HoTT Types

Derived from the HoTT Book

March 21, 2014

Currently this doc contains the intro and chapter 1 from the HoTT Book, plus some notes on some of the basic ideas like type and judgment. The idea is to winnow out some of the strictly mathematical stuff leaving the core "philosophical" stuff, and annotate the text with some comments and quotes from Martin-Löf, Brandom, etc. The purpose is to more fully articulate the link between HoTT's ideas of type and judgment (etc.) to the philosophical debates about language, assertion, proposition from which they emerged. Why? Because I find those bits of the HoTT a little murky, and philosophers like Brandom have a lot to say about the issues.

DRAFT

Contents

1 HoTT Introduction 3

1.1 Type theory 4

1.2 Homotopy type theory 5

1.3 Univalent foundations 7

1.4 Higher inductive types 8

1.5 Sets in univalent foundations 9

1.6 Informal type theory 10

1.7 Constructivity 11

1.8 Open problems 15

1.9 How to read this book 17

2 Type theory 21

2.1 Type theory versus set theory 21

2.2 Function types 26

2.3 Universes and families 29

2.4 Dependent function types (Π-types) 30

2.5 Product types 32

2.6 Dependent pair types (Σ-types) 36

2.7 Coproduct types 40

2.8 The type of booleans 41

2.9 The natural numbers 43

2.10 Pattern matching and recursion 46

2.11 *Propositions as types* 48

2.12 *Identity types* 55

2.13 *Path induction* 57

2.14 *Equivalence of path induction and based path induction* 60

2.15 *Disequality* 63

2.16 *Notes* 63

3 *Philosophical Niceties* 66

3.1 *a : A* 67

3.2 *Justification* 69

DRAFT

HoTT 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 ultimate theory will probably not look exactly like the one described in this book, but it will surely be *at least* as capable and powerful; we therefore believe that univalent

DRAFT

foundations will eventually become a viable alternative to set theory as the "implicit foundation" for the unformalized mathematics done by most mathematicians.

1.1 Type theory

Type theory was originally invented by Bertrand Russell [**?**], 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 [**???**]. 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 [**?**]. Per Martin-Löf [**????**], 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 [**??**]) for the history of type theory).

"[T]he intuitionistic type theory ([**?**]), which I began to develop *solely with the philosophical motive* of clarifying the syntax and semantics of intuitionistic mathematics, may equally well be viewed as a programming language." [**?**] (Emphasis added.)

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. We believe that the new idea of regarding types, not as strange sets (perhaps constructed without using classical logic), but as spaces, viewed

DRAFT

from the perspective of homotopy theory, is a significant step forward. In particular, it solves the problem of understanding how the notion of equality of elements of a type differs from that of elements of a set.

In homotopy theory one is concerned with spaces and continuous mappings between them, up to homotopy. A *homotopy* between a pair of continuous maps $f : X \to Y$ and $g : X \to Y$ is a continuous map $H : X \times [0,1] \to Y$ 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*.

1.2 Homotopy type theory

Homotopy type theory (HoTT) 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 function $f : A \to B$ in type theory is regarded as a continuous map 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 between 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.

DRAFT

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 partial equivalence relations 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 [**?**] and Voevodsky [**?**], but it was inspired by Hofmann and Streicher's earlier groupoid interpretation [**?**]. 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 prin-

DRAFT

ciple 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.

1.3 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

DRAFT

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.

1.4 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

DRAFT

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.

1.5 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 [**?**].

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 (ZF). 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 [**?**], 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, i.e. those which are homotopy equivalent to a discrete space. It is a theorem that the category of such "sets" satisfies Lawvere's axioms (or related ones, depending on the details of the theory). 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

DRAFT

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.

1.6 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

DRAFT

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 demonstrated 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. 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.

1.7 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

DRAFT

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:

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

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).

The logic of propositions-as-types is flexible and supports many variations, such as using only a subclass of types to represent propositions. In homotopy type theory, there are natural such subclasses arising from the fact 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. Particularly important for logic is the case of homotopy (-1)-types, which we call *mere propositions*. Classically, every (-1)-type is empty or contractible; we interpret these possibilities as the truth values "false" and "true" respectively.

Using all types as propositions yields a "constructive" conception of logic (for more on which, see [**???**]), which gives type theory its good computational character. For instance, every proof that something ex-

DRAFT

ists 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 logic does not faithfully represent certain important classical principles of reasoning, such as the axiom of choice and the law of excluded middle. For these we need to use the "(-1)-truncated" logic, in which only the homotopy (-1)-types represent propositions; and under this interpretation, the system is fully compatible with classical mathematics. Homotopy type theory is thus compatible with both constructive and classical conceptions of logic, and many more besides.

More specifically, consider on one hand the *axiom of choice*: "if for every $x : A$ there exists a $y : B$ such that $R(x,y)$, there is a function $f : A \to B$ such that for all $x : A$ we have $R(x,f(x))$." The pure propositions-as-types notion of "there exists" is strong enough to make this statement simply provable — yet it does not have all the consequences of the usual axiom of choice. However, in (-1)-truncated logic, this statement is not automatically true, but is a strong assumption with the same sorts of consequences as its counterpart in classical set theory.

On the other hand, consider the *law of excluded middle*: "for all A, either A or not A." Interpreting this in the pure propositions-as-types logic yields a statement that is inconsistent with the univalence axiom. For since proving "A" means exhibiting an element of it, this assumption would give a uniform way of selecting an element from every nonempty type — a sort of Hilbertian choice operator. 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. However, the "(-1)-truncated law of excluded middle", though also not automatically true, may consistently be assumed with most of the same consequences as in classical mathematics.

In other words, while the pure propositions-as-types logic is "constructive" in the strong algorithmic sense mentioned above, the default (-1)-truncated logic is "constructive" in a different sense (namely, that of the logic formalized by Heyting under the name "intuitionistic"); and to the latter we may freely add the axioms of choice and excluded middle to obtain a logic that may be called "classical". 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

DRAFT

$-1 < n < \infty$). 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 [**???**]. 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, [**?**]. 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

DRAFT

these constructions directly and constructively. In particular, none of the "synthetic" homotopy theory in **??** 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 **??**.

(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 **??**.

(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 **??**.

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 [**?**] (see [**?**] for higher inductive types). Thus, they are known to be 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., [**??**]).

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

1.8 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

DRAFT

Types	Logic	Sets	Homotopy
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
$\mathbf{0}$,	$\bot, \top$	$\emptyset, \{\emptyset\}$	$\emptyset, *$
$A + B$	$A \vee B$	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	equality $=$	$\{ (x, x) \mid x \in A \}$	path space A^I

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

questions.

Perhaps the most pressing of them is the "constructivity" of the Univalence Axiom, posed by Voevodsky in [**?**]. 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 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.

DRAFT

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 **??** 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 **????**. Obviously, however, much work remains to be done.

The homotopy type theory community maintains a web site and group blog at http://homotopytypetheory.org, as well as a discussion email list. Newcomers are always welcome!

1.9 How to read this book

This book is divided into two parts. **??**, "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 **??** is fairly extensive.

??, "Mathematics", consists of four chapters that build on the basic

DRAFT

notions of **??** to exhibit some of the new things we can do with univalent foundations in four different areas of mathematics: homotopy theory (**??**), category theory (**??**), set theory (**??**), and real analysis (**??**). The chapters in **??** 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 **??**. 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 **??**. Fortunately, not all of **??** is necessary in order to read the chapters in **??**. Each chapter in **??** begins with a brief overview of its subject, what univalent foundations has to contribute to it, and the necessary background from **??**, 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 **??** more deeply than this, but are not ready to read all of **??**, we provide here a brief summary of **??**, with remarks about which parts are necessary for which chapters in **??**.

?? 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 **??**, as there are many subtle differences between type theory and other foundations such as set theory.

?? 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 **??**. It also introduces the *univalence axiom* (**??**) — the first of the two basic innovations of homotopy type theory. Thus, it is quite basic and we encourage everyone to read it, especially **??**–**??**.

?? 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 **????**, less important for **??**, and not so necessary for **??**.

