Homotopy Type Theory:
Univalent Foundations of Mathematics

University of Waterloo Graduate Course, Spring 2014

Cecilia Flori, Tobias Fritz
Perimeter Institute for Theoretical Physics

June 10, 2014

Contents

Topic 1: What is HoTT and why?

Introduction

Homotopy type theory (HoTT) is a newly emerging field of mathematics which is currently being developed as a foundation of mathematics which is in some sense closer to mathematical practice than conventional foundations are. Type theory—without the "homotopy" aspect—has a long history, to a large extent within computer science and the theory of programming languages. It is the base of computerized interactive theorem provers like `Coq` and `Agda`, which also serve as programming languages allowing *formal verification*: the writing of software together with proofs of its correctness. Homotopy type theory, which we will study, is one particularly interesting variant or incarnation of type theory, but there are many other type theories with various applications. At present, it seems plausible that homotopy type theory has not yet reached its final form and that there remain even better variants of type theory yet to be discovered. Please keep this in mind during the course.

We will soon briefly explain what some of the issues with conventional foundations, as in first-order logic plus set theory, are. Type theory does not only overcome these, but also has many other desirable features which we will become intimately familiar with:

- Proofs-as-programs and constructivism: every proof is necessarily *constructive*. This means that it can be regarded as an explicit algorithm which computes a certain object, using given data corresponding to the assumptions of the proven theorem. We will see many examples of this in later topics. Non-constructive axioms like the law of excluded middle $P \vee \neg P$ or the axiom of choice can still be used, but they have to be added as explicit assumptions.
- The distinction between logic and set theory has no analogue in type-theoretic foundations. Both logical propositions and sets are represented as types. Both proofs of propositions and elements of sets become elements of types. Optimally, type theory has no axioms, but only *inference rules*, sometimes called *deduction rules*, from which everything else follows.
- If two things are equal, they can be equal in many different ways. If two equalities between two things are equal, they can themselves be equal in many different ways, etc... This is quite specific to *homotopy* type theory, but may already be a familiar phenomenon from *homotopy theory*. In fact, homotopy theory and other geometric structures live at the most fundamental level: there is no need to define real numbers or topological spaces in order to do geometry! Homotopy type theory is the *internal logic* of homotopy theory:

homotopy theory + type theory = homotopy type theory = the theory of reasoning about homotopy types.

Finally, while elements of a set in conventional foundations do themselves have to be sets and therefore have a lot of internal structure, elements of a type have no structure beyond their relation with other elements. This is what we now discuss in some detail.

Oddities in conventional foundations

The conventional foundations of mathematics comprise *first-order logic* and *set theory*. Since the ZFC axioms of set theory were established in the early 20th century, these axioms have explicitly or implicitly been underlying almost all of mathematics. This should make it clear that conventional logic and set theory serve as a perfectly adequate foundation, and there is nothing wrong with using them as a "working mathematician". So before starting to discuss type theory in more detail, we would like to expose some of the weaknesses and dark corners of set theory.

The possibility to ask meaningless questions. In set-theoretic foundations, every mathematical object must be a set, and every other piece of mathematics is built on sets and the element relation between sets. In particular, this has to apply to things like numbers 3 or π, which are themselves sets. In particular, it is possible to ask questions of the form "is $3 \in \pi$?" However, if asked, most people would reply that this is a meaningless question. Nevertheless it is a well-formed question with an answer, and this answer depends on how we represent both 3 and π as sets.

Let us illustrate this with the simpler question "is $1 \in 3$?" To answer this question, we first of all have to represent the natural numbers as sets. There are various ways to do this, but two of the most common ones are:

1. Natural numbers as von Neumann ordinals (the standard definition):

$$0 \coloneqq \{\}, \quad 1 \coloneqq \{\{\}\}, \quad 2 \coloneqq \{\{\},\{\{\}\}\}, \quad 3 \coloneqq \{\{\},\{\{\}\},\{\{\},\{\{\}\}\}\}, \quad \ldots, \quad n+1 \coloneqq \{0,\ldots,n\}.$$

   If now we ask whether $1 \in 3$, then the answer is yes!

2. Natural numbers as singleton sets:

$$0 \coloneqq \{\}, \quad 1 \coloneqq \{\{\}\}, \quad 2 \coloneqq \{\{\{\}\}\}, \quad 3 \coloneqq \{\{\{\{\}\}\}\}, \quad \ldots, \quad n+1 \coloneqq \{n\}.$$

   If now we ask whether $1 \in 3$, then the answer is no!

As this simple example shows, it is an awkward requirement of set-theoretic foundations that every mathematical object needs to be a set all of whose elements again need to be sets. This causes mathematical objects like numbers to have more internal structure than they really should have!

How to define Cartesian products? Imagine you are given two sets A and B and you need to construct their cartesian product $A \times B$. Clearly, this should be the set of all pairs of elements of A and B,

$$A \times B \coloneqq \{(x,y) \mid x \in A,\ y \in B\}\,.$$

But now the question is, *what is a pair*? Again, every mathematical object needs to be a set, and the same applies to a pair (x,y). The obvious definition $(x,y) \coloneqq \{x,y\}$ does not work, since this would give rise to $(x,y) = (y,x)$. Again there are various ways to actually achieve such a representation, for example

1. $$(x,y) \coloneqq \{\{x\},\{x,y\}\},$$

2. $$(x,y) \coloneqq \{x,\{x,y\}\},$$

3. $$(x,y) \coloneqq \{\{0,x\},\{1,y\}\},$$

where 0 and 1 again need to be defined as sets, for example using the natural numbers above. Again, it is unclear why any of these definitions should be better than the others; or for that matter, it seems inappropriate for such a pair to have any internal structure at all, other than its two constituents x and y.

Equality in set-theoretic foundations. In set theory, two sets are equal if and only if they have the same elements:

$$\forall x (x \in A \Leftrightarrow x \in B) \quad \Longleftrightarrow \quad A = B.$$

While this is the *axiom of extensionality*, it can also be informally understood as a definition of equality in set theory. Although quite intuitive, this notion of equality is too strict for many purposes: for example, take two groups that are isomorphic, like the permutation group on a two-element set and the group of numbers $\{-1,+1\}$ with respect to multiplication. These are not equal as sets, and are therefore also not equal as groups, although they share exactly the same group-theoretical properties. Again, the issue is that these groups have more internal structure than is relevant in a group-theoretical context.

The situation in type-theoretic foundations

We now turn to discussing how the type-theoretic approach fares in each of these examples and then will continue with an outline of other features of (homotopy) type theory.

The central concept in any type theory is the concept of *type*. Types may be familiar from most programming languages, in which every variable has a type. Concerning foundations of mathematics, types can be seen as the analogues of sets, but also as the analogoues of propositions. In HoTT, there are also yet other types that neither look like a set nor like a proposition; in general, a type behaves like a space having certain geometrical properties not possessed by sets or propositions.

Whereas sets are defined in terms of the internal structure given by their elements, the types in type theory have a much more 'operational' flavour. The relevant operational rules are typically these:

1. Formation rule: what one needs to have in order to construct the type.
2. Introduction rule: how to construct elements of the type.
3. Elimination rule: how to construct functions out of the type.
4. Computation rule: what happens when applying a function constructed via the elimination rule to an element constructed via the introduction rule.

Natural numbers. The collection of natural numbers forms a type which we denote by $\mathbb{N}$. The four rules now take the concrete form:

1. Formation: $\mathbb{N}$ is a type; no premises are needed.
2. Introduction: $0 : \mathbb{N}$ is an element which can be constructed without any premises; and for any $n : \mathbb{N}$, there is another element $s(n) : \mathbb{N}$, which represents the successor of n.
3. Elimination: From any type A, element $x_0 \in A$ and function $f : \mathbb{N} \times A \to A$, we can construct a function $\mathrm{ind}_{\mathbb{N}}(A, x_0, f) : \mathbb{N} \to A$ by making the recursive definition
$$\mathrm{ind}_{\mathbb{N}}(A, x_0, f)(0) :\equiv x_0, \qquad \mathrm{ind}_{\mathbb{N}}(A, x_0, f)(s(n)) :\equiv f(n, \mathrm{ind}_{\mathbb{N}}(A, x_0, f)(n)).$$
4. Computation: Evaluating the function $\mathrm{ind}_{\mathbb{N}}(A, x_0, f)$ on 0 gives x_0; evaluating it on $s(x)$ for some $x : \mathbb{N}$ gives $f(\mathrm{ind}_{\mathbb{N}}(A, x_0, f)(x))$.

We will later meet more powerful versions of the elimination and computation rules. Together with the standard inference rules of type theory, these rules are enough for obtaining all definitions and theorems of basic arithmetic in pretty much the usual way. (See `Coq` file.)

For example, the factorial can be constructed as

$$0! :\equiv 1, \qquad (n+1)! :\equiv (n+1) \cdot n!,$$

which means that the factorial is given by $\mathrm{ind}_{\mathbb{N}}(\mathbb{N}, 1, f)$ for $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ being given by $f(n, m) :\equiv (n+1) \cdot m$.

In this formalism, a natural number does not have any internal structure; the mathematics resides in its *relation* to the other numbers. This is why meaningless questions like "$1 \in 3$?" cannot be asked here.

Cartesian products. Although Cartesian products are very different from natural numbers, the type-theoretic rules turn out to be of the same four kinds:

1. Formation: For every types A and B one can form the Cartesian product $A \times B$.
2. Introduction: From every $x : A$ and $y : B$ one obtains $(x, y) : A \times B$.
3. Elimination: A function $f : A \times B \to C$ can be constructed from having an element $c : C$ assigned to every $a : A$ and $b : B$.
4. Computation: Applying this function f to an explicit pair (a, b) results in the associated $c : C$.

As an example of the elimination rule, we can construct the projection $A \times B \to A$: to any $a : A$ and $b : B$, we can trivially associate $a : A$, and the elimination rule constructs a function $A \times B \to A$.

Propositional equality. In many flavours of type theory, one can consider equality between elements $x, y : A$. However, such an equality $x = y$ is itself a type! Its elements $p : x = y$ correspond to the proofs of the equality. This can be iterated: for two $p, q : x = y$, we can consider the type $p = q$, etc... Again, this so-called *identity type* satisfies variants of the four familiar rules:

1. Formation: To every $x, y : A$, one can associate the identity type $x = y$.
2. Introduction: For every $x : A$, one can construct $\mathrm{refl}_x : x = x$, a canonical proof that x is equal to itself.
3. Elimination: One obtains a *family* of maps $\prod_{x,y:A}(x = y \to B)$ if one has a family of elements in $\prod_{x \in A} B$, corresponding to the images of the refl_x.

4. Computation: If such a family of maps is evaluated on some refl_x, it returns exactly the specified value there.

Again, we will later on meet a more powerful variant of the elimination and computation rules. To see how these work, suppose that we want to show that $x = y$ implies that $y = x$, i.e. we want to obtain a family of maps

$$\prod_{x,y:A} (x = y \to y = x) \tag{1.1}$$

By the elimination rule, it suffices to define these maps on $\mathrm{refl}_x : x = x$ for every $x : A$. On these, the codomain is $x = x$ as well, and we can choose $\mathrm{refl}_x : x = x$ as its own image.

Besides its inference rules, HoTT in its present form has essentially *only one* axiom, which states roughly that isomorphic objects are actually equal:

1.1 Axiom (Univalence Axiom, informal weak version)**.**

$$A \simeq B \longrightarrow A = B.$$

We will see that the univalence axiom has a large number of important consequences. The hope is that even this axiom may become redundant after a suitable improvement of the inference rules, but this has not yet been realized.

The circle. The identity type which we just discussed is among the most fundamental concepts in HoTT. This is no different from conventional foundations, where equality is likewise a very fundamental concept, even if it behaves quite differently. As another example of this difference, we show how identity types are relevant for dealing with types which cannot be represented as sets, i.e. types which form non-trivial spaces.

As we will see, there is a type S^1 obeying the following rules:

1. Formation: S^1 is a type; no premises are needed.
2. Introduction: There is an element $\mathrm{base} : S^1$ representing a base point and an identity

$$\mathrm{loop} : \mathrm{base} = \mathrm{base}$$

   representing a path from that base point to itself.
3. Elimination: A function $f : S^1 \to A$ is constructed by specifying that the base point gets mapped to some $x : A$ and the loop to some $p : x = x$.

$$f(\mathrm{base}) :\equiv x, \quad f(\mathrm{loop}) :\equiv p.$$

4. Computation: Upon applying this f to base or to loop, it returns x and p, respectively (modulo some subtleties that will be discussed in due time).

Topic 2: Typed lambda calculus

Some theorems in Coq

Before we start the formal development of type theory, it should be helpful to have some ideas of what the upcoming type-theoretic rules are going to mean. Consider the following examples of code in the interactive theorem prover Coq:

```
(* Lawvere's fixed point theorem in Coq *)

Lemma eq_app (A B : Type) (f1 f2 : A → B) (a:A) : (f1 = f2) → (f1 a = f2 a).
Proof.
        intro assumption.
        rewrite assumption.
        reflexivity.
Qed.

Theorem Lawvere (X Y : Type) (f : X → X → Y) : (forall g : X → Y, exists x : X, g =
    f x) → (forall h : Y → Y, exists y : Y, h y = y).
Proof.
        intro assumption.
        intro h.
        specialize (assumption (fun x ⇒ h (f x x))).
        destruct assumption.
        exists (f x x).
        apply (eq_app X Y (fun x ⇒ h (f x x)) (f x) x).
        apply H.
Qed.
```

The lemma states the following: given that A and B are taken to be types, $f1$ and $f2$ functions $A \to B$ and a an element of A, then an equality $f1 = f2$ implies an equality $f1(a) = f2(a)$. The theorem can be parsed in a similar way.

So what exactly is going on here? What do these symbols and keywords stand for? And what are the inference rules that have been applied in order to prove this theorem?

The answer is that Coq is based on a type theory called the Calculus of Constructions. We will not study this exact type theory in this course, so suffice it to say that it has many features in common with HoTT. We now start our formal developments with the introduction of two very simple type theories: a trivial one containing just a bare minimum of structure, and then we expand it to the simply typed lambda calculus in which one can reason about functions. The basic structures occurring in these type theories will be just as in the above Coq proofs, so it will help to keep in mind how these are structured.

HoTT will be an extension of these two very basic type theories.

Trivial type theory

In this basic (and completely inexpressive) type theory—and also in all the upcoming ones—there are five kinds of *judgements*:

$$\Gamma\ \mathtt{ctx}, \qquad \Gamma \vdash A : \mathtt{Type}, \qquad \Gamma \vdash a : A, \tag{2.1}$$

$$\Gamma \vdash A \equiv B : \mathtt{Type}, \qquad \Gamma \vdash a \equiv b : A. \tag{2.2}$$

The symbol '$\equiv$' is called *judgemental equality.* Note that a judgemental equality statement $A \equiv B : \mathtt{Type}$, does not mean that $A \equiv B$ is a type; it rather indicates that A and B are judgementally equal *as types.* Similarly, $a \equiv b : A$ indicates that a and b are judgementally equal as elements of A.

The first kind of judgement expresses that the expression Γ is a well-formed *context.* In the above `Coq` examples, the context is given by all the stuff in brackets between the name of the statement and the colon separating it from the actual statement; a context represents a list of assumptions which can then be used in other judgements, in which Γ stands for a context. The turnstile symbol '$\vdash$' is shorthand for the claim

In the list of assumptions given by Γ, the following is true: ...

and is denoted by an unbracketed colon in `Coq`. The stuff to the right of Γ represents the actual statement. The judgements in the second line are about *judgemental equality*; the reason to use the funny notation '$\equiv$' for judgemental equality is that we will later encounter another kind of equality denoted '$=$'. Roughly speaking, judgemental equality states that two things are 'trivially equal' or 'equal by definition'.

Note that all of these explanations are intuitive interpretations of what the different judgements mean; we have not yet specified any formal rules on how to create or manipulate judgements. This is what we do now:

1. Context formation:

   The first context formation rule looks very trivial:

$$\frac{}{\cdot\ \mathtt{ctx}}$$

   There is also a rule to construct less trivial contexts:

$$\frac{(x_1 : A_1, \ldots, x_n : A_n)\ \mathtt{ctx}}{(x_1 : A_1, \ldots, x_n : A_n, B : \mathtt{Type})\ \mathtt{ctx}}$$

   In each of these two rules, the stuff above the horizontal bar is a list of judgements which one already needs to have derived in order to formally conclude the new judgement below the bar. In this sense, we have written down an *inference rule.* In the first rule, this list is empty, while below the bar, '$\cdot$' stands for the empty expression; so this rule says that the empty expression is always a context.

   In the second rule, B needs to be a new variable symbol distinct from the one that are already present; in the above `Coq` examples, this means that each new variable symbol in the context must be assigned a different name.

   From now on, we will be sloppy about the use of brackets, and not distinguish between $x : A, y : B$ and $(x : A, y : B)$ and $(x : A)\ (y : B)$. As in the above examples, we can also abbreviate $(a : A, b : A)$ to $a, b : A$.

   The remaining context formation rule is this:

$$\frac{x_1 : A_1, \ldots, x_{n-1} : A_{n-1} \vdash A_n : \mathtt{Type}}{(x_1 : A_1, \ldots, x_n : A_n)\ \mathtt{ctx}}$$

   Here, A_n is an expression which may in principle contain the variables $x_1, \ldots, x_{n-1}$. However, in the basic type theories that we discuss today, there are no rules which one could apply in order to obtain such a *dependent type.*

2. Assumptions are derivable:

$$\frac{(x_1 : A_1, \ldots, x_n : A_n)\ \mathtt{ctx}}{x_1 : A_1, \ldots, x_n : A_n \vdash x_i : A_i}$$

   This inference rule says that if one has a context which consists of a list of variables of certain types, then one can derive that each variable x_i has the type which is assumed in the context. Since there are no other ways for obtaining contexts other than applying the above context formation rules, it is guaranteed that every context is simply a list of typing assumptions, so that the present rule is always applicable.

3. Judgemental equality rules:

$$\frac{\Gamma \vdash A : \mathtt{Type}}{\Gamma \vdash A \equiv A : \mathtt{Type}} \qquad \frac{\Gamma \vdash A \equiv B : \mathtt{Type}}{\Gamma \vdash B \equiv A : \mathtt{Type}} \qquad \frac{\Gamma \vdash A \equiv B : \mathtt{Type} \quad \Gamma \vdash B \equiv C : \mathtt{Type}}{\Gamma \vdash A \equiv C : \mathtt{Type}}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a \equiv a : A} \qquad \frac{\Gamma \vdash a \equiv b : A}{\Gamma \vdash b \equiv a : A} \qquad \frac{\Gamma \vdash a \equiv b : A \quad \Gamma \vdash b \equiv c : A}{\Gamma \vdash a \equiv c : A} \quad .$$

   These rules state that judgemental equality is an equivalence relation, both on types and on elements of types.

4. Judgemental equality preserves judgements:

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A \equiv B : \mathtt{Type}}{\Gamma \vdash a : B} \qquad \frac{\Gamma \vdash a \equiv b : A \quad \Gamma \vdash A \equiv B : \mathtt{Type}}{\Gamma \vdash a \equiv b : B}$$

As already mentioned, these inference rules represent derivations of new judgements from already known judgements. Proving any sort of non-trivial statement requires many applications of inference rules; one can visualize these in a *proof tree*:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot\ \mathtt{ctx}}}{(A : \mathtt{Type})\ \mathtt{ctx}}}{A : \mathtt{Type} \vdash A : \mathtt{Type}}}{\vdash (A : \mathtt{Type}, a : A)\ \mathtt{ctx}}}{A : \mathtt{Type}, a : A \vdash a : A}}{A : \mathtt{Type}, a : A \vdash a \equiv a : A} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot\ \mathtt{ctx}}}{(A : \mathtt{Type})\ \mathtt{ctx}}}{A : \mathtt{Type} \vdash A : \mathtt{Type}}}{\vdash (A : \mathtt{Type}, a : A)\ \mathtt{ctx}}}{A : \mathtt{Type}, a : A \vdash A : \mathtt{Type}}}{A : \mathtt{Type}, a : A \vdash A \equiv A : \mathtt{Type}}}{A : \mathtt{Type}, a : A \vdash a \equiv a : A}$$

What has happened here is that we have derived the simple judgement $A : \mathtt{Type}, a : A \vdash a \equiv a : A$ using the inference rules of our trivial type theory. Each horizontal bar stands for an application of one inference rule. Note that we already obtained the target statement on the left branch, but we can nevertheless derive it again, even making use of itself. Also, note that the left and the right branch actually coincide in the upper part.

