_{\mathsf{c}}$, but it is a useful property.) We define the type of Cauchy approximations to be

$$\mathcal{C} :\equiv \{\, x : \mathbb{Q}_+ \to \mathbb{R}_{\mathsf{c}} \mid \forall(\epsilon, \delta : \mathbb{Q}_+).\, x_\delta \sim_{\delta+\epsilon} x_\epsilon \,\}.$$

The second constructor of $\mathbb{R}_{\mathsf{c}}$ gives a function $\mathsf{lim} : \mathcal{C} \to \mathbb{R}_{\mathsf{c}}$.

Lemma 11.3.8. *Every real merely is a limit point:* $\forall(u : \mathbb{R}_{\mathsf{c}}).\, \exists(x : \mathcal{C}).\, u = \mathsf{lim}(x)$. *In other words,* $\mathsf{lim} : \mathcal{C} \to \mathbb{R}_{\mathsf{c}}$ *is surjective.*

Proof. By $\mathbb{R}_{\mathsf{c}}$-induction, we may divide into cases on u. Of course, if u is a limit $\mathsf{lim}(x)$, the statement is trivial. So suppose u is a rational point $\mathsf{rat}(q)$; we claim u is equal to $\mathsf{lim}(\lambda\_.\, \mathsf{rat}(q))$. By the path-constructor of $\mathbb{R}_{\mathsf{c}}$, it suffices to show $\mathsf{rat}(q) \sim_\epsilon \mathsf{lim}(\lambda\_.\, \mathsf{rat}(q))$ for all $\epsilon : \mathbb{Q}_+$. And by the second constructor of $\sim$, for this it suffices to find $\delta : \mathbb{Q}_+$ such that $\mathsf{rat}(q) \sim_{\epsilon-\delta} \mathsf{rat}(q)$. But by the first constructor of $\sim$, we may take any $\delta : \mathbb{Q}_+$ with $\delta < \epsilon$. □

Lemma 11.3.9. *If A is a set and $f : \mathcal{C} \to A$ respects coincidence of Cauchy approximations, in the sense that*

$$\forall(x, y : \mathcal{C}).\, \mathsf{lim}(x) = \mathsf{lim}(y) \Rightarrow f(x) = f(y),$$

then f factors uniquely through $\mathsf{lim} : \mathcal{C} \to \mathbb{R}_{\mathsf{c}}$.

Proof. Since lim is surjective, by Theorem 10.1.5, $\mathbb{R}_{\mathsf{c}}$ is the quotient of $\mathcal{C}$ by the kernel of lim. But this is exactly the statement of the lemma. □

For the second special case of the induction principle, suppose instead that we are given only $C : \mathbb{R}_{\mathsf{c}} \to \mathbb{R}_{\mathsf{c}} \to \mathbb{Q}_+ \to \mathsf{Prop}$, and define $A(\_) :\equiv \mathbf{1}$ and $B(u, v, \_, \_, \epsilon, \_) :\equiv C(u, v, \epsilon)$. Then the required data reduces to the following, where q, r denote rational numbers, ϵ, δ, η positive rational numbers, and x, y Cauchy approximations:

- if $-\epsilon < q - r < \epsilon$, then $C(\mathsf{rat}(q), \mathsf{rat}(r), \epsilon)$,
- if $\mathsf{rat}(q) \sim_{\epsilon-\delta} y_\delta$ and $C(\mathsf{rat}(q), y_\delta, \epsilon - \delta)$, then $C(\mathsf{rat}(q), \mathsf{lim}(y), \epsilon)$,
- similarly, if $x_\delta \sim_{\epsilon-\delta} \mathsf{rat}(r)$ and $C(x_\delta, \mathsf{rat}(r), \epsilon - \delta)$, then $C(\mathsf{lim}(y), \mathsf{rat}(q), \epsilon)$,
- if $x_\delta \sim_{\epsilon-\delta-\eta} y_\eta$ and $C(x_\delta, y_\eta, \epsilon - \delta - \eta)$, then $C(\mathsf{lim}(x), \mathsf{lim}(y), \epsilon)$.

The resulting conclusion is $\forall(u, v : \mathbb{R}_{\mathsf{c}}).\, \forall(\epsilon : \mathbb{Q}_+).\, (u \sim_\epsilon v) \to C(u, v, \epsilon)$. We call this principle **$\sim$-induction**; it says essentially that if we take $\mathbb{R}_{\mathsf{c}}$ as given, then $\sim_\epsilon$ is inductively generated (as a family of types) by *its* constructors. For example, we can use this to show that $\sim$ is symmetric.

Lemma 11.3.10. *For any $u, v : \mathbb{R}_{\mathsf{c}}$ and $\epsilon : \mathbb{Q}_+$, we have $(u \sim_\epsilon v) = (v \sim_\epsilon u)$.*

Proof. Since both are mere propositions, by symmetry it suffices to show one implication. Thus, let $C(u, v, \epsilon) :\equiv (v \sim_\epsilon u)$. By $\sim$-induction, we may reduce to the case that $u \sim_\epsilon v$ is derived from one of the four interesting constructors of $\sim$. In the first case when u and v are both rational, the result is trivial (we can apply the first constructor again). In the other three cases, the inductive hypothesis (together with commutativity of addition in $\mathbb{Q}$) yields exactly the input to another of the constructors of $\sim$ (the second and third constructors switch, while the fourth stays put). □

The general induction principle, which we may call $(\mathbb{R}_c, \sim)$**-induction**, is therefore a sort of joint $\mathbb{R}_c$-induction and $\sim$-induction. Consider, for instance, its non-dependent version, which we call $(\mathbb{R}_c, \sim)$**-recursion**, which is the one that we will have the most use for. Ordinary $\mathbb{R}_c$-recursion tells us that to define a function $f : \mathbb{R}_c \to A$ it suffices to:

(i) for every $q : \mathbb{Q}$ construct $f(\mathsf{rat}(q)) : A$,
(ii) for every Cauchy approximation $x : \mathbb{Q}_+ \to \mathbb{R}_c$, construct $f(x) : A$, assuming that $f(x_\epsilon)$ has already been defined for all $\epsilon : \mathbb{Q}_+$,
(iii) prove $f(u) = f(v)$ for all $u, v : \mathbb{R}_c$ satisfying $\forall(\epsilon : \mathbb{Q}_+).\, u \sim_\epsilon v$.

However, it is generally quite difficult to show (iii) without knowing something about how f acts on ϵ-close Cauchy reals. The enhanced principle of $(\mathbb{R}_c, \sim)$-recursion remedies this deficiency, allowing us to specify an *arbitrary* "way in which f acts on ϵ-close Cauchy reals", which we can then prove to be the case by a simultaneous induction with the definition of f. This is the family of relations $B : A \to A \to \mathbb{Q}_+ \to \mathsf{Prop}$. If we write the mere property $B(a, b, \epsilon)$ as "$a \frown_\epsilon b$", then defining a function $f : \mathbb{R}_c \to A$ by $(\mathbb{R}_c, \sim)$-recursion requires the following cases.

- For every $q : \mathbb{Q}$, construct $f(\mathsf{rat}(q)) : A$.
- For every Cauchy approximation $x : \mathbb{Q}_+ \to \mathbb{R}_c$, construct $f(x) : A$, assuming inductively that $f(x_\epsilon)$ has already been defined for all $\epsilon : \mathbb{Q}_+$ and form a "Cauchy approximation with respect to $\frown$", i.e. that $\forall(\epsilon, \delta : \mathbb{Q}_+).\, (f(x_\epsilon) \frown_{\epsilon+\delta} f(x_\delta))$.
- Prove that the relations $\frown$ are *separated*, i.e. that $(\forall(\epsilon : \mathbb{Q}_+).\, a \frown_\epsilon b) \to (a = b)$ for any $a, b : A$.
- Prove that if $-\epsilon < q - r < \epsilon$ for $q, r : \mathbb{Q}$, then $f(\mathsf{rat}(q)) \frown_\epsilon f(\mathsf{rat}(r))$.
- For any $q : \mathbb{Q}$ and any Cauchy approximation y, prove that $f(\mathsf{rat}(q)) \frown_\epsilon f(\mathsf{lim}(y))$, assuming inductively that $\mathsf{rat}(q) \sim_{\epsilon-\delta} y_\delta$ and $f(\mathsf{rat}(q)) \frown_{\epsilon-\delta} f(y_\delta)$ for some $\delta : \mathbb{Q}_+$, and that $\eta \mapsto f(x_\eta)$ is a Cauchy approximation with respect to $\frown$.
- For any Cauchy approximation x and any $r : \mathbb{Q}$, prove that $f(\mathsf{lim}(x)) \frown_\epsilon f(\mathsf{rat}(r))$, assuming inductively that $x_\delta \sim_{\epsilon-\delta} \mathsf{rat}(r)$ and $f(x_\delta) \frown_{\epsilon-\delta} f(\mathsf{rat}(r))$ for some $\delta : \mathbb{Q}_+$, and that $\eta \mapsto f(x_\eta)$ is a Cauchy approximation with respect to $\frown$.
- For any Cauchy approximations x, y, prove that $f(\mathsf{lim}(x)) \frown_\epsilon f(\mathsf{lim}(y))$, assuming inductively that $x_\delta \sim_{\epsilon-\delta-\eta} y_\eta$ and $f(x_\delta) \frown_{\epsilon-\delta-\eta} f(y_\eta)$ for some $\delta, \eta : \mathbb{Q}_+$, and that $\theta \mapsto f(x_\theta)$ and $\theta \mapsto f(y_\theta)$ are Cauchy approximations with respect to $\frown$.

Note that in the last four proofs, we are free to use the specific definitions of $f(\mathsf{rat}(q))$ and $f(\mathsf{lim}(x))$ given in the first two data. However, the proof of separatedness must apply to *any* two elements of A, without any relation to f: it is a sort of "admissibility" condition on the family of relations $\frown$. Thus, we often verify it first, immediately after defining $\frown$, before going on to define $f(\mathsf{rat}(q))$ and $f(\mathsf{lim}(x))$.

Under the above hypotheses, $(\mathbb{R}_c, \sim)$-recursion yields a function $f : \mathbb{R}_c \to A$ such that $f(\mathsf{rat}(q))$ and $f(\mathsf{lim}(x))$ are judgmentally equal to the definitions given for them in the first two clauses. Moreover, we may also conclude

$$\forall(u, v : \mathbb{R}_c).\, \forall(\epsilon : \mathbb{Q}_+).\, (u \sim_\epsilon v) \to (f(u) \frown_\epsilon f(v)). \tag{11.3.11}$$

As a paradigmatic example, $(\mathbb{R}_{\mathsf{c}}, \sim)$-recursion allows us to extend functions defined on $\mathbb{Q}$ to all of $\mathbb{R}_{\mathsf{c}}$, as long as they are sufficiently continuous.

Definition 11.3.12. A function $f : \mathbb{Q} \to \mathbb{R}_{\mathsf{c}}$ is **Lipschitz** if there exists $L : \mathbb{Q}_+$ (the **Lipschitz constant**) such that

$$|q - r| < \epsilon \Rightarrow (f(q) \sim_{L\epsilon} f(r))$$

for all $\epsilon : \mathbb{Q}_+$ and $q, r : \mathbb{Q}$. Similarly, $g : \mathbb{R}_{\mathsf{c}} \to \mathbb{R}_{\mathsf{c}}$ is **Lipschitz** if there exists $L : \mathbb{Q}_+$ such that

$$(u \sim_\epsilon v) \Rightarrow (g(u) \sim_{L\epsilon} g(v))$$

for all $\epsilon : \mathbb{Q}_+$ and $u, v : \mathbb{R}_{\mathsf{c}}$..

In particular, note that by the first constructor of $\sim$, if $f : \mathbb{Q} \to \mathbb{Q}$ is Lipschitz in the obvious sense, then so is the composite $\mathbb{Q} \xrightarrow{f} \mathbb{Q} \hookrightarrow \mathbb{R}_{\mathsf{c}}$.

Lemma 11.3.13. *Suppose $f : \mathbb{Q} \to \mathbb{R}_{\mathsf{c}}$ is Lipschitz with constant $L : \mathbb{Q}_+$. Then there exists a Lipschitz map $\bar{f} : \mathbb{R}_{\mathsf{c}} \to \mathbb{R}_{\mathsf{c}}$, also with constant L, such that $\bar{f}(\mathsf{rat}(q)) \equiv f(q)$ for all $q : \mathbb{Q}$.*

Proof. We define $\bar{f}$ by $(\mathbb{R}_{\mathsf{c}}, \sim)$-recursion, with codomain $A :\equiv \mathbb{R}_{\mathsf{c}}$. We define the relation $\frown : \mathbb{R}_{\mathsf{c}} \to \mathbb{R}_{\mathsf{c}} \to \mathbb{Q}_+ \to \mathsf{Prop}$ to be

$$(u \frown_\epsilon v) :\equiv (u \sim_{L\epsilon} v).$$

For $q : \mathbb{Q}$, we define

$$\bar{f}(\mathsf{rat}(q)) :\equiv \mathsf{rat}(f(q)).$$

For a Cauchy approximation $x : \mathbb{Q}_+ \to \mathbb{R}_{\mathsf{c}}$, we define

$$\bar{f}(\mathsf{lim}(x)) :\equiv \mathsf{lim}(\lambda\epsilon.\, \bar{f}(x_{\epsilon/L})).$$

For this to make sense, we must verify that $y :\equiv \lambda\epsilon.\, \bar{f}(x_{\epsilon/L})$ is a Cauchy approximation. However, the inductive hypothesis for this step is that for any $\delta, \epsilon : \mathbb{Q}_+$ we have $\bar{f}(x_\delta) \frown_{\delta+\epsilon} \bar{f}(x_\epsilon)$, i.e. $\bar{f}(x_\delta) \sim_{L\delta+L\epsilon} \bar{f}(x_\epsilon)$. Thus we have

$$y_\delta \equiv f(x_{\delta/L}) \sim_{\delta+\epsilon} f(x_{\epsilon/L}) \equiv y_\epsilon.$$

For the proof of separatedness, we simply observe that $\forall(\epsilon : \mathbb{Q}_+).\, a \frown_\epsilon b$ means $\forall(\epsilon : \mathbb{Q}_+).\, a \sim_{L\epsilon} b$, which implies $\forall(\epsilon : \mathbb{Q}_+).\, a \sim_\epsilon b$ and thus $a = b$.

To complete the $(\mathbb{R}_{\mathsf{c}}, \sim)$-recursion, it remains to verify the four conditions on $\frown$. This basically amounts to proving that $\bar{f}$ is Lipschitz for all the four constructors of $\sim$.

(i) When u is $\mathsf{rat}(q)$ and v is $\mathsf{rat}(r)$ with $-\epsilon < |q - r| < \epsilon$, the assumption that f is Lipschitz yields $f(q) \sim_{L\epsilon} f(r)$, hence $\bar{f}(\mathsf{rat}(q)) \frown_\epsilon \bar{f}(\mathsf{rat}(r))$ by definition.

(ii) When u is $\mathsf{lim}(x)$ and v is $\mathsf{rat}(q)$ with $x_\eta \sim_{\epsilon-\eta} \mathsf{rat}(q)$, then the induction hypothesis is $\bar{f}(x_\eta) \sim_{L\epsilon-L\eta} \mathsf{rat}(f(q))$, which proves $\bar{f}(\mathsf{lim}(x)) \sim_{L\epsilon} \bar{f}(\mathsf{rat}(q))$ by the third constructor of $\sim$.

(iii) The symmetric case when u is rational and v is a limit is essentially identical.

(iv) When u is $\mathsf{lim}(x)$ and v is $\mathsf{lim}(y)$, with $\delta, \eta : \mathbb{Q}_+$ such that $x_\delta \sim_{\epsilon-\delta-\eta} y_\eta$, the induction hypothesis is $\bar{f}(x_\delta) \sim_{L\epsilon-L\delta-L\eta} \bar{f}(y_\eta)$, which proves $\bar{f}(\mathsf{lim}(x)) \sim_{L\epsilon} \bar{f}(\mathsf{lim}(y))$ by the fourth constructor of $\sim$.

This completes the $(\mathbb{R}_\mathsf{c}, \sim)$-recursion, and hence the construction of $\bar{f}$. The desired equality $\bar{f}(\mathsf{rat}(q)) \equiv f(q)$ is exactly the first computation rule for $(\mathbb{R}_\mathsf{c}, \sim)$-recursion, and the additional condition (11.3.11) says exactly that $\bar{f}$ is Lipschitz with constant L. □

At this point we have gone about as far as we can without a better characterization of $\sim$. We have specified, in the constructors of $\sim$, the conditions under which we want Cauchy reals of the two different forms to be ϵ-close. However, how do we know that in the resulting inductive-inductive type family, these are the *only* witnesses to this fact? We have seen that inductive type families (such as identity types, see §5.8) and higher inductive types have a tendency to contain "more than was put into them", so this is not an idle question.

In order to characterize $\sim$ more precisely, we will define a family of relations $\approx_\epsilon$ on $\mathbb{R}_\mathsf{c}$ *recursively*, so that they will compute on constructors, and prove that this family is equivalent to $\sim_\epsilon$.

Theorem 11.3.14. *There is a family of mere relations* $\approx : \mathbb{R}_\mathsf{c} \to \mathbb{R}_\mathsf{c} \to \mathbb{Q}_+ \to \mathsf{Prop}$ *such that*

$$(\mathsf{rat}(q) \approx_\epsilon \mathsf{rat}(r)) :\equiv (-\epsilon < q - r < \epsilon) \tag{11.3.15}$$
$$(\mathsf{rat}(q) \approx_\epsilon \mathsf{lim}(y)) :\equiv \exists(\delta : \mathbb{Q}_+).\, \mathsf{rat}(q) \approx_{\epsilon-\delta} y_\delta \tag{11.3.16}$$
$$(\mathsf{lim}(x) \approx_\epsilon \mathsf{rat}(r)) :\equiv \exists(\delta : \mathbb{Q}_+).\, x_\delta \approx_{\epsilon-\delta} \mathsf{rat}(r) \tag{11.3.17}$$
$$(\mathsf{lim}(x) \approx_\epsilon \mathsf{lim}(y)) :\equiv \exists(\delta, \eta : \mathbb{Q}_+).\, x_\delta \approx_{\epsilon-\delta-\eta} y_\eta. \tag{11.3.18}$$

Moreover, we have

$$(u \approx_\epsilon v) \leftrightarrow \exists(\theta : \mathbb{Q}_+).\, (u \approx_{\epsilon-\theta} v) \tag{11.3.19}$$
$$(u \approx_\epsilon v) \to (v \sim_\delta w) \to (u \approx_{\epsilon+\delta} w) \tag{11.3.20}$$
$$(u \sim_\epsilon v) \to (v \approx_\delta w) \to (u \approx_{\epsilon+\delta} w). \tag{11.3.21}$$

The additional conditions (11.3.19)–(11.3.21) turn out to be required in order to make the inductive definition go through. Condition (11.3.19) is called being **rounded**. Reading it from right to left gives **monotonicity** of $\approx$,

$$(\delta < \epsilon) \wedge (u \approx_\delta v) \Rightarrow (u \approx_\epsilon v)$$

while reading it left to right to **openness** of $\approx$,

$$(u \approx_\epsilon v) \Rightarrow \exists(\epsilon : \mathbb{Q}_+).\, (\delta < \epsilon) \wedge (u \approx_\delta v).$$

Conditions (11.3.20) and (11.3.21) are forms of the triangle inequality, which say that $\approx$ is a "module" over $\sim$ on both sides.

Proof. We will define $\approx : \mathbb{R}_\mathsf{c} \to \mathbb{R}_\mathsf{c} \to \mathbb{Q}_+ \to \mathsf{Prop}$ by double $(\mathbb{R}_\mathsf{c}, \sim)$-recursion. First we will apply $(\mathbb{R}_\mathsf{c}, \sim)$-recursion with codomain the subset of $\mathbb{R}_\mathsf{c} \to \mathbb{Q}_+ \to \mathsf{Prop}$ consisting of those families

of predicates which are rounded and satisfy the one appropriate form of the triangle inequality. Thinking of these predicates as half of a binary relation, we will write them as $(u, \epsilon) \mapsto (\diamondsuit \approx_\epsilon u)$, with the symbol $\diamondsuit$ referring to the whole relation. Now we can write A precisely as

$$A :\equiv \left\{ \diamondsuit : \mathbb{R}_{\mathsf{c}} \to \mathbb{Q}_+ \to \mathsf{Prop} \;\middle|\; \begin{array}{l} \Big(\forall(u : \mathbb{R}_{\mathsf{c}}).\, \forall(\epsilon : \mathbb{Q}_+).\, \big((\diamondsuit \approx_\epsilon u) \leftrightarrow \exists(\theta : \mathbb{Q}_+).\, (\diamondsuit \approx_{\epsilon-\theta} u)\big)\Big) \\ \quad \wedge \Big(\forall(u, v : \mathbb{R}_{\mathsf{c}}).\, \forall(\eta, \epsilon : \mathbb{Q}_+).\, (u \sim_\epsilon v) \to \\ \qquad \big((\diamondsuit \approx_\eta u) \to (\diamondsuit \approx_{\eta+\epsilon} v)\big) \wedge \big((\diamondsuit \approx_\eta v) \to (\diamondsuit \approx_{\eta+\epsilon} u)\big)\Big) \end{array} \right\}$$

As usual with subsets, we will use the same notation for an inhabitant of A and its first component $\diamondsuit$. As the family of relations required for $(\mathbb{R}_{\mathsf{c}}, \sim)$-recursion, we consider the following, which will ensure the other form of the triangle inequality:

$$(\diamondsuit \frown_\epsilon \heartsuit) :\equiv \forall(u : \mathbb{R}_{\mathsf{c}}).\, \forall(\eta : \mathbb{Q}_+).\, ((\diamondsuit \approx_\eta u) \to (\heartsuit \approx_{\epsilon+\eta} u)) \wedge ((\heartsuit \approx_\eta u) \to (\diamondsuit \approx_{\epsilon+\eta} u)).$$

We observe that these relations are separated. For assuming $\forall(\epsilon : \mathbb{Q}_+).\, (\diamondsuit \frown_\epsilon \heartsuit)$, to show $\diamondsuit = \heartsuit$ it suffices to show $(\diamondsuit \approx_\epsilon u) \leftrightarrow (\heartsuit \approx_\epsilon u)$ for all $u : \mathbb{R}_{\mathsf{c}}$. But $\diamondsuit \approx_\epsilon u$ implies $\diamondsuit \approx_{\epsilon-\theta} u$ for some θ, by roundedness, which together with $\diamondsuit \frown_\epsilon \heartsuit$ implies $\heartsuit \approx_\epsilon u$; and the converse is identical.

Now the first two data the recursion principle requires are the following.

- For any $q : \mathbb{Q}$, we must give an element of A, which we denote $(\mathsf{rat}(q) \approx_{(-)} -)$.
- For any Cauchy approximation x, if we assume defined a function $\mathbb{Q}_+ \to A$, which we will denote by $\epsilon \mapsto (x_\epsilon \approx_{(-)} -)$, with the property that

$$\forall(u : \mathbb{R}_{\mathsf{c}}).\, \forall(\delta, \epsilon, \eta : \mathbb{Q}_+).\, (x_\delta \approx_\eta u) \to (x_\epsilon \approx_{\eta+\delta+\epsilon} u), \tag{11.3.22}$$

  we must give an element of A, which we will denote $(\mathsf{lim}(x) \approx_{(-)} -)$.

In both cases, we give the required definition by using a nested $(\mathbb{R}_{\mathsf{c}}, \sim)$-recursion, with codomain the subset of $\mathbb{Q}_+ \to \mathsf{Prop}$ consisting of rounded families of mere propositions. Thinking of these propositions as zero halves of a binary relation, we will write them as $\epsilon \mapsto (\bullet \approx_\epsilon \triangle)$, with the symbol $\triangle$ referring to the whole family. Now we can write the codomain of these inner recursions precisely as

$$C :\equiv \left\{ \triangle : \mathbb{Q}_+ \to \mathsf{Prop} \;\middle|\; \forall(\epsilon : \mathbb{Q}_+).\, \Big((\bullet \approx_\epsilon \triangle) \leftrightarrow \exists(\theta : \mathbb{Q}_+).\, (\bullet \approx_{\epsilon-\theta} \triangle)\Big) \right\}$$

We take the required family of relations to be the remnant of the triangle inequality:

$$(\triangle \smile_\epsilon \square) :\equiv \forall(\eta : \mathbb{Q}_+).\, ((\bullet \approx_\eta \triangle) \to (\bullet \approx_{\epsilon+\eta} \square)) \wedge ((\bullet \approx_\eta \square) \to (\bullet \approx_{\epsilon+\eta} \triangle)).$$

These relations are separated by the same argument as for $\frown$, using roundedness of all elements of C.

Note that if such an inner recursion succeeds, it will yield a family of predicates $\diamondsuit : \mathbb{R}_c \to \mathbb{Q}_+ \to \mathsf{Prop}$ which are rounded (since their image in $\mathbb{Q}_+ \to \mathsf{Prop}$ lies in C) and satisfy

$$\forall (u, v : \mathbb{R}_c).\, \forall (\epsilon : \mathbb{Q}_+).\, (u \sim_\epsilon v) \to \big((\diamondsuit \approx_{(-)} u) \smile_\epsilon (\diamondsuit \approx_{(-)} u)\big).$$

Expanding out the definition of $\smile$, this yields precisely the third condition for $\diamondsuit$ to belong to A; thus it is exactly what we need.

It is at this point that we can give the definitions (11.3.15)–(11.3.18), as the first two clauses of each of the two inner recursions, corresponding to rational points and limits. In each case, we must verify that the relation is rounded and hence lies in C. In the rational-rational case (11.3.15) this is clear, while in the other cases it follows from an inductive hypothesis. (In (11.3.16) the relevant inductive hypothesis is that $(\mathsf{rat}(q) \approx_{(-)} y_\delta) : C$, while in (11.3.17) and (11.3.18) it is that $(x_\delta \approx_{(-)} -) : A$.)

The remaining data of the sub-recursions consist of showing that (11.3.15)–(11.3.18) satisfy the triangle inequality on the right with respect to the constructors of $\sim$. There are eight cases — four in each sub-recursion — corresponding to the eight possible ways that u, v, and w in (11.3.20) can be chosen to be rational points or limits. First we consider the cases when u is $\mathsf{rat}(q)$.

(i) Assuming $\mathsf{rat}(q) \approx_\phi \mathsf{rat}(r)$ and $-\epsilon < |r - s| < \epsilon$, we must show $\mathsf{rat}(q) \approx_{\phi+\epsilon} \mathsf{rat}(s)$. But by definition of $\approx$, this reduces to the triangle inequality for rational numbers.