???? 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 **??**. Only a few lemmas from **??** are used here and there, while the general

DRAFT

discussions in **??????** are helpful for providing the intuition required for **??**. The generalized sorts of inductive definition discussed in **??** are also used in a few places in **????**.

?? introduces the second basic innovation of homotopy type theory — *higher inductive types* — with many examples. Higher inductive types are the primary object of study in **??**, and some particular ones play important roles in **????**. They are not so necessary for **??**, although one example is used in **??**.

Finally, **??** discusses homotopy n-types and related notions such as n-connected types. These notions are important for **??**, but not so important in the rest of **??**, although the case $n = -1$ of some of the lemmas are used in **??**.

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

Of the chapters in **??**, **??** (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 $\pi_1(\mathbb{S}^1) = \mathbb{Z}$ to the Freudenthal suspension theorem.

In **??** (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.)

?? (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 **??** (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

DRAFT

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.

DRAFT

Type theory

2.1 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 16. 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 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

[1] Confusingly, it is also a common practice (dating back to Euclid) to use the word "proposition" synonymously 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.

DRAFT

"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 **??**.

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 "$\mathrm{id}[A]ab$". (In *homotopy* type theory, of course, this

DRAFT

equality proposition can behave in unfamiliar ways: see **????**, and the rest of the book). When $\mathrm{id}[A]ab$ is inhabited, we say that a and b are **(propositionally) equal**.

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 $a \equiv b : A$ or simply $a \equiv b$. 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*. Inside the theory, 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 (though the algorithm is necessarily meta-theoretic, not internal to the theory).

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 may 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"
$a \equiv b : A$	"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, "$p : \mathrm{id}\, xy$" should be parsed as "$p : (\mathrm{id}\, xy)$", which makes sense since "$\mathrm{id}\, xy$" is a type, and not as "$\mathrm{id}\,(p : x)y$",

[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 **??**.

DRAFT

which is senseless since "$p : x$" is a judgment and cannot be equal to anything. Similarly, "$A \equiv \mathrm{id}\, xy$" can only be parsed as "$A \equiv (\mathrm{id}\, xy)$", although in extreme cases such as this, one ought to add parentheses anyway to aid reading comprehension. Moreover, later on we will fall into the common notation of chaining together equalities — e.g. writing $a = b = c = d$ to mean "$a = b$ and $b = c$ and $c = d$, hence $a = d$" — and we will also include judgmental equalities in such chains. Context usually suffices to make the intent clear.

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 §2.4).

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 $p : \mathrm{id}[A]xy$, we may construct an element $p^{-1} : \mathrm{id}[A]yx$. The collection of all such assumptions is called the **context**; from 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 $\mathrm{id}[A]xy$, we can prove $\mathrm{id}[A]yx$. 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 **??**.

Note that under this meaning of the word *assumption*, we can assume a propositional equality (by assuming a variable $p : x = y$), but we cannot assume a judgmental equality $x \equiv y$, since it is not a type that can have an element. However, we can do something else which looks kind of like assuming a judgmental equality: if we have a type or an element which involves a variable $x : A$, then we can *substitute* any particular element $a : A$ for x to obtain a more specific type or element. We will sometimes use language like "now assume $x \equiv a$" to refer to this process of substitution, even though it is not an *assumption* in the technical sense introduced above.

By the same token, we cannot *prove* a judgmental equality either,

DRAFT

since it is not a type in which we can exhibit a witness. Nevertheless, we will sometimes state judgmental equalities as part of a theorem, e.g. "there exists $f : A \to B$ such that $f(x) \equiv y$". This should be regarded as the making of two separate judgments: first we make the judgment $f : A \to B$ for some element f, then we make the additional judgment that $f(x) \equiv y$.

In the rest of this chapter, we attempt to give an informal presentation of type theory, sufficient for the purposes of this book; we give a more formal account in **??**. 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 here; see however the beginning of **??** and also **??**.

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 a 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 **??** 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 **??????** 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

DRAFT

a traditional rule-based type theory.

2.2 Function types

Given types A and B, we can construct the type $A \to B$ of **functions** with domain A and codomain B. We also sometimes refer to functions as **maps**. 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 B, denoted $f(a)$ and called the **value** of f at a. It is common in type theory to omit the parentheses and denote $f(a)$ simply by $f\,a$, and we will sometimes do this as well.

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

$$f(x) :\equiv \Phi \tag{2.1}$$

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

Now we can compute $f(a)$ by replacing the variable x in Φ 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 **??**.) 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 Φ of type B which may use $x : A$, as above, we write $\lambda(x:A).\,\Phi$ to indicate the same function defined by (2.1). Thus, we have

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

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

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

As another example, for any types A and B and any element $y : B$, we have a **constant function** $(\lambda(x:A).\,y) : A \to B$.

We generally omit the type of the variable x in a λ-abstraction and write $\lambda x.\,\Phi$, since the typing $x : A$ is inferable from the judgment that the function $\lambda x.\,\Phi$ 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

DRAFT

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

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

We may also sometimes use a blank "$-$" in the expression Φ 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 following **computation rule**[3], which is a definitional equality:

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

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

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

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 that applies f to its argument" we consider it to be definitionally equal to f:[4]

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

This equality is the **uniqueness principle for function types**, because it shows that f is uniquely determined by its values.

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

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

as

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

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

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

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

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

DRAFT

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 argument of the λ-abstraction. 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.)

[5] Use of this equality is often referred to as **α-conversion.**

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** (after the mathematician Haskell Curry).

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 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. When omitting parentheses around function arguments entirely, we write $f\,a\,b$ for $(f\,a)\,b$, with the default associativity now being to the left so that f is applied to its arguments in the correct order.

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

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

where $\Phi : C$ assuming $x : A$ and $y : B$. Using λ-abstraction this

DRAFT

corresponds to

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

which may also be written as

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

We can also implicitly abstract over multiple variables by writing multiple blanks, e.g. $g(-,-)$ means $\lambda x.\, \lambda y.\, g(x,y)$. Currying a function of three or more arguments is a straightforward extension of what we have just described.

2.3 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 (see **??**), is inhabited. For instance, using a representation of sets as trees, we can directly encode Russell's paradox [**?**].

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 i^{th} universe are also elements of the $(i+1)^{\text{st}}$ universe, i.e. if $A : \mathcal{U}_i$ then also $A : \mathcal{U}_{i+1}$. This is convenient, but has the slightly unpleasant consequence that elements no longer have unique types, and is a bit tricky in other ways that need not concern us here; see the Notes.

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**. It is convenient but a bit dangerous, since it allows us to write valid-looking proofs that reproduce the paradoxes of self-reference. If there is any doubt about whether an argument is correct, the way to check it is to try to assign levels consistently to all universes appearing in it. When some universe $\mathcal{U}$ is assumed, we may refer to types belonging to $\mathcal{U}$ as **small types**.

DRAFT

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** (or sometimes *dependent types*); they correspond to families of sets as used in set theory.

An example of a type 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. (We cannot *define* the family Fin yet — indeed, we have not even introduced its domain $\mathbb{N}$ yet — but we will be able to soon; see **??**.) We may denote the elements of $\mathsf{Fin}(n)$ by $0_n, 1_n, \ldots, (n-1)_n$, with subscripts to emphasize that the elements of $\mathsf{Fin}(n)$ are different from those of $\mathsf{Fin}(m)$ if n is different from m, and all are different from the ordinary natural numbers (which we will introduce in **??**).

A more trivial (but very important) example of a type family is the **constant** type family at a type $B : \mathcal{U}$, which is of course the constant function $(\lambda(x:A).B) : A \to \mathcal{U}$.

As a *non*-example, in our version of type theory there is no type family "$\lambda(i:\mathbb{N}).\mathcal{U}_i$". Indeed, there is no universe large enough to be its codomain. Moreover, we do not even identify the indices i of the universes $\mathcal{U}_i$ with the natural numbers $\mathbb{N}$ of type theory (the latter to be introduced in **??**).