Additional rules for simply typed lambda calculus

As one may imagine, our trivial type theory has very low expressivity in the sense that no interesting judgements can be derived. Over the following lectures, we will successively enhance it by adding more and more structure. We now starting to do this by adding *function types*. A function type represents the collection of all functions with given domain and codomain types. This allows us to talk about functions in our type theory, which thereby becomes the *simply typed lambda calculus*.

The *type former* for function types is denoted '$\to$'. The inference rules for function types are as follows:

1. Formation rules:

$$\frac{\Gamma \vdash A : \mathtt{Type} \qquad \Gamma \vdash B : \mathtt{Type}}{\Gamma \vdash A \to B : \mathtt{Type}} \qquad \frac{\Gamma \vdash A \equiv A' : \mathtt{Type} \qquad \Gamma \vdash B \equiv B' : \mathtt{Type}}{\Gamma \vdash (A \to B) \equiv (A' \to B') : \mathtt{Type}}$$

   This rule tells us that for every two types A and B, one can *form* the function type $A \to B$. In the expression $A \to B$, the arrow '$\to$' is a new symbol which has not been part of our trivial type theory. We also learn that two functions types $A \to B$ and $A' \to B'$ become judgementally equal as soon as their domain and codomain are.

   It is interesting to note that the usual notation for a function $f : A \to B$ can now be understood to be part of a typing judgement.

2. Introduction rules:

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda(x : A).b : A \to B} \qquad \frac{\Gamma \vdash A \equiv A' : \mathtt{Type} \qquad \Gamma \vdash B \equiv B' : \mathtt{Type} \qquad \Gamma, x : A \vdash b \equiv b' : B}{\Gamma \vdash \lambda(x : A).b \equiv \lambda(x : A').b' : A \to B}$$

   The first rule tells us that if we have an expression b of type B which may possibly depend on $x : A$ (and on the other variables in Γ), then we may *introduce* a function $A \to B$ by regarding the expression b as a function of x. Constructing a function in this way is called *lambda abstraction.* We will usually abbreviate $\lambda(x : A).b$ to $\lambda x.b$, since the type of x is clear from the context.

   The second rule states that this construction preserves judgemental equality in all its arguments.

3. Elimination rules:

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B}$$

This rules tells us how to apply a function. Since there is no function in the judgement below the bar, one can say that the function has been *eliminated.* Again, there is another rule stating that this function elimination preserves equality judgements:

$$\frac{\Gamma \vdash A \equiv A' : \mathtt{Type} \qquad \Gamma \vdash B \equiv B' : \mathtt{Type} \qquad \Gamma \vdash f \equiv f' : A \to B \qquad \Gamma \vdash a \equiv a' : A}{\Gamma \vdash f(a) \equiv f'(a') : B}$$

4. Computation rules:

If b is an expression containing a variable x and a is another expression, then we write $b[a/x]$ for the new expression in which a has been substituted for x.

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

This rule tells us how the introduction rule and the elimination rule interact: if one has a function obtained by lambda abstraction from an expression b and applies this function to an expression a, then the result is the same 'on the nose' as replacing every occurrence of the function variable x by a. Applying the computation rule in this way is also known as *β-conversion.*

5. Uniqueness principle:

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash f \equiv (\lambda x.f(x)) : A \to B}$$

This principle tells us that any function is equal to evaluating that function on a variable and then considering the resulting expression as a function in that variable. This rule is also known as *η-conversion.*

As an example of how to apply these rules of inference, we now construct the composition of functions, which is the judgement

$$A, B, C : \mathtt{Type}, f : A \to B, g : B \to C \vdash \lambda x.g(f(x)) : A \to C. \tag{2.3}$$

In words: if A, B, C are types, and we have function variables $f : A \to B$ and $g : B \to C$, then we obtain a function $\lambda x.g(f(x)) : A \to C$, which we call the *composition* of f and g.

If we write Γ for the context in (2.3), then that this Γ is indeed a context can be seen as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{A,B,C:\mathtt{Type} \vdash A:\mathtt{Type}} \qquad \dfrac{\vdots}{A,B,C:\mathtt{Type} \vdash B:\mathtt{Type}}}{A,B,C:\mathtt{Type} \vdash A \to B:\mathtt{Type}}}{(A,B,C:\mathtt{Type}, f:A \to B)\ \mathtt{ctx}}}{A,B,C:\mathtt{Type}, f:A\to B \vdash B:\mathtt{Type}} \qquad \dfrac{\vdots}{A,B,C:\mathtt{Type}, f:A\to B \vdash C:\mathtt{Type}}}{\dfrac{A,B,C:\mathtt{Type}, f:A\to B \vdash B \to C:\mathtt{Type}}{(A,B,C:\mathtt{Type}, f:A\to B, g:B\to C)\ \mathtt{ctx}}}$$

Here, we have abbreviated $(A : \mathtt{Type}, B : \mathtt{Type}, C : \mathtt{Type})$ to $A, B, C : \mathtt{Type}$ and already omitted those parts of the proof tree which are completely trivial. The right branch is the same as the left branch, so also that has been omitted. If one continues all branches all the way up, one will find that each branch eventually ends with the empty context, as required for a valid derivation of a judgement.

The proof tree for (2.3) itself, i.e. for the construction of function composition, now looks like this:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma\ \mathtt{ctx}}}{\Gamma \vdash A:\mathtt{Type}}}{(\Gamma, x:A)\ \mathtt{ctx}}}{\Gamma, x:A \vdash g:B\to C} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma\ \mathtt{ctx}}}{\Gamma \vdash A:\mathtt{Type}}}{(\Gamma, x:A)\ \mathtt{ctx}}}{\Gamma, x:A \vdash f:A\to B} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma\ \mathtt{ctx}}}{\Gamma \vdash A:\mathtt{Type}}}{(\Gamma, x:A)\ \mathtt{ctx}}}{\Gamma, x:A \vdash x:A}}{\Gamma, x:A \vdash f(x):B}}{\dfrac{\Gamma, x:A \vdash g(f(x)):C}{\Gamma \vdash \lambda x.g(f(x)):A\to C}}$$

We already showed that Γ `ctx` is a valid judgement. So in order to complete the branches of this proof tree, we can simply stick a copy of the above proof tree for Γ `ctx` onto each leaf.

One can also consider this function composition itself as a function of f and g:

$$A, B, C : \texttt{Type} \vdash \lambda f.\lambda g.\lambda x.g(f(x)) : (A \to B) \to ((B \to C) \to (A \to C))$$

We omit the proof tree for this statement.

Higher function types. As we have just seen, it is frequently useful to form function types of function types. As a more basic example of this, consider $f : A \to (B \to C)$; this says that f is a function which takes an argument $x : A$ and returns a function which takes an argument $y : B$ and returns some value $f(x)(y) : C$. One usually abbreviate this $f(x)(y)$ to $f(x, y)$,

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

and understands it as a function with two arguments. Since these kinds of functions are much more frequent than the ones of type $(A \to B) \to C$, one typically omits the brackets in $A \to (B \to C)$ and simply writes $A \to B \to C$.

Some metatheoretical properties

As the above `Coq` example might illustrate, the most important kind of judgements are the *typing judgements* $\Gamma \vdash x : A$. We are now going to discuss some *metatheoretical* properties of typing judgements. This refers to statements *about* judgements which cannot be formulated as judgements themselves, so that they live outside of the theory rather than inside it. We will not develop this metatheory in any detail and not prove any of the following remarks, but it is nevertheless very important to be aware of these considerations and results of this kind.

Rigorous theorems about some of the metatheoretical properties which we discuss can be found here:

Gilles Barthe, Thierry Coquand: *An Introduction to Dependent Type Theory.*

Do not try reading this at this stage—unless you already know a lot more type theory than we have introduced so far!

Adding common-sense inference rules. Besides the inference rules which we listed above, there are many others which can be derived from them by combining them into partial proof trees. There are yet other which cannot be derived from them, but they nevertheless seem so obvious and 'common sense' that it would be extremely strange if one were not allowed to use them. Take for example the *substitution rule*:

$$\frac{\Gamma \vdash a : A \quad \Gamma, x : A, \Delta \vdash b : B}{\Gamma, \Delta[a/x] \vdash b[a/x] : B[a/x]}$$

Here, also Δ is an arbitrary expression playing the role of a list of typing assumptions so that $(\Gamma, x : A, \Delta)$ is a context. The notation '$\Delta[a/x]$' stands for the same expression Δ, with every occurrence of the variable x replaced by the expression a; similarly for $b[a/x]$ and $B[a/x]$. Intuitively, this rule tells us that we can always substitute a free variable of type A by any expression of type A which can be formed in the present context. As soon as we switch to a more informal mode of reasoning, we will make judicious use of this rule and similar ones.

Assuming this additional inference rule modifies the type theory in the sense that this rule cannot be derived from the other ones. Nevertheless, it is not necessary for the derivation of any judgement: any judgement which can be derived using the substitution rule can also be derived without it. For this reason, there is no harm in using it, although it is—strictly speaking—not part of our type theory. All of this is very reminiscent of the cut elimination theorem in sequent calculus.

There are other rules of a similar flavor which can be assumed to hold, but which are not necessary for the derivation of any judgement. As soon as we switch to a more informal mode of reasoning in order to do actual mathematics within type theory, we will become guilty of secretly using such rules without making them explicit.

Every expression has at most one type. Type theories may or may not have the property that every expression in a given context has at most one type. More precisely, what we mean by this property is that if $\Gamma \vdash x : A$ and $\Gamma \vdash x : B$, then $\Gamma \vdash A \equiv B : \texttt{Type}$. It can be shown that the two type theories we have been setting up in this lecture do have this property, and most other type theories also enjoy this property at least in some approximate sense.

On the other hand, there are also expressions that are *ill-typed*: an expression x is ill-typed in a context Γ if there exists no A such that $\Gamma \vdash x : A$. Some expressions are ill-typed in any context. For example, there is no way to assign a type to the expression $f(f)$: in order to apply f, its type must be a function type $f : A \to B$; but then, its argument would have to be of type A, which it is not in this particular case.

As soon as one considers a type theory with *universes*—as we will do later on—becomes more subtle. Typically, while it does still hold in a certain approximate or intuitive sense, it is technically violated. We will learn more about this soon.

Judgemental equality judgements imply typing judgements. Any sensible type theory should have the property that if $\Gamma \vdash a \equiv b : A$, then $\Gamma \vdash a : A$ and $\Gamma \vdash b : A$. This is indeed the case for the type theories that we have been looking at so far, and will also be true in all the extensions that we will be considering.

Contexts in judgements are contexts. A similar consistency property is that if $\Gamma \vdash x : A$ is a typing judgement, then necessarily Γ `ctx`. Also, if $\Gamma \vdash x \equiv y : A$, then Γ `ctx` as well. We also have that if $(\Gamma, x : A)$ `ctx`, then also $\Gamma \vdash A : \texttt{Type}$, since the latter is the only way to obtain that $(\Gamma, x : A)$ `ctx`. This property is important for the above introduction and elimination rules for function types: strictly speaking, the introduction rule should be written as

$$\frac{\Gamma \vdash A : \texttt{Type} \qquad \Gamma \vdash B : \texttt{Type} \qquad \Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda(x : A).b : A \to B}$$

But the metatheoretical properties discussed here guarantee that $\Gamma \vdash A : \texttt{Type}$ and $\Gamma \vdash B : \texttt{Type}$ are valid judgements as soon as $\Gamma, x : A \vdash b : B$ is so.

Type checking: typing judgements have canonical proofs. The notion of *type checking* may be familiar from programming languages which have types: it is a verification guaranteeing that a certain program, which for example takes a number and outputs a string, does indeed comply with this specification. For us, type checking means determining whether an alleged typing judgement $\Gamma \vdash x : A$ is indeed valid or not.

In other words, type checking asks whether there exists a proof tree showing that the given $\Gamma \vdash x : A$ is indeed a valid judgement. In the simply type lambda calculus, and also in many other type theories, there is a simple algorithm to do that which can be illustrated by example as follows. In the judgement,

$$A, B, C : \texttt{Type} \vdash \lambda f.\lambda g.\lambda x.g(f(x)) : (A \to B) \to ((B \to C) \to (A \to C))$$

the 'outermost' part of the expression which is to be type-checked is λf; therefore, any derivation of this judgement must necessarily use the introduction rule for functions as its final step, since this is the only rule which creates lambda abstractions. Before applying this step, we therefore are dealing with the alleged judgement

$$A, B, C : \texttt{Type}, f : A \to B \vdash \lambda g.\lambda x.g(f(x)) : (B \to C) \to (A \to C)$$

which remains to be type-checked. The same argument can be applied two more times, and we then need to check whether

$$A, B, C : \texttt{Type}, f : A \to B, g : B \to C, x : A \vdash g(f(x)) : C$$

is a valid judgement. But even from here, we can continue in a similar way: the only way to obtain a function application in an expression is by using the elimination rule for functions. Continuing up in this way, one can reconstruct the entire proof tree.

So, at least intuitively, the expression x in a typing judgement $\Gamma \vdash x : A$ contains complete information about its entire proof tree! (Again, there are some subtleties here, since the proof tree for a typing judgement is not strictly unique.)

Topic 3: Propositions as types

Propositions as types

We have seen that the main mathematical objects in a type theory are types. But remember that in conventional foundations, as based on logic and set theory, there are not only sets, but also logical propositions! So what are propositions in type-theoretic foundations? The short answer: propositions are types as well!

But wait, then what are supposed to be the elements of these types? Again, the short answer: they are the *proofs* of that proposition! In particular, in type theory, proofs are not special objects, but on par with perhaps more familiar mathematical objects such as numbers, functions etc. Also, proofs matter: as we will see, the inference rule

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash x : A}{\Gamma \vdash f(x) : B} \tag{3.1}$$

can be understood as follows, in case that A and B play the role of propositions: if f proves that A implies B and a proves that A, then $f(a)$ proves that B. So a proof in the sense of an element $x : A$ can be seen as a witness of the correctness of the proposition A.

So now if proofs are elements of propositions and propositions are types, then what are the logical connectives? Equation (3.1) may give some idea: they correspond to *type formers* like '→'. If A and B are types which represent propositions, then the function $A \to B$ represents the proposition 'A implies B'. Thinking of elements as proofs, this makes perfect sense: the possible proofs of an implication $A \to B$ corresponds then to all the possible functions which turn a proof of A into a proof of B.

Definitions vs theorems. An intriguing aspect of the propositions-as-types correspondence is that the difference between definitions and proofs of theorems ceases to exist; which judgements should be considered as definitions and which ones as theorems or lemmas is an issue of exposition which only matters when we explain our mathematics to our fellow mathematicians.

For example in `Coq`, it is possible to define function composition through a proof:

```
(* Function composition *)

Definition compose (A B C : Type) : (A → B) → (B → C) → (A → C).
Proof.
        intros f g x.
        apply g.
        apply f.
        exact x.
Defined.
```

What is going on here is that the individual steps in the proof are a different way of writing down the individual parts of the expression $\lambda f.\lambda g.\lambda x.g(f(x))$. On the other hand, we can also interpret $(A \to B) \to (B \to C) \to (A \to C)$ as the statement that if A implies B and B implies C, then A implies C. This is a theorem to which the exact same proof applies:

```
(* Transitivity of implication *)

Theorem trans (A B C : Type) : (A → B) → (B → C) → (A → C).
Proof.
```

```
        intros f g x.
        apply g.
        apply f.
        exact x.
Qed.
```

An important aspect of this unification of definitions and proofs is that the style of mathematics in HoTT is at times quite different. In particular, HoTT is *proof-relevant*: often, it not only matters *that* one has a proof of some theorem, but it is also relevant *what* that proof is. For example, a proof of some theorem may be referenced later as part of the *statement* of another theorem: the above example of the proof of transitivity of implication illustrates this nicely, since in some later situation we may want to interpret this proof as function composition and formulate another theorem *about* function composition. This is why doing mathematics in HoTT requires a good memory. Computer proof assistants like `Coq` can assist with this.

Logical connectives as type formers. We now reproduce a table from the HoTT book which lists the basic type formers and corresponding logical connectives of arity zero, one and two:

Logic	Type theory
True	$\mathbf{1}$
False	0
A and B	$A \times B$
A or B	$A + B$
A implies B	$A \to B$
A if and only if B	$(A \to B) \times (B \to A)$
Not A	$A \to \mathbf{0}$

We have already encountered the type former '→' in Topic 2. We now proceed to define the other type formers by writing down the inference rules which can be used for working with them, before commenting further on the significance of the 'propositions-as-types' idea.

The unit type '1'

Again, the rules for the unit type are of the familiar kinds:

1. Formation rule:
$$\frac{\Gamma\ \texttt{ctx}}{\Gamma \vdash \mathbf{1} : \texttt{Type}}$$
In words: $\mathbf{1}$ can be constructed as a type in any context.
2. Introduction rule:
$$\frac{\Gamma\ \texttt{ctx}}{\Gamma \vdash \star : \mathbf{1}}$$
In any context, we can construct an element of $\mathbf{1}$ denoted $\star : \mathbf{1}$.
3. Elimination rule:
$$\frac{\Gamma, x : \mathbf{1} \vdash C : \texttt{Type} \qquad \Gamma \vdash c : C[\star/x] \qquad \Gamma \vdash a : \mathbf{1}}{\Gamma \vdash \mathrm{ind}_{\mathbf{1}}(x.C, c, a) : C[a/x]}$$
Here, the notation '$x.C$' means that x, which may occur as a variable in the expression C, is now regarded as a *bound* variable in the same expression. With the rules that we have until now, our theory does not allow the derivation of judgements of the form $\Gamma, x : \mathbf{1} \vdash C : \texttt{Type}$ in which C depends explicitly on the variable x; but we will soon encounter rules which permit the construction of such *dependent types*, and then this becomes relevant.
This rules states that if C is a type, possibly depending on or indexed by some variable $x : \mathbf{1}$, then in order to construct an element of $C[a/x]$ for some given expression $a : \mathbf{1}$, it is sufficient to construct an element of

$C[\star/x]$. This is an indirect way of expressing the idea that the unit type $\mathbf{1}$ has exactly one element. This is consistent with the philosophy that the relevant aspects of a mathematical object ($\mathbf{1}$ in this case) are not its internal structure, but the ways in which it interacts with other mathematical objects. The reason for writing '$\mathrm{ind}_{\mathbf{1}}$' is that the elimination rule can also be interpreted as *induction* over the unit type $\mathbf{1}$, in the sense of a 'reduction to the base case', where the base case is $\star : \mathbf{1}$.

If one thinks of C as a proposition indexed by a variable $x : \mathbf{1}$, then this rule states that in order to prove C for a certain expression a in place of x, then it is sufficient to prove it in the case where x is replaced by $\star$.

Again, this rule needs to be supplemented by a corresponding rule stating that $\mathrm{ind}_{\mathbf{1}}$ preserves judgemental equality in all its arguments:

$$\frac{\Gamma, x : \mathbf{1} \vdash C \equiv C' : \mathtt{Type} \qquad \Gamma \vdash c \equiv c' : C[\star/x] \qquad \Gamma \vdash a \equiv a' : \mathbf{1}}{\Gamma \vdash \mathrm{ind}_{\mathbf{1}}(x.C, c, a) \equiv \mathrm{ind}_{\mathbf{1}}(x.C', c', a') : C[a/x]}$$

In the conclusion, we also could have replaced $C[a/x]$ by $C[a'/x]$, since the assumption $a \equiv a'$ guarantees that $C[a'/x] \equiv C[a/x]$. (Again, this is a metatheoretical property.)

From now on, we will omit explicit mention of such 'companion' rules stating preservation of judgemental equalities. We take it to be implicitly understood that if we write down some inference rule for a typing judgement, then the corresponding rule for 'propagating' a judgemental equality from top to bottom is assumed as well.

4. Computation rule:

$$\frac{\Gamma, x : \mathbf{1} \vdash C : \mathtt{Type} \qquad \Gamma \vdash c : C[\star/x]}{\Gamma \vdash \mathrm{ind}_{\mathbf{1}}(x.C, c, \star) \equiv c : C[\star/x]}$$

This rule tells us what happens if we apply the elimination rule to the canonical element $\star : \mathbf{1}$, namely we simply recover the given c up to judgemental equality.