(ii) We assume $\phi, \epsilon, \delta : \mathbb{Q}_+$ such that $\mathsf{rat}(q) \approx_\phi \mathsf{rat}(r)$ and $\mathsf{rat}(r) \sim_{\epsilon-\delta} y_\delta$, and inductively that

$$\forall (\psi : \mathbb{Q}_+).\, (\mathsf{rat}(q) \approx_\psi \mathsf{rat}(r)) \to (\mathsf{rat}(q) \approx_{\psi+\epsilon-\delta} y_\delta). \tag{11.3.23}$$

We assume also that $\psi, \delta \mapsto (\mathsf{rat}(q) \approx_\psi y_\delta)$ is a Cauchy approximation with respect to $\smile$, i.e.

$$\forall (\psi, \xi, \zeta : \mathbb{Q}_+).\, (\mathsf{rat}(q) \approx_\psi y_\xi) \to (\mathsf{rat}(q) \approx_{\psi+\xi+\zeta} y_\zeta), \tag{11.3.24}$$

although we will not need this assumption in this case. Indeed, (11.3.23) with $\psi :\equiv \phi$ yields immediately $\mathsf{rat}(q) \approx_{\phi+\epsilon-\delta} y_\delta$, and hence $\mathsf{rat}(q) \approx_{\phi+\epsilon} \mathsf{lim}(y)$ by definition of $\approx$.

(iii) We assume $\phi, \epsilon, \delta : \mathbb{Q}_+$ such that $\mathsf{rat}(q) \approx_\phi \mathsf{lim}(y)$ and $y_\delta \sim_{\epsilon-\delta} \mathsf{rat}(r)$, and inductively that

$$\forall (\psi : \mathbb{Q}_+).\, (\mathsf{rat}(q) \approx_\psi y_\delta) \to (\mathsf{rat}(q) \approx_{\psi+\epsilon-\delta} \mathsf{rat}(r)). \tag{11.3.25}$$
$$\forall (\psi, \xi, \zeta : \mathbb{Q}_+).\, (\mathsf{rat}(q) \approx_\psi y_\xi) \to (\mathsf{rat}(q) \approx_{\psi+\xi+\zeta} y_\zeta). \tag{11.3.26}$$

Now by definition, $\mathsf{rat}(q) \approx_\phi \mathsf{lim}(y)$ means we have $\theta : \mathbb{Q}_+$ with $\mathsf{rat}(q) \approx_{\phi-\theta} y_\theta$. By assumption (11.3.26), therefore, we have also $\mathsf{rat}(q) \approx_{\phi+\delta} y_\delta$. But now by assumption (11.3.25), we have $\mathsf{rat}(q) \approx_{\phi+\epsilon} \mathsf{rat}(r)$, as desired.

(iv) We assume $\phi, \epsilon, \delta, \eta : \mathbb{Q}_+$ such that $\mathsf{rat}(q) \approx_\phi \mathsf{lim}(y)$ and $y_\delta \sim_{\epsilon-\delta-\eta} z_\eta$, and inductively that

$$\forall (\psi : \mathbb{Q}_+).\, (\mathsf{rat}(q) \approx_\psi y_\delta) \to (\mathsf{rat}(q) \approx_{\psi+\epsilon-\delta-\eta} z_\eta) \tag{11.3.27}$$
$$\forall (\psi, \xi, \zeta : \mathbb{Q}_+).\, (\mathsf{rat}(q) \approx_\psi y_\xi) \to (\mathsf{rat}(q) \approx_{\psi+\xi+\zeta} y_\zeta) \tag{11.3.28}$$
$$\forall (\psi, \xi, \zeta : \mathbb{Q}_+).\, (\mathsf{rat}(q) \approx_\psi z_\xi) \to (\mathsf{rat}(q) \approx_{\psi+\xi+\zeta} z_\zeta) \tag{11.3.29}$$

Again, $\mathsf{rat}(q) \approx_\phi \mathsf{lim}(y)$ means we have $\xi : \mathbb{Q}_+$ with $\mathsf{rat}(q) \approx_{\phi-\xi} y_\xi$, while (11.3.28) then implies $\mathsf{rat}(q) \approx_{\phi+\delta} y_\delta$ and (11.3.27) implies $\mathsf{rat}(q) \approx_{\phi+\epsilon-\eta} z_\eta$. But by definition of $\approx$, this implies $\mathsf{rat}(q) \approx_{\phi+\epsilon} \mathsf{lim}(z)$ as desired.

Now we move on to the cases when u is $\mathsf{lim}(x)$, with x a Cauchy approximation. In this case, the ambient induction hypothesis of the definition of $(\mathsf{lim}(x) \approx_{(-)} -) : A$ is that we have $(x_\delta \approx_{(-)} -) : A$, so that in addition to being rounded they satisfy the triangle inequality on the right.

(v) Assuming $\mathsf{lim}(x) \approx_\phi \mathsf{rat}(r)$ and $-\epsilon < |r - s| < \epsilon$, we must show $\mathsf{lim}(x) \approx_{\phi+\epsilon} \mathsf{rat}(s)$. But by definition of $\approx$, the former means $x_\delta \approx_{\phi-\delta} \mathsf{rat}(r)$, so that above-mentioned triangle inequality implies $x_\delta \approx_{\epsilon+\phi-\delta} \mathsf{rat}(s)$, hence $\mathsf{lim}(x) \approx_{\phi+\epsilon} \mathsf{rat}(s)$ as desired.

(vi) We assume $\phi, \epsilon, \delta : \mathbb{Q}_+$ such that $\mathsf{lim}(x) \approx_\phi \mathsf{rat}(r)$ and $\mathsf{rat}(r) \sim_{\epsilon-\delta} y_\delta$, and two unneeded inductive hypotheses. By definition, we have $\eta : \mathbb{Q}_+$ such that $x_\eta \approx_{\phi-\eta} \mathsf{rat}(r)$, so the inductive triangle inequality gives $x_\eta \approx_{\phi+\epsilon-\eta-\delta} y_\delta$. The definition of $\approx$ then immediately yields $\mathsf{lim}(x) \approx_{\phi+\epsilon} \mathsf{lim}(y)$.

(vii) We assume $\phi, \epsilon, \delta : \mathbb{Q}_+$ such that $\mathsf{lim}(x) \approx_\phi \mathsf{lim}(y)$ and $y_\delta \sim_{\epsilon-\delta} \mathsf{rat}(r)$, and two unneeded inductive hypotheses. By definition, then, we have $\xi, \theta : \mathbb{Q}_+$ such that $x_\xi \approx_{\phi-\xi-\theta} y_\theta$. Since y is a Cauchy approximation, we have $y_\theta \sim_{\theta+\delta} y_\delta$, so the inductive triangle inequality gives $x_\xi \approx_{\phi+\delta-\xi} y_\delta$ and then $x_\xi \sim_{\phi+\epsilon-\xi} \mathsf{rat}(r)$. The definition of $\approx$ then gives $\mathsf{lim}(x) \approx_{\phi+\epsilon} \mathsf{rat}(r)$, as desired.

(viii) Finally, we assume $\phi, \epsilon, \delta, \eta : \mathbb{Q}_+$ such that $\mathsf{lim}(x) \approx_\phi \mathsf{lim}(y)$ and $y_\delta \sim_{\epsilon-\delta-\eta} z_\eta$. Then as before we have $\xi, \theta : \mathbb{Q}_+$ with $x_\xi \approx_{\phi-\xi-\theta} y_\theta$, and two applications of the triangle inequality suffices as before.

This completes the two inner recursions, and thus the definitions of the families of relations $(\mathsf{rat}(q) \approx_{(-)} -)$ and $(\mathsf{lim}(x) \approx_{(-)} -)$. Since all are elements of A, they are rounded and satisfy the triangle inequality on the right with respect to $\sim$. What remains is to verify the conditions relating to $\frown$, which is to say that these relations satisfy the triangle inequality on the *left* with respect to the constructors of $\sim$. The four cases correspond to the four choices of rational or limit points for u and v in (11.3.21), and since they are all mere propositions, we may apply $\mathbb{R}_\mathsf{c}$-induction and assume that w is also either rational or a limit. This yields another eight cases, whose proofs are essentially identical to those just given; so we will not subject the reader to them. □

We can now prove:

Theorem 11.3.30. *For any $u, v : \mathbb{R}_\mathsf{c}$ and $\epsilon : \mathbb{Q}_+$ we have $(u \sim_\epsilon v) = (u \approx_\epsilon v)$.*

Proof. Since both are mere propositions, it suffices to prove bidirectional implication. For the left-to-right direction, we use $\sim$-induction with $C(u, v, \epsilon) :\equiv (u \approx_\epsilon v)$. Thus, it suffices to consider the four constructors of $\sim$. In each case, u and v are specialized to either rational points or limits, so that the definition of $\approx$ evaluates, and the inductive hypothesis always applies.

For the right-to-left direction, we use $\mathbb{R}_\mathsf{c}$-induction to assume that u and v are rational points or limits, allowing $\approx$ to evaluate. But now the definitions of $\approx$, and the inductive hypotheses, supply exactly the data required for the relevant constructors of $\sim$. □

Stretching a point, one might call $\approx$ a fibration of "codes" for $\sim$, with the two directions of the above proof being encode and decode respectively. By the definition of $\approx$, from Theorem 11.3.30 we get equivalences

$$\begin{aligned}(\mathsf{rat}(q) \sim_\epsilon \mathsf{rat}(r)) &= (-\epsilon < q - r < \epsilon)\\ (\mathsf{rat}(q) \sim_\epsilon \mathsf{lim}(y)) &= \exists(\delta : \mathbb{Q}_+).\, \mathsf{rat}(q) \sim_{\epsilon-\delta} y_\delta\\ (\mathsf{lim}(x) \sim_\epsilon \mathsf{rat}(r)) &= \exists(\delta : \mathbb{Q}_+).\, x_\delta \sim_{\epsilon-\delta} \mathsf{rat}(r)\\ (\mathsf{lim}(x) \sim_\epsilon \mathsf{lim}(y)) &= \exists(\delta, \eta : \mathbb{Q}_+).\, x_\delta \sim_{\epsilon-\delta-\eta} y_\eta.\end{aligned}$$

Our proof also provides the following additional information.

Corollary 11.3.31. *$\sim$ is rounded and satisfies the triangle inequality:*

$$(u \sim_\epsilon v) \simeq \exists(\theta : \mathbb{Q}_+).\, u \sim_{\epsilon-\theta} v \tag{11.3.32}$$

$$(u \sim_\epsilon v) \to (v \sim_\delta w) \to (u \sim_{\epsilon+\delta} w) \tag{11.3.33}$$

With the triangle inequality in hand, we can show that "limits" of Cauchy approximations actually behave like limits.

Lemma 11.3.34. *For any $u : \mathbb{R}_\mathsf{c}$, Cauchy approximation y, and $\epsilon, \delta : \mathbb{Q}_+$, if $u \sim_\epsilon y_\delta$ then $u \sim_{\epsilon+\delta} \mathsf{lim}(y)$.*

Proof. We use $\mathbb{R}_\mathsf{c}$-induction on u. If u is $\mathsf{rat}(q)$, then this is exactly the second constructor of $\sim$. Now suppose u is $\mathsf{lim}(x)$, and that each x_η has the property that for any y, ϵ, δ, if $x_\eta \sim_\epsilon y_\delta$ then $x_\eta \sim_{\epsilon+\delta} \mathsf{lim}(y)$. In particular, taking $y :\equiv x$ and $\delta :\equiv \eta$ in this assumption, we conclude that $x_\eta \sim_{\eta+\theta} \mathsf{lim}(x)$ for any $\eta, \theta : \mathbb{Q}_+$.

Now let y, ϵ, δ be arbitrary and assume $\mathsf{lim}(x) \sim_\epsilon y_\delta$. By roundedness, there is a θ such that $\mathsf{lim}(x) \sim_{\epsilon-\theta} y_\delta$. Then by the above observation, for any η we have $x_\eta \sim_{\eta+\theta/2} \mathsf{lim}(x)$, and hence $x_\eta \sim_{\epsilon+\eta-\theta/2} y_\delta$ by the triangle inequality. Hence, the fourth constructor of $\sim$ yields $\mathsf{lim}(x) \sim_{\epsilon+2\eta+\delta-\theta/2} \mathsf{lim}(y)$. Thus, if we choose $\eta :\equiv \theta/4$, the result follows. □

Lemma 11.3.35. *For any Cauchy approximation y and any $\delta, \eta : \mathbb{Q}_+$ we have $y_\delta \sim_{\delta+\eta} \mathsf{lim}(y)$.*

Proof. Take $u :\equiv y_\delta$ and $\epsilon :\equiv \eta$ in the previous lemma. □

Remark 11.3.36. We might have expected to have $y_\delta \sim_\delta \mathsf{lim}(y)$, but this fails in examples. For instance, consider x defined by $x_\epsilon :\equiv \epsilon$. Its limit is clearly 0, but we do not have $|\epsilon - 0| < \epsilon$, only $\leq$.

As an application, Lemma 11.3.35 enables us to show that the extensions of Lipschitz functions from Lemma 11.3.13 are unique.

Lemma 11.3.37. *Let $f, g : \mathbb{R}_\mathsf{c} \to \mathbb{R}_\mathsf{c}$ be continuous, in the sense that*

$$\forall(u : \mathbb{R}_\mathsf{c}).\, \forall(\epsilon : \mathbb{Q}_+).\, \exists(\delta : \mathbb{Q}_+).\, \forall(v : \mathbb{R}_\mathsf{c}).\, (u \sim_\delta v) \to (f(u) \sim_\epsilon f(v))$$

and analogously for g. If $f(\mathsf{rat}(q)) = g(\mathsf{rat}(q))$ for all $q : \mathbb{Q}$, then $f = g$.

Proof. We prove $f(u) = g(u)$ for all u by $\mathbb{R}_c$-induction. The rational case is just the hypothesis. Thus, suppose $f(x_\delta) = g(x_\delta)$ for all δ. We will show that $f(\mathsf{lim}(x)) \sim_\epsilon g(\mathsf{lim}(x))$ for all ϵ, so that the path-constructor of $\mathbb{R}_c$ applies.

Since f and g are continuous, there exist θ, η such that for all v, we have

$$(\mathsf{lim}(x) \sim_\theta v) \to (f(\mathsf{lim}(x)) \sim_{\epsilon/2} f(v))$$
$$(\mathsf{lim}(x) \sim_\eta v) \to (g(\mathsf{lim}(x)) \sim_{\epsilon/2} g(v)).$$

Choosing $\delta < \min(\theta, \eta)$, by Lemma 11.3.35 we have both $\mathsf{lim}(x) \sim_\theta y_\delta$ and $\mathsf{lim}(x) \sim_\eta y_\delta$. Hence

$$f(\mathsf{lim}(x)) \sim_{\epsilon/2} f(y_\delta) = g(y_\delta) \sim_{\epsilon/2} g(\mathsf{lim}(x))$$

and thus $f(\mathsf{lim}(x)) \sim_\epsilon g(\mathsf{lim}(x))$ by the triangle inequality. □

11.3.3 The algebraic structure of Cauchy reals

We first define the additive structure $(\mathbb{R}_c, 0, +, -)$. Clearly, the neutral element 0 is just $\mathsf{rat}(0)$, while the additive inverse $- : \mathbb{R}_c \to \mathbb{R}_c$ is obtained as the extension of the additive inverse $- : \mathbb{Q} \to \mathbb{Q}$, using Lemma 11.3.13 with Lipschitz factor 1. We have to work a bit harder for addition.

Lemma 11.3.38. *Suppose $f : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$ satisfies, for all $q, r, s : \mathbb{Q}$,*

$$|f(q,s) - f(r,s)| \leq |q - r| \qquad \text{and} \qquad |f(q,r) - f(q,s)| \leq |r - s|.$$

Then there is a function $\bar{f} : \mathbb{R}_c \times \mathbb{R}_c \to \mathbb{R}_c$ such that $\bar{f}(\mathsf{rat}(q), \mathsf{rat}(r)) = f(q,r)$ for all $q, r : \mathbb{Q}$. Furthermore, for all $u, v, w : \mathbb{R}_c$ and $q : \mathbb{Q}_+$,

$$u \sim_\epsilon v \Rightarrow \bar{f}(u,w) \sim_\epsilon \bar{f}(v,w) \quad \text{and} \quad v \sim_\epsilon w \Rightarrow \bar{f}(u,v) \sim_\epsilon \bar{f}(u,w).$$

Proof. We use $(\mathbb{R}_c, \sim)$-recursion to construct the curried form of $\bar{f}$ as a map $\mathbb{R}_c \to A$ where A is the space of non-expanding real-valued functions:

$$A :\equiv \{\, h : \mathbb{R}_c \to \mathbb{R}_c \mid \forall(\epsilon : \mathbb{Q}_+).\, \forall(u, v : \mathbb{R}_c).\, u \sim_\epsilon v \Rightarrow h(u) \sim_\epsilon h(v) \,\}.$$

We shall also need a suitable $\frown_\epsilon$ on A, which we define as

$$(h \frown_\epsilon k) :\equiv \forall(u : \mathbb{R}_c).\, h(u) \sim_\epsilon k(u).$$

Clearly, if $\forall(\epsilon : \mathbb{Q}_+).\, h \frown_\epsilon k$ then $h(u) = k(u)$ for all $u : \mathbb{R}_c$, so $\frown$ is separated.

For the base case we define $\bar{f}(\mathsf{rat}(q)) : A$, where $q : \mathbb{Q}$, as the extension of the Lipschitz map $\lambda r.\, f(q,r)$ from $\mathbb{Q} \to \mathbb{Q}$ to $\mathbb{R}_c \to \mathbb{R}_c$, as constructed in Lemma 11.3.13 with Lipschitz constant 1. Next, for a Cauchy approximation x, we define $\bar{f}(\mathsf{lim}(x)) : \mathbb{R}_c \to \mathbb{R}_c$ as

$$\bar{f}(\mathsf{lim}(x))(v) :\equiv \mathsf{lim}(\lambda\epsilon.\, \bar{f}(x_\epsilon)(v)).$$

For this to be a valid definition, $\lambda\epsilon.\, \bar{f}(x_\epsilon)(v)$ should be a Cauchy approximation, so consider any $\delta, \epsilon : \mathbb{Q}$. Then by assumption $\bar{f}(x_\delta) \frown_{\delta+\epsilon} \bar{f}(x_\epsilon)$, hence $\bar{f}(x_\delta)(v) \sim_{\delta+\epsilon} \bar{f}(x_\epsilon)(v)$. Furthermore,

$\bar{f}(\mathsf{lim}(x))$ is non-expanding because $\bar{f}(x_\epsilon)$ is such by induction hypothesis. Indeed, if $u \sim_\epsilon v$ then, for all $\epsilon : \mathbb{Q}$,

$$\bar{f}(x_{\epsilon/3})(u) \sim_{\epsilon/3} \bar{f}(x_{\epsilon/3})(v),$$

therefore $\bar{f}(\mathsf{lim}(x))(u) \sim_\epsilon \bar{f}(\mathsf{lim}(x))(v)$.

We still have to check four more conditions, let us illustrate just one. Suppose $\epsilon : \mathbb{Q}_+$ and for some $\delta : \mathbb{Q}_+$ we have $\mathsf{rat}(q) \sim_{\epsilon-\delta} y_\delta$ and $\bar{f}(\mathsf{rat}(q)) \frown_{\epsilon-\delta} \bar{f}(y_\delta)$. To show $\bar{f}(\mathsf{rat}(q)) \frown_\epsilon \bar{f}(y)$, consider any $v : \mathbb{R}_\mathsf{c}$ and observe that

$$\bar{f}(q)(v) \sim_{\epsilon-\delta} \bar{f}(y_\delta)(v)$$

and because $y_\delta \sim y$ also

$$\bar{f}(y_\delta)(v) \sim_\delta \bar{f}(y)(v),$$

therefore $\bar{f}(q)(v) \sim_\epsilon \bar{f}(y)(v)$, as required. □

We may apply Lemma 11.3.38 to any bivariate rational function which is non-expanding separately in each variable. Addition is such a function, therefore we get $+ : \mathbb{R}_\mathsf{c} \times \mathbb{R}_\mathsf{c} \to \mathbb{R}_\mathsf{c}$. Furthermore, the extension is unique as long as we require it to be non-expanding in each variable, and just as in the univariate case, identities on rationals extend to identities on reals. Since composition of non-expanding maps is again non-expanding, we may conclude that addition satisfies the usual properties, such as commutativity and associativity. Therefore, $(\mathbb{R}_\mathsf{c}, 0, +, -)$ is a commutative group.

We may also apply Lemma 11.3.38 to the functions $\min : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$ and $\max : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$, which turns $\mathbb{R}_\mathsf{c}$ into a lattice. The partial order $\leq$ on $\mathbb{R}_\mathsf{c}$ is defined in terms of max as

$$(u \leq v) :\equiv (\max(u, v) = v).$$

The relation $\leq$ is a partial order because it is such on $\mathbb{Q}$, and the axioms of partial order are expressible as equations in terms of min and max, so they transfer to $\mathbb{R}_\mathsf{c}$.

Another function which extends to $\mathbb{R}_\mathsf{c}$ by the same method is the absolute value $|-|$. Again, it has the expected properties because they transfer from $\mathbb{Q}$ to $\mathbb{R}_\mathsf{c}$.

From $\leq$ we get the strict order $<$ by

$$(u < v) :\equiv \exists(q, r : \mathbb{Q}_+).\, (u \leq q) \wedge (q < r) \wedge (r \leq v).$$

That is, $u < v$ holds when there merely exists a pair of rational numbers $q < r$ such that $x \leq \mathsf{rat}(q)$ and $\mathsf{rat}(r) \leq v$. It is not hard to check that $<$ is irreflexive and transitive, and has other properties that are expected for an ordered field. The archimedean principle follows directly from the definition of $<$.

Theorem 11.3.39 (Archimedean principle for $\mathbb{R}_\mathsf{c}$)**.** *For every $u, v : \mathbb{R}_\mathsf{c}$ such that $u < v$ there merely exists $q : \mathbb{Q}$ such that $u < q < v$.*

Proof. From $u < v$ we merely get $r, s : \mathbb{Q}$ such that $u < r < s < v$, and we may take $q :\equiv (r + s)/2$. □

We now have enough structure on $\mathbb{R}_\mathsf{c}$ to express $u \sim_\epsilon v$ with standard concepts.

Lemma 11.3.40. *If $q : \mathbb{Q}$ and $u : \mathbb{R}_c$ satisfy $u \leq \mathsf{rat}(q)$, then for any $v : \mathbb{R}_c$ and $\epsilon : \mathbb{Q}_+$, if $u \sim_\epsilon v$ then $v \leq \mathsf{rat}(q + \epsilon)$.*

Proof. Note that the function $\max(\mathsf{rat}(q), -) : \mathbb{R}_c \to \mathbb{R}_c$ is Lipschitz with constant 1. First consider the case when $u = \mathsf{rat}(r)$ is rational. For this we use induction on v. If v is rational, then the statement is obvious. If v is $\mathsf{lim}(y)$, we assume inductively that for any ϵ, δ, if $\mathsf{rat}(r) \sim_\epsilon y_\delta$ then $y_\delta \leq \mathsf{rat}(q + \epsilon)$, i.e. $\max(\mathsf{rat}(q + \epsilon), y_\delta) = \mathsf{rat}(q + \epsilon)$.

Now assuming ϵ and $\mathsf{rat}(r) \sim_\epsilon \mathsf{lim}(y)$, we have θ such that $\mathsf{rat}(r) \sim_{\epsilon-\theta} \mathsf{lim}(y)$, hence $\mathsf{rat}(r) \sim_\epsilon y_\delta$ whenever $\delta < \theta$. Thus, the inductive hypothesis gives $\max(\mathsf{rat}(q + \epsilon), y_\delta) = \mathsf{rat}(q + \epsilon)$ for such δ. But by definition,

$$\max(\mathsf{rat}(q + \epsilon), \mathsf{lim}(y)) \equiv \mathsf{lim}(\lambda\delta.\ \max(\mathsf{rat}(q + \epsilon), y_\delta)).$$

Since the limit of an eventually constant Cauchy approximation is that constant, we have

$$\max(\mathsf{rat}(q + \epsilon), \mathsf{lim}(y)) = \mathsf{rat}(q + \epsilon),$$

hence $\mathsf{lim}(y) \leq \mathsf{rat}(q + \epsilon)$.

Now consider a general $u : \mathbb{R}_c$. Since $u \leq \mathsf{rat}(q)$ means $\max(\mathsf{rat}(q), u) = \mathsf{rat}(q)$, the assumption $u \sim_\epsilon v$ and the Lipschitz property of $\max(\mathsf{rat}(q), -)$ imply $\max(\mathsf{rat}(q), v) \sim_\epsilon \mathsf{rat}(q)$. Thus, since $\mathsf{rat}(q) \leq \mathsf{rat}(q)$, the first case implies $\max(\mathsf{rat}(q), v) \leq \mathsf{rat}(q + \epsilon)$, and hence $v \leq \mathsf{rat}(q + \epsilon)$ by transitivity of $\leq$. □

Lemma 11.3.41. *Suppose $q : \mathbb{Q}$ and $u : \mathbb{R}_c$ satisfy $u < \mathsf{rat}(q)$. Then:*

(i) For any $v : \mathbb{R}_c$ and $\epsilon : \mathbb{Q}_+$, if $u \sim_\epsilon v$ then $v < \mathsf{rat}(q + \epsilon)$.

(ii) There exists $\epsilon : \mathbb{Q}_+$ such that for any $v : \mathbb{R}_c$, if $u \sim_\epsilon v$ we have $v < \mathsf{rat}(q)$.

Proof. By definition, $u < \mathsf{rat}(q)$ means there is $r : \mathbb{Q}$ with $r < q$ and $u \leq \mathsf{rat}(r)$. Then by Lemma 11.3.40, for any ϵ, if $u \sim_\epsilon v$ then $v \leq \mathsf{rat}(r + \epsilon)$. Conclusion (i) follows immediately since $r + \epsilon < q + \epsilon$, while for (ii) we can take any $\epsilon < q - r$. □

We are now able to show that the auxiliary relation $\sim$ is what we think it is.

Theorem 11.3.42. $(u \sim_\epsilon v) \simeq (|u - v| < \mathsf{rat}(\epsilon))$ *for all $u, v : \mathbb{R}_c$ and $\epsilon : \mathbb{Q}_+$.*

Proof. The Lipschitz properties of subtraction and absolute value imply that if $u \sim_\epsilon v$, then $|u - v| \sim_\epsilon |u - u| = 0$. Thus, for the left-to-right direction, it will suffice to show that if $u \sim_\epsilon 0$, then $|u| < \mathsf{rat}(\epsilon)$. We proceed by $\mathbb{R}_c$-induction on u.

If u is rational, the statement follows immediately since absolute value and order extend the standard ones on $\mathbb{Q}_+$. If u is $\mathsf{lim}(x)$, then by roundedness we have $\theta : \mathbb{Q}_+$ with $\mathsf{lim}(x) \sim_{\epsilon-\theta} 0$. By the triangle inequality, therefore, we have $x_{\theta/3} \sim_{\epsilon-2\theta/3} 0$, so the inductive hypothesis yields $|x_{\theta/3}| < \mathsf{rat}(\epsilon - 2\theta/3)$. But $x_{\theta/3} \sim_{2\theta/3} \mathsf{lim}(x)$, hence $|x_{\theta/3}| \sim_{2\theta/3} |\mathsf{lim}(x)|$ by the Lipschitz property, so Lemma 11.3.41(i) implies $|\mathsf{lim}(x)| < \mathsf{rat}(\epsilon)$.