2.4 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, called **dependent functions**. The name "Π-type" is used because this type can also be regarded as the 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 a constant family, 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 generalizations 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 $\Phi : B(x)$ possibly involving the variable $x : A$, and we write

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

DRAFT

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

$$\lambda x.\, \Phi \;:\; \prod_{x:A} B(x). \tag{2.2}$$

As with non-dependent functions, we can **apply** a dependent function $f : \prod_{(x:A)} B(x)$ to an argument $a : A$ to obtain an element $f(a) : B(a)$. The equalities are the same as for the ordinary function type, i.e. we have the computation rule given $a : A$ we have $f(a) \equiv \Phi'$ and $(\lambda x.\, \Phi)(a) \equiv \Phi'$, where Φ' is obtained by replacing all occurrences of x in Φ by a (avoiding variable capture, as always). Similarly, we have the uniqueness principle $f \equiv (\lambda x. f(x))$ for any $f : \prod_{(x:A)} B(x)$.

As an example, recall from **??** that there is a type family $\mathsf{Fin} : \mathbb{N} \to \mathcal{U}$ whose values are the standard finite sets, with elements $0_n, 1_n, \ldots, (n-1)_n : \mathsf{Fin}(n)$. There is then a dependent function $\mathsf{fmax} : \prod_{(n:\mathbb{N})} \mathsf{Fin}(n+1)$ which returns the "largest" element of each nonempty finite type, $\mathsf{fmax}(n) :\equiv n_{n+1}$. As was the case for Fin itself, we cannot define fmax yet, but we will be able to soon; see **??**.

Another important class of dependent function types, which we can define now, 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 $\mathsf{id} : \prod_{(A:\mathcal{U})} A \to A$, which we define by $\mathsf{id} :\equiv \lambda(A:\mathcal{U}).\, \lambda(x:A).\, x$.

We sometimes write some arguments of a dependent function as subscripts. For instance, we might equivalently define the polymorphic identity function by $\mathsf{id}_A(x) :\equiv x$. Moreover, if an argument can be inferred from context, we may omit it altogether. For instance, if $a : A$, then writing $\mathsf{id}(a)$ is unambiguous, since id must mean id_A in order for it to be applicable to a.

Another, less trivial, example of a polymorphic function is the "swap" operation that switches the order of the arguments of a (curried) two-argument function:

$$\mathsf{swap} : \prod_{(A:\mathcal{U})} \prod_{(B:\mathcal{U})} \prod_{(C:\mathcal{U})} (A \to B \to C) \to (B \to A \to C)$$

We can define this by

$$\mathsf{swap}(A, B, C, g) :\equiv \lambda b.\, \lambda a.\, g(a)(b).$$

We might also equivalently write the type arguments as subscripts:

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

Note that as we did for ordinary functions, we use currying to define dependent functions with several arguments (such as swap). However, in the dependent case the second domain may depend on the first one,

DRAFT

and the codomain may depend on both. That is, given $A : \mathcal{U}$ and type families $B : A \to \mathcal{U}$ 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 $C : \prod_{(x:A)} (B(x) \to \mathcal{U})$.) In the case when B is constant and equal to A, we may condense the notation and write $\prod_{(x,y:A)}$; for instance, the type of swap could also be written as

$$\mathsf{swap} : \prod_{(A,B,C:\mathcal{U})} (A \to B \to C) \to (B \to A \to C).$$

Finally, 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 $f(a,b) : C(a,b)$.

2.5 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** $:\mathcal{U}$. 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 to be some particular object $^* :$. 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.

Remark 2.3. There is a general pattern for introduction of a new kind of type in type theory, and because products are our second example following this pattern,[6] it is worth emphasizing the general form: To specify a type, we specify:

(i) how to form new types of this kind, via **formation rules**. (For example, we can form the function type $A \to B$ when A is a type and when B is a type. We can form the dependent function type $\prod_{(x:A)} B(x)$ when A is a type and $B(x)$ is a type for $x : A$.)

(ii) how to construct elements of that type. These are called the type's **constructors** or **introduction rules**. (For example, a function type has one constructor, λ-abstraction. Recall that a direct definition like $f(x) :\equiv 2x$ can equivalently be phrased as a λ-abstraction $f :\equiv \lambda x.\, 2x$.)

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

(iv) a **computation rule**[7], which expresses how an eliminator acts on a constructor. (For example, for functions, the computation rule states that $(\lambda x.\, \Phi)(a)$ is judgmentally equal to the substitution of a for x in Φ.)

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

[7] also referred to as β-**reduction**

DRAFT

(v) an optional **uniqueness principle**[8], which expresses uniqueness of maps into or out of that type. For some types, the uniqueness principle characterizes maps into the type, by stating that every element of the type is uniquely determined by the results of applying eliminators to it, and can be reconstructed from those results by applying a constructor—thus expressing how constructors act on eliminators, dually to the computation rule. (For example, for functions, the uniqueness principle says that any function f is judgmentally equal to the "expanded" function $\lambda x.f(x)$, and thus is uniquely determined by its values.) For other types, the uniqueness principle says that every map (function) *from* that type is uniquely determined by some data. (An example is the coproduct type introduced in **??**, whose uniqueness principle is mentioned in **??**.)

When the uniqueness principle is not taken as a rule of judgmental equality, it is often nevertheless provable as a *propositional* equality from the other rules for the type. In this case we call it a **propositional uniqueness principle**. (In later chapters we will also occasionally encounter *propositional computation rules*.)

[8] also referred to as η-**expansion**

The inference rules in **??** are organized and named accordingly; see, for example, **??**, where each possibility is realized.

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 , namely we have $^* :$. We expect that "every element of $A \times B$ is a pair", which is the uniqueness principle for products; we do not assert this as a rule of type theory, but we will prove it later on as a propositional equality.

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, we expect to be able to define such a function by prescribing the result when f is applied to a pair (a, b). We can prescribe these results by providing a function $g : A \to B \to C$. Thus, we introduce a new rule (the elimination rule for products), which says that for any such g, we can define a function $f : A \times B \to C$ by

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

We avoid writing $g(a, b)$ here, in order to emphasize that g is not a function on a product. (However, later on in the book we will often write $g(a, b)$ both for functions on a product and for curried functions of two variables.) This defining equation is the computation rule for product types.

Note that in set theory, we would justify the above definition of f by the fact that every element of $A \times B$ is a pair, so that it suffices

DRAFT

to define f on pairs. By contrast, type theory reverses the situation: we assume that a function on $A \times B$ is well-defined as soon as we specify its values on tuples, and from this (or more precisely, from its more general version for dependent functions, below) we will be able to *prove* that every element of $A \times B$ is a pair. From a category-theoretic perspective, we can say that we define the product $A \times B$ to be left adjoint to the "exponential" $B \to C$, which we have already introduced.

As an example, we can derive the **projection** functions

$$\mathsf{pr}_1 : A \times B \to A$$
$$\mathsf{pr}_2 : A \times B \to B$$

with the defining equations

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

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

$$\mathsf{rec}_{A\times B} : \prod_{C:\mathcal{U}} (A \to B \to C) \to A \times B \to C \tag{2.4}$$

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 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. The name "recursor" is a bit unfortunate here, since no recursion is taking place. It comes from 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 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.

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:

$$\mathsf{rec} : \textstyle\prod_{(C:\mathcal{U})} C \to \ \to C$$

DRAFT

with the defining equation

$$\mathsf{rec}(C, c, *) :\equiv c.$$

Although we include it to maintain the pattern of type definitions, the recursor for is completely useless, because we could have defined such a function directly by simply ignoring the argument of type .