Although the terminology 'unit type' suggests that $\mathbf{1}$ may have only one element, the candidate judgement

$$\Gamma, a : \mathbf{1} \vdash a \equiv \star : \mathbf{1}$$

is not derivable from these inference rules! It would be analogous to the 'uniqueness principle' that we postulated for function types. It is not perfectly clear to us why one postulates a uniqueness principle for function types but not for the unit type. However, we will see later in which sense the unit type does have exactly one element.

The empty type '0'

1. Formation rule:

$$\frac{\Gamma \ \mathrm{ctx}}{\Gamma \vdash \mathbf{0} : \mathtt{Type}}$$

2. Elimination rule:

$$\frac{\Gamma, x : \mathbf{0} \vdash C : \mathtt{Type} \qquad \Gamma \vdash a : \mathbf{0}}{\Gamma \vdash \mathrm{ind}_{\mathbf{0}}(x.C, a) : C[a/x]}$$

This rule has a beautiful interpretation in terms of propositions-as-types: if C represents a proposition in a context in which $x : \mathbf{0}$, i.e. x is a proof of False, then one automatically obtains a proof of $C[a/x]$ for any a. In other words: falsity implies anything!

Here, the absence of an introduction rule does not mean that the empty type somehow deviates from the usual paradigm—to the contrary! Usually, there is one introduction rule for every way in which an element of the type under consideration can be constructed. In the case of the empty type, there is no way to construct an element, and hence there is no introduction rule.

Product types

Intuitively, for two types A and B, the product type $A \times B$ has as elements the pairs (x, y). But again, instead of defining the type by saying which elements it contains, we rather specify operational rules which tell us how to *use* the product type and its elements.

1. Formation rule:
$$\frac{\Gamma \vdash A : \mathtt{Type} \qquad \Gamma \vdash B : \mathtt{Type}}{\Gamma \vdash A \times B : \mathtt{Type}}$$
This rules tells us that a product type $A \times B$ can be formed for any given types A and B in any context.

2. Introduction rule:
$$\frac{\Gamma \vdash x : A \qquad \Gamma \vdash y : B}{\Gamma \vdash (x, y) : A \times B}$$
Any pair of elements of A and B yields an element of $A \times B$, which we denote using the usual notation for a pair.

3. Elimination rule:
$$\frac{\Gamma, p : A \times B \vdash C : \mathtt{Type} \qquad \Gamma, x : A, y : B \vdash g : C[(x,y)/p] \qquad \Gamma \vdash q : A \times B}{\Gamma \vdash \mathrm{ind}_{A\times B}(p.C, x.y.g, q) : C[q/p]}$$
This rule shows that to derive an assertion C involving any element $q : A \times B$, it suffices to derive it for all explicitly given pairs $(x, y) : A \times B$.

4. Computation rule:
$$\frac{\Gamma, p : A \times B \vdash C : \mathtt{Type} \qquad \Gamma, x : A, y : B \vdash g : C[(x,y)/p] \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathrm{ind}_{A\times B}(p.C, x.y.g, (a,b)) \equiv g[a/x, b/y] : C[(a,b)/p]}$$
As usual, the computation rule tells us what happens when we apply the elimination rule on an element obtained via the introduction rule.

As an example application, we can construct the *product projections* $\mathrm{pr}_1 : A \times B \to A$ and $\mathrm{pr}_2 : A \times B \to B$ which map each pair to the corresponding component,

$$\mathrm{pr}_1 :\equiv \lambda(q : A \times B).\mathrm{ind}_{A\times B}(p.A, x.y.x, q)$$
$$\mathrm{pr}_2 :\equiv \lambda(q : A \times B).\mathrm{ind}_{A\times B}(p.B, x.y.y, q).$$

Again, there is no rule which would tell us that any $p \equiv (\mathrm{pr}_1(p), \mathrm{pr}_2(p))$ for any p, and in fact this statement is not derivable from the rules of HoTT, although it is intuitive. We will get back to this point later.

Coproduct Types

What we mean by a 'coproduct' $A + B$ of types A and B is what correponds to a *disjoint union* in a set-theoretic context: intuitively, an element of $A + B$ is either an element of A or an element of B. The coproduct type is operationally specified through the following inference rules:

1. Formation rule:
$$\frac{\Gamma \vdash A : \mathtt{Type} \qquad \Gamma \vdash B : \mathtt{Type}}{\Gamma \vdash A + B : \mathtt{Type}}$$
This rule exemplifies how to form a coproduct type $A + B$ given A and B.

2. Introduction rules:
$$\frac{\Gamma \vdash A : \mathtt{Type} \qquad \Gamma \vdash B : \mathtt{Type} \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathrm{inl}(a) : A + B}$$
This rule associates to every expression $a : A$ an element $\mathrm{inl}(a) : A + B$. Similarly, there is another introduction rule that associates to every element $b : B$ the corresponding $\mathrm{inr}(b) : B$,
$$\frac{\Gamma \vdash A : \mathtt{Type} \qquad \Gamma \vdash B : \mathtt{Type} \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathrm{inr}(b) : A + B}$$
Here, 'inl' and 'inr' stand for 'in left' and 'in right', respectively.

3. Elimination rule:

$$\frac{\Gamma, z : A + B \vdash C : \texttt{Type} \qquad \Gamma, x : A \vdash c : C[\mathrm{inl}(x)/z] \qquad \Gamma, y : B \vdash d : C[\mathrm{inr}(y)/z] \qquad \Gamma \vdash e : A + B}{\Gamma \vdash \mathrm{ind}_{A+B}(z.C, x.c, y.d, e) : C[e/z]}$$

   As usual for an elimination rule, this states that in order to prove a statement concerning some element $e : A + B$, it suffices to prove it for all 'base cases', meaning all elements of the form $\mathrm{inl}(a)$ and $\mathrm{inr}(b)$.

4. Computation rules:

$$\frac{\Gamma, z : A + B \vdash C : \texttt{Type} \qquad \Gamma, x : A \vdash c : C[\mathrm{inl}(x)/z] \qquad \Gamma, y : B \vdash d : C[\mathrm{inr}(y)/z] \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathrm{ind}_{A+B}(z.C, x.c, y.d, \mathrm{inl}(a)) \equiv c[a/x] : C[\mathrm{inl}(a)/z]}$$

   Again, this explains what happens when one applies the elimination to one of the 'base cases'. There is an analogous rule for the other base case:

$$\frac{\Gamma, z : A + B \vdash C : \texttt{Type} \qquad \Gamma, x : A \vdash c : C[\mathrm{inl}(x)/z] \qquad \Gamma, y : B \vdash d : C[\mathrm{inr}(y)/z] \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathrm{ind}_{A+B}(z.C, x.c, y.d, \mathrm{inr}(b)) \equiv d[b/y] : C[\mathrm{inr}(b)/z]}$$

This ends our list of type formers which represent the logical connectives for propositional logic. We thus arrive at a situation in which propositions are represented by types and proofs by elements of that type. So proving a proposition means finding an element of that type. Let us give an example to understand this better. Consider the proposition "if A then (if B then A)"; it is a tautology, even in the system that we are considering. In our type-theoretic language, it would be formulated as the type $A \to (B \to A)$ in the context $A, B : \texttt{Type}$. Finding an element of this type means finding an expression E which fits into a typing judgement

$$A, B : \texttt{Type} \vdash E : A \to (B \to A). \tag{3.2}$$

Here is an example of what this expression can be and what the associated proof tree looks like:

$$\cfrac{\cfrac{\cfrac{\cfrac{\vdots}{(A, B : \texttt{Type}, x : A, y : B)\ \texttt{ctx}}}{A, B : \texttt{Type}, x : A, y : B \vdash x : A}}{A, B : \texttt{Type}, x : A \vdash (\lambda(y : B).x) : B \to A}}{A, B : \texttt{Type} \vdash \lambda(x : A).\lambda(y : B).x : A \to (B \to A)}$$

It should be clear how to complete this proof tree by deriving $(A, B : \texttt{Type}, x : A, y : B)$ `ctx`. In `Coq`, which automatically takes care of contexts, this proof would simply look like this:

```
(* Proving a simple tautology *)

Theorem tautology (A B : Type) : A → (B → A).

Proof.
    intro a.
    intro b.
    exact a.
Qed.
```

Proofs as programs. So we have seen that propositions become types whose elements play the role of proofs. But there is another interpretation of elements of types: they are programs doing a computation! For example, function composition

$$A, B, C : \texttt{Type} \vdash \lambda f.\lambda g.\lambda x.g(f(x)) : (A \to B) \to (B \to C) \to (A \to C) \tag{3.3}$$

can be understood as a program which takes an $f : A \to B$ and a $g : B \to C$ as inputs and returns the composite $g \circ f : A \to C$. From this point of view, any type theory is a *programming language*! We will later see how to write more elaborate programs in type theory, like a program for sorting lists.

In particular, every proof of a proposition has an interpretation of a program which computes an actual element of the proposition-as-type.

Type-checking vs theorem-proving. An important distinction is between *finding* an element of a given type in a given context, as in (3.2), and *verifying* that a claimed typing judgement is indeed derivable. In terms of the logical interpretation, the former problems consists in finding a proof of a conjectural statement, while the latter boils down to verifying such a proof. In the programming context, the latter is also known as *type-checking*, i.e. verifying that all constructions used in a program are consistent with the specified variable types.

The reason that it can be checked algorithmically whether a claimed typing judgement is derivable is the following: for every expression formed from the inference rules, the expression essentially 'remembers' which inference rules have been applied in order to construct it. For example with (3.3), it is clear that the last inference rule used in constructing that typing judgement must have been a lambda abstraction, i.e. an application of the introduction rule for function types. Hence the main parts of the proof tree of any typing judgement can be reconstructed by decomposing the given expression step by step, starting with the outermost part.

Again, this simplicity of type-checking is a metatheoretical property. As before, we restrain from a rigorous treatment of this issue and leave it at the present informal discussion.

Constructivism. The fact that every proof in type theory can be interpreted as a program or algorithm means that type theory is inherently *constructive*: proving a proposition boils down to writing down an explicit element of the corresponding type, which has an interpretation as a program which can possibly be run on a computer. This has as a consequence that some statements which are true in classical logic have no analog in type theory. For example, there is no expression E which would fit into a valid judgement of the form

$$A : \mathtt{Type} \vdash E : A + (A \to \mathbf{0}). \tag{3.4}$$

As a proposition, the statement $A + (A \to \mathbf{0})$ translates into $A \vee \neg A$, which is the *law of exluded middle*. One can easily show that this holds for any proposition A in classical logic: if A is true, then certainly $A \vee \neg A$ must be true as well; on the other hand, if A is false, then $\neg A$ is true and $A \vee \neg A$ holds likewise.

The intuitive reason that no judgement of the form (3.4) holds is that there is no program to prove it. After all, such a program would have to return an actual element of $A + (A \to \mathbf{0})$, and since any element of $A + (A \to \mathbf{0})$ is (at least intuitively) either an element of A or an element of $A \to \mathbf{0}$, this means that whatever algorithm E would fit into such a judgement, it would necessarily decide for any given proposition A whether it is true or false. But it is known that no such algorithm can exist, at least in more expressive theories than we currently have; for example, by the negative solution to Hilbert's 10th problem, there is no algorithm which could determine whether a given polynomial equation with integer coefficients has an integer solution or not.

So although the above judgement does not hold, it is still possible to use the law of excluded middle if one is inclined to do so: this can be done by assuming an additional $em : A + (A \to \mathbf{0})$ in the context, which corresponds to a hypothetical proof of excluded middle for A. As an example, consider this complicated-looking typing judgement:

$$\begin{aligned} &A, B : \mathtt{Type}, em : A + (A \to \mathbf{0}) \\ &\qquad \vdash \lambda F.\mathrm{ind}_{A+(A\to\mathbf{0})}(z.A, x.x, y.F(\lambda(a:A).\mathrm{ind}_{\mathbf{0}}(w.B, y(a))), em) : ((A \to B) \to A) \to A. \end{aligned} \tag{3.5}$$

We derive this below. In classical logic, the statement 'if A implies B implies A, then A' is known as *Peirce's law.* As this example shows, type theory is *agnostic* with respect to the law of excluded middle: it does not follow from the basic inference rules, but there is no harm in assuming it as an additional axiom. This has the interesting feature that it makes uses of the law of excluded middle completely explicit.

We now describe the proof tree for (3.5), while slowly moving to a more informal mode of reasoning in order to aid comprehensibility. Constructing the element of $(A \to B) \to A) \to A$ in (3.5) means that we need to assign to every $F : (A \to B) \to A$ an element of A. This is the informal explanation of lambda abstraction, by which we can move one step up in the proof tree and arrive at

$$\begin{aligned} &A, B : \mathtt{Type}, em : A + (A \to \mathbf{0}), F : (A \to B) \to A \\ &\qquad \vdash \mathrm{ind}_{A+(A\to\mathbf{0})}(z.A, x.x, y.F(\lambda(a:A).\mathrm{ind}_{\mathbf{0}}(w.B, y(a))), em) : A. \end{aligned}$$

Since now $\mathrm{ind}_{A+(A\to\mathbf{0})}$ is outermost, one can go up one more step in the proof tree by using the elimination rule for the coproduct $A + (A \to \mathbf{0})$, which plays the rule of a *case distinction* splitting the problem into $x : A$, i.e. A holds, and $y : A \to \mathbf{0}$, i.e. A does not hold. If we abbreviate the present context by Γ, then this reduces the problem to deriving the *four* typing judgements

$$\Gamma, z : A + (A \to \mathbf{0}) \vdash A : \mathtt{Type}$$

$$\Gamma, x : A \vdash x : A$$

$$\Gamma, y : A \to \mathbf{0} \vdash F(\lambda(a : A).\mathrm{ind}_{\mathbf{0}}(w.B, y(a))) : A$$

$$\Gamma \vdash em : A + (A \to \mathbf{0}).$$

The important ones are the second, which tells us that if A holds then we simply return the given proof of A, and the third, which says that if the negation of A holds, then we return the complicated-looking element. These four judgements are all trivial to derive except for the third, so we focus on this. Since the outermost construction in this judgement is a function application (elimination rule for function types), going one step up in the proof tree yields the two branches

$$\Gamma, y : A \to \mathbf{0} \vdash F : (A \to B) \to A$$

$$\Gamma, y : A \to \mathbf{0} \vdash \lambda(a : A).\mathrm{ind}_{\mathbf{0}}(w.B, y(a)) : A \to B,$$

so we find that the complicated-looking element is given by an application of F to a complicated-looking function $A \to B$. Again, deriving the first one is trivial, so we continue with the second. It says that the function $A \to B$ arises from an application of lambda abstraction to the judgement

$$\Gamma, y : A \to \mathbf{0}, a : A \vdash \mathrm{ind}_{\mathbf{0}}(w.B, y(a)) : B.$$

This comes from the elimination rule for $\mathbf{0}$, which requires

$$\Gamma, y : A \to \mathbf{0}, a : A, w : \mathbf{0} \vdash B : \mathtt{Type}$$

$$\Gamma, y : A \to \mathbf{0}, a : A \vdash y(a) : \mathbf{0}$$

Again the first one follows from a simple formation rule, while the second one can now be reduced to judgements that are finally all trivial,

$$\Gamma, y : A \to \mathbf{0}, a : A \vdash y : A \to \mathbf{0}$$

$$\Gamma, y : A \to \mathbf{0}, a : A \vdash a : A.$$

Over the course of the following lectures, we will turn this very formal style of reasoning into more and more informal language. Nevertheless, one should always be able to transform an informal argument back into a formal one, since only then can one achieve the level of accuracy required in mathematical proofs. An informal proof of Peirce's law would be as follows:

Proof. We assume the law of excluded middle, so we can distinguish the two cases that A holds and that the negation of A holds. If A holds, then we simply return the given proof of A; while if the negation of A holds, then we take the proof $y : A \to \mathbf{0}$ of this, compose it with the trivial function $\mathbf{0} \to B$ in order to obtain a function $A \to B$, and then evaluate $F : (A \to B) \to A$ on this composite function. □

It should be clear that this informal style is much more concise and also more comprehensible to a human reader, which is why we prefer it in order to do actual mathematics.

Finally, let us note that the situation with the axiom of choice is similar to the case of excluded middle, but more subtle. We will not discuss this now, though.

But what about quantifiers?

So far, we have only discussed propositional logic in the propositions-as-types paradigm. However, doing any kind of non-trivial mathematics requires the use of quantifiers! For example, in the statement that 'there exists no triple of positive natural numbers (x, y, z) such that $x^3 + y^3 = z^3$', the expression $x^3 + y^3 = z^3$ is a proposition which *depends* on variables $x, y, z : \mathbb{N}$. These variables become bound after the introduction of the existential quantifier to the proposition

$$\exists x, \exists y, \exists z,\ x^3 + y^3 + z^3.$$

Until next time, you want to try and figure out for yourself what the type-theoretic generalization of quantifiers might be!

Topic 4: Universes and dependent types

In Topic 3, we discussed how propositions are types and how proofs are elements of types. Logical connectives turned out to correspond to certain type formers. However, we only analyzed propositional logic and have not yet introduced quantifiers. Before doing so, let us ask first: what should a quantifier quantify over? The obvious answer seems to be that a quantifier should quantify over all elements of a type: for example, a proposition like '$x + y = y + x$' should be interpreted as a proposition[1], universally quantified over $x, y : \mathbb{N}$. So, from this point of view one would expect that a quantifier quantifies over the elements of a type. On the other hand, we may also want to quantify statements like the law of exluded middle $A + (A \to \mathbf{0})$ over all $A : \mathtt{Type}$. In this case, a quantifier should quantify over all types!

For this and other purposes, it is very convenient to be able to regard $A : \mathtt{Type}$ itself as a typing judgement, in which `Type` itself is a type of which A is an element. Doing this is indeed possible, but one has to be careful: a naive solution will run into Girard's paradox, which is a type-theoretic version of Russell's paradox on the set of all sets. This seems to require the introduction of a whole *hierarchy* of bigger and bigger universes, in the spirit of the Grothendieck universes used in category theory.

Universes

There are several different ways of modelling universes in type theory; the one used in HoTT results is *Russell-style* universes. There is an infinite hierarchy of universe types denoted by symbols

$$\mathcal{U}_0, \quad \mathcal{U}_1, \quad \mathcal{U}_2, \ldots.$$

This hierarchy is indexed by a natural number i which is 'external' to the theory in the sense that it is not an element of the type of natural numbers, but rather just a formal symbol which we use to tell apart the different universes. In particular, it is impossible to write down statements within the theory which quantify over all universes. In all kind of actual mathematics within HoTT, one will not get beyond $\mathcal{U}_1$ or sometimes $\mathcal{U}_2$.

These universes play the role of what we previously denoted `Type`. So from now on, types themselves will be elements of a universe; this means that we abandon `Type` completely and write $A : \mathcal{U}_i$ for some appropriate i instead of $A : \mathtt{Type}$. Concerning the many inference rules which we have already set up and which contain `Type`, this means that we replace each such inference rule by the same rule with `Type` replaced by $\mathcal{U}_i$ for each i. For example, the formation rule for contexts

$$\frac{(x_1 : A_1, \ldots, x_n : A_n)\ \mathtt{ctx}}{(x_1 : A_1, \ldots, x_n : A_n, B : \mathtt{Type})\ \mathtt{ctx}}$$

needs to be replaced by one rule

$$\frac{(x_1 : A_1, \ldots, x_n : A_n)\ \mathtt{ctx}}{(x_1 : A_1, \ldots, x_n : A_n, B : \mathcal{U}_i)\ \mathtt{ctx}}$$

for each index i.

Since now types themselves are elements of another type, some of these new versions of the inference rules actually become redundant. For example, the judgemental equality rules for types,

$$\frac{\Gamma \vdash A \equiv B : \mathcal{U}_i \quad \Gamma \vdash B \equiv C : \mathcal{U}_i}{\Gamma \vdash A \equiv C : \mathcal{U}_i}$$

[1]The equality sign '=' denotes *propositional equality*, which is different from $\equiv$ and turns the given statement into a proposition, i.e. into a type itself. We will talk about propositional equality soon in more detail.

are special cases of the judgemental equality rule for elements of types,

$$\frac{\Gamma \vdash a \equiv b : A \quad \Gamma \vdash b \equiv c : A}{\Gamma \vdash a \equiv c : A}$$

The other properties of universes are expressed by the following inference rules:

1. Introduction rule:

$$\frac{\Gamma \ \mathtt{ctx}}{\Gamma \vdash \mathcal{U}_i : \mathcal{U}_{i+1}}$$

   In any context, the universe $\mathcal{U}_i$ is a well-typed expression consisting of only one symbol, and its type is the next higher universe $\mathcal{U}_{i+1}$.