In the other direction, we use $\mathbb{R}_c$-induction on u and v. If both are rational, this is the first constructor of $\sim$.

If u is $\mathsf{rat}(q)$ and v is $\lim(y)$, we assume inductively that for any ϵ, δ, if $|\mathsf{rat}(q) - y_\delta| < \mathsf{rat}(\epsilon)$ then $\mathsf{rat}(q) \sim_\epsilon y_\delta$. Fix an ϵ such that $|\mathsf{rat}(q) - \lim(y)| < \mathsf{rat}(\epsilon)$. Since $\mathbb{Q}$ is order-dense in $\mathbb{R}_\mathsf{c}$, there exists $\theta < \epsilon$ with $|\mathsf{rat}(q) - \lim(y)| < \mathsf{rat}(\theta)$. Now for any δ, η we have $\lim(y) \sim_{2\delta} y_\delta$, hence by the Lipschitz property

$$|\mathsf{rat}(q) - \lim(y)| \sim_{\delta+\eta} |\mathsf{rat}(q) - y_\delta|.$$

Thus, by Lemma 11.3.41(i), we have $|\mathsf{rat}(q) - y_\delta| < \mathsf{rat}(\theta + 2\delta)$. So by the inductive hypothesis, $\mathsf{rat}(q) \sim_{\theta+2\delta} y_\delta$, and thus $\mathsf{rat}(q) \sim_{\theta+4\delta} \lim(y)$ by the triangle inequality. Thus, it suffices to choose $\delta :\equiv (\epsilon - \theta)/4$.

The remaining two cases are entirely analogous. □

Next, we would like to equip $\mathbb{R}_\mathsf{c}$ with multiplicative structure. For each $q : \mathbb{Q}$ the map $r \mapsto q \cdot r$ is Lipschitz with constant[1] $|q| + 1$, and so we can extend it to multiplication by q on the real numbers. Therefore $\mathbb{R}_\mathsf{c}$ is a vector space over $\mathbb{Q}$. In general, we can define multiplication of real numbers as

$$u \cdot v :\equiv \tfrac{1}{2} \cdot ((u+v)^2 - u^2 - v^2), \tag{11.3.43}$$

so we just need squaring $u \mapsto u^2$ as a map $\mathbb{R}_\mathsf{c} \to \mathbb{R}_\mathsf{c}$. Squaring is not a Lipschitz map, but it is Lipschitz on every bounded domain, which allows us to patch it together. Define the open and closed intervals

$$[u, v] :\equiv \{\, x : \mathbb{R}_\mathsf{c} \mid u \le x \le v \,\} \qquad \text{and} \qquad (u, v) :\equiv \{\, x : \mathbb{R}_\mathsf{c} \mid u < x < v \,\}.$$

Theorem 11.3.44. *There exists a unique function $(-)^2 : \mathbb{R}_\mathsf{c} \to \mathbb{R}_\mathsf{c}$ which extends squaring $q \mapsto q^2$ of rational numbers and satisfies*

$$\forall (n : \mathbb{N}).\, \forall (u, v : [-n, n]).\, |u^2 - v^2| \le 2 \cdot n \cdot |u - v|.$$

Proof. We first observe that for every $u : \mathbb{R}_\mathsf{c}$ there merely exists $n : \mathbb{N}$ such that $-n \le u \le n$, see Exercise 11.7, so the map

$$e : \left(\sum_{n:\mathbb{N}} [-n, n] \right) \to \mathbb{R}_\mathsf{c} \qquad \text{defined by} \qquad e(n, (x, \_)) :\equiv x$$

is surjective. Next, for each $n : \mathbb{N}$, the squaring map

$$s_n : \{\, q : \mathbb{Q} \mid -n \le q \le n \,\} \to \mathbb{Q} \qquad \text{defined by} \qquad s_n(q, \_) :\equiv q^2$$

is Lipschitz with constant $2n$, so we can use Lemma 11.3.13 to extend it to a map $\bar{s}_n : [-n, n] \to \mathbb{R}_\mathsf{c}$ with Lipschitz constant $2n$, see Exercise 11.8 for details. The maps $\bar{s}_n$ are compatible: if $m < n$ for some $m, n : \mathbb{N}$ then s_n restricted to $[-m, m]$ must agree with s_m because both are Lipschitz, and therefore continuous in the sense of Lemma 11.3.37. Therefore, by Theorem 10.1.5 the map

$$\left(\sum_{n:\mathbb{N}} [-n, n] \right) \to \mathbb{R}_\mathsf{c}, \qquad \text{given by} \qquad (n, (x, \_)) \mapsto s_n(x)$$

factors uniquely through $\mathbb{R}_\mathsf{c}$ to give us the desired function. □

[1] We defined Lipschitz constants as *positive* rational numbers.

At this point we have the ring structure of the reals and the archimedean order. To establish $\mathbb{R}_c$ as an archimedean ordered field, we still need inverses.

Theorem 11.3.45. *A Cauchy real is invertible if, and only if, it is apart from zero.*

Proof. First, suppose $u : \mathbb{R}_c$ has an inverse $v : \mathbb{R}_c$ By the archimedean principle there is $q : \mathbb{Q}$ such that $|v| < q$. Then $1 = |uv| < |u| \cdot v < |u| \cdot q$ and hence $|u| > 1/q$, which is to say that $u \# 0$.

For the converse we construct the inverse map

$$(-)^{-1} : \{\, u : \mathbb{R}_c \mid u \# 0 \,\} \to \mathbb{R}_c$$

by patching together functions, similarly to the construction of squaring in Theorem 11.3.44. We only outline the main steps. For every $q : \mathbb{Q}$ let

$$[q, \infty) :\equiv \{\, u : \mathbb{R}_c \mid q \leq u \,\} \qquad \text{and} \qquad (-\infty, q] :\equiv \{\, u : \mathbb{R}_c \mid u \leq -q \,\}.$$

Then, as q ranges over $\mathbb{Q}_+$, the types $(-\infty, q]$ and $[q, \infty)$ jointly cover $\{\, u : \mathbb{R}_c \mid u \# 0 \,\}$. On each such $[q, \infty)$ and $[-\infty, q)$ the inverse function is obtained by an application of Lemma 11.3.13 with Lipschitz constant $1/q^2$. Finally, Theorem 10.1.5 guarantees that the inverse function factors uniquely through $\{\, u : \mathbb{R}_c \mid u \# 0 \,\}$. □

We summarize the algebraic structure of $\mathbb{R}_c$ with a theorem.

Theorem 11.3.46. *The Cauchy reals form an archimedean ordered field.*

11.3.4 Cauchy reals are Cauchy complete

We constructed $\mathbb{R}_c$ by closing $\mathbb{Q}$ under limits of Cauchy approximations, so it better be the case that $\mathbb{R}_c$ is Cauchy complete. Thanks to Theorem 11.3.42 there is no difference between a Cauchy approximation $x : \mathbb{Q}_+ \to \mathbb{R}_c$ as defined in the construction of $\mathbb{R}_c$, and a Cauchy approximation in the sense of Definition 11.2.10 (adapted to $\mathbb{R}_c$).

Thus, given a Cauchy approximation $x : \mathbb{Q}_+ \to \mathbb{R}_c$ is is quite natural to expect that $\lim(x)$ is its limit, where the notion of limit is defined as in Definition 11.2.10. But this is so by Theorem 11.3.42 and Lemma 11.3.35. We have proved:

Theorem 11.3.47. *Every Cauchy approximation in $\mathbb{R}_c$ has a limit.*

An archimedean ordered field in which every Cauchy approximation has a limit is called **Cauchy complete**. The Cauchy reals are the least such field.

Theorem 11.3.48. *The Cauchy reals embed into every Cauchy complete archimedean ordered field.*

Proof. Suppose F is a Cauchy complete archimedean ordered field. Because limits are unique, there is an operator lim which takes Cauchy approximations in F to their limits. We define the embedding $e : \mathbb{R}_c \to F$ by $(\mathbb{R}_c, \sim)$-recursion as

$$e(\mathsf{rat}(q)) :\equiv q \qquad \text{and} \qquad e(\lim(x)) :\equiv \lim(e \circ x).$$

A suitable $\frown$ on F is

$$(a \frown_\epsilon b) :\equiv |a - b| < \epsilon.$$

This is a separated relation because F is archimedean. The rest of the clauses for $(\mathbb{R}_c, \sim)$-recursion are easily checked. One would also have to check that e is an embedding of ordered fields which fixes the rationals. □

11.4 Comparison of Cauchy and Dedekind reals

Let us also say something about the relationship between the Cauchy and Dedekind reals. By Theorem 11.3.46, $\mathbb{R}_{\mathsf{c}}$ is an archimedean ordered field. It is also admissible for Ω, as can be easily checked. (In case Ω is the initial σ-frame it takes a simple induction, while in other cases it is immediate.) Therefore, by Theorem 11.2.14 there is an embedding of ordered fields

$$\mathbb{R}_{\mathsf{c}} \to \mathbb{R}_{\mathsf{d}}$$

which fixes the rational numbers. In general we do not expect $\mathbb{R}_{\mathsf{c}}$ and $\mathbb{R}_{\mathsf{d}}$ to coincide without further assumptions.

Lemma 11.4.1. *If for every $x : \mathbb{R}_{\mathsf{d}}$ there merely exists*

$$c : \prod_{x:\mathbb{R}_{\mathsf{d}}} \prod_{q,r:\mathbb{Q}} (q < r) \to (q < x) + (x < r) \tag{11.4.2}$$

then the Cauchy and Dedekind reals coincide.

Proof. Note that the type in (11.4.2) is an untruncated variant of (11.2.3), which states that $<$ is an intuitionistic linear order. We already know that $\mathbb{R}_{\mathsf{c}}$ embeds into $\mathbb{R}_{\mathsf{d}}$, so it suffices to show that every Dedekind real merely is the limit of a Cauchy sequence of rational numbers.

Consider any $x : \mathbb{R}_{\mathsf{d}}$. By assumption there merely exists c as in the statement of the lemma, and by boundedness of cuts there merely exist $a, b : \mathbb{Q}$ such that $a < x < b$. We construct a sequence $f : \mathbb{N} \to \{ (q, r) \in \mathbb{Q} \times \mathbb{Q} \mid q < r \}$ by recursion:

(i) Set $f(0) :\equiv (a, b)$.

(ii) Suppose $f(n)$ is already defined as (q_n, r_n) such that $q_n < r_n$. Define $s :\equiv (2q_n + r_n)/3$ and $t :\equiv (q_n + 2r_n)/3$. Then $c(x, s, t)$ decides between $s < x$ and $x < t$. If it decides $s < x$ then we set $f(n+1) :\equiv (s, r_n)$, otherwise $f(n+1) :\equiv (q_n, t)$.

Let us write (q_n, r_n) for the n-th term of the sequence f. Then it is easy to see that $q_n < x < r_n$ and $|q_n - r_n| \leq (2/3)^n \cdot |q_0 - r_0|$ for all $n : \mathbb{N}$. Therefore $q_0, q_1, \ldots$ and $r_0, r_1, \ldots$ are both Cauchy sequences converging to the Dedekind cut x. We have shown that for every $x : \mathbb{R}_{\mathsf{d}}$ there merely exists a Cauchy sequence converging to x. □

The lemma implies that either Countable Choice or excluded middle suffice for coincidence of $\mathbb{R}_{\mathsf{c}}$ and $\mathbb{R}_{\mathsf{d}}$.

Corollary 11.4.3. *If excluded middle or Countable Choice holds then $\mathbb{R}_{\mathsf{c}}$ and $\mathbb{R}_{\mathsf{d}}$ are equivalent.*

Proof. If excluded middle holds then $(x < y) \to (x < z) + (z < y)$ can be proved: either $x < z$ or $\neg(x < z)$. In the former case we are done, while in the latter we get $z < y$ because $z \leq x < y$. Therefore, we get (11.4.2) so that we can apply Lemma 11.4.1.

Suppose Countable Choice holds. The set $S = \{ (q, r) \in \mathbb{Q} \times \mathbb{Q} \mid q < r \}$ is equivalent to $\mathbb{N}$, so we may apply Countable Choice to the statement that x is located,

$$\forall((q, r) : S).\, (q < x) \vee (x < r).$$

Note that $(q < x) \vee (x < r)$ is expressible as an existential statement $\exists(b : \mathbf{2}).\, (b = 0_{\mathbf{2}} \to q < x) \wedge (b = 1_{\mathbf{2}} \to x < r)$. The (curried form) of the choice function is then precisely (11.4.2) so that Lemma 11.4.1 is applicable again. □

11.5 Compactness of the interval

We already pointed out that our constructions of reals are entirely compatible with classical logic. Thus, by assuming the law of excluded middle (3.4.1) and the axiom of choice (3.8.1) we could develop classical analysis, which would essentially amount to copying any standard book on analysis.

Nevertheless, anyone interested in computation, for example a numerical analyst, ought to be curious about developing analysis in a computationally meaningful setting. That analysis in a constructive setting is even possible was demonstrated by [Bis67]. As a sample of the differences and similarities between classical and constructive analysis we shall briefly discuss just one topic—compactness of the closed interval $[0,1]$ and a couple of theorems surrounding the concept.

Compactness is no exception to the common phenomenon in constructive mathematics that classically equivalent notions bifurcate. The three most frequently used notions of compactness are:

(i) **metric compact:** "Cauchy complete and totally bounded",
(ii) **Bolzano-Weierstraß compact:** "every sequence has a convergent subsequence",
(iii) **Heine-Borel compact:** "every open cover has a finite subcover".

These are all equivalent in classical mathematics. Let us see how they fare in homotopy type theory. We can use either the Dedekind or the Cauchy reals, so we shall denote the reals just as $\mathbb{R}$. We first recall several basic definitions.

Definition 11.5.1. A **metric space** (M,d) is a set M with a map $d : M \times M \to \mathbb{R}$ satisfying, for all $x,y,z : M$,

$$d(x,y) \geq 0, \qquad d(x,y) = d(y,x),$$
$$d(x,y) = 0 \Leftrightarrow x = y, \qquad d(x,z) \leq d(x,y) + d(y,z).$$

Definition 11.5.2. A **Cauchy approximation** in M is a sequence $x : \mathbb{Q}_+ \to M$ satisfying

$$\forall(\delta,\epsilon).\, d(x_\delta, x_\epsilon) < \delta + \epsilon.$$

The **limit** of a Cauchy approximation $x : \mathbb{Q}_+ \to M$ is a point $\ell : M$ satisfying

$$\forall(\epsilon,\theta : \mathbb{Q}_+).\, d(x_\epsilon, \ell) < \epsilon + \theta.$$

A **complete metric space** is one in which every Cauchy sequence has a limit.

Definition 11.5.3. For a positive rational ϵ, an **ϵ-net** in a metric space (M,d) is an element of

$$\sum_{(n:\mathbb{N})} \sum_{(x_1,\ldots,x_n:M)} \forall(y : M).\, \exists(k \leq n).\, d(x_k, y) < \epsilon.$$

In words, this is a finite sequence of points $x_1,\ldots,x_n$ such that every point in M merely is within ϵ of some x_k.

A metric space (M,d) is **totally bounded** when it has ϵ-nets of all sizes:

$$\prod_{(\epsilon:\mathbb{Q}_+)} \sum_{(n:\mathbb{N})} \sum_{(x_1,\ldots,x_n:M)} \forall(y : M).\, \exists(k \leq n).\, d(x_k, y) < \epsilon.$$

Remark 11.5.4. In the definition of total boundedness we wrote $\sum_{(n:\mathbb{N})} \sum_{(x_1,\ldots,x_n:M)}$, which is sloppy notation. Formally, we should have written $\sum_{(x:\mathsf{List}(M))}$ instead, where $\mathsf{List}(M)$ is the inductive type of finite lists from §5.1. However, that would make the rest of the statement a bit more cumbersome to express.

Note that in the definition of total boundedness we require pure existence of an ϵ-net, not mere existence. This way we obtain a function which assigns to each $\epsilon : \mathbb{Q}_+$ a specific ϵ-net. Such a function might be called a "modulus of total boundedness". In general, when porting classical metric notions to homotopy type theory, we should use propositional truncation sparingly, typically so that we avoid asking for a non-constant map from $\mathbb{R}$ to $\mathbb{Q}$ or $\mathbb{N}$. For instance, here is the "correct" definition of uniform continuity.

Definition 11.5.5. A map $f : M \to \mathbb{R}$ on a metric space is **uniformly continuous** when

$$\prod_{(\epsilon:\mathbb{Q}_+)} \sum_{(\delta:\mathbb{Q}_+)} \forall (x, y : M).\, d(x, y) < \delta \Rightarrow |f(x) - f(y)| < \epsilon.$$

In particular, a uniformly continuous map has a modulus of uniform continuity, which is a function that assigns to each ϵ a corresponding δ.

Let us show that $[0, 1]$ is compact in the first sense.

Theorem 11.5.6. *The closed interval* $[0, 1]$ *is complete and totally bounded.*

Proof. Given $\epsilon : \mathbb{Q}_+$, there is $n : \mathbb{N}$ such that $2/k < \epsilon$, so we may take the ϵ-net $x_i = i/k$ for $i = 0, \ldots, k-1$. This is an ϵ-net because, for every $y : [0, 1]$ there merely exists i such that $0 \leq i < k$ and $(i-1)/k < y < (i+1)/k$, and so $|y - x_i| < 2/k < \epsilon$.

For completeness of $[0, 1]$, consider a Cauchy approximation $x : \mathbb{Q}_+ \to [0, 1]$ and let ℓ be its limit in $\mathbb{R}$. Since max and min are Lipschitz maps, the retraction $r : \mathbb{R} \to [0, 1]$ defined by $r(x) :\equiv \max(0, \min(1, x))$ commutes with limits of Cauchy approximations, therefore

$$r(\ell) = r(\lim x) = \lim(r \circ x) = r(\lim x) = \ell,$$

which means that $0 \leq \ell \leq 1$, as required. □

We thus have at least one good notion of compactness in homotopy type theory. Unfortunately, it is limited to metric spaces because total boundedness is a metric notion. We shall consider the other two notions shortly, but first we prove that a uniformly continuous map on a totally bounded space has a supremum.

Theorem 11.5.7. *A uniformly continuous map* $f : M \to \mathbb{R}$ *on a totally bounded metric space* (M, d) *has a supremum* $m : \mathbb{R}$*. For every* $\epsilon : \mathbb{Q}_+$ *there exists* $u : M$ *such that* $|m - f(u)| < \epsilon$*.*

Proof. Let $h : \mathbb{Q}_+ \to \mathbb{Q}_+$ be the modulus of uniform continuity of f. We define a sequence $x : \mathbb{Q}_+ \to \mathbb{R}$ as follows: for any $\epsilon : \mathbb{Q}$ total boundedness of M gives a $h(\epsilon)$-net $y_0, \ldots, y_n$. Define

$$x_\epsilon :\equiv \max(f(y_0), \ldots, f(y_n)).$$

We claim that x is a Cauchy approximation. Consider any $\epsilon, \eta : \mathbb{Q}$, so that

$$x_\epsilon \equiv \max(f(y_0), \ldots, f(y_n)) \quad \text{and} \quad x_\eta \equiv \max(f(z_0), \ldots, f(z_m))$$

for some $h(\epsilon)$-net $y_0, \ldots, y_n$ and $h(\eta)$-net $z_0, \ldots, z_m$. Every z_i is merely $h(\epsilon)$-close to some y_j, therefore $|f(z_i) - f(y_j)| < \epsilon$, from which we may conclude that

$$f(z_i) < \epsilon + f(y_j) \leq \epsilon + x_\epsilon,$$

therefore $x_\eta < \epsilon + x_\epsilon$. Symmetrically we obtain $x_\eta < \eta + x_\eta$, therefore $|x_\eta - x_\epsilon| < \eta + \epsilon$.

We claim that $m :\equiv \lim x$ is the supremum of f. To prove that $f(x) \leq m$ for all $x : M$ it suffices to show $\neg(m < f(x))$. So suppose to the contrary that $m < f(x)$. There is $\epsilon : \mathbb{Q}_+$ such that $m + \epsilon < f(x)$. But now merely for some y_i participating in the definition of x_ϵ we get $|f(x) - f(y_i) < \epsilon$, therefore $m < f(x) - \epsilon < f(y_i) \leq m$, a contradiction.

We finish the proof by showing that m satisfies the second part of the theorem, because it is then automatically a least upper bound. Given any $\epsilon : \mathbb{Q}_+$, on one hand $|m - f x_{\epsilon/2}| < 3\epsilon/4$, and on the other $|f(x_{\epsilon/2}) - f(y_i)| < \epsilon/4$ merely for some y_i participating in the definition of $x_{\epsilon/2}$, therefore by taking $u :\equiv y_i$ we obtain $|m - f(u)| < \epsilon$ by triangle inequality. □

Now, if in Theorem 11.5.7 we also knew that M were complete, we could hope to weaken the assumption of uniform continuity to continuity, and strengthen the conclusion to existence of a point at which the supremum is attained. The usual proofs of these improvements rely on the the facts that in a complete totally bounded space

(i) continuity implies uniform continuity, and
(ii) every sequence has a convergent subsequence.

The first statement follows easily from Heine-Borel compactness, and the second is just Bolzano-Weierstraß compactness. Unfortunately, these are both somewhat problematic. Let us first show that Bolzano-Weierstraß compactness implies an instance of excluded middle known as the **limited principle of omniscience**: for every $\alpha : \mathbb{N} \to \mathbf{2}$,

$$\left(\sum_{n:\mathbb{N}} \alpha(n) = 1_\mathbf{2}\right) + \left(\prod_{n:\mathbb{N}} \alpha(n) = 0_\mathbf{2}\right). \tag{11.5.8}$$

Computationally speaking, we would not expect this principle to hold, because it asks us to decide whether infinitely many values of a function are $0_\mathbf{2}$.

Theorem 11.5.9. *Bolzano-Weirstraß compactness of* $[0, 1]$ *implies the limited principle of omniscience.*

Proof. Given any $\alpha : \mathbb{N} \to \mathbf{2}$, define the sequence $x : \mathbb{N} \to [0, 1]$ by

$$x_n :\equiv \begin{cases} 0 & \text{if } \alpha(k) = 0_\mathbf{2} \text{ for all } k < n, \\ 1 & \text{if } \alpha(k) = 1_\mathbf{2} \text{ for some } k < n. \end{cases}$$

If the Bolzano-Weierstraß property holds, there exists a strictly increasing $f : \mathbb{N} \to \mathbb{N}$ such that $x \circ f$ is a Cauchy sequence. For a sufficiently large $n : \mathbb{N}$ the n-th term $x_{f(n)}$ is within $1/6$ of its limit. Either $x_{f(n)} < 2/3$ or $x_{f(n)} > 1/3$. If $x_{f(n)} < 2/3$ then x_n converges to 0 and so $\prod_{(n:\mathbb{N})} \alpha(n) = 0_\mathbf{2}$. If $x_{f(n)} > 1/3$ then $x_{f(n)} = 1$, therefore $\sum_{(n:\mathbb{N})} \alpha(n) = 1_\mathbf{2}$. □

While we might not mourn Bolzano-Weierstraß compactness too much, it seems harder to live without Heine-Borel compactness, as attested by the fact that both classical mathematics and Brouwer's Intuitionism accepted it. As we do not want to wade too deeply into general topology, we shall work with basic open sets. In the case of $\mathbb{R}$ these are the open intervals with rational endpoints. A family of such intervals, indexed by a type I, would be a map

$$I \to \sum_{q,r:\mathbb{Q}} (q < r) \to \{\, x : \mathbb{R} \mid q < x < r \,\},$$

but since the pair (q, r) already determines the open interval $\{\, x : \mathbb{R} \mid q < x < r \,\}$, and no harm is done if we throw in empty intervals, we might as well take a **family of basic intervals** to be a map

$$\mathcal{F} : I \to \mathbb{Q} \times \mathbb{Q}.$$

To be quite precise, a family is a dependent pair $(I, \mathcal{F})$, not just $\mathcal{F}$. A **finite family of basic intervals** is one indexed by $\{\, m : \mathbb{N} \mid m < n \,\}$ for some $n : \mathbb{N}$. We usually presented it by a finite list $[(q_0, r_0), \ldots, (q_{n-1}, r_{n-1})]$. Finally, a **finite subfamily** of $(I, \mathcal{F})$ is given by a list of indices $[i_1, \ldots, i_n]$ which then determine the finite family $[\mathcal{F}(i_1), \ldots, \mathcal{F}(i_n)]$.

As long as we are aware of the distinction between a pair (q, r) and the corresponding interval $\{\, x : \mathbb{R} \mid q < x < r \,\}$, we may safely use the same notation (q, r) for both. Intersections and inclusions of intervals are expressible in terms of their endpoints:

$$\begin{aligned} (q, r) \cap (s, t) &:\equiv (\max(q, s), \min(r, t)), \\ (q, r) \subseteq (s, t) &:\equiv (q < r \Rightarrow s \leq q < r \leq t). \end{aligned}$$

We say that $(I, \lambda i.\, (q_i, r_i))$ **(pointwise) covers** $[a, b]$ when

$$\forall (x : [a, b]).\, \exists (i : I).\, q_i < x < r_i. \tag{11.5.10}$$

The **Heine-Borel compactness for** $[0, 1]$ states that every covering family of $[0, 1]$ merely has a finite subfamily which still covers $[0, 1]$.

Theorem 11.5.11. *If excluded middle holds then* $[0, 1]$ *is Heine-Borel compact.*

Proof. Assume for the purpose of reaching a contradiction that a family $(I, \lambda i.\, (a_i, b_i))$ covers $[0, 1]$ but no finite subfamily does. We construct a sequence of closed intervals $[q_n, r_n]$ which are nested, their sizes shrink to 0, and none of them is covered by a finite subfamily of $(I, \lambda i.\, (a_i, b_i))$.

We set $[q_0, r_0] :\equiv [0, 1]$. Assuming $[q_n, r_n]$ has been constructed, let $s :\equiv (2q_n + r_n)/3$ and $t :\equiv (q_n + 2r_n)/3$. Both $[q_n, t]$ and $[s, r_n]$ are covered by $(I, \lambda i.\, (a_i, b_i))$, but they cannot both have a finite subcover, or else so would $[q_n, r_n]$. Either $[q_n, t]$ has a finite subcover or it does not. If it does we set $[q_{n+1}, r_{n+1}] :\equiv [s, r_n]$, otherwise we set $[q_{n+1}, r_{n+1}] :\equiv [q_n, t]$.