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 $g : \prod_{(x:A)} \prod_{(y:B)} C((x,y))$ with defining equation

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

For example, in this way we can prove the propositional uniqueness principle, which says that every element of $A \times B$ is equal to a pair. Specifically, we can construct a function

$$\mathsf{uppt} : \prod_{(x:A\times B)} (\mathrm{id}[A \times B](\mathsf{pr}_1(x), \mathsf{pr}_2(x))x).$$

Here we are using the identity type, which we are going to introduce below in **??**. However, all we need to know now is that there is a reflexivity element $\mathsf{refl}_x : \mathrm{id}[A]xx$ for any $x : A$. Given this, we can define

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

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,

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

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

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

with the defining equation

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

DRAFT

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:

$$\mathsf{ind} : \textstyle\prod_{(C:\to\mathcal{U})} C(*) \to \prod_{(x:)} C(x)$$

with the defining equation

$$\mathsf{ind}(C, c, *) :\equiv c.$$

Induction enables us to prove the propositional uniqueness principle for , which asserts that its only inhabitant is $*$. That is, we can construct

$$\mathsf{upun} : \textstyle\prod_{(x:)} \mathrm{id}\, x^*$$

by using the defining equations

$$\mathsf{upun}(*) :\equiv \mathsf{refl}_*$$

or equivalently by using induction:

$$\mathsf{upun} :\equiv \mathsf{ind}(\lambda x.\ \mathrm{id}\, x^*, \mathsf{refl}_*).$$

2.6 *Dependent pair types (Σ-types)*

Just as we generalized function types (**??**) to dependent function types (§2.4), it is often useful to generalize the product types from **??** 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 **dependent pair type**, or **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

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

Like other binding constructs such as λ-abstractions and Πs, Σs automatically scope over the rest of the expression unless delimited, so e.g. $\sum_{(x:A)} B(x) \times C(x)$ means $\sum_{(x:A)} (B(x) \times C(x))$.

The way to construct elements of a dependent pair 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:

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

DRAFT

All the constructions on Σ-types arise as straightforward generalizations 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:

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

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

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

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:

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

DRAFT

with the defining equation

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

and the corresponding induction operator:

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

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

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

Moreover, the type $\prod_{(x:A)} R(x, \mathsf{pr}_1(g(x)))$ 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:

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

Thus, we have

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

as required.

DRAFT

If we read $\prod$ as "for all" and $\sum$ 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**, and as we have just shown, it can be proved directly from the rules of type theory, rather than having to be taken as an axiom. However, note that 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 **??** we will give another formulation of an "axiom of choice" which is closer to the usual one.

Dependent pair types are often used to define types of mathematical structures, which commonly consist of several dependent pieces of data. To take a simple example, suppose we want to define a **magma** to be a type A together with a binary operation $m : A \to A \to A$. The precise meaning of the phrase "together with" (and the synonymous "equipped with") is that "a magma" is a *pair* (A, m) consisting of a type $A : \mathcal{U}$ and an operation $m : A \to A \to A$. Since the type $A \to A \to A$ of the second component m of this pair depends on its first component A, such pairs belong to a dependent pair type. Thus, the definition "a magma is a type A together with a binary operation $m : A \to A \to A$" should be read as defining *the type of magmas* to be

$$\mathsf{Magma} :\equiv \sum_{(A:\mathcal{U})} (A \to A \to A).$$

Given a magma, we extract its underlying type (its "carrier") with the first projection pr_1, and its operation with the second projection pr_2. Of course, structures built from more than two pieces of data require iterated pair types, which may be only partially dependent; for instance the type of pointed magmas (magmas (A, m) equipped with a basepoint $e : A$) is

$$\mathsf{PointedMagma} :\equiv \sum_{(A:\mathcal{U})} (A \to A \to A) \times A.$$

We generally also want to impose axioms on such a structure, e.g. to make a pointed magma into a monoid or a group. This can also be done using Σ-types; see **??**.

In the rest of the book, we will sometimes make definitions of this sort explicit, but eventually we trust the reader to translate them from English into Σ-types. We also generally follow the common mathematical practice of using the same letter for a structure of this sort and for its carrier (which amounts to leaving the appropriate projection function implicit in the notation): that is, we will speak of a magma A with its operation $m : A \to A \to A$.

DRAFT

Note that the canonical elements of PointedMagma are of the form $(A,(m,e))$ where $A : \mathcal{U}$, $m : A \to A \to A$, and $e : A$. Because of the frequency with which iterated Σ-types of this sort arise, we use the usual notation of ordered triples, quadruples and so on to stand for nested pairs (possibly dependent) associating to the right. That is, we have $(x,y,z) :\equiv (x,(y,z))$ and $(x,y,z,w) :\equiv (x,(y,(z,w)))$, etc.

2.7 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 $b : B$. (The names inl and inr are short for "left injection" and "right injection".) 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

$$f(\mathsf{inl}(a)) :\equiv g_0(a),$$
$$f(\mathsf{inr}(b)) :\equiv g_1(b).$$

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

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

with the defining equations

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

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

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

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

To construct a dependent function $f : \prod_{(x:A+B)} C(x)$ out of a coprod-

DRAFT

uct, we assume as given the family $C : (A + B) \to \mathcal{U}$, and require

$$g_0 : \prod_{a:A} C(\mathsf{inl}(a)),$$
$$g_1 : \prod_{b:B} C(\mathsf{inr}(b)).$$

This yields f with the defining equations:

$$f(\mathsf{inl}(a)) :\equiv g_0(a),$$
$$f(\mathsf{inr}(b)) :\equiv g_1(b).$$

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.

2.8 The type of booleans

The type of booleans $\mathbf{2} : \mathcal{U}$ is intended to have exactly two elements $0_{\mathbf{2}}, 1_{\mathbf{2}} : \mathbf{2}$. It is clear that we could construct this type out of coproduct and unit types as $+$. 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

$$f(0_{\mathbf{2}}) :\equiv c_0,$$
$$f(1_{\mathbf{2}}) :\equiv c_1.$$

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

$$\mathsf{rec}_{\mathbf{2}}(C, c_0, c_1, 0_{\mathbf{2}}) :\equiv c_0,$$
$$\mathsf{rec}_{\mathbf{2}}(C, c_0, c_1, 1_{\mathbf{2}}) :\equiv c_1.$$

DRAFT

Given $C : \mathbf{2} \to \mathcal{U}$, 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

$$f(0_\mathbf{2}) :\equiv c_0,$$
$$f(1_\mathbf{2}) :\equiv c_1.$$

We package this up into the induction principle

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

with the defining equations

$$\mathsf{ind}_\mathbf{2}(C, c_0, c_1, 0_\mathbf{2}) :\equiv c_0,$$
$$\mathsf{ind}_\mathbf{2}(C, c_0, c_1, 1_\mathbf{2}) :\equiv c_1.$$

As an example, using the induction principle we can deduce that, as we expect, every element of $\mathbf{2}$ is either $1_\mathbf{2}$ or $0_\mathbf{2}$. As before, in order to state this we use the equality types which we have not yet introduced, but we need only the fact that everything is equal to itself by $\mathsf{refl}_x : x = x$. Thus, we construct an element of

$$\prod_{x:\mathbf{2}} (x = 0_\mathbf{2}) + (x = 1_\mathbf{2}), \tag{2.5}$$

i.e. a function assigning to each $x : \mathbf{2}$ either an equality $x = 0_\mathbf{2}$ or an equality $x = 1_\mathbf{2}$. We define this element using the induction principle for $\mathbf{2}$, with $C(x) :\equiv (x = 0_\mathbf{2}) + (x = 1_\mathbf{2})$; the two inputs are $\mathsf{inl}(\mathsf{refl}_{0_\mathbf{2}}) : C(0_\mathbf{2})$ and $\mathsf{inr}(\mathsf{refl}_{1_\mathbf{2}}) : C(1_\mathbf{2})$. In other words, our element of (2.5) is

$$\mathsf{ind}_\mathbf{2}\big(\lambda x.\, (x = 0_\mathbf{2}) + (x = 1_\mathbf{2}),\, \mathsf{inl}(\mathsf{refl}_{0_\mathbf{2}}),\, \mathsf{inr}(\mathsf{refl}_{1_\mathbf{2}})\big).$$

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 a subtle and important aspect of type theory.) Thus, we could have defined

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

with

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

DRAFT

We leave it as an exercise to derive the induction principle of a coproduct type from this definition. (See also **????**.)

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

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

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 **??**. Moreover, we do not get the same judgmental equalities; see **??**. This is a recurrent issue when encoding one type as another; we will return to it in **??**.

We may occasionally refer to the elements $0_{\mathbf{2}}$ and $1_{\mathbf{2}}$ of $\mathbf{2}$ as "false" and "true" 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 **??**.) In particular, the type $A \to \mathbf{2}$ is not generally the power set of A; it represents only the "decidable" subsets of A (see **??**).