2. Cumulativity rule:

$$\frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash A : \mathcal{U}_{i+1}}$$

   If A is some universe, then it is also in the next higher universe.

Repeated application of the cumulativity rule shows that if A is in some universe, then it is also in any higher universe. Together with the introduction rule, this implies that $\mathcal{U}_i : \mathcal{U}_j$ for any $j > i$.

Informally, we also write $\mathcal{U}$ instead of $\mathcal{U}_i$ in order to refer to an arbitrary universe. It is then understood that the resulting statement or inference rule is supposed to be instantiated for *all* universes $\mathcal{U}_i$.

So, we can work with the idea of a 'type of all types', made precise by the notion of universes, just like we can work with any other type. For example, for any $A : \mathcal{U}_i$, one can form the function type $A \to \mathcal{U}_i : \mathcal{U}_{i+1}$. To see this, one simply uses the cumulativity rule above together with the formation rule for function types as follows:

$$\frac{\dfrac{\dfrac{\vdots}{A : \mathcal{U}_i \vdash A : \mathcal{U}_i}}{A : \mathcal{U}_i \vdash A : \mathcal{U}_{i+1}} \quad \dfrac{\vdots}{A : \mathcal{U}_i \vdash \mathcal{U}_i : \mathcal{U}_{i+1}}}{A : \mathcal{U}_i \vdash A \to \mathcal{U}_i : \mathcal{U}_{i+1}}$$

Elements of this function type $A \to \mathcal{U}_i$ can be constructed for example by lambda abstraction from a judgement of the form

$$A : \mathcal{U}_i, x : A \vdash P : \mathcal{U}_i, \tag{4.1}$$

where P is some expression possibly involving x. What would such a function mean? If we think of $P : \mathcal{U}_i$ as a proposition, this means that we are dealing with a proposition containing a variable x. This is precisely the kind of thing for which we wanted to introduce quantifiers!

A type depending on a variable of another type is called a *dependent type*. This refers both to judgements of the form (4.1) in which $P : \mathcal{U}_i$ depends on $x : A$ and to functions $A \to \mathcal{U}_i$; thanks to lambda abstraction and function application, these two points of view are equivalent. From now on, we will freely confuse them without explicit mention.

As an example, consider the function $\lambda(A : \mathcal{U}).A \to (A \to \mathbf{0}) \to \mathbf{0}$, which has type $\mathcal{U} \to \mathcal{U}$. Upon thinking of A as a proposition, $(A \to \mathbf{0}) \to \mathbf{0}$ becomes the double negation of A, and hence the type $A \to (A \to \mathbf{0}) \to \mathbf{0}$ represents the proposition that A implies its own double negation. We would now like to express the statement 'for all types A, A implies its double negation' within type theory. This needs a universal quantifier that quantifies over all $A : \mathcal{U}$.

Dependent functions

We now introduce yet another new type former '$\prod$' which will play the role of the universal quantifier. For any given dependent type $P : A \to \mathcal{U}$, it will be possible to form the *dependent function type* $\prod_{x:A} P(x)$. Its elements $f : \prod_{x:A} P(x)$ intuitively correspond to tuples $(f(x))_{x:A}$ with $f(x) : P(x)$. We usually think of such a tuple as a function whose codomain $P(x) : \mathcal{U}$ depends on its argument $x : A$; it is a *dependent function*. In this sense, $\prod_{x:A} P(x)$ is the type of all dependent functions from A to P. In type theory, $\prod_{x:A} P(x)$ is often also written as $\prod(x : A)P(x)$. The latter notation can be more useful in cases in which A itself may be a big expression involving other variables, and it is also more consistent with the notation for lambda abstraction. However, we usually stick with the more conventional notation $\prod_{x:A} P(x)$.

The dependent function type generalizes the ordinary function type former '$\to$'. Hence it should not be surprising that its rules are very similar, and actually specialize to the ones for $\to$. They are given as follows:

1. Formation rule:
$$\frac{\Gamma, x : A \vdash P : \mathcal{U}}{\Gamma \vdash \prod_{x:A} P : \mathcal{U}}$$
Here, the interesting case is when P depends on x.
2. Introduction rule:
$$\frac{\Gamma, x : A \vdash y : P}{\Gamma \vdash \lambda(x : A).y : \prod_{x:A} P}$$
Again, y typically depends on x. As before, we abbreviate $\lambda(x : A).y$ to $\lambda x.y$ if the type of x is clear from the context.
3. Elimination rule:
$$\frac{\Gamma \vdash f : \prod_{x:A} P \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : P[a/x]}$$
As before, the elimination rule is function application.
4. Computation rule:
$$\frac{\Gamma, x : A \vdash y : P \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda(x : A).y)(a) \equiv y[a/x] : P[a/x]}$$
When read from left to right, the judgemental equality $(\lambda x.y)(a) \equiv y[a/x]$ is also known as *beta reduction.*
5. Uniqueness principle:
$$\frac{\Gamma \vdash f : \prod_{x:A} P}{\Gamma \vdash f \equiv (\lambda x.f(x)) : \prod_{x:A} P}$$
When read from right to left, the judgemental equality $f \equiv (\lambda x.f(x))$ is also known as *eta reduction.*

These rules refer to the situation in which P is an expression containing the variable x; in the picture where one considers a dependent type as a function $A \to \mathcal{U}$, one will correspondingly have to replace every occurrence of P by $P(x)$ and $P[a/x]$ by $P(a)$.

If $P : \mathcal{U}$ is a fixed type not depending on $x : A$, then these rules are precisely the ones of the ordinary function type. This does *not* mean that $(\prod_{x:A} P) \equiv (A \to P)$; it rather means that there are functions
$$\left(\prod_{x:A} P\right) \to (A \to P), \qquad (A \to P) \to \left(\prod_{x:A} P\right),$$
where the uniqueness principles imply that the two possible composites of these functions are judgementally equal to the identity functions on $(A \to P)$ and $\prod_{x:A} P$, respectively.

If $P : A \to \mathcal{U}$ represents a proposition depending on a variable $x : A$, then $\prod_{x:A} P(x)$ represents the universally quantified proposition 'for all $x : A$, $P(x)$'. In this interpretation, the introduction and elimination rule acquire the following meaning: whenever one has a proof y that P holds for a variable $x : A$, then one obtains a proof $\lambda(x : A).y$ showing that P holds for all x. If one has a proof f that P holds for all x and a given expression $a : A$, then one obtains a proof $f(a)$ showing that P holds with a in place of x.

Getting back to the above example, let us try to prove that any proposition implies its double negation,
$$\prod_{A:\mathcal{U}} A \to (A \to \mathbf{0}) \to \mathbf{0}. \tag{4.2}$$
One element of this type in the empty context is the expression
$$\lambda(A : \mathcal{U}).\lambda(x : A).\lambda(f : A \to \mathbf{0}).f(a),$$
which constitutes a proof of the double negation statement that we were looking for.

One can arrive at this statement by the following reasoning, which is specific to this one and similar cases and does not constitute a general procedure for constructing proofs of propositions. What we are trying to do is to find an expression E fitting into a judgement
$$\vdash E : \prod_{A:\mathcal{U}} A \to (A \to \mathbf{0}) \to \mathbf{0} \tag{4.3}$$
in the empty context. The obvious way to construct such a dependent function with domain $\mathcal{U}$ is by lambda abstraction, so that we can make the ansatz $E \equiv \lambda(A : \mathcal{U}).E'$, where E' must fit into the judgement
$$A : \mathcal{U} \vdash E' : A \to (A \to \mathbf{0}) \to 0. \tag{4.4}$$

Again, the canonical way to construct such an E' is through lambda abstraction, in fact two lambda abstractions, so that we make the ansatz $E' \equiv \lambda(x:A).\lambda(f:A\to\mathbf{0}).E''$ where E'' should fit into the judgement

$$A:\mathcal{U}, x:A, f:A\to\mathbf{0} \vdash E'':\mathbf{0}.$$

At this point, one can simply set $E'' :\equiv f(x)$.

In Coq, the type (4.2) and its element look as follows:

```
Theorem double_negation : forall A : Type, A → (A → False) → False.
Proof.
      intro A.
      intro x.
      intro f.
      apply f.
      exact x.
Qed.
```

In general, one can use successive applications of lambda abstraction for dependent function types to turn the context of any typing judgement into the empty context. For example, one can regard either (4.4) or (4.3) as the statement that any type implies its double negation. The latter is a statement in the empty context.

Up to now, all the dependent types that we can construct live itself over the universe in the sense that they are functions $P:\mathcal{U}\to\mathcal{U}$. But as soon as we get to inductive type families, the situation will change and we will be able to construct dependent types depending on a variable in any other type.

Dependent pair types

In quite a similar way, the existential quantifier is given by the *dependent pair* type $\sum_{x:A} P(x)$. Just as the dependent function type $\prod_{x:A} P(x)$ generalizes the function type $A\to P$, the dependent pair type $\sum_{x:A} P(x)$ generalizes the cartesian product type $A\times P$. More concretely, while the second component of an explicitly given pair $(x,y):A\times P$ is necessarily of type P, the dependent pair type allows that the type of the second component may vary as a function of the first component.

The rules are as follows:

1. Formation rule:
$$\frac{\Gamma, x:A\vdash P:\mathcal{U}}{\Gamma\vdash \sum_{x:A} P:\mathcal{U}}$$

2. Introduction rule:
$$\frac{\Gamma, x:A\vdash P:\mathcal{U} \qquad \Gamma\vdash a:A \qquad \Gamma\vdash b:P[a/x]}{\Gamma\vdash (a,b):\sum_{x:A} P}$$
This states that an element $(a,b):\sum_{x:A} P$ can be constructed as soon as one has some $a:A$ and some $b:P[a/x]$.

3. Elimination rule:
$$\frac{\Gamma, z:\sum_{x:A} P\vdash C:\mathcal{U} \qquad \Gamma, x:A, y:P\vdash g:C[(x,y)/z] \qquad \Gamma\vdash q:\sum_{x:A} P}{\Gamma\vdash \mathrm{ind}_{\sum_{x:A} P}(z.C, x.y.g, q):C[q/z]}$$
So given a type C possibly depending on some $z:\sum_{x:A} P$, in order to construct some element of $C[q/z]$ for some given $q:\sum_{x:A} P$, it is sufficient to construct an element in $C[(x,y)/z]$ for every explicitly given pair (x,y) given in terms of an $x:A$ and some $y:P$. Again, it is important to remember that P may be an arbitrary expression involving the variable x.

4. Computation rule:
$$\frac{\Gamma, z:\sum_{x:A} P\vdash C:\mathcal{U} \qquad \Gamma, x:A, y:P\vdash g:C[(x,y)/z] \qquad \Gamma\vdash a:A \qquad \Gamma\vdash b:P[a/x]}{\Gamma\vdash \mathrm{ind}_{\sum_{x:A} P}(z.C, x.y.g, (a,b))\equiv g[a/x,b/y]:C[(a,b)/z]}$$

Again in this case, all rules refer to the case that $P:\mathcal{U}$ is an expression possibly depending on $x:A$. If instead the dependent type is given in terms of a function $P:A\to\mathcal{U}$, then every occurence of P should actually be $P(x)$ and $P[a/x]$ should be $P(a)$. An analogous comment applies to the type C which may depend on $z:\sum_{x:A} P$.

In the case that P does not actually depend on $x : A$, these rules are precisely those of the product type $A \times P$.

In terms of the logical interpretation of $\sum$ as the existential quantifier, the introduction rule states that a statement of the form 'there exists an $x : A$ such that P' can be proven by specifying some $a : A$ and a proof of $P[a/x]$. The elimination rule means that in order to prove $C[p/z]$ holds for some given $p : \sum_{x:A} P$, it is sufficient to prove that $C[(a,b)/z]$ holds for any explicitly given pair consisting of $x : A$ and $y : P$. In other words, applying this elimination rule replaces the given p by an explicit pair (x, y), which can be thought of as *extracting* from the given proof $p : \sum_{x:A} P$ an $x : A$ and some $y : P$ as witnesses of the existential statement $\sum_{x:A} P$.

We illustrate the use of the introduction and elimination rules by means of an example. In a situation where we have a dependent type

$$\Gamma, x : A \vdash P : \mathcal{U},$$

we already know that we can reinterpret this dependent type as a function $\lambda(x : A).P : A \to \mathcal{U}$. When we have another type depending on that dependent type,

$$\Gamma, x : A, y : P \vdash Q : \mathcal{U},$$

we can think of it either as a dependent function

$$\lambda(x : A).\lambda(y : P).Q : \prod_{x:A} P \to \mathcal{U}$$

or as an ordinary function

$$\lambda\left(q : \sum_{x:A} P\right).\mathrm{ind}_{\sum_{x:A} P}(z.\mathcal{U}, x.y.Q, q) : \left(\sum_{x:A} P\right) \to \mathcal{U}.$$

Example: the type of magmas

Another good example is as follows. Eventually we want to do actual mathematics inside type theory, and this comprises algebra. As the simplest example of an algebraic structure, let us consider magmas, i.e. sets or types equipped with a binary operation, which is not required to satisfy any particular law. So the structure of a magma on a type A should be given by a function

$$m \,:\, A \to A \to A,$$

so that we can define the type of all magma structures on A as

$$\mathrm{Magma}(A) :\equiv (A \to A \to A).$$

Moreover, we can define the *type of all magmas* in a particular universe $\mathcal{U}$ as

$$\mathrm{Magma}_{\mathcal{U}} :\equiv \sum_{A:\mathcal{U}} \mathrm{Magma}(A).$$

Now we may want to prove a certain statement about all magmas. Such a statement would be a dependent type

$$P : \mathrm{Magma}_{\mathcal{U}} \to \mathcal{U}.$$

In order to prove this statement, we therefore need to find an element in $P(s)$ for every $s : \mathrm{Magma}_{\mathcal{U}}$. The elimination rule for $\sum_{A:\mathcal{U}} \mathrm{Magma}(A)$ then lets us assume without loss of generality that the given s is of the form $s \equiv (A, m)$ for some particular $A : \mathcal{U}$ and $m : \mathrm{Magma}(A)$.

Functional forms of inference rules

Not only can one freely go back and forth between the picture of a dependent type as an expression P involving a variable $x : A$ and a function $\lambda x.P : A \to \mathcal{U}$, but a similar statement also applies to the inference rules, which one can convert into a functional form by successive applications of lambda abstraction. For example, one can consider the type former for the cartesian product as a function

$$\lambda A.\lambda B.A \times B \,:\, \mathcal{U} \to \mathcal{U} \to \mathcal{U}$$

which encapsulates the formation rule in the empty context. In a similar way, the introduction rule can be regarded as a dependent function

$$\lambda A.\lambda B.\lambda x.\lambda y.(x, y) \,:\, \prod_{A,B:\mathcal{U}} A \to B \to (A \times B),$$

for which we typically regard A and B as implicit arguments that are not written down explicitly, since they can be inferred from the other arguments.

Similarly, the functional form of the formation rule and introduction rule for dependent pairs look like this:

$$\lambda A.\lambda P.\sum_{x:A} P(x) \;:\; \prod_{A:\mathcal{U}} (A \to \mathcal{U}) \to \mathcal{U}, \qquad \lambda A.\lambda P.\lambda a.\lambda b.(a,b) \;:\; \prod_{A:\mathcal{U}} \prod_{P:A\to\mathcal{U}} \prod_{x:A} P(x) \to \sum_{x:A} P(x).$$

Elimination rules look somewhat more complicated not just as inference rules, but also in their functional form. If one thinks of an elimination rule as a rule for defining a (dependent) function out of the type under consideration, then the rule states that in order to define such a function, it is sufficient to define it on the 'base cases', which are those elements of the type arising from the introduction rule. So the functional form of the elimination rule for the dependent pair type looks like this:

$$A:\mathcal{U},\ P:A\to\mathcal{U} \vdash \mathrm{ind}'_{\sum_{x:A} P(x)} : \prod_{C:(\sum_{x:A} P(x))\to\mathcal{U}} \left(\prod_{x:A} \prod_{y:P(x)} C((x,y)) \right) \to \prod_{q:\sum_{x:A} P(x)} C(q) \tag{4.5}$$

We have left A and P in the context in order to keep the number of quantifiers reasonably small, and also in order to make clear that one can consider $\mathrm{ind}'_{\sum_{x:A} P(x)}$ as a *universal instance* of the elimination rule which repackages all its instances into a single dependent function. The double brackets in $C((x,y))$ indicate that one first forms the pair (x,y) and then applies the function C to this pair. One can construct the element $\mathrm{ind}'_{\sum_{x:A} P(x)}$ in the obvious way by defining it to be given by

$$\mathrm{ind}'_{\sum_{x:A} P} :\equiv \lambda C.\lambda g.\lambda p.\mathrm{ind}_{\sum_{x:A} P}\,(z.C(z), x.y.g(x,y), p)\,.$$

The actual content is the same: (4.5) represents the statement that in order to prove $C(q)$ for all or some $q:\sum_{x:A} P(x)$ for a given dependent type $C:(\sum_{x:A} P(x))\to\mathcal{U}$, it is sufficient to prove $C((x,y))$ for all explicitly given pairs (x,y). The same holds if one replaces 'prove' by 'find an element of'.

Finally, since computation rules result in judgemental equality judgements, they do not have a functional form, since judgemental equalities are not types themselves and hence one cannot apply quantifiers to them. An expression like $\prod_{x:A} x \equiv x$ does not make sense in type theory: quantifying equations is one reason for introducing *propositional equality* later on.

From now on, we will usually work with these 'functional' versions of inference rules, and in particular of the elimination rules. Their advantage is that (4.5) is an *element* of a type rather than an inference rule, and therefore can be manipulated as such and applied as a function, and this makes dependences on other variables more explicit than saying that a certain expression may depend on a certain variable. Likewise, we will use the functional versions of dependent types, as in the following example.

Example: the projections of a dependent pair

Let $P:A\to\mathcal{U}$ be a dependent type. Given an element $q:\sum_{x:A} P(x)$ of the associated dependent pair type, how do we turn it into an explicitly given pair? There are two options: if, as outlined at the end of the magma example, we want to construct a dependent function

$$f \;:\; \prod_{q:\sum_{x:A} P(x)} S(q)$$

where $S:(\sum_{x:A} P(x))\to\mathcal{U}$ is a 'doubly dependent' type as described above, then applying the elimination rule for $\sum_{x:A} P(x)$ reduces this problem to finding a dependent function in

$$\prod_{x:A} \prod_{y:P(x)} S((x,y)).$$

So in some sense we have 'turned' q into an explicitly given pair using (4.5). Note that this procedure is not a function which takes q and turns it into a pair; it is rather a method of getting rid of q and replacing it by an explicitly given pair.

Alternatively, just as the cartesian product type has product projections that can be constructed from its elimination rule, one can construct projections for dependent pair that do indeed map every $q:\sum_{x:A} P(x)$ to an explicitly given pair $(\mathrm{pr}_1(q), \mathrm{pr}_2(q)) : \sum_{x:A} P(x)$. The first projection is not so hard to construct; in terms of the functional form of the elimination rule, it is given by

$$\mathrm{pr}_1 :\equiv \mathrm{ind}'_{\sum_{x:A} P} \left(\lambda \left(q : \sum_{x:A} P(x) \right).A,\ \lambda(x:A).\lambda(y:P(x)).x \right) \;:\; \left(\sum_{x:A} P(x) \right) \to A.$$

Note that we have not specified any third argument; this means that the resulting expression is still considered as a function of that argument, which means that it is of type $\prod_{q:\sum_{x:A} P(x)} A$, an element of which can also be regarded as a function $(\sum_{x:A} P(x)) \to A$ (slightly informally). The computation rule for the dependent pair type guarantees that if one applies the projection to an explicitly given pair $(x, y) : \sum_{x:A} P(x)$ for $x : A$ and $y : P(x)$, then $\mathrm{pr}_1((x, y)) \equiv x$.

Informally, one can phrase this construction as follows: constructing an element of A from every given $q : \sum_{x:A} P(x)$ can be done by induction on q. This lets us consider the case $q \equiv (x, y)$ without loss of generality, where $x : A$ and $y : P(x)$. In this case, we define the function to return x itself. Finally, we can consider this construction as a function of q, and this function has type $(\sum_{x:A} P(x)) \to A$.