The sequences $q_0, q_1, \ldots$ and $r_0, r_1, \ldots$ are both Cauchy and they converge to a point $x : [0, 1]$ which is contained in every $[q_n, r_n]$. There merely exists $i : I$ such that $a_i < x < b_i$. Because the sizes of the intervals $[q_n, r_n]$ shrink to zero, there is $n : \mathbb{N}$ such that $a_i < q_n \leq x \leq r_n < b_i$, but this means that $[q_n, r_n]$ is covered by a single interval (a_i, b_i), while at the same time it has no finite subcover. A contradiction. □

Without excluded middle, or a pinch of Brouwerian Intuitionism, we seem to be stuck. Nevertheless, Heine-Borel compactness of $[0,1]$ *can* be recovered in a constructive setting, in a fashion that is still compatible with classical mathematics! For this to be done, we need to revisit the notion of cover. The trouble with (11.5.10) is that the truncated existential allows a space to be covered in any haphazard way, and so computationally speaking, we stand no chance of merely extracting a finite subcover. By removing the truncation we get

$$\prod_{(x:[0,1])} \sum_{(i:I)} q_i < x < r_i, \tag{11.5.12}$$

which might help, were it not too demanding of covers. With this definition we could not even show that $(0,3)$ and $(2,5)$ cover $[1,4]$ because that would amount to exhibiting a non-constant map $[1,4] \to \mathbf{2}$, see Exercise 11.6. We take a lesson from pointfree topology: the notion of cover ought to be expressed in terms of open sets, without reference to points. Such a "holistic" view of space will then allow us to analyze the notion of cover, and we shall be able to recover Heine-Borel compactness. Since locale theory uses powersets, which we do not have, we steal ideas from its predicative cousin, formal topology.

Suppose that we have a family $(I, \mathcal{F})$ and an interval (a,b). How might we express the fact that (a,b) is covered by the family, without referring to points? Here is one: if (a,b) equals some $\mathcal{F}(i)$ then it is covered by the family. And another one: if (a,b) is covered by some other family $(J,\mathcal{G})$, and in turn each $\mathcal{G}(j)$ is covered by $(I,\mathcal{F})$, then (a,b) is covered $(I,\mathcal{F})$. Notice that we are listing *rules* which can be used to *deduce* that $(I,\mathcal{F})$ covers (a,b). We should find sufficiently good rules and turn them into an inductive definition.

Definition 11.5.13. The **inductive cover** $\lhd$ is a mere relation

$$\lhd : (\mathbb{Q} \times \mathbb{Q}) \to (\sum_{I:\mathcal{U}} (I \to \mathbb{Q} \times \mathbb{Q})) \to \mathsf{Prop}$$

defined inductively by the following rules, where q, r, s, t are rational numbers and $(I,\mathcal{F})$, $(J,\mathcal{G})$ are families of basic intervals:

(i) *relexivity:* $\mathcal{F}(i) \lhd (I,\mathcal{F})$ for all $i : I$,
(ii) *transitivity:* if $(q,r) \lhd (J,\mathcal{G})$ and $\forall (j:J).\, \mathcal{G}(j) \lhd (I,\mathcal{F})$ then $(q,r) \lhd (I,\mathcal{F})$,
(iii) *monotonicity:* if $(q,r) \subseteq (s,t)$ and $(s,t) \lhd (I,\mathcal{F})$ then $(q,r) \lhd (I,\mathcal{F})$,
(iv) *localization:* if $(q,r) \lhd (I,\mathcal{F})$ then $(q,r) \cap (s,t) \lhd (I, \lambda i.\, (\mathcal{F}(i) \cap (s,t)))$.
(v) if $q < s < t < r$ then $(q,r) \lhd [(q,t),(r,s)]$,
(vi) $(q,r) \lhd (\{\, \mathbb{Q} \times \mathbb{Q} \mid q < s < t < r \,\}, \lambda u.\, u)$.

The definition should be read as a higher-inductive type in which the listed rules are point constructors, and the type is (-1)-truncated. The first four clauses are of a general nature and should be intuitively clear. The last two clauses are specific to the real line: one says that an interval may be covered by two intervals if they overlap, while the other one says that an interval may be covered from within. Incidentally, if $r \leq q$ then (q,r) is covered by the empty family by the last clause.

Experience from formal topology shows that the rules for inductive covers are sufficient for a constructive development of pointfree topology. But we can also provide our own evidence that they are a reasonable notion.

Theorem 11.5.14.

(i) An inductive cover is also a pointwise cover.

(ii) Assuming excluded middle, a pointwise cover is also an inductive cover.

Proof.

(i) Consider a family of basic intervals $(I, \mathcal{F})$, where we write $(q_i, r_i) :\equiv \mathcal{F}(i)$, an interval (a, b) inductively covered by $(I, \mathcal{F})$, and x such that $a < x < b$. We prove by induction on $(a, b) \lhd (I, \mathcal{F})$ that there merely exists $i : I$ such that $q_i < x < r_i$. Most cases are pretty obvious, so we show just two. If $(a, b) \lhd (I, \mathcal{F})$ by reflexivity, then there merely is some $i : I$ such that $(a, b) = (q_i, r_i)$ and so $q_i < x < r_i$. If $(a, b) \lhd (I, \mathcal{F})$ by transitivity via $(J, \lambda j.\, (s_j, t_j))$ then by induction hypothesis there merely is $j : J$ such that $s_j < x < t_j$, and then since $(s_j, t_j) \lhd (I, \mathcal{F})$ again by induction hypothesis there merely exists $i : I$ such that $q_i < x < r_i$. Other cases are just as exciting.

(ii) Suppose $(I, \lambda i.\, (q_i, r_i))$ pointwise covers $(0, 1)$. By Theorem 11.5.11 there is a finite subfamily $[i_1, \ldots, i_n]$ which already pointwise covers $(0, 1)$. Let $\epsilon : \mathbb{Q}_+$ be a Lebesgue number for $(q_{i_1}, r_{i_1}), \ldots, (q_{i_n}, r_{i_n})$ as in Exercise 11.11. There is a positive $k : \mathbb{N}$ such that $2/k < \min(1, \epsilon)$. The intervals $(i/k, (i+2)/k)$, where i ranges over $0, \ldots, k-2$, inductively cover $(0, 1)$ by repeated use of transitivity and Item (v) in Definition 11.5.13. Because each of them is contained in some (q_i, r_i) we may use transitivity and monotonicity to conclude that $(I, \lambda i.\, (q_i, r_i))$ inductively cover $(0, 1)$. □

The upshot of the previous theorem is that, as far as classical mathematics is concerned, there is no difference between a pointwise and an inductive cover. In particular, since it is consistent to assume excluded middle in homotopy type theory, we cannot exhibit an inductive cover which fails to be a pointwise cover. Or to put it in a different way, the difference between pointwise and inductive covers is not what they cover but in the *proofs* that they cover. In any case, inductive covers enjoy the Heine-Borel property.

Lemma 11.5.15. *Suppose $q < s < t < r$ and $(q, r) \lhd (I, \mathcal{F})$. Then there merely exists a finite subfamily of $(I, \mathcal{F})$ which inductively covers (s, t).*

Proof. We prove the statement by induction on $(q, r) \lhd (I, \mathcal{F})$. There are six cases:

(i) Reflexivity: if $(q, r) = \mathcal{F}(i)$ then by monotonicity (s, t) is covered by the finite subfamily $[\mathcal{F}(i)]$.

(ii) Transitivity: suppose $(q, r) \lhd (J, \mathcal{G})$ and $\forall(j).\, \forall(J).\, \mathcal{G}(j) \lhd (I, \mathcal{F})$. By induction hypothesis there merely exists a finite subfamily $[\mathcal{G}(j_1), \ldots, \mathcal{G}(j_n)]$ which covers (s, t). Again by induction hypothesis, each of $\mathcal{G}(j_k)$ is covered by a finite subfamily of $(I, \mathcal{F})$, and we can collect these into a finite subfamily which covers (s, t).

(iii) Monotonicity: if $(q, r) \subseteq (u, v)$ and $(u, v) \lhd (I, \mathcal{F})$ then we may apply the induction hypothesis to $(u, v) \lhd (I, \mathcal{F})$ because $u < s < t < v$.

(iv) Localization: suppose $(q', r') \lhd (I, \mathcal{F})$ and $(q, r) = (q', r') \cap (a, b)$. Because $q' < s < t < r'$, by induction hypothesis there is a finite subcover $[\mathcal{F}(i_1), \dots, \mathcal{F}(i_n)]$ of (s, t). We also know that $a < s < t < b$, therefore $(s, t) = (s, t) \cap (a, b)$ is covered by $[\mathcal{F}(i_1) \cap (a, b), \dots, \mathcal{F}(i_n) \cap (a, b)]$, which is a finite subfamily of $(I, \lambda i.\, (\mathcal{F}(i) \cap (a, b)))$.

(v) If $(q, r) \lhd [(q, v), (u, r)]$ for some $q < u < v < r$ then by monotonicity $(s, t) \lhd [(q, v), (u, r)]$.

(vi) Finally, $(s, t) \lhd (\{\, \mathbb{Q} \times \mathbb{Q} \mid q < u < v < r \,\}, \lambda u.\, z)$ by reflexivity. □

Finally, we are ready to prove that the closed interval is Heine-Borel compact with respect to inductive covers. Say that $(I, \mathcal{F})$ **inductively covers** $[a, b]$ when there merely exists $\epsilon : \mathbb{Q}_+$ such that $(a - \epsilon, b + \epsilon) \lhd (I, \mathcal{F})$.

Corollary 11.5.16. *A closed interval is Heine-Borel compact for inductive covers.*

Proof. Suppose $[a, b]$ is inductively covered by $(I, \mathcal{F})$, so there merely is $\epsilon : \mathbb{Q}_+$ such that $(a - \epsilon, b + \epsilon) \lhd (I, \mathcal{F})$. By Lemma 11.5.15 there is a finite subcover of $(a - \epsilon/2, b + \epsilon/2)$, which is therefore a finite subcover of $[a, b]$. □

We could write another book by going on like this, but let us stop here and hope that we have provided ample justification for the claim that analysis can be developed in homotopy type theory. The curious reader should consult Exercise 11.12 for constructive versions of the mean value theorem.

11.6 The surreal numbers

In this section we consider another example of a higher inductive-inductive type, which draws together many of our threads: Conway's field No of *surreal numbers* [Con76]. The surreal numbers are the natural common generalization of the (Dedekind) real numbers (§11.2) and the ordinal numbers (§10.3). Conway, working in classical mathematics with excluded middle and Choice, defines a surreal number to be a pair of *sets* of surreal numbers, written $\{\, L \mid R \,\}$, such that every element of L is strictly less than every element of R. This obviously looks like an inductive definition, but there are three issues with regarding it as such.

Firstly, the definition requires the relation of (strict) inequality between surreals, so that relation must be defined simultaneously with the type No of surreals. (Conway avoids this issue by first defining *games*, which are like surreals but omit the compatibility condition on L and R.) As with the relation $\sim$ for the Cauchy reals, this simultaneous definition could *a priori* be either inductive-inductive or inductive-recursive. We will choose to make it inductive-inductive, for the same reasons we made that choice for $\sim$.

Moreover, we will define strict inequality $<$ and non-strict inequality $\leq$ for surreals separately (and mutually inductively). Conway defines $<$ in terms of $\leq$, in a way which is sensible classically but not constructively. Furthermore, a negative definition of $<$ would make it unacceptable as a hypothesis of the constructor of a higher inductive type (see §5.6).

Secondly, Conway says that L and R in $\{\, L \mid R \,\}$ should be "sets of surreal numbers", but the naive meaning of this as a predicate $\mathsf{No} \to \mathsf{Prop}$ is not positive, hence cannot be used as input to an inductive constructor. However, this would not be a good type-theoretic translation of what

Conway means anyway, because in set theory the surreal numbers form a proper class, whereas the sets L and R are true (small) sets, not arbitrary subclasses of No. In type theory, this means that No will be defined relative to a universe $\mathcal{U}$, but will itself belong to the next higher universe $\mathcal{U}'$, like the sets Ord and Card of ordinals and cardinals, the cumulative hierarchy V, or even the Dedekind reals in the absence of propositional resizing. We will then require the "sets" L and R of surreals to be $\mathcal{U}$-small, and so it is natural to represent them by *families* of surreals indexed by some $\mathcal{U}$-small type. (This is all exactly the same as what we did with the cumulative hierarchy in §10.5.) That is, the constructor of surreals will have type

$$\prod_{\mathcal{L},\mathcal{R}:\mathcal{U}} (\mathcal{L} \to \mathsf{No}) \to (\mathcal{R} \to \mathsf{No}) \to (\text{some condition}) \to \mathsf{No}$$

which is indeed strictly positive.

Finally, after giving the mutual definitions of No and its ordering, Conway declares two surreal numbers x and y to be *equal* if $x \leq y$ and $y \leq x$. This is naturally read as passing to a quotient of the set of "pre-surreals" by an equivalence relation. However, in the absence of the axiom of choice, such a quotient presents the same problem as the quotient in the usual construction of Cauchy reals: it will no longer be the case that a pair of families *of surreals* yield a new surreal $\{ L \mid R \}$, since we cannot necessarily "lift" L and R to families of pre-surreals. Of course, we can solve this problem in the same way we did for Cauchy reals, by using a *higher* inductive-inductive definition.

Definition 11.6.1. The type No of **surreal numbers**, along with the relations $< : \mathsf{No} \to \mathsf{No} \to \mathcal{U}$ and $\leq : \mathsf{No} \to \mathsf{No} \to \mathcal{U}$, are defined higher inductive-inductively as follows. The type No has the following constructors.

- For any $\mathcal{L}, \mathcal{R} : \mathcal{U}$ and functions $\mathcal{L} \to \mathsf{No}$ and $\mathcal{R} \to \mathsf{No}$, whose values we write as x^L and x^R for $L : \mathcal{L}$ and $R : \mathcal{R}$ respectively, if $\forall(L : \mathcal{L}).\, \forall(R : \mathcal{R}).\, x^L < x^R$, then there is a surreal number x.
- For any $x, y : \mathsf{No}$ such that $x \leq y$ and $y \leq x$, we have $x = y$.

We will refer to the inputs of the first constructor as a *cut*. If x is the surreal number constructed from a cut, then the notation x^L will implicitly assume $L : \mathcal{L}$, and similarly x^R will assume $R : \mathcal{R}$. In this way we can usually avoid naming the indexing types $\mathcal{L}$ and $\mathcal{R}$, which is convenient when there are many different cuts under discussion. Following Conway, we call x^L a *left option* of x and x^R a *right option*.

The path-constructor implies that different cuts can define the same surreal number. Thus, it does not make sense to speak of the left or right options of an arbitrary surreal number x, unless we also know that x is defined by a particular cut. Thus in what follows we will say, for instance, "given a cut defining a surreal number x" in contrast to "given a surreal number x".

The relation $\leq$ has the following constructors.

- Given cuts defining two surreal numbers x and y, if $x^L < y$ for all L, and $x < y^R$ for all R, then $x \leq y$.
- Propositional truncation:[2] for any $x, y : \mathsf{No}$, if $p, q : x \leq y$, then $p = q$.

[2] This constructor may not be necessary ($\leq$ might turn out to be a mere relation automatically), but we include it for convenience.

And the relation $<$ has the following constructors.

- Given cuts defining two surreal numbers x and y, if there is an L such that $x \leq y^L$, then $x < y$.
- Given cuts defining two surreal numbers x and y, if there is an R such that $x^R \leq y$, then $x < y$.
- Propositional truncation: for any $x, y : \mathsf{No}$, if $p, q : x < y$, then $p = q$.

We compare this with Conway's definitions:

- If L, R are any two sets of numbers, and no member of L is $\geq$ any member of R, then there is a number $\{ L \mid R \}$. All numbers are constructed in this way.
- $x \geq y$ iff (no $x^R \leq y$ and $x \leq$ no y^L).
- $x = y$ iff ($x \geq y$ and $y \geq x$).
- $x > y$ iff ($x \geq y$ and $y \ngeq x$).

The inclusion of $x \geq y$ in the definition of $x > y$ is unnecessary if all objects are [surreal] numbers rather than "games". Thus, Conway's $<$ is just the negation of his $\leq$, so that his condition for $\{ L \mid R \}$ to be a surreal is the same as ours. Negating Conway's $\leq$ and canceling double negations, we arrive at our definition of $<$, and we can then reformulate his $\leq$ in terms of $<$ without negations.

We can immediately populate No with many surreal numbers. Like Conway, we write

$$\{ x, y, z, \dots \mid u, v, w, \dots \}$$

for the surreal number defined by a cut where $\mathcal{L} \to \mathsf{No}$ and $\mathcal{R} \to \mathsf{No}$ are families described by $x, y, z, \dots$ and $u, v, w, \dots$. Of course, if $\mathcal{L}$ or $\mathcal{R}$ are $\mathbf{0}$, we leave the corresponding part of the notation empty. There is an unfortunate clash with the standard notation $\{ x : A \mid P(x) \}$ for subsets, but we will not use the latter in this section.

- We define $\iota_{\mathbb{N}} : \mathbb{N} \to \mathsf{No}$ recursively by

$$\begin{aligned} \iota_{\mathbb{N}}(0) &:\equiv \{ \mid \} \\ \iota_{\mathbb{N}}(\mathsf{succ}(n)) &:\equiv \{ \iota_{\mathbb{N}}(n) \mid \} \end{aligned}$$

  That is, $\iota_{\mathbb{N}}(0)$ is defined by the cut consisting of $\mathbf{0} \to \mathsf{No}$ and $\mathbf{0} \to \mathsf{No}$. Similarly, $\iota_{\mathbb{N}}(\mathsf{succ}(n))$ is defined by $\mathbf{1} \to \mathsf{No}$ (picking out $\iota_{\mathbb{N}}(n)$) and $\mathbf{0} \to \mathsf{No}$.
- Similarly, we define $\iota_{\mathbb{Z}} : \mathbb{Z} \to \mathsf{No}$ using the sign-case recursion principle:

$$\begin{aligned} \iota_{\mathbb{Z}}(0) &:\equiv \{ \mid \} \\ \iota_{\mathbb{Z}}(n+1) &:\equiv \{ \iota_{\mathbb{Z}}(n) \mid \} \quad n \geq 0 \\ \iota_{\mathbb{Z}}(n-1) &:\equiv \{ \mid \iota_{\mathbb{Z}}(n) \} \quad n \leq 0 \end{aligned}$$

- By a *dyadic rational* we mean a pair (a, n) where $a : \mathbb{Z}$ and $n : \mathbb{N}$, and such that if $n > 0$ then a is odd. We will write it as $a/2^n$, and identify it with the corresponding rational number. If $\mathbb{Q}_D$ denotes the set of dyadic rationals, we define $\iota_{\mathbb{Q}_D} : \mathbb{Q}_D \to \mathsf{No}$ by induction on n:

$$\begin{aligned} \iota_{\mathbb{Q}_D}(a/2^0) &:\equiv \iota_{\mathbb{Z}}(a) \\ \iota_{\mathbb{Q}_D}(a/2^n) &:\equiv \{ a/2^n - 1/2^n \mid a/2^n + 1/2^n \} \quad n > 0 \end{aligned}$$

Here we use the fact that if $n > 0$ and a is odd, then $a/2^n \pm 1/2^n$ is a dyadic rational with a smaller denominator than $a/2^n$.

- We define $\iota_{\mathbb{R}_\mathsf{d}} : \mathbb{R}_\mathsf{d} \to \mathsf{No}$, where $\mathbb{R}_\mathsf{d}$ is (any version of) the Dedekind reals from §11.2, by
$$\iota_{\mathbb{R}_\mathsf{d}}(x) :\equiv \{\, q \in \mathbb{Q}_D \text{ such that } q < x \mid q \in \mathbb{Q}_D \text{ such that } x < q \,\}$$
Unlike in the previous cases, it is not obvious that this extends $\iota_{\mathbb{Q}_D}$ when we regard dyadic rationals as Dedekind reals. This follows from the simplicity theorem (Theorem 11.6.2).
- Recall the type Ord of *ordinals* from §10.3, which is well-ordered by the relation $<$, where $A < B$ means that $A = B_{/b}$ for some $b : B$. We define $\iota_{\mathsf{Ord}} : \mathsf{Ord} \to \mathsf{No}$ by well-founded recursion (Lemma 10.3.7) on Ord:
$$\iota_{\mathsf{Ord}}(A) :\equiv \{\, \iota_{\mathsf{Ord}}(A_{/a}) \text{ for all } a : A \mid \,\}$$
It will also follow from the simplicity theorem that ι_{Ord} restricted to finite ordinals agrees with $\iota_{\mathbb{N}}$.
- A few more interesting examples taken from Conway:
$$\begin{aligned}
\omega &:\equiv \{\, 0, 1, 2, 3, \ldots \mid \,\} \qquad \text{(also an ordinal)} \\
-\omega &:\equiv \{\, \mid \ldots, -3, -2, -1, 0 \,\} \\
1/\omega &:\equiv \{\, 0 \mid 1, \tfrac{1}{2}, \tfrac{1}{4}, \tfrac{1}{8}, \ldots \,\} \\
\omega - 1 &:\equiv \{\, 0, 1, 2, 3, \ldots \mid \omega \,\} \\
\omega/2 &:\equiv \{\, 0, 1, 2, 3, \ldots \mid \ldots, \omega - 2, \omega - 1, \omega \,\}.
\end{aligned}$$

In identifying surreal numbers presented by different cuts, the following simple observation is useful.

Theorem 11.6.2 (Conway's simplicity theorem)**.** *Suppose x and z are surreal numbers defined by cuts, and that the following hold.*

- *$x^L < z < x^R$ for all L and R.*
- *For every left option z^L of z, there exists a left option $x^{L'}$ with $z^L \leq x^{L'}$.*
- *For every right option z^R of z, there exists a right option $x^{R'}$ with $x^{R'} \leq z^R$.*

Then $x = z$.

Proof. Applying the path-constructor of No, we must show $x \leq z$ and $z \leq x$. The first entails showing $x^L < z$ for all L, which we assumed, and $x < z^R$ for all R. But by assumption, for any z^R there is an $x^{R'}$ with $x^{R'} \leq z^R$ hence $x < z^R$ as desired. Thus $x \leq z$; the proof of $z \leq x$ is symmetric. □

In order to say much more about surreal numbers, however, we need their induction principle. The mutual induction principle for $(\mathsf{No}, \leq, <)$ applies to three families of types:

$$\begin{aligned}
A &: \mathsf{No} \to \mathcal{U} \\
B &: \prod_{(x,y:\mathsf{No})} \prod_{(a:A(x))} \prod_{(b:A(y))} (x \leq y) \to \mathcal{U} \\
C &: \prod_{(x,y:\mathsf{No})} \prod_{(a:A(x))} \prod_{(b:A(y))} (x < y) \to \mathcal{U}.
\end{aligned}$$

It requires the following hypotheses:

- For any cut defining a surreal number x, together with

  (i) for each L, an element $a^L : A(x^L)$, and
  (ii) for each R, an element $a^R : A(x^R)$, such that
  (iii) for all L and R we have $C(x^L, x^R, a^L, a^R, \_)$

  there is a specified element $f_a : A(x)$. We call such data a *dependent cut* over the cut defining x.
- For any $x, y : \mathsf{No}$ with $a : A(x)$ and $b : A(y)$, if $x \leq y$ and $y \leq x$ and also $B(x, y, a, b, \_)$ and $B(y, x, b, a, \_)$, then $a =^A_{\_} b$.
- Given cuts defining two surreal numbers x and y, and dependent cuts a over x and b over y, such that for all L we have $x^L < y$ and $C(x^L, y, a^L, b, \_)$, and for all R we have $x < y^R$ and $C(x, y^R, a, b^R, \_)$, then $B(x, y, a, b, \_)$.
- B is a family of mere propositions.
- Given cuts defining two surreal numbers x and y, dependent cuts a over x and b over y, and an L_0 such that $x \leq y^{L_0}$ and $B(x, y^{L_0}, a, b^{L_0}, \_)$, we have $C(x, y, a, b, \_)$.
- Given cuts defining two surreal numbers x and y, dependent cuts a over x and b over y, and an R_0 such that $x^{R_0} \leq y$ and $B(x^{R_0}, y, a^{R_0}, b, \_)$, we have $C(x, y, a, b, \_)$.
- C is a family of mere propositions.

Under these hypotheses we deduce a function $f : \prod_{(x:\mathsf{No})} A(x)$ such that

$$(x \leq y) \to B(x, y, f(x), f(y), \_)$$
$$(x < y) \to C(x, y, f(x), f(y), \_).$$

Its computation rule on the point-constructor can be written as $f(x) \equiv f_{f[x]}$, where x is a surreal number defined by a cut, and $f[x]$ denotes the dependent cut over x defined by applying f (and using the fact that f takes $<$ to C).

As with the Cauchy reals, we have special cases resulting from trivializing some of A, B, and C. Taking B and C to be constant at $\mathbf{1}$, we have No-*induction*, which for simplicity we state only for mere properties:

- Given $P : \mathsf{No} \to \mathsf{Prop}$, if $P(x)$ holds whenever x is a surreal number defined by a cut such that $P(x^L)$ and $P(x^R)$ hold for all L and R, then $P(x)$ holds for all $x : \mathsf{No}$.

This should be compared with Conway's remark:

> In general when we wish to establish a proposition $P(x)$ for all numbers x, we will prove it inductively by deducing $P(x)$ from the truth of all the propositions $P(x^L)$ and $P(x^R)$. We regard the phrase "all numbers are constructed in this way" as justifying the legitimacy of this procedure.

With No-induction, we can prove

Theorem 11.6.3 (Conway's Theorem 0)**.**

(i) *For any* $x : \mathsf{No}$, *we have* $x \leq x$.

(ii) *For any* $x : \mathsf{No}$ *defined by a cut, we have* $x^L < x$ *and* $x < x^R$ *for all* L *and* R.

Proof. Note first that if $x \leq x$, then whenever x occurs as a left option of some cut y, we have $x < y$ by the first constructor of $<$, and similarly whenever x occurs as a right option of a cut y, we have $y < x$ by the second constructor of $<$. In particular, (i)⇒(ii).

We prove (i) by No-induction on x. Thus, assume x is defined by a cut such that $x^L \leq x^L$ and $x^R \leq x^R$ for all L and R. But by our observation above, these assumptions imply $x^L < x$ and $x < x^R$ for all L and R, yielding $x \leq x$ by the constructor of $\leq$. □

Corollary 11.6.4. No *is a 0-type.*

Proof. The mere relation $R(x, y) :\equiv (x \leq y) \wedge (y \leq x)$ implies identity by the path-constructor of No, and contains the diagonal by Theorem 11.6.3(i). Thus, Theorem 7.2.2 applies. □

By contrast, Conway's Theorem 1 (transitivity of $\leq$) is somewhat harder to establish with our definition; see Corollary 11.6.16.

We will also need the joint recursion principle, $(\mathsf{No}, \leq, <)$*-recursion*, which it is convenient to state as follows. Suppose A is a type equipped with relations $\preccurlyeq : A \to A \to \mathsf{Prop}$ and $\lhd : A \to A \to \mathsf{Prop}$. Then we can define $f : \mathsf{No} \to A$ by doing the following.

(i) For any x defined by a cut, assuming $f(x^L)$ and $f(x^R)$ to be defined such that $f(x^L) \lhd f(x^R)$ for all L and R, we must define $f(x)$. (We call this the *primary clause* of the recursion.)

(ii) Prove that $\preccurlyeq$ is *antisymmetric*: if $a \preccurlyeq b$ and $b \preccurlyeq a$, then $a = b$.

(iii) For x, y defined by cuts such that $x^L < y$ for all L and $x < y^R$ for all R, and assuming inductively that $f(x^L) \lhd f(y)$ for all L, $f(x) \lhd f(y^R)$ for all R, and also that $f(x^L) \lhd f(x^R)$ and $f(y^L) \lhd f(y^R)$ for all L and R, we must prove $f(x) \preccurlyeq f(y)$.