2.9 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 one which is of course also 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

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

DRAFT

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 "starting point" and "next step" data:

$$\begin{aligned} c_0 &: \mathbb{N} \to \mathbb{N} \\ c_0(n) &:\equiv n \\ c_s &: \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}) \\ c_s(m, g)(n) &:\equiv \mathsf{succ}(g(n)). \end{aligned}$$

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 $2 + 2 \equiv 4$.

DRAFT

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

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

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:

$$\mathsf{double} :\equiv \mathsf{rec}_{\mathbb{N}}\big(\mathbb{N},\, 0,\, \lambda n.\, \lambda y.\, \mathsf{succ}(\mathsf{succ}(y))\big) \tag{2.6}$$

$$\mathsf{add} :\equiv \mathsf{rec}_{\mathbb{N}}\big(\mathbb{N} \to \mathbb{N},\, \lambda n.\, n,\, \lambda n.\, \lambda g.\, \lambda m.\, \mathsf{succ}(g(m))\big). \tag{2.7}$$

Of course, all functions definable only using the primitive recursion principle will be *computable*. (The presence of higher function types — that is, functions with other functions as arguments — does, however, mean we can define more than the usual primitive recursive functions; see e.g. **??**.) This is appropriate in constructive mathematics; in **????** 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

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

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

DRAFT

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

$$\mathsf{assoc} : \textstyle\prod_{(i,j,k:\mathbb{N})} \mathrm{id}\, i + (j+k)(i+j) + k,$$

it is sufficient to supply

$$\mathsf{assoc}_0 : \textstyle\prod_{(j,k:\mathbb{N})} \mathrm{id}\, 0 + (j+k)(0+j) + k$$

and

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

To derive assoc_0, recall that $0 + n \equiv n$, and hence $0 + (j+k) \equiv j + k \equiv (0+j) + k$. Hence we can just set

$$\mathsf{assoc}_0(j,k) :\equiv \mathsf{refl}_{j+k}.$$

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

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

Thus, the output type of assoc_s is equivalently $\mathrm{id}\, \mathsf{succ}(i + (j+k))\mathsf{succ}((i+j)+k)$. But its input (the "inductive hypothesis") yields $\mathrm{id}\, i + (j+k)(i+j) + k$, 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 in **??**, using the induction principle of identity types.) We call this latter fact $\mathsf{ap}_{\mathsf{succ}} : (\mathrm{id}[\mathbb{N}]mn) \to (\mathrm{id}[\mathbb{N}]\mathsf{succ}(m)\mathsf{succ}(n))$, so we can define

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

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

2.10 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:

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

DRAFT

or by giving the defining equations:

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

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

$$f(\mathsf{inl}(a)) :\equiv \Phi_0$$
$$f(\mathsf{inr}(b)) :\equiv \Phi_1$$

where Φ_0 and Φ_1 are expressions that may involve the variables a and b respectively, we can express these equations equivalently in terms of the recursor by using λ-abstraction:

$$f :\equiv \mathsf{rec}_{A+B}(C, \lambda a.\, \Phi_0, \lambda b.\, \Phi_1).$$

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

$$\mathsf{double}(0) :\equiv 0 \tag{2.8}$$
$$\mathsf{double}(\mathsf{succ}(n)) :\equiv \mathsf{succ}(\mathsf{succ}(\mathsf{double}(n))) \tag{2.9}$$

involve *the function* double *itself* on the right-hand side. However, we would still like to be able to give these equations, rather than (2.6), 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 (2.9) 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

$$f(0) :\equiv \Phi_0$$
$$f(\mathsf{succ}(n)) :\equiv \Phi_s$$

where Φ_0 is an expression of type C, and Φ_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

$$f :\equiv \mathsf{rec}_{\mathbb{N}}(C,\ \Phi_0,\ \lambda n.\, \lambda r.\, \Phi'_s)$$

where Φ'_s is obtained from Φ_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

DRAFT

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 **??????**. 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.

2.11 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 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 the collection of 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 mathemat-

DRAFT

ical objects and govern how they interact, propositions are nothing but special types — namely, types whose elements are 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. And if we want to use "A and B" to prove something else, we are free to use both A and B in doing so, analogously to how the induction principle for $A \times B$ allows us to construct a function out of it by using elements of A and 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. And the basic way to use an implication "if A then B" is deduce B if we know A, analogously to how we can apply a function $f : A \to B$ to an element of A to produce an element of B. We strongly encourage the reader to do the exercise of verifying that the rules governing the other type constructors translate sensibly into logic.

Of special note is that the empty type $\mathbf{0}$ corresponds to falsity. When speaking logically, we refer to an inhabitant of $\mathbf{0}$ as a **contradiction**: thus there is no way to prove a contradiction,[9] while from a contradiction anything can be derived. We also define the **negation** of a type A as

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

Thus, a witness of $\neg A$ is a function $A \to \mathbf{0}$, which we may construct by assuming $x : A$ and deriving an element of $\mathbf{0}$. Note that although the

[9] More precisely, there is no *basic* way to prove a contradiction, i.e. $\mathbf{0}$ has no constructors. If our type theory were inconsistent, then there would be some more complicated way to construct an element of $\mathbf{0}$.

DRAFT

logic we obtain is "constructive", as discussed in the introduction, this sort of "proof by contradiction" (assume A and derive a contradiction, concluding $\neg A$) is perfectly valid constructively: it is simply invoking the *meaning* of "negation". The sort of "proof by contradiction" which is disallowed is to assume $\neg A$ and derive a contradiction as a way of proving A. Constructively, such an argument would only allow us to conclude $\neg\neg A$, and the reader can verify that there is no obvious way to get from $\neg\neg A$ (that is, from $(A \to \mathbf{0}) \to \mathbf{0}$) to A.

The above translation of logical connectives into type-forming operations is referred to as **propositions as types**: it gives us a way to translate propositions and their proofs, written in English, into types and their elements. For example, suppose we want to prove the following tautology (one of "de Morgan's laws"):

$$\text{"If not } A \text{ and not } B\text{, then not } (A \text{ or } B)\text{"}. \tag{2.10}$$

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 (2.10), according to the rules given above, is

$$(A \to \mathbf{0}) \times (B \to \mathbf{0}) \to (A + B \to \mathbf{0}) \tag{2.11}$$

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 (2.11). 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) of types $A \to \mathbf{0}$ and $B \to \mathbf{0}$. When translating into type theory, we have to give these variables names; let us call them x and y. At this point our partial definition of an element of (2.11) can be written as

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

with a "hole" $\square$ of type $A + B \to \mathbf{0}$ indicating what remains to be done. (We could equivalently write $f :\equiv \mathsf{rec}_{(A\to\mathbf{0})\times(B\to\mathbf{0})}(A + B \to \mathbf{0}, \lambda x.\, \lambda y.\, \square)$, using the recursor instead of pattern matching.) The next phrase "also

DRAFT

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:

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

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

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

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

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

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. (Note the convenient coincidence of the phrase "applying a function" with that of "applying a hypothesis" or theorem.) 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" propositions-as-types logic of type theory is *constructive*. This means that it does not include certain classical principles, such as the law of excluded middle (LEM) or proof by contradiction, and others which depend on them, such as this instance of de Morgan's law.

Philosophically, constructive logic is so-called because it confines itself to constructions that can be carried out *effectively*, which is to

DRAFT

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 **??**). 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 **????** 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

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

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 $Q(x)$.

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

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

DRAFT

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:

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

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 $n \leqslant m$ 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:

$$(n \leqslant m) :\equiv \textstyle\sum_{(k:\mathbb{N})} (\mathrm{id}\, n + km).$$

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

$$(n < m) :\equiv \textstyle\sum_{(k:\mathbb{N})} (\mathrm{id}\, n + \mathsf{succ}(k)m)$$

or

$$(n < m) :\equiv (n \leqslant m) \times \neg(\mathrm{id}\, nm).$$

DRAFT

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 **????**).

There is also another interpretation of the type $\sum_{(x:A)} P(x)$. Since an inhabitant of it is an element $x : A$ together with a witness that $P(x)$ holds, instead of regarding $\sum_{(x:A)} P(x)$ as the proposition "there exists an $x : A$ such that $P(x)$", we can regard it as "the type of all elements $x : A$ such that $P(x)$", i.e. as a "subtype" of A.