The second projection is a bit more challenging, since the type of the result depends on q; this type is given by $P(\mathrm{pr}_1(q))$. In other words, one can construct the second projection as being given by

$$\mathrm{pr}_2 :\equiv \mathrm{ind}'_{\sum_{x:A} P(x)}\left(\lambda\left(q : \sum_{x:A} P(x)\right).P(\mathrm{pr}_1(q)),\ \lambda(x : A).\lambda(y : P(x)).y\right) : \prod_{q:\sum_{x:A} P(x)} P(\mathrm{pr}_1(q)).$$

Similarly to pr_1, the computation rule implies that $\mathrm{pr}_2((x, y)) \equiv y$ in this situation. Note that in order to show that the function $\lambda(x : A).\lambda(y : P(x)).y$ has the required type $\prod_{x:A} \prod_{y:P(x)} P(\mathrm{pr}_1((x, y)))$, one needs to use the computation rule for pr_1.

Informally, one can phrase this construction as follows: one can construct an element of $P(\mathrm{pr}_1(q))$ for every $q : \sum_{x:A} P(x)$ by induction on q, which reduces to the problem to the case $q \equiv (x, y)$. In this case, we can simply return y. The resulting element can be constructed as a function of q.

Topic 5: Inductive types

The notion of *induction* should be familiar from induction proofs over the natural numbers. The basic idea is that in order to prove a statement about all natural numbers, it is sufficient to:

- prove it in the case that the number is zero,
- prove it in the case that the number is the successor of another number for which the statement has already been proven.

Under the propositions-as-types correspondence, this also gives rise to the principle of *recursion*: in order to construct a sequence of elements of a set, it is sufficient to:

- construct an element of the set corresponding to the first (or 'zeroth') element of the sequence,
- construct a function which takes an already constructed element of the sequence as input and outputs the following element of the sequence.

We will see how this induction and recursion correspond to the elimination rule that we will postulate for the type of natural numbers. In fact, we will see that induction and recursion are not at all only a property of the natural numbers; in fact, there are many *inductive types* for which induction proofs and recursive constructions make sense.

In fact, we have already encountered many examples of inductive types: most of the types that we have considered so far can be regarded as inductive types, assuming that one uses a sufficiently general definition of what an inductive type is. This statement explains the notation '$\mathrm{ind}(\ldots)$' for elements obtained via application of an elimination rule. Even universes are a kind of inductive type, when defined in a slightly different way (Tarski-style). There are so many schemes which formalize the idea of an inductive type at a very high level of generality, like inductive-recursive and inductive-inductive types, that we can impossibly discuss all of these. Hence we limit ourselves to giving an informal idea of what inductive types and their various generalizations are about and refer to the research literature for more details. We will meet one generalization of inductive types, namely *higher inductive types*, later on as a new kind of construction specific to *homotopy* type theory.

The natural numbers as an inductive type. An inductive type is specified by a collection of *constructors* which tell us how to obtain elements of the type under consideration. As a running example, we consider the type $\mathbb{N}$ of natural numbers, for which there are two constructors:

- $0 : \mathbb{N}$.
- $\mathtt{succ} : \mathbb{N} \to \mathbb{N}$.

In words: 0 is a natural number, and whenever x is a natural number, then so is $\mathtt{succ}(x)$, which stands for the successor of x. The important point here is that both constructors return a natural number. Intuitively, the inductive type $\mathbb{N}$ is then specified as 'freely generated' by these two constructors in the sense that the elements of $\mathbb{N}$ correspond precisely to all those expressions which can be formed by repeatedly applying these constructors:

$$0, \quad \mathtt{succ}(0), \quad \mathtt{succ}(\mathtt{succ}(0)), \quad \mathtt{succ}(\mathtt{succ}(\mathtt{succ}(0))), \quad \ldots$$

This intuition applies to any inductive type: roughly speaking, its elements are the expressions formed by repeatedly applying the constructors, where a constructor may take any number of arguments from the inductive type itself.

As is always in type theory, this intuitive explanation of what the elements of the type $\mathbb{N}$ are does not constitute a definition of $\mathbb{N}$ in type theory, since types cannot be defined via their elements. The definition of $\mathbb{N}$ is rather given by specifying a set of new inference rules which refer to $\mathbb{N}$. In line with the idea of making our exposition more and more informal, while the reader should retain the ability to fill in all formal details if necessary, we state these rules now in a semi-informal manner, which has the advantage of being more comprehensible to a human than the inference rules written out in all detail.

1. Formation rule: $\mathbb{N} : \mathcal{U}$ in any context Γ.
2. Introduction rules:
   - $0 : \mathbb{N}$ in any context.
   - In a context in which $x : \mathbb{N}$, also $\mathtt{succ}(x) : \mathbb{N}$.
3. Elimination rule: In order to construct an element of $C(x)$ for a dependent type $C : \mathbb{N} \to \mathcal{U}$ and a given $x : \mathbb{N}$, it is enough to construct an element of $z : C(0)$ and for every $n : \mathbb{N}$, a function $f(n) : C(n) \to C(\mathtt{succ}(n))$. This data gives rise to $\mathrm{ind}(C, z, f, x) : C(x)$.
4. Computation rule: In terms of the data used in the elimination rule, we have

$$\mathrm{ind}(C, z, f, 0) \equiv z : C(0), \qquad \mathrm{ind}(C, z, f, \mathtt{succ}(x)) \equiv f(x, \mathrm{ind}(C, z, f, x)) : C(\mathtt{succ}(x)).$$

As for all inductive types, the basic idea behind the elimination and computation rules is this: in order to prove a statement about a particular natural number x, it is enough to prove it for the 'base cases', i.e. for those elements of the type which are obtained by applying the introduction rule; since x is arbitrary, this can also be interpreted as proving a statement about *all* natural numbers. In case that this introduction rule takes one or more arguments from the type itself, then corresponding induction hypotheses may be assumed. If one plugs in one of the base cases into the elimination rule, then the resulting element is judgementally equal to the given data for that base case.

Again, one can construct a universal instance of the elimination rule, which looks like this:

$$\mathrm{ind}'_{\mathbb{N}} : \prod_{C:\mathbb{N}\to\mathcal{U}} C(0) \to \left(\prod_{n:\mathbb{N}} C(n) \to C(\mathtt{succ}(n))\right) \to \prod_{x:\mathbb{N}} C(x).$$

As a final remark, let it be noted that the introduction of $\mathbb{N}$ has modified the type theory, since we have introduced a new set of inference rules. Generally, introducing a new inductive type always modifies the theory due to the introduction of new inference rules. From a different point of view, which is the one that we will adopt, the word 'theory' in 'type theory' refers to the type-theoretic formalism that we have set up, together with the stipulation that all inductive types exist. In particular, there is a variable collection of inference rules in the theory depending on which inductive types have been written down. This is similar to the situation in set theory, where the *axiom schema of replacement* likewise comprises an infinite number of axioms.

Booleans. A very simple inductive type is $\mathtt{Bool}$, the type of Booleans. It can likewise be formed in any context, and there are two constructors not taking any arguments:

- $0 : \mathtt{Bool}$.
- $1 : \mathtt{Bool}$.

The induction principle (elimination rule) for $\mathtt{Bool}$ states that, in order to construct a dependent function in $\prod_{x:\mathtt{Bool}} C(x)$ for some $C : \mathtt{Bool} \to \mathcal{U}$, it suffices to specify elements $y : C(0)$ and $z : C(1)$, and this results in a function $\mathrm{ind}_{\mathtt{Bool}}(C, y, z) : \prod_{x:\mathtt{Bool}} C(x)$. The computation rules are

$$\mathrm{ind}_{\mathtt{Bool}}(C, y, z, 1) \equiv y, \qquad \mathrm{ind}_{\mathtt{Bool}}(C, y, z, 0) \equiv z.$$

Note that we use the symbol '0' in an ambiguous way: strictly speaking, we should not use the same symbol for $0 : \mathbb{N}$ and $0 : \mathtt{Bool}$. However, which one we mean should be clear from the context.

Binary trees. Another inductive type, important in computer programming applications, is $\mathtt{BinTree}$, the type of *binary trees* (see Figure 5.1). There are two ways to construct a binary tree:

- $\mathtt{treeroot} : \mathtt{BinTree}$, the binary tree corresponding to only one node called "root",
- $\mathtt{join} : \mathtt{BinTree} \to \mathtt{BinTree} \to \mathtt{BinTree}$, a constructor taking two binary trees and appending them as the two branches of a new root node.

This is a nice example of an inductive type with a constructor taking more than one argument from the type itself. As an example, Figure 5.1 depicts the binary tree

$$\mathtt{join}\Big(\mathtt{join}(\mathtt{join}(\mathtt{treeroot}, \mathtt{treeroot}), \mathtt{treeroot}),$$
$$\mathtt{join}(\mathtt{join}(\mathtt{treeroot}, \mathtt{join}(\mathtt{treeroot}, \mathtt{treeroot})), \mathtt{join}(\mathtt{treeroot}, \mathtt{treeroot}))\Big).$$

We leave it to the reader to write down the elimination and computation rules for $\mathtt{BinTree}$.

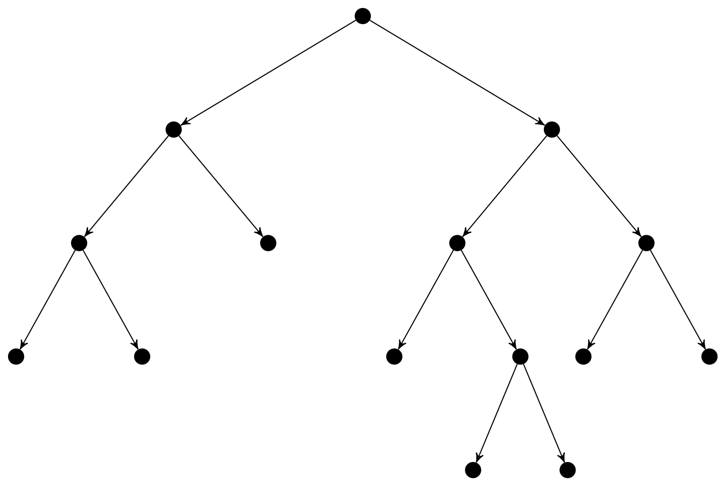

Figure 5.1: Example of a binary tree.

Equalities between inductive types

The types `Bool` and $\mathbf{1}+\mathbf{1}$ seem very similar: the first one is generated by two constructors without any arguments, while the second one is generated by $\mathrm{inl} : \mathbf{1} \to \mathbf{1}+\mathbf{1}$ and $\mathrm{inr} : \mathbf{1} \to \mathbf{1}+\mathbf{1}$, both of which essentially correspond to functions taking no arguments, since the unit type $\mathbf{1}$ has, on the intuitive level, only one element. So how do these two types relate?

As a first step, we construct a map $f : \texttt{Bool} \to \mathbf{1}+\mathbf{1}$. We explain its construction first informally, and then formally. One obtains a function out of `Bool` by specifying its values $f(0)$ and $f(1)$ and then applying the elimination rule. In this case, we can simply define $f(0) :\equiv \mathrm{inl}(\star)$ and $f(1) :\equiv \mathrm{inr}(\star)$.

Formally, we have the following proof tree:

$$\dfrac{\dfrac{\dfrac{\vdots}{y : \texttt{Bool}, x : \texttt{Bool} \vdash \mathbf{1}+\mathbf{1} : \mathcal{U}} \quad \dfrac{\vdots}{y : \texttt{Bool} \vdash \mathrm{inl}(\star) : \mathbf{1}+\mathbf{1}} \quad \dfrac{\vdots}{y : \texttt{Bool}, \vdash \mathrm{inr}(\star) : \mathbf{1}+\mathbf{1}} \quad \dfrac{\vdots}{y : \texttt{Bool} \vdash y : \texttt{Bool}}}{y : \texttt{Bool} \vdash \mathrm{ind}_{\texttt{Bool}}(x.\mathbf{1}+\mathbf{1}, \mathrm{inl}(\star), \mathrm{inr}(\star), y) : \mathbf{1}+\mathbf{1}}}{\vdash \lambda(y : \texttt{Bool}).\mathrm{ind}_{\texttt{Bool}}(x.\mathbf{1}+\mathbf{1}, \mathrm{inl}(\star), \mathrm{inr}(\star), y) : \texttt{Bool} \to \mathbf{1}+\mathbf{1}}$$

In terms of the functional form, we also could have written the desired function more concisely as $\mathrm{ind}'_{\texttt{Bool}}(\lambda x.\mathbf{1} + \mathbf{1}, \mathrm{inl}(\star), \mathrm{inr}(\star)) : \texttt{Bool} \to \mathbf{1}+\mathbf{1}$. Summing up, we constructed $f : \texttt{Bool} \to \mathbf{1}+\mathbf{1}$, simply by mapping $0 \mapsto \mathrm{inl}(\star)$ and $1 \mapsto \mathrm{inr}(\star)$ and applying induction over `Bool`.

Similarly, we can construct a function $g : \mathbf{1}+\mathbf{1} \to \texttt{Bool}$. Informally, we define $g(z)$ for $z : \mathbf{1}+\mathbf{1}$ by induction on z, which reduces the problem to constructing $g(z)$ in the cases $z \equiv \mathrm{inl}(x)$ and $z \equiv \mathrm{inr}(y)$ for $x, y : \mathbf{1}$. In the former case, we put $g(\mathrm{inl}(x)) :\equiv 0$, while in the latter case, $g(\mathrm{inr}(y)) :\equiv 1$. Formally, this is expressed by the following proof tree:

$$\dfrac{\dfrac{\dfrac{\vdots}{x : \mathbf{1}+\mathbf{1}, p : \mathbf{1}+\mathbf{1} \vdash \texttt{Bool} : \mathcal{U}} \quad \dfrac{\vdots}{x : \mathbf{1}+\mathbf{1}, a : \mathbf{1} \vdash 0 : \texttt{Bool}} \quad \dfrac{\vdots}{x : \mathbf{1}+\mathbf{1}, b : \mathbf{1} \vdash 1 : \texttt{Bool}} \quad \dfrac{\vdots}{x : \mathbf{1}+\mathbf{1} \vdash x : \mathbf{1}+\mathbf{1}}}{x : \mathbf{1}+\mathbf{1} \vdash \mathrm{ind}_{\mathbf{1}+\mathbf{1}}(p.\texttt{Bool}, a.0, b.1, x) : \texttt{Bool}}}{\vdash \lambda(x : \mathbf{1}+\mathbf{1}).\mathrm{ind}_{\mathbf{1}+\mathbf{1}}(p.\texttt{Bool}, a.0, b.1, x) : \mathbf{1}+\mathbf{1} \to \texttt{Bool}}$$

In terms of the functional form, we could also have written the desired function more concisely as $\mathrm{ind}'_{\mathbf{1}+\mathbf{1}}(\lambda p.\texttt{Bool}, \lambda a.0, \lambda b.1) : \mathbf{1}+\mathbf{1} \to \texttt{Bool}$.

We can now consider the compositions $g \circ f : \texttt{Bool} \to \texttt{Bool}$ and $f \circ g : \mathbf{1}+\mathbf{1} \to \mathbf{1}+\mathbf{1}$ and ask whether they are judgementally equal to the identity function $\lambda x.x$. For the first composition, we obtain, for any $x : \texttt{Bool}$,

$$(g \circ f)(x) \equiv g(f(x)) \equiv g(\mathrm{ind}_{\texttt{Bool}}(\mathbf{1}+\mathbf{1}, \mathrm{inl}(\star), \mathrm{inr}(\star), x)).$$

However, already when trying to evaluate this further, we are stuck: we cannot apply any computation rule since neither x nor the argument of g is a 'base case' obtained by applying a constructor. Of course, intuitively we know that x corresponds to one of the base cases. But this is merely an intuitive restatement of the elimination rule, which lets us reduce the proof of a proposition to proving it in each base case. And the above judgemental equality is not a proposition, since it is not itself a type, hence the elimination rule is not applicable!

A similarly vexing problem arises for the other composition $f \circ g$. For any $x : \mathbf{1} + \mathbf{1}$, we have

$$(f \circ g)(x) \equiv f(g(x)) \equiv f(\mathrm{ind}_{\mathbf{1}+\mathbf{1}}(p.\mathtt{Bool}, 0, 1, x)),$$

and no computation rule applies in this situation.

There are many other situation with inductive types which 'ought to' be the same, but really are not. We will soon learn how to properly deal with this kind of problem using a notion of *propositional equality* together with the *univalence axiom*. These will indeed give rise to an equality between the types under consideration, and we will be able to prove the equation $\mathbf{1} + \mathbf{1} = \mathtt{Bool}$.

Inductive Type Families

Coproducts. Also the coproduct type $A + B$ can similarly be considered as an inductive type with two constructors:

- $\mathrm{inl} : A \to A + B$.
- $\mathrm{inr} : B \to A + B$.

Again, for any dependent type $C : A + B \to \mathcal{U}$, the elimination rule tells us that in order to construct an element of $C(x)$ for any given $x : A + B$, or equivalently a dependent function in $\prod_{x:A+B} C(x)$, it is enough to construct dependent functions $f : \prod_{a:A} C(\mathrm{inl}(a))$ and $g : \prod_{b:B} C(\mathrm{inr}(b))$, since these give rise to $\mathrm{ind}_{A+B}(C, f, g) : \prod_{x:A+B} C(x)$. The computation rule tells us that

$$\mathrm{ind}_{A+B}(C, f, g, \mathrm{inl}(a)) \equiv f(a), \qquad \mathrm{ind}_{A+B}(C, f, g, \mathrm{inr}(b)) \equiv g(b).$$

But actually, it is better not to consider $A + B$ as a single type: rather, it is a whole *family* of inductive types *parametrized* by A and B, corresponding to the fact that the formation rule for "+" involves A and B as types that need to be defined in the context under consideration. What this means is that we actually get a family of types $+ : \mathcal{U} \to \mathcal{U} \to \mathcal{U}$, taking two types as input and outputting their coproduct.

Lists. Another example of an inductive type depending on another type as parameter is $\mathtt{List}(A)$, the type of all finite sequences (or lists) of elements of A. For this type, the constructors are:

- $\mathtt{nil} : \mathtt{List}(A)$ constructs the empty list,
- $\mathtt{cons} : A \to \mathtt{List}(A) \to \mathtt{List}(A)$, which maps an element $a : A$ and a list $L : \mathtt{List}(A)$ to the list $\mathtt{cons}(a, L) : \mathtt{List}(A)$, which corresponds to L together with a appended at the end. (Or appended at the beginning—this is a matter of interpretation.)

Strictly speaking, one should indicate the dependence on A in these constructors and write $\mathtt{nil}(A)$ and $\mathtt{cons}(A)$ instead of $\mathtt{nil}$ and $\mathtt{cons}$, respectively. But as long as one considers only a fixed A, no confusion can arise and there is no harm in omitting A as an explicit argument.

Lists of fixed length. There is another way in which an inductive type may depend on another type. To see why this might be the case, note that all the inductive types that we defined so far have the characteristic that the constructors have fixed codomain. But sometimes, it is desirable to define a whole family of inductive types *at once* by having constructors which take one or more arguments in some type(s) of this family, while returning an element of a potentially different type in the same family. In this case, we say that the type family is a dependent type *indexed* by the base type. Every parameter can also be considered to be an index, although in a trivial way.

As an example of this, we reconsider lists, but this time as indexed by their length: for any $n : \mathbb{N}$, there should be a type of lists $\mathtt{List}'(A, n)$ such that appending an element to a list should be considered a function $A \to \prod_{n:\mathbb{N}} \mathtt{List}'(A, n) \to \mathtt{List}'(A, \mathtt{succ}(n))$. In other words, $\mathtt{List}'(A)$ is a type family indexed by $\mathbb{N}$ and having constructors

- $\mathtt{nil}' : \mathtt{List}'(A, 0)$,
- $\mathtt{cons}' : A \to \prod_{n:\mathbb{N}} \mathtt{List}'(A, n) \to \mathtt{List}'(A, \mathtt{succ}(n))$.