(iv) For x, y defined by cuts and an L_0 such that $x \leq y^{L_0}$, and assuming inductively that $f(x) \preccurlyeq f(y^{L_0})$, and also that $f(x^L) \lhd f(x^R)$ and $f(y^L) \lhd f(y^R)$ for all L and R, we must prove $f(x) \lhd f(y)$.

(v) For x, y defined by cuts and an R_0 such that $x^{R_0} \leq y$, and assuming inductively that $f(x^{R_0}) \preccurlyeq f(y)$, and also that $f(x^L) \lhd f(x^R)$ and $f(y^L) \lhd f(y^R)$ for all L and R, we must prove $f(x) \lhd f(y)$.

The last three clauses can be more concisely described by saying we must prove that f (as defined in the first clause) takes $\leq$ to $\preccurlyeq$ and $<$ to $\lhd$. We will refer to these properties by saying that f *preserves inequalities*. Moreover, in proving that f preserves inequalities, we may assume the particular instance of $\leq$ or $<$ to be obtained from one of its constructors, and we may also use inductive hypotheses that f preserves all inequalities appearing in the input to that constructor.

If we succeed at (i)–(v) above, then we obtain $f : \mathsf{No} \to A$, which computes on cuts as specified by (i), and which preserves all inequalities:

$$\forall (x, y : \mathsf{No}).\ \Big((x \leq y) \to (f(x) \preccurlyeq f(y))\Big) \wedge \Big((x < y) \to (f(x) \lhd f(y))\Big).$$

Like $(\mathbb{R}_{\mathsf{c}}, \sim)$-recursion for the Cauchy reals, this recursion principle is essential for defining functions on No, since we cannot first define a function on "pre-surreals" and only later prove that it respects the notion of equality.

Example 11.6.5. Let us define the *negation* function $\mathsf{No} \to \mathsf{No}$. We apply the joint recursion principle with $A :\equiv \mathsf{No}$, with $(x \trianglelefteq y) :\equiv (y \leq x)$, and $(x \triangleleft y) :\equiv (y < x)$. Clearly this $\trianglelefteq$ is antisymmetric.

For the main clause in the definition, we assume x defined by a cut, with $-x^L$ and $-x^R$ defined such that $-x^L \triangleleft -x^R$ for all L and R. By definition, this means $-x^R < -x^L$ for all L and R, so we can define $-x$ by the cut $\{ -x^R \mid -x^L \}$. This notation, which follows Conway, refers to the cut whose left options are indexed by the type $\mathcal{R}$ indexing the right options of x, and whose right options are indexed by the type $\mathcal{L}$ indexing the left options of x, with the corresponding families $\mathcal{R} \to \mathsf{No}$ and $\mathcal{L} \to \mathsf{No}$ defined by composing those for x with negation.

We now have to verify that f preserves inequalities.

- For $x \leq y$, we may assume $x^L < y$ for all L and $x < y^R$ for all R, and show $-y \leq -x$. But inductively, we may assume $-y < -x^L$ and $-y^R < -x$, which gives the desired result, by definition of $-y$, $-x$, and the constructor of $\leq$.
- For $x < y$, in the first case when it arises from some $x \leq y^{L_0}$, we may inductively assume $-y^{L_0} \leq -x$, in which case $-y < -x$ follows by the constructor of $<$.
- Similarly, if $x < y$ arises from $x^{R_0} \leq y$, the induction hypothesis is $-y \leq -x^R$, yielding $-y < -x$ again.

To do much more than this, however, we will need to characterize the relations $\leq$ and $<$ more explicitly, as we did for the Cauchy reals in Theorem 11.3.30. Also as there, we will have to simultaneously prove a couple of essential properties of these relations, in order for the induction to go through.

Theorem 11.6.6. *There are relations* $\preceq : \mathsf{No} \to \mathsf{No} \to \mathsf{Prop}$ *and* $\prec : \mathsf{No} \to \mathsf{No} \to \mathsf{Prop}$ *such that if* x *and* y *are surreals defined by cuts, then*

$$(x \preceq y) :\equiv \big(\forall(L).\, x^L \prec y\big) \wedge \big(\forall(R).\, x \prec y^R\big)$$
$$(x \prec y) :\equiv \big(\exists(L).\, x \preceq y^L\big) \vee \big(\exists(R).\, x^R \preceq y\big).$$

Moreover, we have

$$(x \prec y) \to (x \preceq y) \tag{11.6.7}$$

and all the reasonable transitivity properties making $\prec$ *and* $\preceq$ *into a "bimodule" over* $\leq$ *and* $<$*:*

$$\begin{array}{ll} (x \leq y) \to (y \preceq z) \to (x \preceq z) & (x \preceq y) \to (y \leq z) \to (x \preceq z) \\ (x \leq y) \to (y \prec z) \to (x \prec z) & (x \preceq y) \to (y < z) \to (x \prec z) \\ (x < y) \to (y \preceq z) \to (x \prec z) & (x \prec y) \to (y \leq z) \to (x \prec z) \end{array} \tag{11.6.8}$$

Proof. We define $\preceq$ and $\prec$ by double $(\mathsf{No}, \leq, <)$-induction on x, y. The first induction is a simple recursion, whose codomain is the subset A of $(\mathsf{No} \to \mathsf{Prop}) \times (\mathsf{No} \to \mathsf{Prop})$ consisting of pairs of predicates of which one implies the other and which satisfy "transitivity on the right", i.e. (11.6.7) and the right column of (11.6.8) with $(x \preceq -)$ and $(x \prec -)$ replaced by the two given predicates. As in the proof of Theorem 11.3.14, we regard these predicates as half of binary relations, writing

them as $y \mapsto (\diamondsuit \preceq y)$ and $y \mapsto (\diamondsuit \prec y)$, with $\diamondsuit$ denoting the pair of relations. We equip A with the following two relations:

$$(\diamondsuit \trianglelefteq \heartsuit) :\equiv \forall(y : \mathsf{No}).\Big((\heartsuit \preceq y) \to (\diamondsuit \preceq y)\Big) \wedge \Big((\heartsuit \prec y) \to (\diamondsuit \prec y)\Big)$$
$$(\diamondsuit \triangleleft \heartsuit) :\equiv \forall(y : \mathsf{No}).\Big((\heartsuit \preceq y) \to (\diamondsuit \prec y)\Big)$$

Note that $\trianglelefteq$ is antisymmetric, since if $\diamondsuit \trianglelefteq \heartsuit$ and $\heartsuit \trianglelefteq \diamondsuit$, then $(\heartsuit \preceq y) \leftrightarrow (\diamondsuit \preceq y)$ and $(\heartsuit \prec y) \leftrightarrow (\diamondsuit \prec y)$ for all y, hence $\diamondsuit = \heartsuit$ by univalence for mere propositions and function extensionality. Moreover, to say that a function $\mathsf{No} \to A$ preserves inequalities is exactly to say that, when regarded as a pair of binary relations on No, it satisfies "transitivity on the left" (the left column of (11.6.8)).

Now for the primary clause of the recursion, we assume given x defined by a cut, and relations $(x^L \prec -)$, $(x^R \prec -)$, $(x^L \preceq -)$, and $(x^R \preceq -)$ for all L and R, of which the strict ones imply the non-strict ones, which satisfy transitivity on the right, and such that

$$\forall(L, R).\forall(y : \mathsf{No}).\Big((x^R \preceq y) \to (x^L \prec y)\Big) \tag{11.6.9}$$

We now have to define $(x \prec y)$ and $(x \preceq y)$ for all y. Here in contrast to Theorem 11.3.14, rather than a nested recursion, we use a nested induction, in order to be able to inductively use transitivity on the left with respect to the inequalities $x^L < x$ and $x < x^R$. Define $A' : \mathsf{No} \to \mathcal{U}$ by taking $A'(y)$ to be the subset A' of $\mathsf{Prop} \times \mathsf{Prop}$ consisting of two mere propositions, denoted $\triangle \preceq y$ and $\triangle \prec y$ (with $\triangle : A'(y)$), such that

$$(\triangle \prec y) \to (\triangle \preceq y) \tag{11.6.10}$$
$$\forall(L).(\triangle \preceq y) \to (x^L \prec y) \tag{11.6.11}$$
$$\forall(R).(x^R \preceq y) \to (\triangle \prec y). \tag{11.6.12}$$

Using notation analogous to $\trianglelefteq$ and $\triangleleft$ for recursion, we equip A' with the two relations defined for $\triangle : A'(y)$ and $\square : A'(z)$ by

$$(\triangle \sqsubseteq \square) :\equiv \Big((\triangle \preceq y) \to (\square \preceq z)\Big) \wedge \Big((\triangle \prec y) \to (\square \prec z)\Big)$$
$$(\triangle \sqsubset \square) :\equiv \Big((\triangle \preceq y) \to (\square \prec z)\Big).$$

(These are the type families B and C in the general induction principle.) Again, $\sqsubseteq$ is evidently antisymmetric in the appropriate sense. Moreover, a function $\prod_{(y:\mathsf{No})} A'(y)$ which preserves inequalities is precisely a pair of predicates of which one implies the other, which satisfy transitivity on the right, and transitivity on the left with respect to the inequalities $x^L < x$ and $x < x^R$. Thus, this inner induction will provide what we need to complete the primary clause of the outer recursion.

For the primary clause of the inner induction, we assume also given y defined by a cut, and properties $(x \prec y^L)$, $(x \prec y^R)$, $(x \preceq y^L)$, and $(x \preceq y^R)$ for all L and R, with the strict ones implying the non-strict ones, transitivity on the left with respect to $x^L < x$ and $x < x^R$, and

on the right with respect to $y^L < y^R$. We can now give the definitions specified in the theorem statement:

$$(x \preceq y) :\equiv (\forall(L).\, x^L \prec y) \wedge (\forall(R).\, x \prec y^R) \tag{11.6.13}$$

$$(x \prec y) :\equiv (\exists(L).\, x \preceq y^L) \vee (\exists(R).\, x^R \preceq y). \tag{11.6.14}$$

For this to define an element of $A'(y)$, we must show first that $(x \prec y) \to (x \preceq y)$. The assumption $x \prec y$ has two cases. On one hand, if there is L_0 with $x \preceq y^{L_0}$, then by transitivity on the right with respect to $y^{L_0} < y^R$, we have $x \prec y^R$ for all R. Moreover, by transitivity on the left with respect to $x^L < x$, we have $x^L \prec y^{L_0}$ for any L, hence $x^L \prec y$ by transitivity on the right. Thus, $x \preceq y$.

On the other hand, if there is R_0 with $x^{R_0} \preceq y$, then by transitivity on the left with respect to $x^L < x^{R_0}$ we have $x^L \prec y$ for all L. And by transitivity on the left and right with respect to $x < x^{R_0}$ and $y < y^R$, we have $x \prec y^R$ for any R. Thus, $x \preceq y$.

We also need to show that these definitions are transitive on the left with respect to $x^L < x$ and $x < x^R$. But if $x \preceq y$, then $x^L \prec y$ for all L by definition; while if $x^R \preceq y$, then $x \prec y$ also by definition.

Thus, (11.6.13) and (11.6.14) do define an element of $A'(y)$. We now have to verify that this definition preserves inequalities, as a dependent function into A', i.e. that these relations are transitive on the right. Remember that in each case, we may assume inductively that they are transitive on the right with respect to all inequalities arising in the inequality constructor.

- Suppose $x \preceq y$ and $y \leq z$, the latter arising from $y^L < z$ and $y < z^R$ for all L and R. Then the inductive hypothesis (of the inner recursion) applied to $y < z^R$ yields $x \prec z^R$ for any R. Moreover, by definition $x \preceq y$ implies that $x^L \prec y$ for any L, so by the inductive hypothesis of the outer recursion we have $x^L \prec z$. Thus, $x \preceq z$.
- Suppose $x \preceq y$ and $y < z$. First, suppose $y < z$ arises from $y \leq z^{L_0}$. Then the inner inductive hypothesis applied to $y \leq z^{L_0}$ yields $x \preceq z^{L_0}$, hence $x \prec z$.

  Second, suppose $y < z$ arises from $y^{R_0} \leq z$. Then by definition, $x \preceq y$ implies $x \prec y^{R_0}$, and then the inner inductive hypothesis for $y^{R_0} \leq z$ yields $x \prec z$.
- Suppose $x \prec y$ and $y \leq z$, the latter arising from $y^L < z$ and $y < z^R$ for all L and R. By definition, $x \prec y$ implies there merely exists R_0 with $x^{R_0} \preceq y$ or L_0 with $x \preceq y^{L_0}$. If $x^{R_0} \preceq y$, then the outer inductive hypothesis yields $x^{R_0} \preceq z$, hence $x \prec z$. If $x \preceq y^{L_0}$, then the inner inductive hypothesis for $y^{L_0} < z$ (which holds by the constructor of $y \leq z$) yields $x \prec z$.

This completes the inner induction. Thus, for any x defined by a cut, we have $(x \prec -)$ and $(x \preceq -)$ defined by (11.6.13) and (11.6.14), and transitive on the right.

To complete the outer recursion, we need to verify these definitions are transitive on the left. After a No-induction on z, we end up with three cases that are essentially identical to those just described above for transitivity on the right. Hence, we omit them. □

Theorem 11.6.15. *For any x, y : No we have $(x < y) = (x \prec y)$ and $(x \leq y) = (x \preceq y)$.*

Proof. From left to right, we use $(\mathsf{No}, \leq, <)$-induction where $A(x) :\equiv \mathbf{1}$, with $\preceq$ and $\prec$ supplying the relations $\leqslant$ and $\lhd$. In all the constructor cases, x and y are defined by cuts, so the definitions of $\preceq$ and $\prec$ evaluate, and the inductive hypotheses apply.

From right to left, we use No-induction to assume that x and y are defined by cuts. But now the definitions of $\preceq$ and $\prec$, and the inductive hypotheses, supply exactly the data required for the relevant constructors of $\leq$ and $<$. □

Corollary 11.6.16. *The relations $\leq$ and $<$ on No satisfy*

$$\forall(x, y : \mathsf{No}).\, (x < y) \to (x \leq y)$$

and are transitive:

$$\begin{aligned}(x \leq y) &\to (y \leq z) \to (x \leq z)\\ (x \leq y) &\to (y < z) \to (x < z)\\ (x < y) &\to (y \leq z) \to (x < z).\end{aligned}$$

As with the Cauchy reals, the joint $(\mathsf{No}, \leq, <)$-recursion principle remains essential when defining all operations on No.

Example 11.6.17. We define $+ : \mathsf{No} \to \mathsf{No} \to \mathsf{No}$ by a double recursion. For the outer recursion, we take the codomain to be the subset of $\mathsf{No} \to \mathsf{No}$ consisting of functions g such that $(x < y) \to (g(x) < g(x))$ and $(x \leq y) \to (g(x) \leq g(y))$ for all x, y. For such g, h we define $(g \leqslant h) :\equiv \forall(x : \mathsf{No}).\, g(x) \leq h(x)$ and $(g \lhd h) :\equiv \forall(x : \mathsf{No}).\, g(x) < h(x)$. Clearly $\leqslant$ is antisymmetric.

For the primary clause of the recursion, we suppose x defined by a cut, and we define $(x + \_)$ by an inner recursion on No with codomain No, with relations $\sqsubseteq$ and $\sqsubset$ coinciding with $\leq$ and $<$. For the primary clause of the inner recursion, we suppose also y defined by a cut, and give Conway's definition:

$$x + y :\equiv \{\, x^L + y, x + y^L \mid x^R + y, x + y^R \,\}.$$

In other words, the left options of $x + y$ are all numbers of the form $x^L + y$ for some left option x^L, or $x + y^L$ for some left option y^L. The presence of the options $x^L + y$ and $x^R + y$ in $x + y$ implies directly that $x + y \in A'(y)$. Now we verify that this definition preserves inequality:

- If $y \leq z$ arises from knowing that $y^L < z$ and $y < z^R$ for all L and R, then the inner inductive hypothesis gives $x + y^L < x + z$ and $x + y < x + z^R$, while the outer inductive hypotheses give $x^L + y < x^L + z$ and $x^R + y < x^R + z$. And since each $x^L + z$ is by definition a left option of $x + z$, we have $x^L + z < x + z$, and similarly $x + y < x^R + y$. Thus, using transitivity, $x^L + y < x + z$ and $x + y < x^R + z$, and so we may conclude $x + y \leq x + z$ by the constructor of $\leq$.
- If $y < z$ arises from an L_0 with $y \leq z^{L_0}$, then inductively $x + y \leq x + z^{L_0}$, hence $x + y < x + z$ since $x + z^{L_0}$ is a right option of $x + z$.
- Similarly, if $y < z$ arises from $y^{R_0} \leq z$, then $x + y < x + z$ since $x + y^{R_0} \leq x + z$.

This completes the inner recursion. For the outer recursion, we have to verify that $+$ preserves inequality on the left as well. After an No-induction, this proceeds in exactly the same way.

In the Appendix to Part Zero of [Con76], Conway discusses how the surreal numbers may be formalized in ZFC set theory: by iterating along the ordinals and passing to sets of representatives of lowest rank for each equivalence class, or by representing numbers with "sign-expansions". He then remarks that

> The curiously complicated nature of these constructions tells us more about the nature of formalizations within ZF than about our system of numbers...

and goes on to advocate for a general theory of "permissible kinds of construction" which should include

(i) Objects may be created from earlier objects in any reasonably constructive fashion.

(ii) Equality among the created objects can be any desired equivalence relation.

Condition (i) can be naturally read as justifying general principles of *inductive definition*, such as those presented in §§5.6 and 5.7. In particular, the condition of strict positivity for constructors can be regarded as a formalization of what it means to be "reasonably constructive". Condition (ii) then suggests we should extend this to *higher* inductive definitions of all sorts, in which we can impose path-constructors making objects equal in any reasonable way. For instance, in the next paragraph Conway says:

> ...we could also, for instance, freely create a new object (x, y) and call it the ordered pair of x and y. We could also create an ordered pair $[x, y]$ different from (x, y) but co-existing with it...If instead we wanted to make (x, y) into an unordered pair, we could define equality by means of the equivalence relation $(x, y) = (z, t)$ if and only if $x = z, y = t$ *or* $x = t, y = z$.

The freedom to introduce new objects with new names, generated by certain forms of constructors, is precisely what we have in the theory of inductive definitions. Just as with our two copies of the natural numbers $\mathbb{N}$ and $\mathbb{N}'$ in §5.2, if we wrote down an identical definition to the cartesian product type $A \times B$, we would obtain a distinct product type $A \times' B$ whose canonical elements we could freely write as $[x, y]$. And we could make one of these a type of unordered pairs by adding a suitable path-constructor.

To be sure, Conway's point was not to complain about ZF in particular, but to argue against all foundational theories at once:

> ...this proposal is not of any particular theory as an alternative to ZF... What is proposed is instead that we give ourselves the freedom to create arbitrary mathematical theories of these kinds, but prove a metatheorem which ensures once and for all that any such theory could be formalised in terms of any of the standard foundational theories.

One might respond that, in fact, Univalent Foundations is not one of the "standard foundational theories" which Conway had in mind, but rather the *metatheory* in which we may express our ability to create new theories, and about which we may prove Conway's metatheorem. For instance, the surreal numbers are one of the "mathematical theories" Conway has in mind, and we have seen that they can be constructed and justified inside Univalent Foundations. Similarly, Conway remarked earlier that

> ...set theory would be such a theory, sets being constructed from earlier ones by processes corresponding to the usual axioms, and the equality relation being that of having the same members.

This description closely matches the higher-inductive construction of the cumulative hierarchy of set theory in §10.5. Conway's metatheorem would then correspond to the fact we have referred to several times that we can construct a model of Univalent Foundations inside ZFC (which is outside the scope of this book).

However, Univalent Foundations is so rich and powerful in its own right that it would be foolish to relegate it to only a metatheory in which to construct set-like theories. We have seen that even at the level of sets (0-types), the higher inductive types in Univalent Foundations yield direct constructions of objects by their universal properties (§6.11), such as a constructive theory of Cauchy completion (§11.3). But most importantly, the potential to model homotopy theory and category theory directly in the foundational system (Chapters 8 and 9) gives Univalent Foundations an advantage which no set-theoretic foundation can match.

Notes

In traditional constructive mathematics constructions of real numbers always seem to require certain compromises. For example, the Dedekind reals do not work well without powersets or some other form of impredicativity, while Cauchy reals work well in the presence of Countable Choice. It seems that with our construction of Cauchy reals as a higher inductive-inductive type we found a third possibility, which requires neither powersets nor Countable Choice.

Defining algebraic operations on Dedekind reals, especially multiplication, is somewhat tricky and/or tedious. Our definition works, but there are other approaches which may be preferable for technical reasons. For instance, Richman [Ric08] defines multiplication on the Dedekind reals first on the positive cuts and then extends it algebraically to all Dedekind cuts, while Conway [Con76] has observed that the definition of multiplication for surreal numbers works well for Dedekind reals.

The fact that $\mathbb{R}_c$ is the least Cauchy complete archimedean ordered field, as was proved in Theorem 11.3.48, indicates that our Cauchy reals coincide with those of [ES01].

The intricate relationship between various notions of compactness in a constructive setting is discussed in [BIS02]. Palmgren [Pal07] has a good comparison between pointwise analysis and pointfree topology.

...

The surreal numbers were defined by [Con76], using a sort of inductive definition but without justifying it explicitly in terms of any foundational system. For this reason, some later authors have tended to use sign-expansions or other more explicit presentations which can be coded more obviously into set theory. The idea of representing them in type theory was first considered by Hancock, while Setzer and Forsberg [FS12] noted that the surreals and their inequality relations $<$ and $\leq$ naturally form an inductive-inductive definition. The *higher* inductive-inductive version presented here, which builds in the correct notion of equality for surreals, is new.

Exercises

Exercise 11.1. Give an alternative definition of the Dedekind reals by first defining the square and then use Eq. (11.3.43). Check that one obtains an abelian ring.

Exercise 11.2. Suppose we remove the boundedness condition in Definition 11.2.1. Then we obtain the **extended reals** which contain $-\infty :\equiv (\emptyset, \mathbb{Q})$ and $\infty :\equiv (\mathbb{Q}, \emptyset)$. Which definitions of arithmetical operations on cuts still make sense for extended reals? What algebraic structure do we get?

Exercise 11.3. By considering one-sided cuts we obtain **lower** and **upper** Dedekind reals, respectively. For example, a lower real is given by a predicate $L : \mathbb{Q} \to \Omega$ which is

(i) *bounded:* $\exists(q : \mathbb{Q}).\, L(q)$ and
(ii) *rounded:* $L(q) = \exists(r : \mathbb{Q}).\, q < r \wedge L(r)$.

(We could also require $\exists(r : \mathbb{Q}).\, \neg L(r)$ to exclude the cut $\infty :\equiv \mathbb{Q}$.) Which arithmetical operations can you define on the lower reals? In particular, what happens with the additive inverse?

Exercise 11.4. Suppose we remove the locatedness condition in Definition 11.2.1. Then we obtain the **interval domain** $\mathbb{I}$ because cuts are allowed to have "gaps", which are just intervals. Define the partial order $\sqsubseteq$ on $\mathbb{I}$ by

$$((L, U) \sqsubseteq (L', U')) :\equiv (\forall(q : \mathbb{Q}).\, L(q) \Rightarrow L'(q)) \wedge (\forall(q : \mathbb{Q}).\, U(q) \Rightarrow U'(q)).$$

What are the maximal elements of $\mathbb{I}$ with respect to $\mathbb{I}$? Define the "endpoint" operations which assign to an element of the interval domain its lower and upper endpoints. Are the endpoints reals, lower reals, or upper reals (see Exercise 11.3)? Which definitions of arithmetical operations on cuts still make sense for the interval domain?

Exercise 11.5. Show that, for all $x, y : \mathbb{R}_{\mathsf{d}}$,

$$\neg(x < y) \Rightarrow y \leq x$$

and

$$(x \leq y) \simeq \prod_{\epsilon:\mathbb{Q}_+} x < y + \epsilon.$$

Does $\neg(x \leq y)$ imply $y < x$?

Exercise 11.6.

(i) Assuming excluded middle, construct a non-constant map $\mathbb{R}_{\mathsf{d}} \to \mathbb{Z}$.
(ii) Suppose $f : \mathbb{R}_{\mathsf{d}} \to \mathbb{Z}$ is a map such that $f(0) = 0$ and $f(x) \neq 0$ for all $x > 0$. Derive from this the limited principle of omniscience (11.5.8).

Exercise 11.7. Show that in an ordered field F density of $\mathbb{Q}$ and the traditional archimedean axiom are equivalent:

$$(\forall(x, y : F).\, x < y \Rightarrow \exists(q : \mathbb{Q}).\, x < q < y) \Leftrightarrow (\forall x : F \exists(k : \mathbb{Z}).\, x < z).$$

Exercise 11.8. Suppose $a, b : \mathbb{Q}$ and $f : \{\, q : \mathbb{Q} \mid a \leq q \leq b \,\} \to \mathbb{R}_c$ is Lipschitz with constant L. Show that there exists a unique extension $\bar{f} : [a, b] \to \mathbb{R}_c$ of f which is constant with Lipschitz constant L. Hint: rather than redoing Lemma 11.3.13 for closed intervals, observe that there is a retraction $r : \mathbb{R}_c \to [-n, n]$ and apply Lemma 11.3.13 to $f \circ r$.

Exercise 11.9. **Markov principle** says that for all $f : \mathbb{N} \to \mathbf{2}$,

$$(\neg\neg\exists(n : \mathbb{N}).\, f(n) = 1_{\mathbf{2}}) \Rightarrow \exists(n : \mathbb{N}).\, f(n) = 1_{\mathbf{2}}.$$

This is a particular instance of the law of double negation (3.4.2). Show that $\forall(x, y : \mathbb{R}_d).\, x \neq y \Rightarrow x \# y$ implies Markov principle. Does the converse holds as well?

Exercise 11.10. Verify that the following "no zero divisors" property holds for the real numbers: $xy \# 0 \Leftrightarrow x \# 0 \wedge y \# 0$.

Exercise 11.11. Suppose $(q_1, r_1), \ldots, (q_n, r_n)$ pointwise cover (a, b). Then there is $\epsilon : \mathbb{Q}_+$ such that whenever $a < x < y < b$ and $|x - y| < \epsilon$ then there merely exists i such that $q_i < x < r_i$ and $q_i < y < r_i$. Such an ϵ is called a **Lebesgue number** for the given cover.

Exercise 11.12. Prove the following approximate version of the mean value theorem:

> *If $f : [0, 1] \to \mathbb{R}$ is uniformly continuous and $f(0) < 0 < f(1)$ then for every $\epsilon : \mathbb{Q}_+$ there merely exists $x : [0, 1]$ such that $|f(x)| < \epsilon$.*

Hint: do not try to use the bisection method because it leads to the axiom of choice. Instead, approximate f with a piece-wise linear map. How do you construct a piece-wise linear map?

Exercise 11.13. Check whether everything in [Knu74] can be done using the higher inductive-inductive surreals of §11.6.