We will return to this interpretation in **??**. For now, we note that it allows us to incorporate axioms into the definition of types as mathematical structures which we discussed in **??**. For example, suppose we want to define a **semigroup** to be a type A equipped with a binary operation $m : A \to A \to A$ (that is, a magma) and such that for all $x, y, z : A$ we have $m(x, m(y, z)) = m(m(x, y), z)$. This latter proposition is represented by the type

$$\prod_{(x,y,z:A)} m(x, m(y, z)) = m(m(x, y), z),$$

so the type of semigroups is

$$\mathsf{Semigroup} :\equiv \sum_{(A:\mathcal{U})} \sum_{(m:A\to A\to A)} \prod_{(x,y,z:A)} m(x, m(y, z)) = m(m(x, y), z),$$

i.e. the subtype of Magma consisting of the semigroups. From an inhabitant of Semigroup we can extract the carrier A, the operation m, and a witness of the axiom, by applying appropriate projections. We will return to this example in **??**.

Note also 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

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

We will return to this issue in **??**.

We have described 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 projection of the given inhabitant).

DRAFT

As a consequence of the proof-relevant nature of this logic, we may have "A if and only if B" (which, recall, means $(A \to B) \times (B \to A)$), and yet the types A and B exhibit different behavior. For instance, it is easy to verify that "$\mathbb{N}$ if and only if ", and yet clearly $\mathbb{N}$ and differ in important ways. The statement "$\mathbb{N}$ if and only if " tells us only that *when regarded as a mere proposition*, the type $\mathbb{N}$ represents the same proposition as (in this case, the true proposition). We sometimes express "A if and only if B" by saying that A and B are **logically equivalent**. This is to be distinguished from the stronger notion of *equivalence of types* to be introduced in **????**: although $\mathbb{N}$ and are logically equivalent, they are not equivalent types.

In **??** we will introduce a class of types called "mere propositions" for which equivalence and logical equivalence coincide. Using these types, we will introduce a modification to the above-described logic that is sometimes appropriate, in which the additional information contained in disjunctions and existentials is discarded.

Finally, we note that the propositions-as-types correspondence can be viewed in reverse, allowing us to regard any type A as a proposition, which we prove by exhibiting an element of A. Sometimes we will state this proposition as "A is **inhabited**". That is, when we say that A is inhabited, we mean that we have given a (particular) element of A, but that we are choosing not to give a name to that element. Similarly, to say that A is *not inhabited* is the same as to give an element of $\neg A$. In particular, the empty type $\mathbf{0}$ is obviously not inhabited, since $\neg\mathbf{0} \equiv (\mathbf{0} \to \mathbf{0})$ is inhabited by $\mathsf{id}_{\mathbf{0}}$.[10]

[10] This should not be confused with the statement that type theory is consistent, which is the *meta-theoretic* claim that it is not possible to obtain an element of $\mathbf{0}$ by following the rules of type theory.

2.12 Identity types

While the previous constructions can be seen as generalizations of standard set theoretic constructions, our way of handling identity seems to be specific to 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 "$\mathrm{id}\, ab$" 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 "$\mathrm{id}[A]ab$" to specify the type A. If we have an element of $\mathrm{id}[A]ab$, we may say that a and b are **equal**, or sometimes **propositionally equal** if we want to emphasize that this is different from the judgmental equality $a \equiv b$ discussed in **??**.

DRAFT

Just as we remarked in **??** 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 the type $\mathrm{id}\,ab$ from also including more information. Indeed, this is the cornerstone of the homotopical interpretation, where we regard witnesses of $\mathrm{id}\,ab$ 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 $\mathrm{id}\,ab$ 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 **??**; for now we focus on the basic rules for the identity type. Just like all the other types considered in this chapter, it will have rules for formation, introduction, elimination, and computation, which behave formally in exactly the same way.

The formation rule says that given a type $A : \mathcal{U}$ and two elements $a, b : A$, we can form the type $(\mathrm{id}[A]ab) : \mathcal{U}$ in the same universe. The basic way to construct an element of $\mathrm{id}\,ab$ is to know that a and b are the same. Thus, the introduction rule is a dependent function

$$\mathsf{refl} : \textstyle\prod_{(a:A)}(\mathrm{id}[A]aa)$$

called **reflexivity**, which says that every element of A is equal to itself (in a specified way). We regard refl_a as being the constant path at the point a.

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

The induction principle (i.e. the elimination rule) 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

$$f : \textstyle\prod_{(x,y:A)} \prod_{(p:\mathrm{id}[A]xy)} C(x) \to C(y)$$

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 be-

DRAFT

tween 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. Just as $\mathsf{rec}_{\mathbb{N}}$ gives a specified map $\mathbb{N} \to C$ for any other type C of a certain sort, indiscernability of identicals gives a specified map from $\mathrm{id}[A]xy$ to certain other reflexive, binary relations on A, namely those of the form $C(x) \to C(y)$ for some unary predicate $C(x)$. 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 this recursion principle to an induction principle, which not only considers maps out of $\mathrm{id}[A]xy$ 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.

2.13 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 **??**. It can be seen as stating that the family of identity types is freely generated by the elements of the form $\mathsf{refl}_x : \mathrm{id}\, xx$.

Path induction: Given a family

$$C : \textstyle\prod_{(x,y:A)} (\mathrm{id}[A]xy) \to \mathcal{U}$$

and a function

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

there is a function

$$f : \textstyle\prod_{(x,y:A)} \prod_{(p:\mathrm{id}[A]xy)} C(x,y,p)$$

such that

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

Note that just like the induction principles for products, coproducts, natural numbers, and so on, path induction allows us to define *specified* functions which exhibit appropriate computational behavior. Just as we have *the* function $f : \mathbb{N} \to C$ defined by recursion from $c_0 : C$ and $c_s : \mathbb{N} \to C \to C$, which moreover satisfies $f(0) \equiv c_0$ and $f(\mathsf{succ}(n)) \equiv c_s(n, f(n))$, we have *the* function $f : \prod_{(x,y:A)} \prod_{(p:\mathrm{id}[A]xy)} C(x,y,p)$ defined by path induction from $c : \prod_{(x:A)} C(x,x,\mathsf{refl}_x)$, which moreover satisfies $f(x,x,\mathsf{refl}_x) \equiv c(x)$.

DRAFT

To understand the meaning of 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 $\mathrm{id}\, xy$. 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 $p : \mathrm{id}\, xy$ 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 $p : \mathrm{id}\, xy$ between them, it suffices to consider all the cases where the elements are x, x and the path is $\mathsf{refl}_x : \mathrm{id}\, xx$. If we were viewing types just as sets, it would be unclear what this buys us, but since there may be many different identifications $p : \mathrm{id}\, xy$ between x and y, it makes sense to keep track of them in considering families over the type $\mathrm{id}[A]xy$. In **??** 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:

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

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 $p : \mathrm{id}\, ab$, path induction requires us 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 $p : \mathrm{id}\, ab$ 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 $\mathrm{id}[A]ax$ is generated by the element $\mathsf{refl}_a : \mathrm{id}\, aa$. As we show below, this second principle is equivalent to the first; it is just sometimes a more convenient formulation.

DRAFT

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

$$C : \prod_{(x:A)} (\mathrm{id}[A]ax) \to \mathcal{U}$$

and an element

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

Then we obtain a function

$$f : \prod_{(x:A)} \prod_{(p:\mathrm{id}\, ax)} C(x, p)$$

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 $\mathrm{id}[A]ax$, 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 $\mathsf{refl}_a : \mathrm{id}\, aa$.

Packaged as a function, based path induction becomes:

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

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 2.12. 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

DRAFT

says that the *family* of types $(\mathrm{id}[A]xy)$, 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 $\sum_{(x,y:A)}(\mathrm{id}\,xy)$), is inductively generated by the constant loops at each point x. As we will see in **??**, in homotopy theory the space corresponding to $\sum_{(x,y:A)}(\mathrm{id}\,xy)$ 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. The analogous fact is also true in type theory: we can prove by path induction on $p : x = y$ that $\mathrm{id}[\sum_{(x,y:A)}(\mathrm{id}\,xy)](x,y,p)(x,x,\mathsf{refl}_x)$.