In Coq, this looks as follows:

```
Inductive List' (A : Type) : nat → Type :=
| nil' : List' A 0
| cons' : A → forall n : nat, List' A n → List' A (S n).
```

More precisely, the functional forms of the inference rules are as follows: the formation rule is

$$\mathtt{List}' : \mathcal{U} \to \mathbb{N} \to \mathcal{U},$$

since $\mathtt{List}'$ produces a type $\mathtt{List}'(A,n)$ from any $A : \mathcal{U}$ and any $n : \mathbb{N}$. The introduction rules are (dependent) functions

$$\mathtt{nil}' : \prod_{A:\mathcal{U}} \mathtt{List}'(A,0), \qquad \mathtt{cons}' : \prod_{A:\mathcal{U}} A \to \prod_{n:\mathbb{N}} \mathtt{List}'(A,n) \to \mathtt{List}'(A,\mathtt{succ}(n)).$$

For fixed A, the universal instance of the elimination rule is

$$\mathrm{ind}_{\mathtt{List}'(A)} : \prod_{C:\prod_{n:\mathbb{N}}\mathtt{List}'(A,n)\to\mathcal{U}} C(0,\mathtt{nil}') \\ \to \left(\prod_{m:\mathbb{N},\, x:A,\, u:\mathtt{List}'(A,m)} C(m,u) \to C(\mathtt{succ}(m),\mathtt{cons}'(x,m,u)) \right) \to \prod_{n:\mathbb{N}} \prod_{x:\mathtt{List}'(A,n)} C(n,x).$$

Finally, the two computation rules state that

$$\mathrm{ind}_{\mathtt{List}'(A)}(C,b,s,0,\mathtt{nil}'(A)) \equiv b,$$

$$\mathrm{ind}_{\mathtt{List}'(A)}(C,b,s,\mathtt{succ}(m),\mathtt{cons}'(x,m,u)) \equiv s(m,x,u,\mathrm{ind}_{\mathtt{List}'(A)}(C,b,s,m,u)).$$

The dependent type $\mathtt{List}'(A) : \mathbb{N} \to \mathcal{U}$ refines $\mathtt{List}(A) : \mathcal{U}$ in the sense that $\sum_{n:\mathbb{N}} \mathtt{List}'(A,n)$ obeys the same rules as $\mathtt{List}(A)$, but $\mathtt{List}'(A)$ contains more information in the sense that it also indexes every list by its length. In fact, it is straightforward to construct functions

$$\mathtt{List}(A) \to \sum_{n:\mathbb{N}} \mathtt{List}'(A,n), \qquad \left(\sum_{n:\mathbb{N}} \mathtt{List}'(A,n)\right) \to \mathtt{List}(A),$$

by applying elimination rules on the domain and the appropriate introduction rules on the codomain, and these functions 'ought to be' mutually inverse to each other, thereby defining an isomorphism between $\mathtt{List}(A)$ and $\sum_{n:\mathbb{N}} \mathtt{List}'(A,n)$. But again, the computation rules are not powerful enough to show that both compositions yield the identity, and therefore we will have to resort to the upcoming propositional equality and univalence axiom. This is exactly what we will start doing next time: our main example of an inductive type family are the so-called *identity types* which will give us a the new notion of propositional equality.

Example: a bit of arithmetic

In order to become more familiar with inductive types and induction principles, let us define the basic arithmetic of natural numbers. First of all, we would like to try and define a function

$$\mathtt{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

implementing addition of natural numbers. By induction over the first argument, it is sufficient to define $\mathtt{add}(0) : \mathbb{N} \to \mathbb{N}$, and $\mathtt{add}(\mathtt{succ}(n)) : \mathbb{N} \to \mathbb{N}$ in terms of $\mathtt{add}(n) : \mathbb{N} \to \mathbb{N}$. For the first, we simply take it to be the identity function,

$$\mathtt{add}(0) :\equiv \lambda n.n : \mathbb{N} \to \mathbb{N},$$

since adding 0 to any other number should reproduce that number. For the other function, we define

$$\mathtt{add}(\mathtt{succ}(n)) :\equiv \lambda m.\mathtt{succ}(\mathtt{add}(n,m)) : \mathbb{N} \to \mathbb{N},$$

since adding $\mathtt{succ}(n)$ to any other number should yield the successor of the sum of n and the other number. As an alternative (and not judgementally equal!) definition, we could have defined either or both of $\mathtt{add}(0)$ and $\mathtt{add}(\mathtt{succ}(n))$ also by a second induction over their remaining argument.

Defining multiplication can now be done similarly in terms of addition. We define

$$\mathtt{mult}(0) :\equiv \lambda n.0 : \mathbb{N} \to \mathbb{N},$$

and

$$\mathtt{mult}(\mathtt{succ}(n)) :\equiv \lambda m.\mathtt{add}(n,\mathtt{mult}(n,m)) : \mathbb{N} \to \mathbb{N},$$

and by induction, this yields a function $\mathtt{mult} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$. Again, there are variants on this definition, but they all result in the same theory of arithmetic, which is precisely the usual one.

Topic 6: Identity types

We can now move on to one of the central concepts which turns type theory into *homotopy* type theory, namely the concept of *identity types*: for every type A and $x, y : A$, there is an identity type $x =_A y$ which turns the statement that x is equal to y into a type itself, as one would expect from the propositions-as-types correspondence. The elements of this type can be thought of as witnesses of the equality of x and y. One of the main features of identity types, which it has in common with homotopy, is that one can iterate them: for two witnesses $p, q : x =_A y$, one can ask whether these two witnesses are themselves equal by considering the identity type $p =_{x=_A y} q$.

Strictly speaking, there are two different definitions of identity types. Both of these are indexed by $A : \mathcal{U}$ and by the two elements $x, y : A$ that one wants to compare. In other words, both of these variants have type formers given by functions $\prod_{A:\mathcal{U}} A \to A \to \mathcal{U}$. In order to distinguish these two inductive type families, we start by writing these identity types as $\mathrm{id}_u(A, x, y)$ and $\mathrm{id}_b(A, x, y)$, where the subscript disambiguates the two different versions, called 'based' and 'unbased' identity types.

Based identity types. The based identity type $\mathrm{id}_b : \prod_{A:\mathcal{U}} A \to A \to \mathcal{U}$ has the first two arguments $A : \mathcal{U}$ and $x : A$ as parameters, while the third argument $y : A$ is an index.

More concretely, for every fixed $A : \mathcal{U}$ and $x : A$, the dependent type $\mathrm{id}_b(A, x) : A \to \mathcal{U}$ is an inductive type family with only one constructor,

- $\mathrm{refl}_x : \mathrm{id}_b(A, x, x)$,

which is a canonical witness of the equality of x to itself. Since there are no other constructors at all, this may seem very similar to the unit type—but as an inductive type *family*, it behaves quite differently and we will see that it equips the theory with a very rich structure.

The elimination rule then states the following: for every dependent type $C : \prod_{y:A} \mathrm{id}_b(A, x, y) \to \mathcal{U}$, one can construct an element of $C(y, p)$ for any given $y : A$ and $p : \mathrm{id}_b(A, x, y)$ by reducing to the 'base case' given by $y \equiv x$ and $p \equiv \mathrm{refl}_x$. In other words, there is an induction principle whose functional form looks like this:

$$\mathrm{ind}_{\mathrm{id}_b(A,x)} : \prod_{C:\prod_{y:A} \mathrm{id}_b(A,x,y)\to\mathcal{U}} C(x, \mathrm{refl}_x) \to \prod_{y:A} \prod_{p:\mathrm{id}_b(A,x,y)} C(y, p). \tag{6.1}$$

As an example application, we show that propositional equality is symmetric, meaning that we construct a dependent function

$$\prod_{x,y:A} \mathrm{id}_b(A, x, y) \to \mathrm{id}_b(A, y, x)$$

as follows. For given $x, y : A$ and $p : \mathrm{id}_b(A, x, y)$, we can apply the elimination rule in order to reduce to the case that $y \equiv x$ and $p \equiv \mathrm{refl}_x$, in which case it remains to construct an element of $\mathrm{id}_b(A, x, x)$, for which we can take again refl_x. More precisely, we have

$$\lambda x.\mathrm{ind}_{\mathrm{id}_b(A,x)}(\lambda y.\lambda p.\mathrm{id}_b(A, y, x), \mathrm{refl}_x) \; : \; \prod_{x,y:A} \mathrm{id}_b(A, x, y) \to \mathrm{id}_b(A, y, x).$$

So what does the induction principle (6.1) do, intuitively? In case in which we apply it to a dependent type $C : A \to \mathcal{U}$, i.e. a type whose dependence on $\mathrm{id}_b(A, x, y)$ is trivial, it specializes to an induction principle of the form

$$C(x) \to \prod_{y:A} \prod_{p:\mathrm{id}_b(A,x,y)} C(y),$$

which has a very simple interpretation: if we have an element of $C(x)$, then we can also obtain an element of $C(y)$ as soon as an identity between x and y is given; usually, applying such a procedure is known as *substitution*! Or, in logical terms: whenever $C(x)$ is a proposition, then the variable x can be *substituted* by any y equal to it, and this induction principle turns any proof of $C(x)$ into a proof of $C(y)$.

Finally, the computation rule says that if we apply the elimination rule in a 'base case', then we recover the given data. In other words,

$$\mathrm{ind}_{\mathrm{id}_b(A,x)}(C, d, x, \mathrm{refl}_x) \equiv d.$$

where $d : C(x, \mathrm{refl}_x)$ is arbitrary.

Unbased identity types. The other incarnation of identity types is the *unbased* identity type $\mathrm{id}_u : \prod_{A:\mathcal{U}} A \to A \to \mathcal{U}$, in which only the first argument is a parameter, while both other arguments are indices. More concretely, $\mathrm{id}_u(A) : A \to A \to \mathcal{U}$ is an inductive type family with again only one constructor,

- $\mathrm{refl} : \prod_{x:A} \mathrm{id}_u(A, x, x)$.

For consistency of notation with based identity types, we also write refl_x instead of $\mathrm{refl}(x)$, although this creates the ambiguity that the expression refl_x stands for the constructor both in the based and unbased cases. But since we will soon stop distinguishing between based and unbased identity types anyway, this ambiguity is actually desired. Hence, the functional form of the elimination rule takes the form

$$\mathrm{ind}_{\mathrm{id}_u(A)} : \prod_{C:\prod_{x,y:A} \mathrm{id}_u(A,x,y) \to \mathcal{U}} \left(\prod_{x:A} C(x, x, \mathrm{refl}_x) \right) \longrightarrow \prod_{x,y:A} \prod_{p:\mathrm{id}_u(A,x,y)} C(x, y, p).$$

The computation rule is this:

$$\mathrm{ind}_{\mathrm{id}_u(A)}(C, s, x, x, \mathrm{refl}_x) \equiv s,$$

where $s : \prod_{x:A} C(x, x, \mathrm{refl}_x)$ is arbitrary.

Equivalence of based and unbased path induction

So how do the based and unbased identity types relate? We will now show that the based identity type also satisfies the inference rules of the unbased identity type and vice versa, where both types are considered as inductive type families $\prod_{A:\mathcal{U}} A \to A \to \mathcal{U}$. This makes the situation analogous to the one for `Bool` vs. $\mathbf{1} + \mathbf{1}$, which also satisfy the same inference rules. Using the notion of equivalence and the upcoming univalence axiom, it will become possible to show that the types $\mathrm{id}_b(A, x, y)$ and $\mathrm{id}_u(A, x, y)$ are themselves equal, and so there is no need to distinguish between them.

We start by showing that the based identity type satisfies the inference rules of the unbased one. For the formation rule, this is trivial, since the formation rules are the same:

$$\mathrm{id}_b,\ \mathrm{id}_u : \prod_{A:\mathcal{U}} A \to A \to \mathcal{U}.$$

Likewise, when considering all three arguments as parameters/indices of the type family, even the introduction rules are the same:

$$\mathrm{refl} : \prod_{A:\mathcal{U}} \prod_{x:A} \mathrm{id}(A, x, x),$$

where id stands for either id_b or id_u.

For the introduction and computation rules, the situation is much less straighforward. Starting with the elimination rule (6.1), which we rewrite slightly differently by including the dependence on x explicitly,

$$\mathrm{ind}_{\mathrm{id}_b(A)} : \prod_{x:A} \prod_{C:\prod_{y:A} \mathrm{id}_b(A,x,y) \to \mathcal{U}} C(x, \mathrm{refl}_x) \to \prod_{y:A} \prod_{p:\mathrm{id}_b(A,x,y)} C(y, p), \tag{6.2}$$

we will derive an element of the corresponding unbased induction principle,

$$\prod_{D:\prod_{x,y:A} \mathrm{id}_b(A,x,y) \to \mathcal{U}} \left(\prod_{x:A} D(x, x, \mathrm{refl}_x) \right) \to \prod_{x,y:A} \prod_{p:\mathrm{id}_b(A,x,y)} D(x, y, p). \tag{6.3}$$

But this is easy to do, since given any $D : \prod_{x,y:A} \mathrm{id}_b(A, x, y) \to \mathcal{U}$, any $s : \prod_{x:A} D(x, x, \mathrm{refl}_x)$, and any $x : A$, we can construct an element of $D(x, y, p)$ by applying $\mathrm{ind}_{\mathrm{id}_b(A)}$ to x and the dependent type $C :\equiv D(x) : \prod_{y:A} \mathrm{id}_b(A, x, y) \to \mathcal{U}$, and thereby obtaining a function

$$\mathrm{ind}_{\mathrm{id}_b(A)}(x, D(x)) : D(x, x, \mathrm{refl}_x) \to \prod_{y:A} \prod_{p:\mathrm{id}_b(A,x,y)} D(x, y, p),$$

which gives us an element of $D(x,y,p)$ upon evaluating this function on $s(x)$, as desired. In conclusion, this means that the expression

$$\lambda D.\lambda s.\lambda x.\lambda y.\lambda p.\mathrm{ind}_{\mathrm{id}_b(A)}(x, D(x), s(x), y, p)$$

is an element of (6.3). It is now straightforward to show that this expression satisfies the appropriate computation rule: upon evaluating it on some D, s and x, and then on $y \equiv x$ and $p \equiv \mathrm{refl}_x$, one indeed recovers $s(x)$ by the computation rule for $\mathrm{ind}_{\mathrm{id}_b(A)}$.

Showing that $\mathrm{id}_u(A)$ also satisfies the based path induction is more difficult, since the first point $x : A$ in based path induction is fixed, so it is not clear how to apply unbased path induction, which can be applied only in cases in which we have an assumption of the form $\prod_{x:A} D(x, x, \mathrm{refl}_x)$. In order to get around this obstacle, it is helpful to consider *all instances* of based path induction at once, in the sense of considering the type

$$\prod_{x,y:A} \prod_{p:\mathrm{id}_u(A,x,y)} \prod_{C:\prod_{z:A} \mathrm{id}_u(A,x,z)\to\mathcal{U}} C(x, \mathrm{refl}_x) \to C(y, p), \tag{6.4}$$

which lives in the next higher universe $\mathcal{U}'$. By (unbased!) path induction, in order to construct an element of this type it is enough to do so in the case that $y \equiv x$ and $p \equiv \mathrm{refl}_x$, in which case we can simply take

$$\lambda x.\lambda C.\lambda c.c : \prod_{x:A} \prod_{C:\prod_{z:A} \mathrm{id}_u(A,x,z)\to\mathcal{U}} C(x, \mathrm{refl}_x) \to C(x, \mathrm{refl}_x)$$

Now that we have an element of (6.4), we can easily turn it into an element of

$$\prod_{x:A} \prod_{C:\prod_{y:A} \mathrm{id}_u(A,x,y)\to\mathcal{U}} C(x, \mathrm{refl}_x) \to \prod_{y:A} \prod_{p:\mathrm{id}_u(A,x,y)} C(y,p),$$

by considering the element as a function taking several arguments and exchanging the order of these arguments. The computation rule is straightforward to show.

Now that we know that id_b and id_u satisfy the exactly the same inference rules, we can conclude that if we prove some statement for id_b, the same statement will also be provable for id_u in place of id_b, and vice versa. Once we have introduced the univalence axiom, we can show that these two type families are actually themselves equal. So there is no need to distinguish them, and in fact we will omit this distinction from now on and simply write $x =_A y$ for either $\mathrm{id}_b(A, x, y)$ or $\mathrm{id}_u(A, x, y)$ and speak of "the" identity type. Since the type A is clear from the context, we will usually just write $x = y$. Our earlier symmetry result can then be written as $(x = y) \to (y = x)$.

Interpretation in terms of paths and homotopies

Finally, we can now explain the "homotopy" in homotopy type theory. While everything else that we have done so far also applies to many other flavours of type theory, this particular behaviour of identity types that we find here is specific to HoTT. In contrast to more conventional type theories, in which the types are interpreted as sets, types in HoTT behave like *spaces* in the sense of homotopy types or, equivalently, as ∞-groupoids. (These two points of view are equivalent by the homotopy hypothesis.)

So we think of any type A as a topological space whose points are the elements $x, y : A$. While this is nothing else than in intuition if one regards HoTT as a foundation of mathematics, there is also a sense in which it can turned into a precise statement inside conventional foundations by finding a *model* of HoTT in conventional foundations. Doing this is important for knowing that if conventional foundations are consistent, then so is HoTT. This is an important result: no contradiction has been found in all of mathematics as based on conventional foundations in close to 100 years. Since HoTT can be modelled inside conventional foundations, this makes it at least as unlikely that HoTT is inconsistent as it is unlikely that conventional foundations are inconsistent.

In any case, in the interpretation that types are spaces and elements of types are points, an element of an identity type $p : x = y$ corresponds to a *path* moving from the point x to the point y—in particular, the letter p actually stands for "path". The identity type $x = y$ does then itself need to be a space, namely the space of all paths from x to y! For paths $p, q : x = y$, the corresponding identity type between p and q then represents the space of all *paths between paths*; commonly, a path between paths is known as a *homotopy* of paths.

Topic 7: Using identity types

Now we would like to learn how to use identity types and how to do some actual mathematics with them. By now we have essentially introduced all inference rules of HoTT; the main missing ingredient is the univalence axiom. At this stage it is possible to do a substantial amount of mathematics, and some of this we will do here.

Throughout $A, B : \mathcal{U}$ are fixed types.

Example: Propositional uniqueness principles

Recall the *uniqueness principle* for functions, which is an inference rule stating that $f \equiv \lambda x.f(x)$ for any (dependent) function f. For other types, we did not have such uniqueness principles; for example for $u : \mathbf{1}$, such a uniqueness principle would assert that $u \equiv \star$, but we have not postulated this and it is not derivable. However, at this point we can show that such a uniqueness principle holds with propositional equality in place of judgemental equality:

7.1 Theorem. $\prod_{u:\mathbf{1}} u = \star$.

The proof of this is exemplative of many other proofs in HoTT. It is a good example of how one can derive non-trivial equations in HoTT, although the only way to prove an equation is to reduce it in some way to reflexivity.

Proof. In order to find an element $w : \prod_{u:\mathbf{1}} u = \star$, we use induction on u. This reduces the problem to finding $w(\star) : \star = \star$, which we can define to be $w(\star) :\equiv \mathrm{refl}_\star$. □

The same theorem and proof in Coq:

```
Theorem prop_unique : forall u : unit, u = tt.
Proof.
        intro u.
        induction u.
        reflexivity.
Defined.
```

Analogous propositional uniqueness principles can now be derived for many other types as well:

7.2 Theorem. $\prod_{x:A\times B} x = (\mathrm{pr}_1(x), \mathrm{pr}_2(x))$.

Proof. In order to find an element of the type $\prod_{x:A\times B} x = (\mathrm{pr}_1(x), \mathrm{pr}_2(x))$, we use induction on x. This reduces the problem to finding an element

$$v : \prod_{a:A} \prod_{b:B} (a,b) = (\mathrm{pr}_1((a,b)), \mathrm{pr}_2((a,b))).$$

Since $\mathrm{pr}_1((a,b)) \equiv a$ and $\mathrm{pr}_2((a,b)) \equiv b$, we can take $v :\equiv \lambda a.\lambda b.\mathrm{refl}_{(a,b)}$. □

```
Theorem prop_unique_product (A B : Type) : forall x : A * B, x = (fst x, snd x).
Proof.
        intro x.
        induction x.
        simpl.
        reflexivity.
Defined.
```

One can generalize this to the dependent pair type: for any dependent type $P : A \to \mathcal{U}$, one can similarly derive $x = (\mathrm{pr}_1(x), \mathrm{pr}_2(x))$ for any $x : \sum_{a:A} P(a)$. We will prove a more general statement later on, when we characterize the identity type $p = q$ for $p, q : \sum_{a:A} P(a)$.