Appendix

Appendix

Formal type theory

Just as one can develop mathematics in set theory without explicitly using the axioms of Zermelo–Fraenkel set theory, in this book we have developed mathematics in univalent foundations without explicitly referring to a formal system of homotopy type theory. Nevertheless, it is important to *have* a precise description of homotopy type theory as a formal system in order to, for example,

- state and prove its metatheoretic properties, including logical consistency,
- construct models, e.g. in simplicial sets, model categories, higher toposes, etc., and
- implement it in proof assistants like Coq or Agda.

Even the logical consistency of homotopy type theory, namely that in the empty context there is no term $a : \mathbf{0}$, is not obvious: if we had erroneously chosen a definition of equivalence for which $\mathbf{0} \simeq \mathbf{1}$, then univalence would imply that $\mathbf{0}$ has an element, since $\mathbf{1}$ does. Nor is it obvious that, for example, our definition of S^1 as a higher inductive type yields a type which behaves like the ordinary circle.

There are two aspects of type theory which we must pin down before addressing such questions. Recall from the Introduction that type theory comprises a set of rules specifying when the judgments $a : A$ and $a \equiv a' : A$ hold—for example, products are characterized by the rule that whenever $a : A$ and $b : B$, $(a, b) : A \times B$. To make this precise, we must first define precisely the syntax of terms—the objects $a, a', A, \ldots$ which these judgments relate; then, we must define precisely the judgments and their rules of inference—the manner in which judgments can be derived from other judgments.

In this appendix, we present two formulations of Martin-Löf type theory, and of the extensions that constitute homotopy type theory. The first presentation (Appendix A.1) describes the syntax of terms and the forms of judgments as an extension of the untyped λ-calculus, while leaving the rules of inference informal. The second (Appendix A.2) defines the terms, judgments, and rules of inference inductively in the style of natural deduction, as is customary in much type-theoretic literature.

Preliminaries

In Chapter 1, we presented the two basic **judgments** of type theory. The first, $a : A$, asserts that a term a has type A. The second, $a \equiv b : A$, states that the two terms a and b are **judgmentally equal** at type A. These judgments are inductively defined by a set of inference rules described in Appendix A.2.

To construct an element a of a type A is to derive $a : A$; in the book, we give informal arguments which describe the construction of a, but formally, one must specify a precise term a and a full derivation that $a : A$.

However, the main difference between the presentation of type theory in the book and in this appendix is that here judgments are explicitly formulated in an ambient **context**, or list of assumptions, of the form

$$x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n.$$

An element $x_i : A_i$ of the context expresses the assumption that the variable x_i has type A_i. The variables $x_1, \ldots, x_n$ appearing in the context must be distinct. We abbreviate contexts with the letters Γ and Δ.

The judgment $a : A$ in context Γ is written

$$\Gamma \vdash a : A$$

and means that $a : A$ under the assumptions listed in Γ. When the list of assumptions is empty, we write simply

$$\vdash a : A$$

or

$$\cdot \vdash a : A$$

where $\cdot$ denotes the empty context. The same applies to the equality judgment

$$\Gamma \vdash a \equiv b : A$$

However, such judgments are sensible only for **well-formed** contexts, a notion captured our third and final judgment

$$\vdash (x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n)\ \mathsf{ctx}$$

expressing that each A_i is a valid type in the context $x_1 : A_1, x_2 : A_2, \ldots, x_{i-1} : A_{i-1}$. Therefore, if $\Gamma \vdash a : A$ and $\vdash \Gamma\ \mathsf{ctx}$, then we know that each A_i contains only the variables $x_1, \ldots, x_{i-1}$, and that a and A contain only the variables $x_1, \ldots, x_n$.

In informal mathematical presentations, the context is implicit. At each point in a proof, the mathematician knows which variables are available and what types they have, either by historical convention (n is usually a number, f is a function, etc.) or because variables are explicitly introduced with sentences such as "let x be a real number". We discuss some benefits of using explicit contexts in Appendices A.2.4 and A.2.5.

We write $B[a/x]$ for the **substitution** of a term a for free occurrences of the variable x in the term B, with possible capture-avoiding renaming of bound variables, as discussed in §1.2. The general form of substitution

$$B[a_1, \ldots, a_n/x_1, \ldots, x_n]$$

substitutes expressions $a_1, \ldots, a_n$ for the variables $x_1, \ldots, x_n$ simultaneously.

To **bind a variable** x **in an expression** B means to incorporate both of them into a larger expression, called an **abstraction**, whose purpose is to express the fact that x is "local" to B, i.e., it is not to be confused with other occurrences of x appearing elsewhere. Bound variables are familiar to programmers, but less so to mathematicians. Various notations are used for binding, such as $x \mapsto B$, $\lambda x.\, B$, and $x\,.\,B$, depending on the situation. We may write $C[a]$ for the substitution of a term a for the variable in the abstracted expression, i.e., we may define $(x.B)[a]$ to be $B[a/x]$. As discussed in §1.2, changing the name of a bound variable everywhere within an expression does not change the expression. Thus, to be very precise, an expression is an equivalence class of syntactic forms which differ in names of bound variables.

One may also regard each variable x_i of a judgment

$$x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \vdash a : A$$

to be bound in its **scope**, consisting of the expressions $A_{i+1}, \ldots, A_n$, a, and A.

A.1 The first presentation

The objects and types of our type theory may be written as terms using the following syntax, which is an extension of λ-calculus with *variables* $x, x', \ldots$, *primitive constants* $c, c', \ldots$, *defined constants* $f, f', \ldots$, and term forming operations

$$t ::= x \mid \lambda x.\, t \mid t(t') \mid c \mid f$$

The notation used here means that a term t is either a variable x, or it has the form $\lambda x.\, t$ where x is a variable and t is a term, or it has the form $t(t')$ where t and t' are terms, or it is a primitive constant c, or it is a defined constant f. The syntactic markers 'λ', '(', ')', and '.' are punctuation for guiding the human eye.

We use $t(t_1, \ldots, t_n)$ as an abbreviation for the repeated application $t(t_1)(t_2)\ldots(t_n)$. We may also use *infix* notation, writing $t_1 \star t_2$ for $\star(t_1, t_2)$ when $\star$ is a primitive or defined constant.

Each defined constant has zero, one or more **defining equations**. There are two kinds of defined constant. An *explicit* defined constant f has a single defining equation

$$f(x_1, \ldots, x_n) :\equiv t,$$

where t does not involve f. For example, we might introduce the explicit defined constant $\circ$ with defining equation

$$\circ(x, y)(z) :\equiv x(y(z)),$$

and use infix notation $x \circ y$ for $\circ(x, y)$. This of course is just composition of functions.

The second kind of defined constant is used in connection with a form of type having some primitive constants that are used to introduce elements into types of that form. With each such primitive constant c there is a defining equation of the form

$$f(x_1, \ldots, x_n, c(y_1, \ldots, y_m)) :\equiv t,$$

where f may occur in t, but now only in such a way that, in the context where f is introduced it will be a totally defined typed function. The paradigm examples of such defined functions are the functions defined by primitive recursion on the natural numbers. We may call this kind of definition of a function a *total recursive definition*. In computer science and logic this kind of definition of a function on a recursive data type has been called a **definition by structural recursion**.

Convertibility $t \downarrow t'$ between terms t and t' is the equivalence relation generated by the defining equations for constants, the **β-conversion**

$$(\lambda x.\, t)(u) :\equiv t[u/x],$$

and the rules which make it a *congruence* with respect to application and λ-abstraction:

- if $t \sim t'$ and $s \sim s'$ then $t(s) \sim t'(s')$, and
- if $t \sim t'$ then $(\lambda x.\, t) \sim (\lambda x.\, t')$.

The equality judgment $t \equiv u : A$ is then derived by the following single rule:

- if $t : A$, $u : A$, and $t \downarrow u$, then $t \equiv u : A$.

Judgmental equality is an equivalence relation.

A.1.1 Type universes

We postulate a hierarchy of **universes** denoted by primitive constants

$$\mathcal{U}_0, \quad \mathcal{U}_1, \quad \mathcal{U}_2, \quad \ldots$$

The first two rules for universes say that they form a cumulative hierarchy of types:

- $\mathcal{U}_m : \mathcal{U}_n$ for $m < n$,
- if $A : \mathcal{U}_m$ and $m \leq n$, then $A : \mathcal{U}_n$,

and the third expresses the idea that an object of a universe can serve as a type and stand to the right of a colon in judgments:

- if $\Gamma \vdash A : \mathcal{U}_n$, and x is a new variable,[1] then $\vdash (\Gamma, x : A)$ ctx.

In the body of the book, an equality judgment $A \equiv B : \mathcal{U}_n$ between types A and B is usually abbreviated to $A \equiv B$. This is an instance of typical ambiguity, as we can always switch to a larger universe, which however does not affect the validity of the judgment.

The following conversion rule allows us to replace a type by one equal to it in a typing judgment:

- if $a : A$ and $A \equiv B$ then $a : B$.

[1] By "new" we mean that it does not appear in Γ or A.

A.1.2 Dependent function types (Π-types)

We introduce a primitive constant c_Π. An expression of the form $c_\Pi(A, \lambda x.\, B)$ is written as $\prod_{(x:A)} B$. Judgments concerning such expressions and expressions of the form $\lambda x.\, b$ are introduced by the following rules:

- if $\Gamma \vdash A : \mathcal{U}_n$ and $\Gamma, x : A \vdash B : \mathcal{U}_n$, then $\Gamma \vdash \prod_{(x:A)} B : \mathcal{U}_n$
- if $\Gamma, x : A \vdash b : B$ then $\Gamma \vdash (\lambda x.\, b) : (\prod_{(x:A)} B)$
- if $\Gamma \vdash g : \prod_{(x:A)} B$ and $\Gamma \vdash t : A$ then $\Gamma \vdash g(t) : B[t/x]$

If x does not occur freely in B, we abbreviate $\prod_{(x:A)} B$ as the non-dependent function type $A \to B$ and derive the following rule:

- if $\Gamma \vdash g : A \to B$ and $\Gamma \vdash t : A$ then $\Gamma \vdash g(t) : B$

Using non-dependent function types and leaving implicit the context Γ, the rules above can be written in the following alternative style that we use in the rest of this section of the appendix.

- if $A : \mathcal{U}_n$ and $B : A \to \mathcal{U}_n$, then $\prod_{(x:A)} B(x) : \mathcal{U}_n$
- if $x : A \vdash b : B$ then $\lambda x.\, b : \prod_{(x:A)} B(x)$
- if $g : \prod_{(x:A)} B(x)$ and $t : A$ then $g(t) : B(t)$

A.1.3 Dependent pair types (Σ-types)

We introduce primitive constants c_Σ and c_{pair}. An expression of the form $c_\Sigma(A, \lambda a.\, B)$ is written as $\sum_{(a:A)} B$, and an expression of the form $c_{\mathsf{pair}}(a, b)$ is written as (a, b). We write $A \times B$ instead of $\sum_{(x:A)} B$ if x is not free in B.

Judgments concerning such expressions are introduced by the following rules:

- if $A : \mathcal{U}_n$ and $B : A \to \mathcal{U}_n$, then $\sum_{(x:A)} B(x) : \mathcal{U}_n$
- if, in addition, $a : A$ and $b : B(a)$, then $(a, b) : \sum_{(x:A)} B(x)$

If we have A and B as above, $C : \sum_{(x:A)} B(x) \to \mathcal{U}_m$, and

$$d : \prod_{(x:A)} \prod_{(y:B(x))} C((x, y))$$

we can introduce a defined constant

$$f : \prod_{(p:\sum_{(x:A)} B(x))} C(p)$$

with the defining equation

$$f((x, y)) :\equiv d(x, y).$$

C, d, x, y may contain extra implicit parameters $x_1, \ldots, x_n$ if they were obtained in some non-empty context; therefore, the fully explicit recursion schema is

$$f(x_1, \ldots, x_n, (x(x_1, \ldots, x_n), y(x_1, \ldots, x_n))) :\equiv d(x_1, \ldots, x_n, (x(x_1, \ldots, x_n), y(x_1, \ldots, x_n)))$$

A.1.4 Coproduct types

We introduce primitive constants c_+, c_{inl}, and c_{inr}. We write $A + B$ instead of $c_+(A, B)$, $\mathsf{inl}(a)$ instead of $c_{\mathsf{inl}}(a)$, and $\mathsf{inr}(a)$ instead of $c_{\mathsf{inr}}(a)$:

- if $A, B : \mathcal{U}_n$ then $A + B : \mathcal{U}_n$
- moreover, $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$

If we have A and B as above, $C : A + B \to \mathcal{U}_m$, $d : \prod_{(x:A)} C(\mathsf{inl}(x))$, and $d' : \prod_{(y:B)} C(\mathsf{inr}(y))$, then we can introduce a defined constant $f : \prod_{(z:A+B)} C(z)$ with the defining equations

$$f(\mathsf{inl}(x)) :\equiv d(x)$$
$$f(\mathsf{inr}(y)) :\equiv d'(y)$$

A.1.5 The finite types

We introduce primitive constants $\star$, $\mathbf{0}$, $\mathbf{1}$, satisfying the following rules:

- $\mathbf{0} : \mathcal{U}_0$, $\mathbf{1} : \mathcal{U}_0$
- $\star : \mathbf{1}$

Given $C : \mathbf{0} \to \mathcal{U}_n$ we can introduce a defined constant $f : \prod_{(x:\mathbf{0})} C(x)$, with no defining equations.

Given $C : \mathbf{1} \to \mathcal{U}_n$ and $d : C(\star)$ we can introduce a defined constant $f : \prod_{(x:\mathbf{1})} C(x)$, with defining equation $f(\star) :\equiv d$.

A.1.6 Natural numbers

The type of natural numbers is obtained by introducing primitive constants $\mathbb{N}$, 0, and succ with the following rules:

- $\mathbb{N} : \mathcal{U}_0$,
- $0 : \mathbb{N}$,
- $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$.

Furthermore, we can define functions by primitive recursion. If we have $C : \mathbb{N} \to \mathcal{U}_k$ we can introduce a defined constant $f : \prod_{(x:\mathbb{N})} C(x)$ whenever we have

$$d : C(0)$$
$$e : \prod_{(x:\mathbb{N})} (C(x) \to C(\mathsf{succ}(x)))$$

with the defining equations

$$f(0) :\equiv d$$
$$f(\mathsf{succ}(x)) :\equiv e(x, f(x))$$

A.1.7 *W*-types

For W-types we introduce primitive constants c_{W} and c_{sup}. An expression of the form $c_{\mathsf{W}}(A, \lambda x.\, B)$ is written as $\mathsf{W}_{(x:A)}B$, and an expression of the form $c_{\mathsf{sup}}(x, u)$ is written as $\mathsf{sup}(x, u)$:

- if $A : \mathcal{U}_n$ and $B : A \to \mathcal{U}_n$, then $\mathsf{W}_{(x:A)}B(x) : \mathcal{U}_n$
- if moreover, $a : A$ and $g : B(a) \to \mathsf{W}_{(x:A)}B(x)$ then $\mathsf{sup}(a, g) : \mathsf{W}_{(x:A)}B(x)$.

Here also we can define functions by total recursion. If we have A and B as above and $C : \mathsf{W}_{(x:A)}B(x) \to \mathcal{U}_m$, then we can introduce a defined constant $f : \prod_{(z:\mathsf{W}_{(x:A)}B(x))} C(z)$ whenever we have

$$d : \prod_{(x:A)}\prod_{(u:B(x)\to\mathsf{W}_{(x:A)}B(x))}((\prod_{(y:B(x))}C(u(y))) \to C(\mathsf{sup}(x, u)))$$

with the defining equation

$$f(\mathsf{sup}(x, u)) :\equiv d(x, u, f \circ u).$$

A.1.8 Identity types

We introduce primitive constants $c_{=}$ and c_{refl}. We write $a =_A b$ for $c_{=}(A, a, b)$ and refl_a for $c_{\mathsf{refl}}(A, a)$, when $a : A$ is understood:

- If $A : \mathcal{U}_n$, $a : A$, and $b : A$ then $a =_A b : \mathcal{U}_n$.
- If $a : A$ then $\mathsf{refl}_a : a =_A a$.

Given $a : A$, if $y : A, z : a =_A y \vdash C : \mathcal{U}_m$ and $\vdash d : C[a, \mathsf{refl}_a/y, z]$ then we can introduce a defined constant

$$f : \prod_{(y:A)}\prod_{(z:a=_A y)} C$$

with defining equation

$$f(a, \mathsf{refl}_a) :\equiv d.$$

A.2 The second presentation

In this section, there are three kinds of judgments

$$\vdash \Gamma\ \mathsf{ctx} \qquad\qquad \Gamma \vdash a : A \qquad\qquad \Gamma \vdash a \equiv a' : A$$

which we specify by providing inference rules for deriving them. A typical **inference rule** has the form

$$\frac{\mathcal{J}_1 \qquad \cdots \qquad \mathcal{J}_k}{\mathcal{J}}\ \textsc{Name}$$

It says that we may derive the **conclusion** $\mathcal{J}$, provided that we have already derived the **hypotheses** $\mathcal{J}_1, \ldots, \mathcal{J}_k$. On the right we write the NAME of the rule, and there may be extra side conditions that need to be checked before the rule is applicable.

A **derivation** of a judgment is a tree constructed from such inference rules, with the judgment at the root of the tree. For example, with the rules given below, the following is a derivation of $\cdot \vdash \lambda x.\, x : \mathbf{1} \to \mathbf{1}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \cdot\ \mathsf{ctx}}\ \textsc{ctx-emp}}{\vdash \mathbf{1} : \mathcal{U}_0}\ \mathbf{1}\textsc{-form}}{\vdash x{:}\mathbf{1}\ \mathsf{ctx}}\ \textsc{ctx-ext}}{x{:}\mathbf{1} \vdash x : \mathbf{1}}\ \mathsf{Vble}}{\cdot \vdash \lambda x.\, x : \mathbf{1} \to \mathbf{1}}\ \Pi\textsc{-intro}$$

A.2.1 Contexts

A context is a list

$$x_1{:}A_1, x_2{:}A_2, \ldots, x_n{:}A_n$$

which indicates that the distinct variables $x_1, \ldots, x_n$ are assumed to have types $A_1, \ldots, A_n$, respectively. The list may be empty. We abbreviate contexts with the letters Γ and Δ, and we may juxtapose them to form larger contexts.

The judgment $\vdash \Gamma\ \mathsf{ctx}$ formally expresses the fact that Γ is well-formed context, and is governed by the rules of inference

$$\frac{}{\vdash \cdot\ \mathsf{ctx}}\ \textsc{ctx-emp} \qquad\qquad \frac{x_1{:}A_1, \ldots, x_{n-1}{:}A_{n-1} \vdash A_n : \mathcal{U}_i}{\vdash (x_1{:}A_1, \ldots, x_n{:}A_n)\ \mathsf{ctx}}\ \textsc{ctx-ext}$$

with a side condition for the second rule: x_n must be distinct from $x_1, \ldots, x_{n-1}$.

A.2.2 Structural rules

The fact that the context holds assumptions is expressed by the rule which says that we may derive those typing judgments which are listed in the context:

$$\frac{\vdash (x_1{:}A_1, \ldots, x_n{:}A_n)\ \mathsf{ctx}}{x_1{:}A_1, \ldots, x_n{:}A_n \vdash x_i : A_i}\ \mathsf{Vble}$$

The following important principles, called **substitution** and **weakening**, need not be explicitly assumed. Rather, it is possible to show, by induction on the structure of all possible derivations, that whenever the hypotheses of these rules are derivable, their conclusion is also derivable.[2] For the typing judgments these principles are manifested as

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x{:}A, \Delta \vdash b : B}{\Gamma, \Delta[a/x] \vdash b[a/x] : B[a/x]}\ \mathsf{Subst}_1 \qquad\qquad \frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, \Delta \vdash b : B}{\Gamma, x{:}A, \Delta \vdash b : B}\ \mathsf{Wkg}_1$$

[2] Such rules are called **admissible**.

and for judgmental equalities they become

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x{:}A, \Delta \vdash b \equiv c : B}{\Gamma, \Delta[a/x] \vdash b[a/x] \equiv c[a/x] : B[a/x]}\ \mathsf{Subst}_2 \qquad \frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, \Delta \vdash b \equiv c : B}{\Gamma, x{:}A, \Delta \vdash b \equiv c : B}\ \mathsf{Wkg}_2$$

In addition to the judgmental equality rules given for each type former, we also assume that judgmental equality is an equivalence relation respected by typing.

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a \equiv a : A} \qquad \frac{\Gamma \vdash a \equiv b : A}{\Gamma \vdash b \equiv a : A} \qquad \frac{\Gamma \vdash a \equiv b : A \qquad \Gamma \vdash b \equiv c : A}{\Gamma \vdash a \equiv c : A}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash A \equiv B : \mathcal{U}_i}{\Gamma \vdash a : B} \qquad \frac{\Gamma \vdash a \equiv b : A \qquad \Gamma \vdash A \equiv B : \mathcal{U}_i}{\Gamma \vdash a \equiv b : B}$$

Additionally, for all the type formers below, we assume rules stating that each constructor preserves definitional equality in each of its arguments; for instance, along with the Π-INTRO rule, we assume the rule

$$\frac{\Gamma \vdash A \equiv A' : \mathcal{U}_i \qquad \Gamma, x{:}A \vdash B \equiv B' : \mathcal{U}_i \qquad \Gamma, x{:}A \vdash b \equiv b' : B}{\Gamma \vdash \lambda x.\, b \equiv \lambda x.\, b' : \prod_{(x:A)} B}\ \Pi\text{-INTRO-EQ}$$

However, we omit these rules for brevity.

A.2.3 Type universes

We postulate an infinite hierarchy of type universes

$$\mathcal{U}_0, \quad \mathcal{U}_1, \quad \mathcal{U}_2, \quad \ldots$$

Each universe is contained in the next, and any type in $\mathcal{U}_i$ is also in $\mathcal{U}_{i+1}$:

$$\frac{\vdash \Gamma\ \mathsf{ctx}}{\Gamma \vdash \mathcal{U}_i : \mathcal{U}_{i+1}}\ \mathcal{U}\text{-INTRO} \qquad \frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash A : \mathcal{U}_{i+1}}\ \mathcal{U}\text{-CUMUL}$$

We shall set up the rules of type theory in such a way that $\Gamma \vdash a : A$ implies $\Gamma \vdash A : \mathcal{U}_i$ for some i. In other words, if A plays the role of a type then it is in some universe. Another property of our type system is that $\Gamma \vdash a \equiv b : A$ implies $\Gamma \vdash a : A$ and $\Gamma \vdash b : A$.

A.2.4 Dependent function types (Π-types)

In §1.2, we introduced non-dependent functions $A \to B$ in order to define a family of types as a function $\lambda(x : A).\, B : A \to \mathcal{U}_i$, which then gives rise to a type of dependent functions $\prod_{(x:A)} B$. But with explicit contexts we may replace $\lambda(x : A).\, B : A \to \mathcal{U}_i$ with the judgment

$$x{:}A \vdash B : \mathcal{U}_i.$$

Consequently, we may define dependent functions directly, without reference to non-dependent ones. This way we follow the general principle that each type former, with its constants and rules, should be introduced independently of all other type formers. In fact, henceforth each type former is introduced systematically by:

- a **formation rule**, stating when the type former can be applied;
- some **introduction rules**, stating how to inhabit the type;
- **elimination rules**, or an induction principle, stating how to use an element of the type;
- **computation rules**, which are judgmental equalities explaining what happens when elimination and introduction rules are combined.

For the dependent function type these are:

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, x{:}A \vdash B : \mathcal{U}_i}{\Gamma \vdash \prod_{(x:A)} B : \mathcal{U}_i}\ \Pi\text{-FORM} \qquad \frac{\Gamma, x{:}A \vdash b : B}{\Gamma \vdash \lambda(x:A).\, b : \prod_{(x:A)} B}\ \Pi\text{-INTRO}$$

$$\frac{\Gamma \vdash f : \prod_{(x:A)} B \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B[a/x]}\ \Pi\text{-ELIM} \qquad \frac{\Gamma, x{:}A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda(x:A).\, b)(a) \equiv b[a/x] : B[a/x]}\ \Pi\text{-COMP}$$

The expression $\lambda(x:A).\, b$ binds free occurrences of x in b, as does $\prod_{(x:A)} B$ for B.

When x does not occur freely in B so that B does not depend on A, we obtain as a special case the ordinary function type $A \to B :\equiv \prod_{(x:A)} B$. We take this as the *definition* of $\to$.

We may abbreviate an expression $\lambda(x:A).\, b$ as $\lambda x.\, b$, with the understanding that the omitted type A should be filled in appropriately before typechecking.

A.2.5 Dependent pair types (Σ-types)

In §1.6, we needed $\to$ and $\prod$ types in order to define the introduction and elimination rules for $\sum$; as with $\prod$, contexts allow us to state the rules for $\sum$ independently:

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, x{:}A \vdash B : \mathcal{U}_i}{\Gamma \vdash \sum_{(x:A)} B : \mathcal{U}_i}\ \Sigma\text{-FORM}$$

$$\frac{\Gamma, x{:}A \vdash B : \mathcal{U}_i \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B[a/x]}{\Gamma \vdash (a, b) : \sum_{(x:A)} B}\ \Sigma\text{-INTRO}$$

$$\frac{\Gamma, z{:}\sum_{(x:A)} B \vdash C : \mathcal{U}_i \qquad \Gamma, x{:}A, y{:}B \vdash g : C[(x,y)/z] \qquad \Gamma \vdash p : \sum_{(x:A)} B}{\Gamma \vdash \mathsf{ind}_{\sum_{(x:A)} B}(z.C, x.y.g, p) : C[p/z]}\ \Sigma\text{-ELIM}$$

$$\frac{\begin{array}{c}\Gamma, z{:}\sum_{(x:A)} B \vdash C : \mathcal{U}_i \qquad \Gamma, x{:}A, y{:}B \vdash g : C[(x,y)/z] \\ \Gamma \vdash a' : A \qquad \Gamma \vdash b' : B[a'/x]\end{array}}{\Gamma \vdash \mathsf{ind}_{\sum_{(x:A)} B}(y.C, x.y.g, (a', b')) \equiv g[a', b'/x, y] : C[(a', b')/z]}\ \Sigma\text{-COMP}$$

The expression $\sum_{(x:A)} B$ binds free occurrences of x in B. Furthermore, because $\mathsf{ind}_{\sum_{(x:A)} B}$ has some arguments with free variables beyond those in Γ, we bind (following the variable names above) y in C, and a and b in g. These bindings are written as $y.C$ and $x.y.g$, to indicate the names of the bound variables. In particular, we treat $\mathsf{ind}_{\sum_{(x:A)} B}$ as a primitive, two of whose arguments contain binders; this is superficially similar to, but different from, $\mathsf{ind}_{\sum_{(x:A)} B}$ being a function that takes functions as arguments.

When B does not contain free occurrences of x, we obtain as a special case the cartesian product $A \times B :\equiv \sum_{(x:A)} B$. We take this as the *definition* of the cartesian product.