Similarly, based path induction says that for a fixed $a : A$, the *family* of types $(\mathrm{id}[A]ay)$, 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 $\sum_{(y:A)}(\mathrm{id}\,ay)$) is contractible to the constant loop on the chosen point. Again, the corresponding fact is also true in type theory: we can prove by based path induction on $p : a = y$ that $\mathrm{id}[\sum_{(y:A)}(\mathrm{id}\,ay)](y,p)(a,\mathsf{refl}_a)$. Note also that according to the interpretation of Σ-types as subtypes mentioned in **??**, the type $\sum_{(y:A)}(\mathrm{id}\,ay)$ 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 path induction nor based path induction 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 $C : (\mathrm{id}[A]aa) \to \mathcal{U}$ 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 $p : \mathrm{id}\,xx$ may be not equal to reflexivity as an element of $(\mathrm{id}\,xx)$, but the pair (x,p) will nevertheless be equal to the pair (x,refl_x) as elements of $\sum_{(y:A)}(\mathrm{id}\,xy)$.

2.14 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 the based

DRAFT

path induction principle. Indeed, let us assume the premises of path induction:

$$C : \prod_{x,y:A} (\mathrm{id}[A]xy) \to \mathcal{U},$$
$$c : \prod_{x:A} C(x, x, \mathsf{refl}_x).$$

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

$$C' : \prod_{y:A} (\mathrm{id}[A]xy) \to \mathcal{U},$$
$$C' :\equiv C(x),$$
$$c' : C'(x, \mathsf{refl}_x),$$
$$c' :\equiv c(x).$$

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

$$g : \prod_{(y:A)} \prod_{(p:\mathrm{id}\ xy)} C'(y, p)$$

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

$$f : \textstyle\prod_{(x,y:A)} \prod_{(p:\mathrm{id}[A]xy)} C(x, y, p)$$

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

$$D : \prod_{x,y:A} (\mathrm{id}[A]xy) \to \mathcal{U},$$
$$D(x, y, p) :\equiv \prod_{C:\prod_{(z:A)} (\mathrm{id}[A]xz) \to \mathcal{U}} C(x, \mathsf{refl}_x) \to C(y, p).$$

DRAFT

Then we can construct the function

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

and hence using path induction obtain

$$f : \textstyle\prod_{(x,y:A)} \prod_{(p:\mathrm{id}[A]xy)} D(x, y, p)$$

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

$$f : \textstyle\prod_{(x,y:A)} \prod_{(p:\mathrm{id}[A]xy)} \prod_{(C:\prod_{(z:A)}(\mathrm{id}[A]xz)\to\mathcal{U})} C(x, \mathsf{refl}_x) \to C(y, p).$$

Now given $x : A$ and $p : \mathrm{id}[A]ax$, 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

$$\mathsf{ind}'_{=_A}(a, C, c, x, p) :\equiv \mathsf{ind}_{=_A}\big((\lambda x, y.\, \lambda p.\, \textstyle\prod_{(C:\prod_{(z:A)}(\mathrm{id}[A]xz)\to\mathcal{U})} C(x, \mathsf{refl}_x) \to C(y, p)),$$
$$(\lambda x.\, \lambda C.\, \lambda d.\, d), a, x, p, C, c\big)$$

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

$$D : \textstyle\prod_{(x,y:A)}(\mathrm{id}[A]xy) \to \mathcal{U}_{i+1}$$

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 **????**, and that these two principles imply $\mathsf{ind}'_{=_A}$ directly. We leave the details to the reader as **??**.

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.

Remark 2.13. We emphasize that despite having some unfamiliar features, propositional equality is *the* equality of mathematics in homotopy type theory. This distinction does not belong to judgmental equality, which is rather a metatheoretic feature of the rules of type theory. For instance, the associativity of addition for natural numbers proven

DRAFT

in **??** is a *propositional* equality, not a judgmental one. The same is true of the commutative law (**??**). Even the very simple commutativity $n + 1 = 1 + n$ is not a judgmental equality for a generic n (though it is judgmental for any specific n, e.g. $3 + 1 \equiv 1 + 3$, since both are judgmentally equal to 4 by the computation rules defining $+$). We can only prove such facts by using the identity type, since we can only apply the induction principle for $\mathbb{N}$ with a type as output (not a judgment).

2.15 Disequality

Finally, let us also say something about **disequality**, which is negation of equality:[11]

$$(x \neq_A y) :\equiv \neg(\mathrm{id}[A]xy).$$

If $x \neq y$, we say that x and y are **unequal** or **not equal**. Just like negation, disequality 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 of the classical law of double negation, see **??**.

Sometimes it is useful to phrase disequality in a positive way. For example, in **??** 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 $x \neq 0$.

[11] We use "inequality" to refer to $<$ and $\leq$. Also, 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.

2.16 Notes

The type theory presented here is a version of Martin-Löf's intuitionistic type theory [**????**], which itself is based on and influenced by the foundational work of Brouwer [**?**], Heyting [**?**], Scott [**?**], de Bruijn [**?**],Howard [**?**], Tait [**??**], and Lawvere [**?**]. Three principal variants of Martin-Löf's type theory underlie the NuPRL [**?**], Coq [**?**], and Agda [**?**] 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 [**?**], rather than the *extensional* version [**?**]. 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. From the homotopical perspective, extensional type theory confines itself to homotopically discrete sets (see **??**), whereas the intensional theory admits types with higher-dimensional structure. The NuPRL system [**?**] is extensional, whereas

DRAFT

both Coq [**?**] and Agda [**?**] 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)* [**?**], stating that any two proofs of equality are judgmentally equal, and *Axiom K* [**?**], 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 uniqueness principle (η-conversion) $f \equiv \lambda x.f(x)$, as a principle of judgmental equality. This principle is used, for example, in **??**, to show that univalence implies propositional function extensionality. Uniqueness principles are sometimes considered for other types. For instance, the uniqueness principle for cartesian products would be a judgmental version of the propositional equality uppt which we constructed in **??**, saying that $u \equiv (\mathsf{pr}_1(u), \mathsf{pr}_2(u))$. 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 uniqueness principle 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 **??**.

It is important for our purposes that (propositional) equality of functions is taken to be *extensional* (in a different sense than that used above!). This is not a consequence of the rules in this chapter; it will be expressed by **??**. This decision is significant for our purposes, because it specifies that equality of functions is as expected in mathematics. Although we include **??** as an axiom, it may be derived from the univalence axiom and the uniqueness principle for functions (see **??**), as well as from the existence of an interval type (see **??**).

Regarding inductive types such as products, Σ-types, coproducts, natural numbers, and so on (see **??**), there are additional choices regarding precisely how to 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 **??**. 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

DRAFT

matching justified in terms of induction.

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

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 [**?**], 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.

It is also not necessary that the universes be indexed by natural numbers and linearly ordered. For some purposes, it is more appropriate to assume only that every universe is an element of some larger universe, together with a "directedness" property that any two universes are jointly contained in some larger one. There are many other possible variations, such as including a universe "$\mathcal{U}_\omega$" that contains all $\mathcal{U}_i$ (or even higher "large cardinal" type universes), or by internalizing the hierarchy into a type family $\lambda i.\, \mathcal{U}_i$. The latter is in fact done in Agda.

The path induction principle for identity types was formulated by Martin-Löf [**?**]. The based path induction rule in the setting of Martin-Löf type theory is due to Paulin-Mohring [**?**]; it can be seen as an intensional generalization of the concept of "pointwise functionality" for hypothetical judgments from NuPRL [**?**, Section 8.1]. The fact that Martin-Löf's rule implies Paulin-Mohring's was proved by Streicher using Axiom K (see **??**), by Altenkirch and Goguen as in **??**, and finally by Hofmann without universes (as in **??**); see [**?**, §1.3 and Addendum].

DRAFT

Philosophical Niceties

ML Type Theory is centered (more or less) on one of the major logico-philosophical topics of the 20th century, namely the nature of assertion and its relation to propositions and inferences.