Functions acting on paths

If we have $x, y : A$ together with $p : x = y$ and a function $f : A \to B$, does this yield an element of $f(x) = f(y)$? In other words, does function application preserve propositional equality? In terms of the topological interpretation, one can say that we now analyse how functions act on paths.

7.3 Lemma. *For any function $f : A \to B$ and $x, y : A$, there is a function*

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

defined such that for all $x : A$, $\mathrm{ap}_f(\mathrm{refl}_x) \equiv \mathrm{refl}_{f(x)}$.

Strictly speaking, we also need to consider the two elements $x, y : A$ as arguments of the function ap_f, whose type therefore should be

$$\mathrm{ap}_f : \prod_{x,y:A} (x =_A y) \to (f(x) =_B f(y)).$$

But since the first two arguments x and y are determined by the third argument $p : x = y$, we consider x and y to be "implicit" arguments which are not written down explicitly, except for in the following proof.

Proof. For any $x, y : A$ and any $p : x =_A y$, we want to find $\mathrm{ap}_f(x, y, p) : f(x) =_B f(y)$. By induction on p as an element of the (say, unbased) identity type, it is sufficient to do so in the case that $y \equiv x$ and $p \equiv \mathrm{refl}_x$. But in this case, we can put $\mathrm{ap}_f(x, x, \mathrm{refl}_x) :\equiv \mathrm{refl}_{f(x)} : f(x) =_B f(x)$. □

In Coq, the statement and proof look as follows:

```
Lemma ap {A B : Type} (f : A → B) : forall x y : A, (x = y) → (f(x) = f(y)).
Proof.
        intros x y p.
        induction p.
        reflexivity.
Defined.
```

We interpret Lemma 7.3 as saying that *every* function f is automatically continuous, since it maps paths to paths! So in HoTT, it is impossible to construct a function that is not continuous.

Example: commutativity of addition

As the possibly first example of actual mathematics in HoTT, let us revisit the natural numbers and prove that addition is commutative. The following proof structure coincides exactly with how one would prove commutativity of addition in conventional foundations! This is true for many other things as well.

7.4 Lemma. $\prod_{x,y:\mathbb{N}} \mathtt{succ}(y + x) = y + \mathtt{succ}(x)$.

Recall that we had defined addition $x + y :\equiv \mathtt{add}(x, y)$ by induction over the first argument, using the definitions $\mathtt{add}(0, y) :\equiv y$ and $\mathtt{add}(\mathtt{succ}(x), y) :\equiv \mathtt{succ}(\mathtt{add}(x, y))$.

Proof. In order to find an element of the type $\mathtt{succ}(y + x) = y + \mathtt{succ}(x)$ for any $x, y : \mathbb{N}$, we can use induction on y. For $y \equiv 0$, we have $\mathrm{refl}_{\mathtt{succ}(x)} : \mathtt{succ}(x) = \mathtt{succ}(x)$, and by the definition of $+$ we also have judgemental equalities $\mathtt{succ}(0 + x) \equiv \mathtt{succ}(x)$ and $0 + \mathtt{succ}(x) \equiv \mathtt{succ}(x)$. Therefore, we also have $\mathrm{refl}_{\mathtt{succ}(x)} : \mathtt{succ}(0 + x) = 0 + \mathtt{succ}(x)$.

For the induction step, we need to show that one can construct an element of the identity type $\mathtt{succ}(\mathtt{succ}(y) + x) = \mathtt{succ}(y) + \mathtt{succ}(x)$ from any given element $v : \mathtt{succ}(y + x) = y + \mathtt{succ}(x)$. Since $\mathrm{ap}_{\mathtt{succ}}(v) : \mathtt{succ}(\mathtt{succ}(y + x)) = \mathtt{succ}(y + \mathtt{succ}(x))$ and we have judgemental equalities

$$\mathtt{succ}(\mathtt{succ}(y) + x) \equiv \mathtt{succ}(\mathtt{succ}(y + x)), \qquad \mathtt{succ}(y) + \mathtt{succ}(x) \equiv \mathtt{succ}(y + \mathtt{succ}(x))$$

by the definition of $+$, we also have $\mathrm{ap}_{\mathtt{succ}}(v) : \mathtt{succ}(\mathtt{succ}(y) + x) = \mathtt{succ}(y) + \mathtt{succ}(x)$. □

Let us write $\mathtt{add\_succ} : \prod_{x,y:\mathbb{N}} y + \mathtt{succ}(x) = \mathtt{succ}(y + x)$ for the element constructed in the proof.
The same statement and proof in Coq:

```
Lemma add_succ : forall x y : nat, x + S y = S (x + y).
Proof.
        intros x y.
        induction x.
                simpl.
                reflexivity.

                simpl.
                apply (ap S).
                exact IHx.
Defined.
```

7.5 Theorem. $\prod_{x,y:\mathbb{N}} x + y = y + x$.

Proof. In order to find an element of the type $x + y = y + x$ for any x and y, we can use induction on x. This reduces the problem to finding elements of the following two types:

- $0 + y = y + 0$,
- $(x + y = y + x) \longrightarrow (\mathtt{succ}(x) + y = y + \mathtt{succ}(x))$

We start with the first. Since $+$ was defined by induction over the first argument, we have $0 + y \equiv y$, so it remains to show that $y = y + 0$. This in turn can be shown by induction on y as follows. For $y \equiv 0$, we have $0 \equiv 0 + 0$ again by definition of $+$, and hence we have a judgemental equality of types $(0 = 0) \equiv (0 = 0 + 0)$, and therefore $\mathrm{refl}_0 : 0 = 0 + 0$. Concerning the induction step, if we have an equality witness $w : y = y + 0$, then we obtain $\mathrm{ap}_{\mathtt{succ}}(w) : \mathtt{succ}(y) = \mathtt{succ}(y + 0)$. Since $\mathtt{succ}(y) + 0 \equiv \mathtt{succ}(y + 0)$ by the inductive definition of $+$, we also have $\mathrm{ap}_{\mathtt{succ}}(w) : \mathtt{succ}(y) = \mathtt{succ}(y) + 0$.

Now for the main induction step. Suppose that we have $v : x + y = y + x$, and we want to find an element of $\mathtt{succ}(x) + y = y + \mathtt{succ}(x)$. For every y, we have

$$\mathrm{ap}_{\mathtt{succ}}(v) : \mathtt{succ}(x) + y = \mathtt{succ}(y + x),$$

where we have used $\mathtt{succ}(x + y) \equiv \mathtt{succ}(x) + y$ by the inductive definition of $+$, and

$$\mathtt{add\_succ}(x, y) : \mathtt{succ}(y + x) = y + \mathtt{succ}(x)$$

thanks to Lemma 7.4. The claim now follows from transitivity of equality; this is what we get to next. □

Concatenating paths

Last time, we already saw that propositional equality is symmetric, in the sense that there exists a canonical function $(x = y) \to (y = x)$ which maps any path $p : x = y$ to its "inverse" $p^{-1} : y = x$. In the path interpretation, this corresponds to the fact that every path can be traversed forwards or backwards. Recall that taking the inverse is a function

$$\_^{-1} : \prod_{x,y:A} (x = y) \to (y = x)$$

which can be defined by (based or unbased) path induction, which reduces the problem to defining the function on $y \equiv x$ and refl_x, in which case we define the function value to be refl_x itself. Although x and y (and even A) should formally also be considered as arguments of this function, we omit them for brevity; if one knows p, one can deduce x and y from the type of p, and then A as the type of x and y. To summarize, the function $p \mapsto p^{-1}$ is defined by path induction on p, together with the stipulation that $\mathrm{refl}_x^{-1} :\equiv \mathrm{refl}_x$ for every $x : A$.

Similarly, it is possible to *concatenate* paths: if p is a path from x to y and q is a path from y to z, then there is a concatenated path $p \cdot q$ from x to z. This is indeed the case: for any $A : \mathcal{U}$, one can construct a function

$$\_\cdot\_ : \prod_{x,y,z:A} (x = y) \to (y = z) \to (x = z) \tag{7.1}$$

as follows. By permuting the z argument with the $(x = y)$-argument, it is enough to construct this function in the form

$$\_\cdot\_ : \prod_{x,y} (x = y) \to \prod_{z:A} (y = z) \to (x = z).$$

Applying (unbased) path induction lets us assume that we are dealing with the case $y \equiv x$ and $\mathsf{refl}_x : x = x$, in which case we simply need to construct an element of the type

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

for which we could simply take the identity function for every z. However, it turns out to be better to *not* use the identity function here, but rather to define such a function again through path induction, which allows us to put $z \equiv x$ and assume $\mathsf{refl}_x : x = x$, in which case we take the value of the function to again be $\mathsf{refl}_x : x = x$.

The reason for not using the identity function, but instead using another path induction, is mainly aesthetics: with the identity function, the resulting computation rule for $\cdot$ is,

$$\mathsf{refl}_x \cdot q \equiv q$$

for all $q : y = z$, while with a second path induction, the computation rule is merely

$$\mathsf{refl}_x \cdot \mathsf{refl}_x \equiv \mathsf{refl}_x,$$

which we prefer, since it makes both arguments behave in the same way.

In logical terms, one can interpret (7.1) as the transitivity of equality: if x is equal to y and y is equal to z, then x is also equal to z. The function (7.1) can be interpreted as converting any two proofs $p : x = y$ and $q : y = z$ into $p \cdot q : x = z$. Generally speaking, thinking in terms of paths between points in a space rather than proofs of equalities is the preferred intuition, and so we stick with this one from now on.

7.6 Theorem. *Concatenation of paths is associative: for any $w, x, y, z : A$ and $p : w = x$, $q : x = y$, $r : y = z$,*

$$p \cdot (q \cdot r) = (p \cdot q) \cdot r.$$

Strictly speaking, what me mean by this theorem is that we claim to be able to construct an element of the type

$$\prod_{A:\mathcal{U}} \prod_{w,x,y,z:A} \prod_{p:w=x} \prod_{q:x=y} \prod_{r:y=z} p \cdot (q \cdot r) = (p \cdot q) \cdot r, \tag{7.2}$$

and this is what will be done in the proof.

Proof. We use similar arguments as in the construction of $\cdot$: permuting some of the parameters w, x, y, z and p, q, r and then applying path induction. Indeed, path induction over p allows us to assume $x \equiv w$ and $p \equiv \mathsf{refl}_w$, in which case we have $q : w = y$ and $r : y = z$ and need to prove

$$\mathsf{refl}_w \cdot (q \cdot r) = (\mathsf{refl}_w \cdot q) \cdot r.$$

We show this by another path induction, this time on q, which gives $y \equiv w$ and $q \equiv \mathsf{refl}_w$, so that we have reduced the problem to finding an element of

$$\mathsf{refl}_w \cdot (\mathsf{refl}_w \cdot r) = (\mathsf{refl}_w \cdot \mathsf{refl}_w) \cdot r.$$

Using the same kind of path induction for r and applying the computation rule $\mathsf{refl}_w \cdot \mathsf{refl}_w \equiv \mathsf{refl}_w$ results in the assertion

$$\mathsf{refl}_w = \mathsf{refl}_w,$$

which has a trivial proof given by $\mathsf{refl}_{\mathsf{refl}_w}$. □

This result is our first *coherence* law. Generally speaking, a coherence law states that any two ways to construct a new object from given data are equal. In this case, it states that the concatenation of a sequence of paths does not depend on the order in which these paths are concatenated—at least up to propositional equality. The element of the above type (7.2) can be shown to satisfy its own coherence law!

There is also a sense in which the reflexivity elements are unit elements for the concatenation of paths, in the sense that concatenating with them gives the other path back—of course, again up to propositional equality:

7.7 Theorem. *For any $x, y : A$ and $p : x = y$,*

$$\mathsf{refl}_x \cdot p = p, \qquad p \cdot \mathsf{refl}_y = p.$$

Finally, in the same sense, concatenating a path with its inverse recovers the reflexivity elements:

7.8 Theorem. *For any $x, y : A$ and $p : x = y$,*

$$p \cdot p^{-1} = \mathsf{refl}_x, \qquad p^{-1} \cdot p = \mathsf{refl}_y.$$

Both theorems can easily be proven by path induction on p.

The concatenation of paths is an algebraic structure on all the identity types over any type A which turns A into a so-called *groupoid*. Can similar things also be done with *higher paths*, i.e. with elements of identity types between paths, identities between those, and so on? Before getting to this, we briefly study how the algebraic structure on paths interacts with the ap function from Lemma 7.3.

7.9 Theorem. *The function* ap *has the following properties:*

1. $\mathrm{ap}_f(p \cdot q) = \mathrm{ap}_f(p) \cdot \mathrm{ap}_f(q)$,
2. $\mathrm{ap}_f(p^{-1}) = \mathrm{ap}_f(p)^{-1}$,
3. *For* $f : A \to B$ *and* $g : B \to C$, *we have* $\mathrm{ap}_{g\circ f}(p) = \mathrm{ap}_g(\mathrm{ap}_f(p))$.
4. $\mathrm{ap}_{\mathrm{id}_A}(p) = p$.

Proof. All these properties follow from path induction on p (and q). □

Concerning the latter two items, it would be helpful to have the stronger statements that $\mathrm{ap}_{g\circ f} = \mathrm{ap}_g \circ \mathrm{ap}_f$, and likewise $\mathrm{ap}_{\mathrm{id}_A} = \mathrm{id}_{x=y}$. However, proving these statements would require to be able to conclude that two functions are equal as soon as they take on equal values on all points; this is something which we cannot prove yet!

A natural question now is, what happens if instead of $f : A \to B$ we consider a dependent function $f : \prod_{x:A} B(x)$? In this case, things are a bit more complicated: if we have $p : x =_A y$, then $f(x) : B(x)$ and $f(y) : B(y)$, so that it does not actually make sense to assert that $f(x) = f(y)$. So, before solving this puzzle, we need to understand dependent types a little better; we consider this below.

Types as higher groupoids

For paths $p, q : x = y$, we can also consider a path between these paths, $\alpha : p = q$. More rigorously, we should indicate the base type over which the identity type $p = q$ is formed, and then we have to write $\alpha : p =_{x=_A y} q$. We will also call such a path between paths a 2-*path*.

Suppose that we have, in addition to this data, also another path $r : y = z$. Geometrically, the situation looks like this:

$$\begin{tikzcd} x \arrow[r, bend left, "p"{name=P}] \arrow[r, bend right, "q"'{name=Q}] \arrow[Rightarrow, from=P, to=Q, "\alpha"] & y \arrow[r, "r"] & z \end{tikzcd}$$

Using path induction on r, we can concatenate this data to a 2-path of type $p \cdot r = q \cdot r$ which we denote by $\alpha \blacklozenge r$,

$$\begin{tikzcd} x \arrow[r, bend left, "p \cdot r"{name=P}] \arrow[r, bend right, "q \cdot r"'{name=Q}] \arrow[Rightarrow, from=P, to=Q, "\alpha \blacklozenge r"] & z \end{tikzcd}$$

More precisely, there is a function which concatenates any 2-path to any 1-path in the sense of having the type

$$\_ \blacklozenge \_ : \prod_{A:\mathcal{U}} \prod_{x,y,z:A} \prod_{p,q:x=y} \prod_{\alpha:p=q} \prod_{r:y=z} (p \cdot r) = (q \cdot r).$$

Again, we only consider α and r as the arguments of this function, since the others can be deduced. Similarly, if we are given another $w : A$ and $o : w = x$ as in

$$\begin{tikzcd} w \arrow[r, "o"] & x \arrow[r, bend left, "p"{name=P}] \arrow[r, bend right, "q"'{name=Q}] \arrow[Rightarrow, from=P, to=Q, "\alpha"] & y \end{tikzcd}$$

then we can likewise use path induction on o in order to define a new 2-path $o \blacklozenge \alpha : o \cdot p = o \cdot q$.

What operations can we construct which concatenate two 2-paths to a new 2-path? For one, if we have $r, p, q : x = y$ and $\alpha : p = q$ and $\beta : q = r$ like this,

$$\begin{tikzcd}
x \arrow[r, bend left=60, "p"{name=P}] \arrow[r, "q"{name=Q, description}] \arrow[r, bend right=60, "r"'{name=R}] & y
\arrow[Rightarrow, from=P, to=Q, "\alpha"] \arrow[Rightarrow, from=Q, to=R, "\beta"]
\end{tikzcd}$$

then we already know that we can form the concatenation $\alpha \cdot \beta : p = r$. Since this is only one kind of concatenation of 2-paths, it is usually called the *vertical concatenation*, due to how the α and β are arranged in the diagram above. On the other hand, if we have this situation,

$$\begin{tikzcd}
x \arrow[r, bend left=50, "p"{name=P}] \arrow[r, bend right=50, "q"'{name=Q}] & y \arrow[r, bend left=50, "r"{name=R}] \arrow[r, bend right=50, "s"'{name=S}] & z
\arrow[Rightarrow, from=P, to=Q, "\alpha"] \arrow[Rightarrow, from=R, to=S, "\beta"]
\end{tikzcd}$$

we can also construct a *horizontal concatenation* by defining

$$\alpha \star \beta :\equiv (\alpha \mathbin{\blacklozenge} r) \cdot (q \mathbin{\blacklozenge} \beta),$$

which is of type $(p \cdot r) = (q \cdot s)$. Alternatively, it would also have been possible to define this horizontal concatenation as $(p \mathbin{\blacklozenge} \beta) \cdot (\alpha \mathbin{\blacklozenge} s)$; the result would have been again equal, since one can find an element of the identity type

$$(\alpha \mathbin{\blacklozenge} r) \cdot (q \mathbin{\blacklozenge} \beta) = (p \mathbin{\blacklozenge} \beta) \cdot (\alpha \mathbin{\blacklozenge} s)$$

by using several consecutive path inductions.

7.10 Theorem. *(Exchange law) In a situation like this,*

$$\begin{tikzcd}
x \arrow[r, bend left=60, "o"{name=O}] \arrow[r, "p"{name=P, description}] \arrow[r, bend right=60, "q"'{name=Q}] & y \arrow[r, bend left=60, "r"{name=R}] \arrow[r, "s"{name=S, description}] \arrow[r, bend right=60, "t"'{name=T}] & z
\arrow[Rightarrow, from=O, to=P, "\alpha"] \arrow[Rightarrow, from=P, to=Q, "\beta"] \arrow[Rightarrow, from=R, to=S, "\gamma"] \arrow[Rightarrow, from=S, to=T, "\delta"]
\end{tikzcd}$$

we have

$$(\alpha \cdot \beta) \star (\gamma \cdot \delta) = (\alpha \star \gamma) \cdot (\beta \star \delta).$$

Proof. As usual, we apply path induction, this time on α, β, γ and δ, which reduces the problem to this case:

$$\begin{tikzcd}
x \arrow[r, bend left=60, "p"{name=P1}] \arrow[r, "p"{name=P2, description}] \arrow[r, bend right=60, "p"'{name=P3}] & y \arrow[r, bend left=60, "s"{name=S1}] \arrow[r, "s"{name=S2, description}] \arrow[r, bend right=60, "s"'{name=S3}] & z
\arrow[Rightarrow, from=P1, to=P2, "\mathsf{refl}_p"] \arrow[Rightarrow, from=P2, to=P3, "\mathsf{refl}_p"] \arrow[Rightarrow, from=S1, to=S2, "\mathsf{refl}_s"] \arrow[Rightarrow, from=S2, to=S3, "\mathsf{refl}_s"]
\end{tikzcd}$$

In this case, the desired equality is

$$(\mathsf{refl}_p \cdot \mathsf{refl}_p) \star (\mathsf{refl}_s \cdot \mathsf{refl}_s) = (\mathsf{refl}_p \star \mathsf{refl}_s) \cdot (\mathsf{refl}_p \star \mathsf{refl}_s).$$

Since we have $\mathsf{refl}_p \cdot \mathsf{refl}_p \equiv \mathsf{refl}_p$, and similar for refl_s and $(\mathsf{refl}_p \star \mathsf{refl}_s)$, this simplifies to

$$\mathsf{refl}_p \star \mathsf{refl}_s = \mathsf{refl}_p \star \mathsf{refl}_s,$$

which is true again by reflexivity. □

There are many more operations on higher-dimensional paths which can be constructed by path induction. We will stop our short discussion of this aspect by noting that all these higher operations can be regarded as one possible definition of an algebraic structure called an ∞-*groupoid*.