A.2.6 Coproduct types

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i}{\Gamma \vdash A + B : \mathcal{U}_i}\ +\text{-FORM}$$

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{inl}(a) : A + B}\ +\text{-INTRO}_1 \qquad \frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathsf{inr}(b) : A + B}\ +\text{-INTRO}_2$$

$$\frac{\begin{array}{c}\Gamma, z{:}(A+B) \vdash C : \mathcal{U}_i \\ \Gamma, x{:}A \vdash c : C[\mathsf{inl}(x)/z] \qquad \Gamma, y{:}B \vdash d : C[\mathsf{inr}(y)/z] \\ \Gamma \vdash e : A + B\end{array}}{\Gamma \vdash \mathsf{ind}_{A+B}(z.C, x.c, y.d, e) : C[e/z]}\ +\text{-ELIM}$$

$$\frac{\Gamma, z{:}(A+B) \vdash C : \mathcal{U}_i \qquad \Gamma, x{:}A \vdash c : C[\mathsf{inl}(x)/z] \qquad \Gamma, y{:}B \vdash d : C[\mathsf{inr}(y)/z] \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{ind}_{A+B}(z.C, x.c, y.d, \mathsf{inl}(a)) \equiv c[a/x] : C[\mathsf{inl}(a)/z]}\ +\text{-COMP}_1$$

$$\frac{\Gamma, z{:}(A+B) \vdash C : \mathcal{U}_i \qquad \Gamma, x{:}A \vdash c : C[\mathsf{inl}(x)/z] \qquad \Gamma, y{:}B \vdash d : C[\mathsf{inr}(y)/z] \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathsf{ind}_{A+B}(z.C, x.c, y.d, \mathsf{inr}(b)) \equiv d[b/y] : C[\mathsf{inr}(b)/z]}\ +\text{-COMP}_2$$

In ind_{A+B}, z is bound in C, x is bound in c, and y is bound in d.

A.2.7 The empty type 0

$$\frac{\vdash \Gamma\ \mathsf{ctx}}{\Gamma \vdash \mathbf{0} : \mathcal{U}_i}\ \mathbf{0}\text{-FORM} \qquad \frac{\Gamma, x{:}\mathbf{0} \vdash C : \mathcal{U}_i \qquad \Gamma \vdash a : \mathbf{0}}{\Gamma \vdash \mathsf{ind}_{\mathbf{0}}(x.C, a) : C[a/x]}\ \mathbf{0}\text{-ELIM}$$

In $\mathsf{ind}_{\mathbf{0}}$, x is bound in C. The empty type has no introduction and computation rules.

A.2.8 The unit type 1

$$\frac{\vdash \Gamma\ \mathsf{ctx}}{\Gamma \vdash \mathbf{1} : \mathcal{U}_i}\ \mathbf{1}\text{-FORM} \qquad \frac{\vdash \Gamma\ \mathsf{ctx}}{\Gamma \vdash \star : \mathbf{1}}\ \mathbf{1}\text{-INTRO}$$

$$\frac{\Gamma, x{:}\mathbf{1} \vdash C : \mathcal{U}_i \qquad \Gamma, y{:}\mathbf{1} \vdash c : C[y/x] \qquad \Gamma \vdash a : \mathbf{1}}{\Gamma \vdash \mathsf{ind}_{\mathbf{1}}(x.C, y.c, a) : C[a/x]}\ \mathbf{1}\text{-ELIM}$$

$$\frac{\Gamma, x{:}\mathbf{1} \vdash C : \mathcal{U}_i \qquad \Gamma, y{:}\mathbf{1} \vdash c : C[y/x]}{\Gamma \vdash \mathsf{ind}_{\mathbf{1}}(x.C, y.c, \star) \equiv c[\star/y] : C[\star/x]}\ \mathbf{1}\text{-COMP}$$

In $\mathsf{ind}_{\mathbf{1}}$, x is bound in C, and y is bound in c.

A.2.9 The natural number type

We give the rules for natural numbers, following §1.9.

$$\frac{\vdash \Gamma\ \mathsf{ctx}}{\Gamma \vdash \mathbb{N} : \mathcal{U}_i}\ \mathbb{N}\text{-FORM} \qquad \frac{\vdash \Gamma\ \mathsf{ctx}}{\Gamma \vdash 0 : \mathbb{N}}\ \mathbb{N}\text{-INTRO}_1 \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{succ}(n) : \mathbb{N}}\ \mathbb{N}\text{-INTRO}_2$$

$$\frac{\Gamma, x{:}\mathbb{N} \vdash C : \mathcal{U}_i \qquad \Gamma \vdash c_0 : C[0/x] \qquad \Gamma, x{:}\mathbb{N}, y{:}C \vdash c_s : C[\mathsf{succ}(x)/x] \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{ind}_{\mathbb{N}}(x.C, c_0, x.y.c_s, n) : C[n/x]}\ \mathbb{N}\text{-ELIM}$$

$$\frac{\Gamma, x{:}\mathbb{N} \vdash C : \mathcal{U}_i \qquad \Gamma \vdash c_0 : C[0/x] \qquad \Gamma, x{:}\mathbb{N}, y{:}C \vdash c_s : C[\mathsf{succ}(x)/x]}{\Gamma \vdash \mathsf{ind}_{\mathbb{N}}(x.C, c_0, x.y.c_s, 0) \equiv c_0 : C[0/x]}\ \mathbb{N}\text{-COMP}_1$$

$$\frac{\Gamma, x{:}\mathbb{N} \vdash C : \mathcal{U}_i \qquad \Gamma \vdash c_0 : C[0/x] \qquad \Gamma, x{:}\mathbb{N}, y{:}C \vdash c_s : C[\mathsf{succ}(x)/x] \qquad \Gamma \vdash n : \mathbb{N}}{\begin{array}{c}\Gamma \vdash \mathsf{ind}_{\mathbb{N}}(x.C, c_0, x.y.c_s, \mathsf{succ}(n)) \\ \equiv c_s[n, \mathsf{ind}_{\mathbb{N}}(x.C, c_0, x.y.c_s, n)/x, y] : C[\mathsf{succ}(n)/x]\end{array}}\ \mathbb{N}\text{-COMP}_2$$

In $\mathsf{ind}_{\mathbb{N}}$, x is bound in C, and x and y are bound in c_s.

Other inductively defined types follow the same general scheme.

A.2.10 Identity types

The presentation here corresponds to the (unbased) path induction principle for identity types in §1.12.

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash a =_A b : \mathcal{U}_i}\ =\text{-FORM} \qquad \frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}_a : a =_A a}\ =\text{-INTRO}$$

$$\frac{\begin{array}{c}\Gamma, x{:}A, y{:}A, p{:}x =_A y \vdash C : \mathcal{U}_i \\ \Gamma, z{:}A \vdash c : C[z, z, \mathsf{refl}_z/x, y, p] \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A \qquad \Gamma \vdash p' : a =_A b\end{array}}{\Gamma \vdash \mathsf{ind}'_{=_A}(x.y.p.C, z.c, a, b, p') : C[a, b, p'/x, y, p]}\ =\text{-ELIM}$$

$$\frac{\Gamma, x{:}A, y{:}A, p{:}x =_A y \vdash C : \mathcal{U}_i \qquad \Gamma, z{:}A \vdash c : C[z, z, \mathsf{refl}_z/x, y, p] \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{ind}'_{=_A}(x.y.p.C, z.c, a, a, \mathsf{refl}_a) \equiv c[a/z] : C[a, a, \mathsf{refl}_a/x, y, p]}\ =\text{-COMP}$$

In $\mathsf{ind}'_{=_A}$, x, y, and p are bound in C, and z is bound in c.

A.2.11 The η-conversion rule

We introduce the η-conversion rule for functions; see §1.2; it introduces a judgmental equality.

$$\frac{\Gamma \vdash f : \prod_{(x:A)} B}{\Gamma \vdash f \equiv (\lambda x.\, f(x)) : \prod_{(x:A)} B}\ \Pi\text{-}\eta$$

Various theorems in the text of the book, such as Corollary 2.7.3, are referred to as η-equivalence rules, but these are derivable propositional equalities, and so need not be asserted as part of the type theory.

A.2.12 Definitions

Although the rules we listed so far allows us to construct everything we need directly, we would still like to be able to use named constants, such as isequiv, as a matter of convenience. Informally, we can think of these constants simply as abbreviations, but the situation is a bit subtler in the formalization.

For example, consider function composition, which we takes $f : A \to B$ and $g : B \to C$ to $g \circ f : A \to C$. Somewhat unexpectedly, to make this work formally, $\circ$ must take as arguments not only f and g, but also their types A, B, C:

$$\circ :\equiv \lambda(A:\mathcal{U}_i).\, \lambda(B:\mathcal{U}_i).\, \lambda(C:\mathcal{U}_i).\, \lambda(g:B\to C).\, \lambda(f:A\to B).\, \lambda(x:A).\, g(f(x))$$

From a practical perspective, we do not want to annotate each application of $\circ$ with A, B and C, as the are usually quite easily guessed from surrounding information. We would like to simply write $g \circ f$. Then, strictly speaking, $g \circ f$ is not an abbreviation for $\lambda(x:A).\, g(f(x))$, because it involves additional **implicit arguments** which we want to suppress.

Inference of implicit arguments, typical ambiguity (§1.3), ensuring that symbols are only defined once, etc., are collectively called **elaboration**. Elaboration must take place prior to checking a derivation, and is thus not usually presented as part of the core type theory. However, it is essentially impossible to use any implementation of type theory which does not perform elaboration; see [Coq12, Nor07] for further discussion.

A.3 Homotopy type theory

In this section we state the additional axioms of homotopy type theory which distinguish it from standard Martin-Löf type theory: function extensionality, the univalence axiom, and higher inductive types. We state them in the style of the second presentation Appendix A.2, although the first presentation Appendix A.1 could be used just as well.

A.3.1 Axioms

There are two basic ways of introducing axioms such as function extensionality, univalence, or any other axiom which does not introduce new syntax or new judgmental equalities: we could either add a primitive constant to the theory which inhabits the axiom, or we could simply

prove all theorems in a context hypothesizing the presence of a variable that inhabits the axiom, as proposed in §1.1.

While these are essentially equivalent, we choose the former approach because these axioms of homotopy type theory are to be part of the core theory, always present.

We begin by recalling some auxiliary definitions.

- In Lemma 2.2.1, we defined the functorial action of $f : A \to B$ on a path $p : x =_A y$ as
$$\mathsf{ap}_f(p) :\equiv \mathsf{ind}'_{=_A}(x.y.p.f(x) =_B f(y), \lambda x.\, \mathsf{refl}_{f(x)}, x, y, p).$$
- In Exercise 1.1, we defined the composition of $g : B \to C$ after $f : A \to B$ as
$$g \circ f :\equiv \lambda x.\, g(f(x)).$$
- In Definition 2.4.1, we said $f, f' : A \to B$ are homotopic ($f \sim f'$) if
$$\prod_{x:A} f(x) =_B g(x).$$
- In Definition 4.2.1, we said $f : A \to B$ is a half-adjoint equivalence ($\mathsf{ishae}(f)$) when
$$\sum_{(g:B\to A)} \sum_{(\eta:g\circ f\sim \mathsf{id}_A)} \sum_{(\epsilon:f\circ g\sim \mathsf{id}_B)} \prod_{x:A} \mathsf{ap}_f(\eta x) = \epsilon(fx).$$
- As in §4.5, we define $\mathsf{isequiv}(f) :\equiv \mathsf{ishae}(f)$, and we say simply that f is an equivalence.
- We write $A \simeq B$ for the type of equivalences $\sum_{(f:A\to B)} \mathsf{isequiv}(f)$.
- In (2.9.2), we defined $\mathsf{happly}_{f,g} : (f = g) \to (f \sim g)$ by
$$\mathsf{happly}_{f,g}(p) :\equiv \mathsf{ind}'_{=_{\prod_{(x:A)} B}}(f'.g'.p'.\prod_{x:A} f'(x) =_{B(x)} g'(x), h'.\lambda x.\, \mathsf{refl}_{h'(x)}, f, g, p).$$

Finally, we assert:

Definition A.3.1 (Axiom of function extensionality (Axiom 2.9.3))**.** We introduce a constant, funext, which asserts that for each f, g, the term $\mathsf{happly}_{f,g}$ is an equivalence $(f = g) \simeq (f \sim g)$:
$$\frac{\Gamma \vdash f : \prod_{(x:A)} B \qquad \Gamma \vdash g : \prod_{(x:A)} B}{\Gamma \vdash \mathsf{funext}(f, g) : \mathsf{isequiv}(\mathsf{happly}_{f,g})}\ \Pi\text{-EXT}$$

- In (2.10.2), we defined the conversion from paths $A =_{\mathcal{U}_i} B$ to equivalences $A \simeq B$ as
$$\mathsf{idtoeqv}_{A,B} :\equiv \lambda p.\, \mathsf{ind}'_{=_{\mathcal{U}_i}}(A'.B'.p'.(A' \simeq B'), A'.(\lambda x.\, x, (\lambda x.\, x, (\lambda x.\, \mathsf{refl}_x, (\lambda x.\, \mathsf{refl}_x, \lambda x.\, \mathsf{refl}_{\mathsf{refl}_x}))))), A, B, p). \tag{A.3.2}$$

We assert:

Definition A.3.3 (Univalence axiom (Axiom 2.10.3))**.** We introduce a constant, $\mathsf{univalence}$, which asserts that for each A, B, the term $\mathsf{idtoeqv}_{A,B}$ is an equivalence $(A =_{\mathcal{U}_i} B) \simeq (A \simeq B)$:
$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i}{\Gamma \vdash \mathsf{univalence}(A, B) : \mathsf{isequiv}(\mathsf{idtoeqv}_{A,B})}\ \mathcal{U}_i\text{-UNIV}$$

A.3.2 The circle

Here we give an example of a basic higher inductive type; others follow the same general scheme, albeit with elaborations.

Note that the rules below do not precisely follow the pattern of the ordinary inductive types in Appendix A.2: the rules refer to the notions of transport and functoriality of maps (§2.2), and the second computation rule is a propositional, not judgmental, equality. These differences are discussed in §6.2.

$$\frac{\vdash \Gamma\ \mathsf{ctx}}{\Gamma \vdash \mathbb{S}^1 : \mathcal{U}_i}\ \mathbb{S}^1\text{-FORM} \qquad \frac{\vdash \Gamma\ \mathsf{ctx}}{\Gamma \vdash \mathsf{base} : \mathbb{S}^1}\ \mathbb{S}^1\text{-INTRO}_1 \qquad \frac{\vdash \Gamma\ \mathsf{ctx}}{\Gamma \vdash \mathsf{loop} : \mathsf{base} =_{\mathbb{S}^1} \mathsf{base}}\ \mathbb{S}^1\text{-INTRO}_2$$

$$\frac{\Gamma, x{:}\mathbb{S}^1 \vdash C : \mathcal{U}_i \qquad \Gamma \vdash b : C[\mathsf{base}/x] \qquad \Gamma \vdash \ell : b =^C_{\mathsf{loop}} b \qquad \Gamma \vdash p : \mathbb{S}^1}{\Gamma \vdash \mathsf{ind}_{\mathbb{S}^1}(x.C, b, \ell, p) : C[p/x]}\ \mathbb{S}^1\text{-ELIM}$$

$$\frac{\Gamma, x{:}\mathbb{S}^1 \vdash C : \mathcal{U}_i \qquad \Gamma \vdash b : C[\mathsf{base}/x] \qquad \Gamma \vdash \ell : b =^C_{\mathsf{loop}} b}{\Gamma \vdash \mathsf{ind}_{\mathbb{S}^1}(x.C, b, \ell, \mathsf{base}) \equiv b : C[\mathsf{base}/x]}\ \mathbb{S}^1\text{-COMP}_1$$

$$\frac{\Gamma, x{:}\mathbb{S}^1 \vdash C : \mathcal{U}_i \qquad \Gamma \vdash b : C[\mathsf{base}/x] \qquad \Gamma \vdash \ell : b =^C_{\mathsf{loop}} b}{\Gamma \vdash \mathbb{S}^1\text{-}\mathsf{loopcomp} : \mathsf{apd}_{(\lambda y.\, \mathsf{ind}_{\mathbb{S}^1}(x.C,b,\ell,y))}(\mathsf{loop}) = \ell}\ \mathbb{S}^1\text{-COMP}_2$$

In $\mathsf{ind}_{\mathbb{S}^1}$, x is bound in C. The notation $b =^C_{\mathsf{loop}} b$ for dependent paths was introduced in §6.2.

A.4 Basic metatheory

This section discusses the meta-theoretic properties of the type theory presented in Appendix A.1, and similar results hold for Appendix A.2. Figuring out which of these still hold when we add the features from Appendix A.3 quickly leads to open questions, as discussed at the end of this section.

Recall that Appendix A.1 defines the terms of type theory as an extension of the untyped λ-calculus. The λ-calculus is has its own notion of computation, namely, β-conversion:

$$(\lambda x.\, t)(u) :\equiv t[u/x]$$

This rule, together with the defining equations for the defined constants form *rewriting rules* that determine reduction steps for a rewriting system. These steps yield a notion of computation in the sense that each rule has a natural direction: one simplifies $(\lambda x.\, t)(u)$ by evaluating the function at its argument.

Moreover, this system is *confluent*, that is, if a simplifies in some number of steps to both a' and a'', there is some b to which both a' and a'' eventually simplify. Thus we can define $t \downarrow u$ to mean that t and u simplify to the same term.

(The situation is similar in Appendix A.2: Athough there we presented the computation rules as undirected equalities $\equiv$, we can give an operational semantics by saying that the application of an eliminator to an introductory form simplifies to its equal, not the other way around.)

It is straightforward to show that the system in Appendix A.1 has the following properties:

Theorem A.4.1. *If $A : \mathcal{U}$ and $A \downarrow A'$ then $A' : \mathcal{U}$. If $t : A$ and $t \downarrow t'$ then $t' : A$.*

We say that a term is **normalizable** (respectively, **strongly normalizable**) if some (respectively, every), sequence of rewriting steps from the term terminates.

Theorem A.4.2. *If $A : \mathcal{U}$ then A is strongly normalizable. If $t : A$ then A and t are strongly normalizable.*

We say that a term is in **normal form** if it cannot be further simplified, and that a term is **closed** if no variable occurs freely in it. A closed normal type has to be a primitive type, i.e., of the form $c(\vec{v})$ for some primitive constant c (where the list $\vec{v}$ of closed normal terms may be omitted if empty, for instance, as with $\mathbb{N}$). In fact, we can explicitly describe all normal forms:

Lemma A.4.3. *The terms in normal form can be described by the following syntax:*

$$v ::= k \mid \lambda x.\, v \mid c(\vec{v}) \mid f(\vec{v}),$$
$$k ::= x \mid k(v) \mid f(\vec{v})(k),$$

where $f(\vec{v})$ represents a partial application of the defined function f. In particular, a type in normal form is of the form k or $c(\vec{v})$.

Theorem A.4.4. *If A is in normal form then the judgment $A : \mathcal{U}$ is decidable. If $A : \mathcal{U}$ and t is in normal form then the judgment $t : A$ is decidable.*

Logical consistency (of the system in Appendix A.1) follows immediately: if we had $a : \mathbf{0}$ in the empty context, then by Theorems A.4.1 and A.4.2, a simplifies to a normal term $a' : \mathbf{0}$. But we can see by Lemma A.4.3 that no such term exists.

Corollary A.4.5. *The system in Appendix A.1 is logically consistent.*

Similarly, we have the *canonicity* property that if $a : \mathbb{N}$ in the empty context, then a simplifies to a normal term $\mathsf{succ}^k(0)$ for some numeral k.

Corollary A.4.6. *The system in Appendix A.1 has the canonicity property.*

Finally, if a, A are in normal form, it is *decidable* whether $a : A$; in other words, because type-checking amounts to verifying the correctness of a proof, this means we can always "recognize a correct proof when we see one."

Corollary A.4.7. *The property of being a proof in the system in Appendix A.1 is decidable.*

The above results do not apply to the extended system of homotopy type theory (i.e., the above system extended by Appendix A.3), since occurrences of the univalence axiom and constructors of higher inductive types never simplify, breaking Lemma A.4.3. It is an open question whether one can simplify applications of these constants in order to restore canonicity. We also do not have a schema describing all permissible higher inductive types, nor are we certain how to correctly formulate their rules (e.g., whether the computation rules on higher constructors should be judgmental equalities).

The consistency of Martin-Löf type theory extended with univalence and higher inductive types could be shown by inventing an appropriate normalization procedure, but currently the only proofs that these systems are consistent are via semantic models—for univalence, a model in Kan complexes due to Voevodsky [KLV12], and for higher inductive types, a model due to Lumsdaine and Shulman [LS13b].

Other metatheoretic issues, and a summary of our current results, are discussed in greater length in the "Constructivity" and "Open problems" sections of the introduction to this book.

Notes

The system of rules with introduction (primitive constants) and elimination and computation rules (defined constant) is inspired by Gentzen natural deduction. The possibility of strengthening the elimination rule for existential quantification was indicated in [How80]. The strengthening of the axioms for disjunction appears in [ML98], and for absurdity elimination and identity type in [ML75b]. The W-types were introduced in [ML82]. They generalize a notion of trees introduced by [Tai68].

The generalized form of primitive recursion for natural numbers and ordinals appear in [Hil26]. This motivated Gödel's system T, [Göd58], which was analyzed by [Tai67], who used, following [Göd58], the terminology "definitional equality" for conversion: two terms are *judgmentally equal* if they reduce to a common term by means of a sequence of applications of the reduction rules. This terminology was also used by de Bruijn [dB73] in his presentation of *AUTOMATH*.

Streicher [Str91, Theorem 4.13], explains how to give the semantics in contextual category of terms in normal form using a simple syntax similar to the one we have presented.

Our second presentation comprises fairly standard presentation of intensional Martin-Löf type theory, with some additional features needed in homotopy type theory. Compared to a reference presentation of [Hof97], the type theory of this book has a few non-critical differences:

- universes à la Russell, in the sense of [ML84]; and
- judgmental η and function extensionality for Π types;

and a few features essential for homotopy type theory:

- the univalence axiom; and
- higher inductive types.

As a matter of convenience, the book primarily defines functions by induction using definition by *pattern matching*. It is possible to formalize the notion of pattern matching, as done in Appendix A.1. However, the standard type-theoretic presentation, adopted in this section, is to introduce a single *dependent eliminator* for each type former, from which functions out of that type must be defined.[3] The two approaches are equivalent; see §1.10 for a longer discussion.

[3] This approach is easier to formalize both syntactically and semantically, as it amounts to the universal property of the type former.

Bibliography

[AB04] Steven Awodey and Andrej Bauer. Propositions as [types]. *J. Logic Comput.*, 14(4):447–471, 2004. 114, 201, 233

[Acz78] Peter Aczel. The type theoretic interpretation of constructive set theory. In *Logic Colloquium '77 (Proc. Conf., Wrocław, 1977)*, volume 96 of *Stud. Logic Foundations Math.*, pages 55–66. North-Holland, Amsterdam, 1978. 343, 344

[AG02] Peter Aczel and Nicola Gambino. Collection principles in dependent type theory. In *Types for proofs and programs*, pages 1–23. Springer, 2002. 114

[AGS12] Steve Awodey, Nicola Gambino, and Kristina Sojakova. Inductive types in homotopy type theory. To appear in LICS 2012; arXiv:1201.3898, 2012. 160

[AKS13] Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. Univalent categories and the rezk completion. arXiv:1303.0584, 2013. 315

[Alt99] Thorsten Altenkirch. Extensional equality in intensional type theory. In *IEEE Symposium on Logic in Computer Science*, 1999. 202

[AMS07] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *Programming Languages meets Program Verification Workshop*, 2007. 202

[AW09] Steve Awodey and Michael A. Warren. Homotopy theoretic models of identity types. *Math. Proc. Camb. Phil. Soc.*, 146:45–55, 2009. 4, 92

[BCH13] B. Barras, T. Coquand, and S. Huber. A generalization of Takeuti-Gandy interpretation. 2013. 11

[Bee85] M.J. Beeson. *Foundations of Constructive Mathematics*. Springer, 1985. 48

[Ber09] Julia E. Bergner. A survey of $(\infty, 1)$-categories. In John C. Baez and J. Peter May, editors, *Towards Higher Categories*, volume 152 of *The IMA Volumes in Mathematics and its Applications*, pages 69–83. Springer, 2009. arXiv:math.CT/0610239. 315

[Bis67] E. Bishop. *Foundations of constructive analysis*. McGraw-Hill Book Co., New York, 1967. 344, 374

[BIS02] Douglas Bridges, Hajime Ishihara, and Peter Schuster. Compactness and continuity, constructively revisited. In *Computer Science Logic*, pages 89–102. Springer, 2002. 391

[Bla79] Georges Blanc. Équivalence naturelle et formules logiques en théorie des catégories. *Arch. Math. Logik Grundlag.*, 19(3-4):131–137, 1978/79. 315

[Bou68] Nicolas Bourbaki. *Theory of Sets*. Hermann, Paris, 1968. 93

[BSP11] Clark Barwick and Christopher Schommer-Pries. On the unicity of the homotopy theory of higher categories. arXiv:1112.0040, 2011. 288

[Bun79] Marta Bunge. Stack completions and Morita equivalence for categories in a topos. *Cahiers Topologie Géom. Différentielle*, 20(4):401–436, 1979. 316

[CAB+86] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice Hall, 1986. 48, 49, 50, 114, 201

[Car95] Aurelio Carboni. Some free constructions in realizability and proof theory. *J. Pure Appl. Algebra*, 103:117–148, 1995. 202

[Chu33] Alonzo Church. A set of postulates for the foundation of logic 2. *Annals of Mathematics*, 34:839 – 864, 1933. 2

[Chu40] Alonzo Church. A formulation of of the simple theory of types. *Journal of Symbolic Logic*, 5:56 – 68, 1940. 2

[Chu41] Alonzo Church. *The Calculi of Lambda Conversation*. Princeton University Press, 1941. 2

[CM85] Robert L. Constable and N. P. Mendler. Recursive definitions in type theory. In Rohit Parikh, editor, *Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 61–78. Springer, 1985. 160

[Con76] J. H. Conway. *On numbers and games*. A K Peters Ltd., Natick, MA, 1976. 380, 390, 391

[Con85] Robert L. Constable. Constructive mathematics as a programming logic I: Some principles of theory. In *Annals of Mathematics*, volume 24, pages 21–37. Elsevier Science Publishers, B.V. (North-Holland), 1985. Reprinted from *Topics in the Theory of Computation*, Selected Papers of the Internationall Conference on Foundations of Computation Theory, FCT '83. 113

[Coq92] Thierry Coquand. The paradox of trees in type theory. *BIT Numerical Mathematics*, 32(1):10–14, 1992. 23

[Coq12] Coq Development Team. *The Coq Proof Assistant Reference Manual*. INRIA-Rocquencourt, 2012. 48, 49, 409

[CP90] Thierry Coquand and Christine Paulin. Inductively defined types. In *COLOG-88 (Tallinn, 1988)*, volume 416 of *Lecture Notes in Comp. Sci.*, pages 50–66. Springer, 1990. 160

[dB73] Nicolaas Govert de Bruijn. *AUTOMATH, a language for mathematics*. Les Presses de l'Université de Montréal, Montreal, Que., 1973. Séminaire de Mathématiques Supérieures, No. 52 (Été 1971). 48, 50, 413