You don't have to understand the arcana of this debate in order to understand type theory (or HoTT), but some familiarity with the main outline is very helpful. Actually, I think it's essential, if you want to understand the HoTT Book's account of *judgement*, presented in HoTT Chapter 1 (reproduced below). Fortunately the presentation is relatively straightforward.

Remark 1. Stress: this is largely a philosophical issue, or perhaps an issue in Philosophy of Language. It's really about how our utterances come to have the significances they do.

Outline:

- Frege's elevation of *force* as essential
- Dealing with embedded (and therefore forceless) propositions
- Wittgenstein
- Dummett
- etc.
- Brandom's recent innovation: decompose "assertion" into "commitment" and "entitlement"

What the HoTT Book refers to as judgment (following ML) could also be called assertion. Brandom's account of the "fine structure" of assertion is very helpful here. Among other things, it provides a very simple explanation of how embedded propositions work. Embedded propositions are unasserted; the problem is how to reconcile this with the fact that they are function as assertions if unembedded. On Brandom's account, [todo...]

In other words, we can have commitment with or without entitlement, and vice-versa.

A set membership statement can be explained in terms of commitments and entitlements. A free occurance of e.g. $a \in A$ is ordinarily taken as an assertion (judgment). We can follow Frege and make this *force* explicit: $\vdash a \in A$. The problem with this, however, is that, in contemporary usage, this would make $a \in A$ *logically* true, which is not what we want. Instead we want a representation of committment to the proposition, as at least ordinarily true, without regard to its logical truth.

TODO: logical v. ordinary truth is pretty hairy for non-logicians so the distinction should be explicated.

In sum: the implicity sense of $a \in A$ is something like:

$$\exists\, \Gamma, a, A \mid \Gamma \vdash a \in A$$

DRAFT

Informally: there exists a set of propositions Γ, a value (or object) a, and a set A such that the propostion $a \in A$ is deducible from Γ.

So the meaning of $a \in A$ essentially involves existential quantification. It is a statement about the world, that it contains the relavant entities, not about the entities themselves.

Remark 2. Not quite; $a \in A$ is surely a statement about a, maybe also about A, no? But still there must be an implicity existential quantification over the propositions that entail the statement.

There is a logical subtlety here. $a \in A$ seems to be about a determinate a and a determinate A, but it isn't, not if we take it to be an existentially quantified statement. That's because $\exists a, A \mid a \in A$ does not pick out determinate individuals; it just says that *some* such individuals exist in the domain of interpretation. True, a and A are said to be bound by $\exists$, but that's not entirely accurate; quantified variables are not bound in the way that constant symbols like π or 0 are bound. Whatever we go on to say about a and A – e.g. $a \in A$ – remains within the scope of the quantifier, so it does not count as a statement about determinate individuals. It's a statement about the world, that it contains entities that satisfy the predicate.

On the other hand, the same seems to be true of $a : A$: though these symbols be bound, we don't know what they are bound to. They are not bound by an implicit existential quantifier; $a : A$ does *not* mean $\exists a, A \mid a : A$.

Remark 3. Plus, quantifiers have to be used with a predicate; strictly speaking, $\exists a, A$ is not a complete statement.

By contrast, the Type Theoretic analogue $a : A$ is a statement about a specific value and a specific type, without any quantification. It is not directly a statement about the world, but about part of the world. Or: it expresses both commitment and entitlement. That's why it cannot be embedded in e.g. "if $a : A$ then it is not the case that $b : B$". Embedded propositions cannot carry force, but $a : A$ always carries force, intrinsically, as it were.

3.1 a : A

Forms go from symbols to terms to sentences; from a to $a + b$ to $a + b =$.

The "judgment" $a : A$ is clearly a compound term, so it cannot merely name something. But is it a sentence? Does it denote a proposition? Or is it analogous to terms like $a + b$ which are names of a sort but involve some additional meaning beyond mere reference.

It seems it must involve a proposition, or let's say propositional content. We take $a : A$ as a statement of fact, rather than a mere reference to some part of the world. Then how is it distinct from $a \in A$?

DRAFT

The HoTT Book says it is "analogous" to the set-theoretic statement $a \in A$, but essentially different, since $a \in A$ is a proposition but $a : A$ is a judgment. It says that, *when working internally in type theory*, $a : A$ cannot be embedded, as in " if $a : A$ then it is not the case that $b : B$", nor can the judgment $a : A$ be disproved.

So let's look closely at what this means. Earlier, HoTT says that (some) judgments involving A "exist at a different level from the *proposition* A itself, which is an internal statement of the theory." (p. 18) There's a bit of circularity there; what is an "internal statement"?

TODO. The nature of "proposition" has been a topic of considerable debate. Review some of the alternative accounts on offer.

The basic idea seems to be based on the well-known concept that propositions by themselves are devoid of force, and must be asserted. HoTT seems to imply that judgments are asserted propositions – or more correctly, assertings of propositions.

This seems a little bit off. Assertion is something only people do. An inked form on a page cannot really be construed as an assertion. So we need to work out the mechanics of how a written form like $a : A$ can be viewed as a "judgment" in this sense. I think Brandom's model of assertion works. It would say, I think, that $a : A$ counts as a judgment (assertion) because by convention we agree to treat it that way, whereas we treat $a \in A$ slightly differently, because of the conventions elaborated by 20th century logic.

When HoTTB refers to "working internally in type theory", it seems, the idea is to consider propositions in isolation from their assertion. Assertion, on this view, is something that comes from outside of the world of propositions. This is perfectly in tune with the idea that asserting is something people do, but that what gets assert*ed* – the *content* of an assertion – is distinct from the assert*ing*.

Remark 4. Sellars called this the notorious -ing/-ed distinction.

This would seem to make $a : A$ an assert*ing*.

We can think of $(a : A)$ as a *given* proposition: one that, while unasserted, has the same force as a propositional assertion. Or another way to put it would be to say that use of $(a : A)$ is inalienably performative.

In fact $(a : A)$ corresponds nicely to a common linguistic practice, namely combining a proper name and a description, as in "Joan of Arc", "King George", or "Slick Willy". Or, more colloquially, "poor Tom", "angry Joe", or "Gimpel the fool". And the primitive nature of types can be clearly illustrated by analogy with the military. In type theory, every object has a type, just as everybody in the military has a rank. You cannot be in the military unless you have a rank. Within the military, the proper way refer to someone in the miliary is to combine rank and

DRAFT

name: General Custer, Sergeant York, Private Bilko. So the difference between $(a : A)$ and $a \in A$ is like the difference between "This is General Custer" and "This is Custer; he is a General".

On the other hand, "This is General Custer" doesn't look much like a *judgment*, although it does look like a *claim*. But not that it is not a claim about the meaning of "General Custer"; rather it is a claim about the relation between "This" and "General Custer". You could be wrong about the name or the rank of whomever you mean by "This", but you cannot be wrong about "General Custer"; that's just a qualified name. Being wrong in this sense about "This is General Custer" is an empirical matter; in type theory, the question of whether $(a : A)$("this is a-of-A") is correct or not never even arises. It doesn't make an empirical assertion, it states a *given*. Or we might say it gives a fact. By contrast, $a \in A$, as a proposition, may be either true or false; when we say "let $a \in A$", we implicitly stipulate that $a \in A$ is to be *assumed* to be true, but it is not *given* as true. In other words, we can gloss it as "$a \in A$ has a truth-value like any proposition, so it could be false, but please assume that it is true."

Another critical distinction: in standard set theory and logic, judgments come from the outside, as it were. But in HoTT, judgments of the form $(a : A)$are internal. They may be derivable inside the system (by production of a proof or witness.) In other words, inference in set theory comes from outside of the world of sets, but inference in HoTT is built in to the structure of types. Inference (construction) is part of the intrinsic meaning of types.

3.2 Justification

The HoTT Book's account of judgments in Chapter 1 section 1 seems to conflate the distinction Brandom makes between commitment and entitlement. "Informally, a deductive system is a collection of rules for deriving things called judgments." But derivation (proof) starts and ends in propositions; commitment is something else. The derivation or proof provide warrant for entitlement to the commitment - justification of the conclusion. So how would Brandom parse "judgment" as HoTT uses the term?

DRAFT