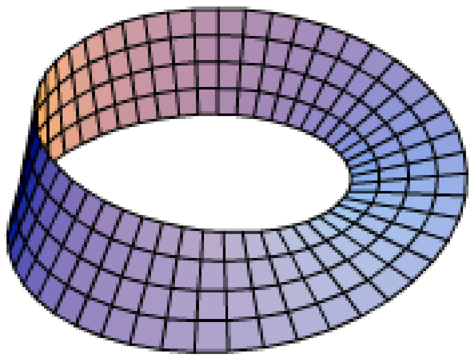

Figure 7.1: A Möbius strip

Dependent types as fibrations

We already learnt that we can think of a type A as a space with elements $x : A$ as "points" and equalities $p : x = y$ as "paths". We also just realized that a function $f : A \to B$ can be interpreted as a continuous map from the space A to the space B.

Now suppose that we apply this in the case $B \equiv \mathcal{U}$. So, the universe $\mathcal{U}$ is itself a gigantic space whose elements are themselves other spaces. (Such a "space of spaces" is often also called a *moduli space.*) So, a dependent type over A or type family over A is a function $D : A \to \mathcal{U}$; intuitively, this means that D assigns to every $x : A$ another space $D(x) : \mathcal{U}$ in a way which depends continuously on x. Conventionally, such a "dependent space" is known as a *fibration*; so, a type family is the HoTT analogue of the notion of fibration. As an example, if we start with the base space A being a circle, and attach to each $x : A$ a copy of the interval, such that as we go once around the circle, the interval comes back with a "twist", then we obtain the Möbius strip of Figure 7.1. What is depicted is not the function $D : A \to \mathcal{U}$, but rather the *total space* $\sum_{x:A} D(x)$, which is the union of all the individual $D(x)$. One can identify the circle S^1 with the central line that winds around the strip; for every x, the space $D(x)$ then consists of the interval perpendicular to that circle. Higher inductive types will let us permit the definition of a type modelling the circle and a type modelling the interval, and then the Möbius strip will be an actual type family that we can construct. Until then, it is nothing but an illustration.

So, the dependent sum $\sum_{x:A} D(x)$ corresponds to the total space of the fibration $D : A \to \mathcal{U}$; but what about the dependent product $\prod_{x:A} D(x)$? Intuitively, an element $s : \prod_{x:A} D(x)$ corresponds to an assignment $x \mapsto s(x)$ with $s(x) : D(x)$ such that the dependence on x is continuous. Under the correspondence to fibrations, such an s is conventionally known as a *section.* The Möbius strip is very special in that it permits a section which embeds the circle as the "base" into the total space $\sum_{x:A} D(x)$; in general, this is not the case.

With this intuition in mind, we can now move on to discussing the behavior of paths with respect to such fibrations.

7.11 Theorem. *Given a type family $D : A \to \mathcal{U}$ over A and a path $p : x =_A y$, there exists a function*

$$\mathrm{transport}^D(p) \,:\, D(x) \to D(y).$$

Proof. Path induction on p, using the definition $\mathrm{transport}^D(\mathrm{refl}_x) \equiv \mathrm{id}_{D(x)}$. □

Whenever the type family D is clear, then we also use the shorthand notation $p_* \,:\, D(x) \to D(y)$ in place of $\mathrm{transport}^D(p)$.

If $D : A \to \mathcal{U}$ is interpreted as a proposition depending on $x : A$, then we can say that $x : A$ "has property D" if $D(x)$ is inhabited. Therefore, the theorem tells us that if x is equal to y, and x has property D, then so does y. More precisely, any witness or proof of the statement "x has property D" can be transported into a witness or proof of the statement "y has property D".

In the topological theory of fibrations, the transport function can be interpreted as a connection, which is a fundamental structure in *differential* geometry. In this sense, a type family is more than just a fibration: it is a fibration equipped with a connection.

Important properties of the transport function are the following, which are all easily proven by the usual path induction:

7.12 Theorem. *If $q : y = z$ is another path, then*

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

for all $t : D(x)$.

So the transport map has the expected behaviour upon concatenation of paths; note the similarity to the previous theorem $\mathrm{ap}_f(q \cdot p) = \mathrm{ap}_f(q) \cdot \mathrm{ap}_f(p)$.

Using transport, we can not only transport elements around, we can also transport paths themselves,

$$\mathrm{ap}_{\mathrm{transport}^D} : \prod_{s,t:D(x)} (s = t) \to (p_*(s) = p_*(t)). \tag{7.3}$$

The behaviour of transport with respect to changing the "base" type A is like this:

7.13 Theorem. *If $f : B \to A$ is a function, then $D \circ f : B \to \mathcal{U}$ is again a type family, and*

$$\mathrm{transport}^{D\circ f}(p,t) = \mathrm{transport}^D(\mathrm{ap}_f(p), t).$$

for all $p : x =_B y$ with $x, y : B$, and $t : D(f(x))$.

Finally, we also note the behaviour of transport with respect to changing the type family, while keeping the base A fixed:

7.14 Theorem. *Given any two type families $D, E : A \to \mathcal{U}$ and a family of dependent functions $f : \prod_{x:A} D(x) \to E(x)$ and any $p : x =_A y$ and $t : D(x)$, we have*

$$\mathrm{transport}^E(p, f(x,t)) = f(y, \mathrm{transport}^D(p,t)).$$

We can express this as saying that transport "commutes" with the application of a dependent function. In terms of category theory, f is a transformation between functors, and the theorem shows that this transformation is *automatically* natural. This is similar to how Lemma 7.3 shows that every function is automatically continuous.

If we apply transport in the case of a non-dependent type, it acts trivially:

7.15 Theorem. *If $D \equiv \lambda x.B : A \to \mathcal{U}$ for some fixed $B : \mathcal{U}$, then for any $x, y : A$ and $p : x =_A y$ there is a family of paths*

$$\mathrm{transportconst}^B(p) : \prod_{t:B} \mathrm{transport}^{\lambda x.B}(p,t) = t$$

We are now in a situation to generalize the function $\mathrm{ap} : (x = y) \to (f(x) = f(y))$ to the case in which f is a dependent function, by simply transporting $f(x)$ from $D(x)$ to $D(y)$ before comparing it to $f(y)$:

7.16 Theorem. *Given any dependent function $f : \prod_{x:A} D(x)$, there exists a function*

$$\mathrm{apd}_f : \prod_{p:x=y} p_*(f(x)) =_{D(y)} f(y).$$

Again we consider the elements $x, y : A$ to be implicit arguments of apd_f for which we omit mention. The proof of the theorem is again straightforward by path induction on p. While the previous ap was a function $(x = y) \to (f(x) = f(y))$, the new apd_f for dependent f is itself a dependent function. Also apd_f has the expected behaviour with respect to concatenation of paths, and more generally there are more complicated analogues of Theorem 7.9, but we omit these.

When f is a non-dependent function, we would expect apd_f to reduce to ap_f, assuming that we identify $f : A \to B$ with the corresponding element of $\prod_{x:A} B$. This is indeed the case:

7.17 Theorem. *For all $f : A \to B$ and $p : x =_A y$, we have*

$$\mathrm{apd}_f(p) = \mathrm{transportconst}^B(p, f(x)) \cdot \mathrm{ap}_f(p).$$

The is well-typed since the path on the left goes from $\mathrm{transport}^{\lambda x.B}(p, f(x))$ to $f(y)$, while the one on the right is a concatenation of a path from $\mathrm{transport}^{\lambda x.B}(p, f(x))$ to $f(x)$ and a path from $f(x)$ to $f(y)$. As usual, the proof is by path induction on p, in which case both sides reduce to $\mathrm{refl}_{f(x)}$.

Path lifting

The previous considerations involving transport were about turning a path $p : x = y$ on the “base” A into a function $\mathrm{transport}^D(p) : D(x) \to D(y)$. But can we also lift a path in the base to a path in D? In order to make sense of this, we need to consider paths in the total space $\sum_{x:A} D(x)$, and then we have:

7.18 Theorem (Path lifting property)**.** *For any $x : A$ and $t : D(x)$, every path $p : x =_A y$ gives a path*

$$\mathrm{lift}(t,p) \;:\; (x,t) =_{\sum_{x:A} D(x)} (y, p_*(t)),$$

and this path lies over p in the sense that

$$\mathrm{ap}_{\mathrm{pr}_1}(\mathrm{lift}(t,p)) = p.$$

Proof. Using path induction on p reduces the proof of both statements to the case $y \equiv x$ and $p \equiv \mathrm{refl}_x$, for which they are trivial. □

Omnipotent path induction?

The previous results may suggest that path induction is a proof method with which one can prove almost anything, since it always reduces the proof to a trivial case. However, this is by no means the case: for example, the statement

$$\prod_{x:A} \prod_{p:x=x} p = \mathrm{refl}_x \tag{7.4}$$

is *not* derivable by path induction. In fact, path induction does not apply here since it would require at least one endpoint of p to be free—based path induction applies to statements of the form $\prod_{x:A} \prod_{p:x=y} C(x,y,p)$ only! We will see soon that together with the univalence axiom, one can show (7.4) to be false. Actually, (7.4) is closely related to Axiom K, which is a commonly imposed axiom resulting in so-called *extensional* type theory, in which two things can be equal in at most one way. Due to the incompatibility with the univalence axiom, and the realization that univalence is a more powerful principle resulting in a foundation of mathematics close to mathematical practice, we do not impose Axiom K.

Example: discriminating the elements of `Bool`

In order to illustrate the use of the results derived here, we show that the two canonical elements of `Bool` are not equal:

7.19 Theorem. $(0 = 1) \to \mathbf{0}$.

Proof. Define the type family $D : \mathtt{Bool} \to \mathcal{U}$ by induction over `Bool` using the values

$$D(0) :\equiv \mathbf{0}, \qquad D(1) :\equiv \mathbf{1}$$

on the base cases. We know that $D(1)$ is inhabited by $* : D(1)$. Hence for any putative equation $p : 0 = 1$, we have

$$(p^{-1})_*(*) \;:\; D(0),$$

which is an element of $\mathbf{0}$ since $D(0) \equiv \mathbf{0}$. □

Topic 8: Homotopy and equivalences

Many pairs of types, for example the product type $A \to B$ and the dependent pair type $\sum_{x:A} B$ in which B does not actually depend on x, satisfy the essentially same sets of inference rules. Correspondingly, we should expect these pairs of types themselves to be equal, i.e. there ought to be a canonical element of the identity type

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

And indeed, it is easy to construct a function $A \times B \to \sum_{x:A} B$ by induction over the product $A \times B$, since for $x : A$ and $y : B$ we have $(x, y) : \sum_{x:A} B$. Concretely, this function is

$$\mathrm{ind}_{A\times B}\left(\sum_{x:A} B,\, x.y.(x,y)\right) \,:\, A \times B \to \sum_{x:A} B. \tag{8.1}$$

Conversely, we can construct a function $\sum_{x:A} B \to (A \times B)$, likewise by a simple induction,

$$\mathrm{ind}_{\sum_{x:A} B}\left(A \times B,\, x.y.(x,y)\right) \,:\, \left(\sum_{x:A} B\right) \to A \times B. \tag{8.2}$$

However, there is no rule that we could use to turn into two such functions into an equality of types!

Moreover, one might naively expect these functions to be *isomorphisms*, i.e. that their composites $A \times B \to A \times B$ and $(\sum_{x:A} B) \to (\sum_{x:A} B)$ are judgementally equal to the identity functions $\lambda p.p$. However, this is not the case: the composite $A \times B \to A \times B$ is given by

$$\lambda p.\mathrm{ind}_{\sum_{x:A} B}\left(A \times B,\, x.y.(x,y),\, \mathrm{ind}_{A\times B}\left(\sum_{x:A} B,\, x.y.(x,y),\, p\right)\right). \tag{8.3}$$

In order to simplify this expression by judgemental equalities, we would have to apply the computation rule of $A \times B$. However, since p is not given as an explicit pair, this is not possible. The exact same problem arises for the other composition.

We will fix the second problem first by introducing *homotopy* for functions, which is a new notion that we introduce now. Next time, we will also fix the first problem by virtue of the *univalence axiom*.

Homotopies between functions

8.1 Definition. *Let $f, g : A \to B$ be functions. The type of all* homotopies *between f and g is*

$$(f \sim g) :\equiv \prod_{x:A} (f(x) = g(x)). \tag{8.4}$$

In words, one can express this as follows: two functions are homotopic if all their values are equal. In terms of the interpretations of types as spaces, this means that two continuous functions are homotopic as soon as there is a path connecting $f(x)$ and $g(x)$ for every x. This may seem a bit strange, since in the usual definition, there is an additional requirement: this path connecting $f(x)$ with $g(x)$ should also depend continuously on x. But this is automatically taken care of: we already noticed that every function is automatically continuous!

8.2 Lemma. *For any two functions $f, g : A \to B$,*

$$(f = g) \to (f \sim g).$$

Proof. Path induction on $p : f = g$. ☐

Homotopy has properties that are very similar to those of equality. In particular, homotopies can be composed and inverted, in the sense that there are functions

$$(f \sim g) \to (g \sim h) \to (f \sim h), \qquad (f \sim g) \to (g \sim f).$$

One obtains these by extending the corresponding operations for the identity type occurring in (8.4), for example the first one

$$\Big(\prod_{x:A}(f(x) = g(x))\Big) \to \Big(\prod_{x:A}(g(x) = h(x))\Big) \to \Big(\prod_{x:A}(f(x) = h(x))\Big)$$

can be obtained as

$$\lambda h.\lambda k.\lambda x.(h(x) \cdot k(x)).$$

Also, homotopies are preserved by functions: for $f, g : A \to B$ and $h : B \to C$ and any homotopy $H : f \sim g$, we have a homotopy

$$h \circ H : h \circ f \sim h \circ g.$$

Function extensionality

If two functions are equal, then they are homotopic:

8.3 Lemma. *For any two functions $f, g : A \to B$,*

$$(f = g) \to (f \sim g).$$

Proof. Path induction reduces to the case $g \equiv f$ and $\mathrm{refl}_f : f = f$, in which case we can take $\lambda x.\mathrm{refl}_{f(x)} : \prod_{x:A}(f(x) = f(x))$. ☐

The converse statement, i.e. the existence of a function $(f \sim g) \to (f = g)$, is known as *function extensionality.* Currently, we cannot prove function extensionality yet, but we will do so with the help of the univalence axiom.

Equivalences

8.4 Definition. *A function $f : A \to B$ is an* equivalence *if there exist functions $g, h : B \to A$ such that $g \circ f \sim \mathrm{id}_A$ and $f \circ h \sim \mathrm{id}_B$. In other words, the statement that f is an equivalence is represented by the type*

$$\mathrm{isequiv}(f) :\equiv \Big(\sum_{g:B\to A}(g \circ f \sim \mathrm{id}_A)\Big) \times \Big(\sum_{h:B\to A}(f \circ h \sim \mathrm{id}_B)\Big). \tag{8.5}$$

Finally, for any two types A and B, we would say that A and B are equivalent if there exists an equivalence $f : A \to B$. This is expressed by a type of equivalences,

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

So, in order to show that the types A and B are equivalent, one has to do several things:

1. Construct a function $f : A \to B$ which is a candidate for an equivalence,
2. Construct functions $g, h : B \to A$ which are candidates 'homotopy inverses' of f,
3. Construct homotopies $\prod_{x:A}(g(f(x)) = x)$ and $\prod_{y:B}(f(h(y)) = y)$.

However, there is no harm in taking g and h to be judgementally equal: for $f : A \to B$, let

$$\mathrm{qinv}(f) :\equiv \sum_{g:B\to A}((g \circ f \sim \mathrm{id}_A) \times (f \circ g \sim \mathrm{id}_B))$$

be the type of proofs that f has a *quasi-inverse*, i.e. an inverse g up to homotopy. Then:

8.5 Theorem. *There are functions*

$$\mathrm{qinv}(f) \to \mathrm{isequiv}(f), \qquad \mathrm{isequiv}(f) \to \mathrm{qinv}(f), \tag{8.6}$$

and hence having a quasi-inverse is logically equivalent to being an equivalence.

Proof. The first function is easy to define: if we know that f has a quasi-inverse, then by (8.6) we get $g : B \to A$ together with homotopies $H : g \circ f \sim \mathrm{id}_A$ and $K : f \circ g \sim \mathrm{id}_B$. This data can then be plugged into (8.5), with $h \equiv g$.

Conversely, suppose that we have $g, h : B \to A$ together with homotopies $H : g \circ f \sim \mathrm{id}_A$ and $K : f \circ h \sim \mathrm{id}_B$. We claim that we can use this given g as a quasi-inverse for f; to show this, it remains to construct a homotopy $f \circ g \sim \mathrm{id}_B$. But this follows from a homotopy between g and h,

$$g \sim g \circ f \circ h \sim h,$$

from which we conclude that $f \circ g \sim f \circ h \sim \mathrm{id}_B$. □

We can now get back to the above problem of the purported equality of $A \times B$ and $\sum_{x:A} B$, since we can now at least show that these types are equivalent: the above functions (8.2) and (8.1) are quasi-inverses of each other. To prove this, we need to show that the composite (8.3) is homotopic to $\mathrm{id}_{A\times B}$, and similarly for the other composite. For (8.3), we therefore need to show

$$\prod_{p:A\times B} \mathrm{ind}_{\sum_{x:A} B}\left(A \times B,\, x.y.(x,y),\, \mathrm{ind}_{A\times B}\left(\sum_{x:A} B,\, x.y.(x,y),\, p\right)\right) = p. \tag{8.7}$$

The main point now is that one can find an element of this type by induction on $p : A \times B$: the elimination rule tells us that in order to find an element of this type, it is sufficient to do so in the case that $p \equiv (a, b)$ is an explicitly given pair. But in this case, we can apply the computation rules of $A \times B$ and $\sum_{x:A} B$ to get

$$\begin{aligned}\mathrm{ind}_{\sum_{x:A} B}&\left(A \times B,\, x.y.(x,y), \mathrm{ind}_{A\times B}\left(\sum_{x:A} B,\, x.y.(x,y),\, (a,b)\right)\right)\\ &\equiv \mathrm{ind}_{\sum_{x:A} B}\left(A \times B,\, x.y.(x,y),\, (a,b)\right) \equiv (a,b),\end{aligned}$$

and hence in this case we can simply use $\mathrm{refl}_{(a,b)}$ as a witness for (8.7).

Here is another example of an equivalence:

8.6 Proposition. *Let $D : A \to \mathcal{U}$ be a dependent type. For any $x, y : A$ and $p : x = y$, we get an equivalence*

$$D(x) \simeq D(y)$$

with underlying function $\mathrm{transport}^D(p) : D(x) \to D(y)$.

Proof. □

¡++¿

For many type formers, like the product $\times$, it is possible to characterize propositional equality on the new type formed in terms of propositional equality on the original types. For example, for the product $\times$, this takes the obvious form:

8.7 Theorem. *For all $x, y : A \times B$,*

$$(x = y) \longrightarrow (\mathrm{pr}_1(x) = \mathrm{pr}_1(y)) \times (\mathrm{pr}_2(x) = \mathrm{pr}_2(y)),$$

and

$$(\mathrm{pr}_1(x) = \mathrm{pr}_1(y)) \times (\mathrm{pr}_2(x) = \mathrm{pr}_2(y)) \longrightarrow (x = y).$$

In words, any equality $x = y$ gives rise to equalities between the components $\mathrm{pr}_1(x) = \mathrm{pr}_1(y)$ and $\mathrm{pr}_2(x) = \mathrm{pr}_2(y)$, and vice versa.

Proof. We construct the first function. By path induction on $p : x = y$, it is sufficient to do so in the case that $y \equiv x$ and $p \equiv \mathrm{refl}_x$. But then, we can simply take the desired element to be given by the pair

$$(\mathrm{refl}_{\mathrm{pr}_1(x)}, \mathrm{refl}_{\mathrm{pr}_2(x)}) \,:\, (\mathrm{pr}_1(x) = \mathrm{pr}_1(x)) \times (\mathrm{pr}_2(x) = \mathrm{pr}_2(x)).$$

Constructing the second function works also by path induction on □

Optimally, we would like to be able to say more: we would expect that the two types that occur in Theorem 8.7, namely

$$x = y, \qquad (\mathrm{pr}_1(x) = \mathrm{pr}_1(y)) \times (\mathrm{pr}_2(x) = \mathrm{pr}_2(y)),$$

should themselves be equal, so that an equality between two pairs can be identified with a pair of equalities between their components. However, we have no way of deriving an equality between them — doing so would require