[Dia75] Radu Diaconescu. Axiom of choice and complementation. *Proc. Amer. Math. Soc.*, 51:176–178, 1975. 344

[dPGM04] Valeria de Paiva, Rajeev Goré, and Michael Mendler. Modalities in constructive logics and type theories. *Journal of Logic and Computation*, 14(4):439–446, 2004. 233

[Dyb91] Peter Dybjer. Inductive sets and families in martin-lf̈'s type theory and their set-theoretic semantics. In Gerard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 280–30. 1991. 160

[Dyb00] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525–549, 2000. 160

[ES01] Martín Hötzel Escardó and Alex K. Simpson. A universal characterization of the closed euclidean interval. In *LICS*, pages 115–125. IEEE Computer Society, 2001. 391

[Fre76] Peter Freyd. Properties invariant within equivalence types of categories. In *Algebra, topology, and category theory (a collection of papers in honor of Samuel Eilenberg)*, pages 55–61. Academic Press, New York, 1976. 315

[FS12] Fredrik Nordvall Forsberg and Anton Setzer. A finite axiomatisation of inductive-inductive definitions. 2012. 391

[Gar09] Richard Garner. On the strength of dependent products in the type theory of Martin-Löf. *Ann. Pure Appl. Logic*, 160(1):1–12, 2009. 92

[Gen36] Gerhard Gentzen. Die widerspruchsfreiheit der reinen zahlentheorie. *Mathematische Annalen*, 112(1):493–565, 1936. 344

[Geo13] 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 Thery. A machine-checked proof of the odd order theorem. In *ITP2013*, 2013. 6

[Göd58] Kurt Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12:280–287, 1958. 413

[Hat02] Allen Hatcher. *Algebraic Topology*. Cambridge University Press, 2002. Available from http://www.math.cornell.edu/~hatcher/AT/ATpage.html. 283

[Hed98] Michael Hedberg. A coherence theorem for martin-löf's type theory. *Journal of Functional Programming*, 8(4):413–436, 1998. 233

[Hey66] A. Heyting. *Intuitionism: an introduction*. Studies in logic and the foundations of mathematics. North-Holland Pub. Co., 1966. 48

[Hil26] David Hilbert. Über das Unendliche. *Math. Ann.*, 95(1):161–190, 1926. 413

[Hof95] Martin Hofmann. *Extensional concepts in intensional type theory*. PhD thesis, University of Edinburgh, 1995. 202

[Hof97] Martin Hofmann. Syntax and semantics of dependent types. In *Semantics and logics of computation (Cambridge, 1995)*, volume 14 of *Publ. Newton Inst.*, pages 79–130. Cambridge Univ. Press, Cambridge, 1997. 413

[How80] William A. Howard. The formulae-as-types notion of construction. In J. Roger Seldin, Jonathan P.; Hindley, editor, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980. original paper manuscript from 1969. 48, 50, 93, 413

[HS98] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 83–111. Oxford Univ. Press, New York, 1998. 4, 92

[Hue80] Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, October 1980. 344

[JM95] A. Joyal and I. Moerdijk. *Algebraic set theory*, volume 220 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 1995. 344

[Joh02] Peter T. Johnstone. *Sketches of an Elephant: A Topos Theory Compendium: Volumes 1 and 2*. Number 43 in Oxford Logic Guides. Oxford Science Publications, 2002. 323

[JT91] André Joyal and Myles Tierney. Strong stacks and classifying spaces. In *Category theory (Como, 1990)*, volume 1488 of *Lecture Notes in Math.*, pages 213–236. Springer, Berlin, 1991. 316

[KECA13] Nicolai Kraus, Martın Escardó, Thierry Coquand, and Thorsten Altenkirch. Generalizations of hedberg's theorem. 2013. To appear in TLCA 2013. 114, 233

[KLN04] Fairouz Kamareddine, Twan Laan, and Rob Nederpelt. *A Modern Perspective on Type Theory: From its Origins until Today*. Number 29 in Applied Logic. Kluwer, 2004. 2

[KLV12] Chris Kapulkin, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. The simplicial model of univalent foundations. arXiv:1211.2851, 2012. 11, 92, 160, 413

[Knu74] Donald Ervin Knuth. *Surreal Numbers*. Addison-Wesley, 1974. 393

[Kom32] Andrej Komolgorov. Zur deutung der intuitionistischen logik. *Mathematische Zeitschrift*, 35:58–65, 1932. 8

[Law05] F. William Lawvere. An elementary theory of the category of sets (long version) with commentary. *Repr. Theory Appl. Categ.*, (11):1–35 (electronic), 2005. Reprinted and expanded from Proc. Nat. Acad. Sci. U.S.A. **52** (1964) [MR0172807], With comments by the author and Colin McLarty. 6, 327, 344

[Law06] F. William Lawvere. Adjointness in foundations. *Repr. Theory Appl. Categ.*, 16:1 – 16, 2006. Reprinted from Dialectica **23** (1969) [MR2223032]. 48, 160

[LH12] 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 '12, pages 337–348, New York, NY, USA, 2012. ACM. 10, 11, 92

[LS13a] Daniel R. Licata and Michael Shulman. Calculating the fundamental group of the circle in homotopy type theory. In *IEEE Symposium on Logic in Computer Science*, 2013. 93

[LS13b] Peter LeFanu Lumsdaine and Michael Shulman. Higher inductive types. In preparation, 2013. 11, 160, 201, 413

[Lum10] Peter LeFanu Lumsdaine. Weak omega-categories from intensional type theory. *Typed lambda calculi and applications*, 6:1–19, 2010. arXiv:0812.0409. 92

[Lur09] Jacob Lurie. *Higher topos theory*. Number 170 in Annals of Mathematics Studies. Princeton University Press, 2009. 10, 134, 233, 277

[Mak95] Michael Makkai. First order logic with dependent sorts, with applications to category theory. Available at `http://www.math.mcgill.ca/makkai/folds/`, 1995. 315

[Mak01] Michael Makkai. On comparing definitions of weak n-category. Available at `http://www.math.mcgill.ca/makkai/`, August 2001. 315

[ML71] Per Martin-Löf. Hauptsatz for the intuitionistic theory of iterated inductive definitions. In *Proceedings of the Second Scandinavian Logic Symposium (University of Oslo 1970)*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 179–216. North-Holland, 1971. 160

[ML75a] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73, Proceedings of the Logic Colloquium*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73 – 118. Elsevier, 1975. 2, 48, 160

[ML75b] Per Martin-Löf. An intuitionistic theory of types: predicative part. In *Logic Colloquium '73 (Bristol, 1973)*, pages 73–118. Studies in Logic and the Foundations of Mathematics, Vol. 80. North-Holland, Amsterdam, 1975. 48, 413

[ML82] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, methodology and philosophy of science, VI (Hannover, 1979)*, volume 104 of *Stud. Logic Found. Math.*, pages 153–175. North-Holland, Amsterdam, 1982. 2, 48, 160, 413

[ML84] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory. Lecture Notes*. Bibliopolis, Naples, 1984. 2, 48, 50, 413

[ML98] Per Martin-Löf. An intuitionistic theory of types. In *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 127–172. Oxford Univ. Press, New York, 1998. 2, 48, 50, 91, 413

[ML06] Per Martin-Löf. 100 years of zermelo's axiom of choice: what was the problem with it? *The Computer Journal*, 49(3):345–350, 2006. 114

[Mog89] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1989. 234

[MP00] Ieke Moerdijk and Erik Palmgren. Wellfounded trees in categories. In *Proceedings of the Workshop on Proof Theory and Complexity, PTAC'98 (Aarhus)*, volume 104, pages 189–218, 2000. 160

[MP02] Ieke Moerdijk and Erik Palmgren. Type theories, toposes and constructive set theory: predicative aspects of AST. *Ann. Pure Appl. Logic*, 114(1-3):155–201, 2002. 344

[MRR88] Ray Mines, Fred Richman, and Wim Ruitenburg. *A course in constructive algebra*. Springer-Verlag, 1988. 344

[MS05] Maria Emilia Maietti and Giovanni Sambin. Toward a minimalist foundation for constructive mathematics. *From Sets and Types to Topology and Analysis: Practicable Foundations for Constructive Mathematics*, 48:91–114, 2005. 114

[MvdB13] Ieke Moerdijk and Benno van den Berg. *W*-types in cartesian model categories. in preparation, 2013. 160

[Nor88] Bengt Nordström. Terminating general recursion. *BIT Numerical Mathematics*, 28(3):605–619, 1988. 344

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers, Göteborg University, 2007. 48, 49, 409

[Pal07] Erik Palmgren. A constructive and functorial embedding of locally compact metric spaces into locales. *Topology Appl.*, 154(9):1854–1880, 2007. 391

[Pal09] Erik Palmgren. Constructivist and structuralist foundations: Bishop's and Lawvere's theories of sets. `http://www.math.uu.se/~palmgren/cetcs.pdf`, 2009. 344

[Pau86] Lawrence C. Paulson. Constructing recursion operators in intuitionistic type theory. *J. Symb. Comput.*, 2(4):325–355, 1986. 344

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. 2

[PM93] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, 1993. LIP research report 92-49.

[PPM90] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In *Mathematical foundations of programming semantics (New Orleans, LA, 1989)*, number 442 in Lecture Notes in Comp. Sci., pages 209–228. Springer, 1990. 160

[PS89] Kent Petersson and Dan Synek. A set constructor for inductive sets in martin-löf's type theory. In *Category Theory and Computer Science*, pages 128–140, 1989. 160

[Rez01] Charles Rezk. A model for the homotopy theory of homotopy theory. *Trans. Amer. Math. Soc.*, 353(3):973–1007 (electronic), 2001. 315

[Rez05] Charles Rezk. Toposes and homotopy toposes. http://www.math.uiuc.edu/~rezk/homotopy-topos-sketch.pdf, 2005. 10, 134

[Ric08] Fred Richman. Real numbers and other completions. *MLQ Math. Log. Q.*, 54(1):98–108, 2008. 391

[RS13] Egbert Rijke and Bas Spitters. Sets in homotopy type theory. *Submitted*, 2013. 344

[Rus08] Bertand Russell. Mathematical logic based on the theory of types. *Amer. Journal of Math.*, 30:222–262, 1908. 2

[Sco70] Dana Scott. Constructive validity. In *the Symposium on Automatic Demonstration, Lecture Notes in Mathematics*, New York, 1970. Springer-Verlag. 48

[Som10] Giovanni Sommaruga. *History and Philosophy of Constructive Type Theory*. Number 290 in Synthese Library. Kluwer, 2010. 2

[Spi11] Arnaud Spiwack. *A Journey Exploring the Power and Limits of Dependent Type Theory*. PhD thesis, École Polytechnique, Palaiseau, France, 2011. 114

[SS12] Urs Schreiber and Michael Shulman. Quantum gauge field theory in cohesive homotopy type theory. *Quantum Physics and Logic*, 2012. 234

[Str91] Thomas Streicher. *Semantics of type theory*. Progress in Theoretical Computer Science. Birkhäuser Boston Inc., Boston, MA, 1991. Correctness, completeness and independence results, With a foreword by Martin Wirsing. 413

[Str93] Thomas Streicher. *Investigations Into Intensional Type Theory*. PhD thesis, LMU München, 1993. 49, 207

[Tai67] W. W. Tait. Intensional interpretations of functionals of finite type. I. *J. Symbolic Logic*, 32:198–212, 1967. 48, 413

[Tai68] W. W. Tait. Constructive reasoning. In *Logic, Methodology and Philos. Sci. III (Proc. Third Internat. Congr., Amsterdam, 1967)*, pages 185–199. North-Holland, Amsterdam, 1968. 48, 50, 413

[Tay96] Paul Taylor. Intuitionistic sets and ordinals. *J. Symbolic Logic*, 61(3):705–744, 1996. 344

[TV02] Bertrand Toën and Gabriele Vezzosi. Homotopical algebraic geometry I: Topos theory. arXiv:math/0207028, 2002. 10

[TvD88a] A. S. Troelstra and D. van Dalen. *Constructivism in mathematics. Vol. I*, volume 121 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., Amsterdam, 1988. An introduction. 8, 114

[TvD88b] A. S. Troelstra and D. van Dalen. *Constructivism in mathematics. Vol. II*, volume 123 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., Amsterdam, 1988. An introduction. 8, 114

[vdBG11] Benno van den Berg and Richard Garner. Types are weak ω-groupoids. *Proceedings of the London Mathematical Society*, 102(2):370–394, 2011. 92

[Voe06] Vladimir Voevodsky. A very short note on the homotopy λ-calculus. 2006. 4

[Voe12] Vladimir Voevodsky. A universe polymorphic type system. `http://uf-ias-2012.wikispaces.com/file/view/Universe+polymorphic+type+sytem.pdf`, 2012. 11, 114

[War08] Michael A. Warren. *Homotopy Theoretic Aspects of Constructive Type Theory*. PhD thesis, Carnegie Mellon University, 2008. 92

[Wik13] Wikipedia. Homotopy groups of spheres, April 2013. 241

[Wil10] Olov Wilander. Setoids and universes. *Math. Structures Comput. Sci.*, 20(4):563–576, 2010. 344

[WR27] Alfred North Whitehead and Bertrand Russell. *Principia mathematica, 3 vol.s*. Cambridge University Press, Cambridge, 1910 – 1913; Second edition, 1925 – 1927. 92

[WR62] Alfred North Whitehead and Bertrand Russell. *Principia mathematica to* *56. Cambridge University Press, New York, 1962. 114

Index of symbols

$x :\equiv a$	definition, p. 19
$a \equiv b$	judgmental equality, p. 19
$a =_A b$, $a = b$	identity type, p. 42
$\mathsf{Id}_A(a, b)$	identity type, p. 42
$\exists(x : A).\, B(x)$	logical notation for mere existential, p. 105
$\forall(x : A).\, B(x)$	logical notation for dependent function type, p. 105
$\mathsf{im}(f)$	image of map f, p. 224
$\mathsf{im}_n(f)$	n-image of map f, p. 224
$\mathsf{ind}_{\mathbf{0}}$	induction for $\mathbf{0}$, p. 31,
$\mathsf{ind}_{\mathbf{1}}$	induction for $\mathbf{1}$, p. 27,
$\mathsf{ind}_{\mathbf{2}}$	induction for $\mathbf{2}$, p. 32,
$\mathsf{ind}_{\mathbb{N}}$	induction for $\mathbb{N}$, p. 35, and
$\mathsf{ind}_{=_A}$	path induction for $=_A$, p. 45,
$\mathsf{ind}'_{=_A}$	based path induction for $=_A$, p. 44,
$\mathsf{ind}_{A\times B}$	induction for $A \times B$, p. 27,
$\mathsf{ind}_{\sum_{(x:A)} B(x)}$	induction for $\sum_{(x:A)} B$, p. 29,
ind_{A+B}	induction for $A + B$, p. 31,
$\mathsf{ind}_{\mathsf{W}_{(x:A)}B(x)}$	induction for $\mathsf{W}_{(x:A)}B$, p. 153
$\lambda x.\, b(x)$	λ-abstraction, p. 24
(a, b)	(dependent) pair, p. 25 and p. 28
$\mathsf{pair}^{=}$	constructor for $=_{A\times B}$, p. 73
$\prod_{(x:A)} B(x)$	dependent function type, p. 24
$\mathsf{pr}_1(t)$	the first projection from a pair, p. 25 and p. 28
$\mathsf{pr}_2(t)$	the second projection from a pair, p. 25 and p. 28
$\mathsf{rec}_{\mathbf{0}}$	recursor for $\mathbf{0}$, p. 30,
$\mathsf{rec}_{\mathbf{1}}$	recursor for $\mathbf{1}$, p. 26,
$\mathsf{rec}_{\mathbf{2}}$	recursor for $\mathbf{2}$, p. 31,
$\mathsf{rec}_{\mathbb{N}}$	recursor for $\mathbb{N}$, p. 34, and
$\mathsf{rec}_{A\times B}$	recursor for $A \times B$, p. 25,

$\mathsf{rec}_{\sum_{(x:A)} B(x)}$	recursor for $\sum_{(x:A)} B$, p. 29,
rec_{A+B}	recursor for $A + B$, p. 30,
$\mathsf{rec}_{\mathsf{W}_{(x:A)}B(x)}$	recursor for $\mathsf{W}_{(x:A)}B$, p. 142
$\{\, x : A \mid P(x) \,\}$	subset type, p. 102
$\sum_{(x:A)} B(x)$	dependent pair type, p. 28
$\mathsf{sup}(a, f)$	constructor for W-type, p. 141
$\mathsf{W}_{(x:A)}B(x)$	W-type (inductive type), p. 140
	UNFINISHED BELOW
$\bigcirc$	`\modal`
$\mathcal{U}_\bigcirc$	`\modaltype`
is_X	`\ism`
$\mathsf{is}_\bigcirc$	`\ismodal`
$\exists_\bigcirc$	`\existsmodal`
$\exists!_\bigcirc$	`\existsmodalunique`
$\bigcirc$-fun	`\modalfunc`
$\mathsf{E}_\bigcirc$	`\Ecirc`
$\mathsf{M}_\bigcirc$	`\Mcirc`
$\blacksquare_\ell$	`\leftwhisker`
$\blacksquare_r$	`\rightwhisker`
η	`\mreturn` `\project`
$\hat{X}$	`\mbind`
ext	`\ext`
$\bar{X}$	`\mmap`
η^{-1}	`\mjoin`
$\mathsf{islocal}_X$	`\islocal`
$\mathcal{L}_X$	`\loc`
$=$	`\idsym`
$X = YZ$	`\id`
$\mathsf{Id}(X, Y)Z$	`\idtype`
Id_X	`\idtypevar`
$:=$	`\defid` (A propositional equality currently being defined)
$Z =^X_Y W$	`\dpath`
sgl	`\sgl` singleton
sctr	`\sctr`
refl	`\reflsym`
refl_X	`\refl`
$\blacksquare$	`\ct` Path concatenation (used infix, in diagrammatic order)
X^{-1}	`\opp` `\rev` Path reversal
$X_*(Y)$	`\trans` `Trans` Transport (covariant)

X_*	`\transf`
$\mathsf{transport}_*^Y$	`\transport`
$\mathsf{transport}^X(Y,Z)$	`\transfib`
$\mathsf{transport}^X\Big(Y,\ Z\Big)$	`\Transfib`
$\mathsf{transport}^X$	`\transfibf`
$X_{**}(Y)$	`\transtwo` 2D transport
$\mathsf{transportconst}_Y^X(Z)$	`\transconst` Constant transport
transportconst	`\transconstf`
ap_X	`\mapfunc` `apfunc` Map on paths
$X(Y)$	`\map` `Ap` `ap`
apd_X	`\mapdepfunc` `apdfunc`
$\mathsf{apd}_X(Y)$	`\mapdep` `apd`
ap_X^2	`\aptwofunc` 2D map on paths
$X(Y)$	`\aptwo`
apd_X^2	`\apdtwofunc`
$\mathsf{apd}_X^2(Y)$	`\apdtwo`
id	`\idfunc`
$\sim$	`\htpy` Homotopies (written infix)
$X \simeq Y$	`\eqv`
$X \ \simeq \ Y$	`\eqvspaced`
$\mathsf{Equiv}(X,Y)$	`\texteqv`
isequiv	`\isequiv`
qinv	`\qinv`
ishae	`\ishae`
linv	`\linv`
rinv	`\rinv`
biinv	`\biinv`
$\mathsf{lcoh}_X(Y,Z)$	`\lcoh`
$\mathsf{rcoh}_X(Y,Z)$	`\rcoh`
$\mathsf{fib}_X(Y)$	`\hfib` `hfiber`
$\mathsf{total}(X)$	`\total` Map on total spaces
$\mathcal{U}$	`\UU` `bbU` `\type` Universe types
X-Type	`\typele` `\ntype` Universes of truncated types
X-$\mathsf{Type}_{\mathcal{U}}$	`\typeleU` `\ntypeU`
(X)-Type	`\typelep` `\ntypep`
(X)-$\mathsf{Type}_{\mathcal{U}}$	`\typelepU` `\ntypepU`
Set	`\set`
$\mathsf{Set}_{\mathcal{U}}$	`\setU`

Prop	`\prop`
$\mathsf{Prop}_{\mathcal{U}}$	`\propU`
$X_\bullet$	`\pointed` Pointed types
Card	`\card`
Ord	`\ord`
$X_{/Y}$	`\ordsl`
ua	`\ua` univalence
idtoeqv	`\idtoeqv`
isContr	`\iscontr`
contr	`\contr`
isSet	`\isset`
isProp	`\isprop`
amereproposition	`\anhprop`
merepropositions	`\hprops`
$\|Y\|_X$	`\trunc` `\pizero`
$\left\|\|Y\|\right\|_X$	`\ttrunc`
$\left\|\left\|Y\right\|\right\|_X$	`\Trunc`
$\|-\|_X$	`\truncf`
$\|Y|_X Z$	`\tproj`
$|-|_X$	`\tprojf`
$\|X\|$	`\brck`
$\left\|\|X\|\right\|$	`\bbrck`
$\left\|\left\|X\right\|\right\|$	`\Brck`
$\|X\|$	`\bproj`
$\|-\|$	`\bprojf`
ext	`\extendsmb`
$\mathsf{ext}(X)$	`\extend`
$\mathbf{0}$	`\emptyt`
$\mathbf{1}$	`\unit`
$\star$	`\ttt`
$\mathbf{2}$	`\bool`
$1_{\mathbf{2}}$	`\btrue`
$0_{\mathbf{2}}$	`\bfalse`
inl	`\inlsym`
inr	`\inrsym`
inl	`\inl`
inr	`\inr`
seg	`\seg`

$F(X)$	`\freegroup`
$F'(X)$	`\freegroupx`
glue	`\glue`
$\mathbb{S}$	`\Sn`
base	`\base`
loop	`\lloop`
surf	`\surf`
Σ	`\susp`
N	`\north`
S	`\south`
merid	`\merid`
_	`\blank`
$\mathbb{B}$	`\bbB`
$\mathbb{P}$	`\bbP`
$\mathcal{S}et$	`\uset`
$\mathcal{C}at$	`\ucat`
$\mathcal{R}el$	`\urel`
$\mathcal{H}ilb$	`\uhilb`
$\mathcal{T}ype$	`\utype`
X^{-1}	`\inv`
idtoiso	`\idtoiso`
isotoid	`\isotoid`
$^{\mathrm{op}}$	`\op`
$\mathbf{y}$	`\y`
$X^\dagger$	`\dgr`
$\cong^\dagger$	`\unitaryiso`
$\mathbb{N}$	`\N \nat`
$\mathbb{N}'$	`\natp`
$0'$	`\zerop`
succ	`\suc`
succ$'$	`\sucp`
add	`\add`
ack	`\ack`
ass	`\ass`
iter	`\ite`
double	`\dbl`
double$'$	`\dblp`
List(X)	`\lst`
nil	`\nil`

cons	`\cons`
$\mathsf{Vec}_X(Y)$	`\vect`
$\mathbb{Z}$	`\Z`
succ	`\Zsuc`
pred	`\Zpred`
$\mathbb{Q}$	`\Q`
funext	`\funext`
happly	`\happly`
$\mathsf{swap}_{X,Y}(Z)$	`\com`
code	`\code`
encode	`\encode`
decode	`\decode`
$\begin{cases} X & \longrightarrow & Y \\ Z & \longmapsto & W \end{cases}$	`\function`
$\mathrm{cone}_X(Y)$	`\cone`
$\mathrm{cocone}_X(Y)$	`\cocone`
$X \circ Y$	`\composecocone`
$Y \circ X$	`\composecone`
$\mathscr{D}$	`\Ddiag`
Map	`\Map`
I	`\interval`
0_I	`\izero`
1_I	`\ione`
$\twoheadrightarrow$	`\epi`
$\rightarrowtail$	`\mono`
$\widetilde{\in}$	`\bin`
$\mathsf{SemigroupStr}(X)$	`\semigroupstr`
Semigroup	`\semigroup`
$\cdot$	`\emptyctx`
$::=$	`\production`
$\downarrow$	`\conv`
ctx	`\ctx`
$\vdash X$ ctx	`\wfctx`
$X \vdash Y : Z$	`\oftp`
$X \vdash Y \equiv Z : W$	`\jdeqtp`
$X \vdash Y$	`\judg`
$X{:}Y$	`\tmtp`
FORM	`\form`

INTRO	\intro
ELIM	\elim
COMP	\comp
Wkg	\Weak
Vble	\Vble
Exch	\Exch
Subst	\Subst
c	\cc
p	\pp
$\widetilde{\mathsf{c}}$	\cct
$\widetilde{\mathsf{p}}$	\ppt
$\widetilde{W}$	\Wtil
is-X-type	\istype
$(n+1)$	\nplusone
$(n-1)$	\nminusone
fact	\fact
$\bar{k}$	\kbar
$\mathbf{N}^{\mathbf{w}}$	\natw
$0^{\mathbf{w}}$	\zerow
$\mathbf{s}^{\mathbf{w}}$	\sucw
$\mathbb{N}$Alg	\nalg
$\mathbb{N}$Hom	\nhom
isHinit$_{\mathsf{W}}$	\ishinitw
isHinit$_{\mathbb{N}}$	\ishinitn
W	\w
WAlg	\walg
WHom	\whom
$\mathbb{R}_{\mathsf{c}}$	\RC
$\mathbb{R}_{\mathsf{d}}$	\RD
$\mathbb{R}$	\R
$\bar{\mathbb{R}}_{\mathsf{d}}$	\barRD
lim	\rclim
rat	\rcrat
eq$_{\mathbb{R}_{\mathsf{c}}}$	\rceq
$\mathcal{C}$	\CAP
$\mathbb{Q}_+$	\Qp
#	\apart
isCut	\dcut
$\lhd$	\cover

$(Y, \lambda X.\, Z)$	`\intfam`
$\frown$	`\bsim`
$\smile$	`\bbsim`
$\diamondsuit \approx$	`\hapx`
$\diamondsuit$	`\hapname`
$\heartsuit \approx$	`\hapxb`
$\heartsuit$	`\hapbname`
$\bullet \approx_X \triangle$	`\tap`
$\triangle$	`\tapname`
$\bullet \approx_X \square$	`\tapb`
$\square$	`\tapbname`
No	`\NO`
$\{ X \mid Y \}$	`\surr`
$\mathcal{L}$	`\LL`
$\mathcal{R}$	`\RR`
$\trianglelefteq$	`\ble`
$\triangleleft$	`\blt`
$\sqsubseteq$	`\bble`
$\sqsubset$	`\bblt`
$\diamondsuit \preceq$	`\hle`
$\diamondsuit \prec$	`\hlt`
$\diamondsuit$	`\hlname`
$\heartsuit \preceq$	`\hleb`
$\heartsuit \prec$	`\hltb`
$\heartsuit$	`\hlbname`
$\triangle \preceq$	`\tle`
$\triangle \prec$	`\tlt`
$\triangle$	`\tlname`
$\square \preceq$	`\tleb`
$\square \prec$	`\tltb`
$\square$	`\tlbname`
set	`\vset`
$\lvert X \rvert$	`\cd`
inj	`\inj`
acc	`\acc`