HoTT seminar: introductory slides

based on the HoTT book

May 23, 2017

What is going on here??

- As you remember, we planned that Jonathan Krebs will introduce us to homotopy

- As you remember, we planned that Jonathan Krebs will introduce us to homotopy
- On Monday evening, he wrote us: *I'm sorry to disrupt the seminar schedule already at the first talk, but I have come down with a bad cold and am unsure if I'll be able to do my presentation . . .*

- As you remember, we planned that Jonathan Krebs will introduce us to homotopy
- On Monday evening, he wrote us: *I'm sorry to disrupt the seminar schedule already at the first talk, but I have come down with a bad cold and am unsure if I'll be able to do my presentation . . .*
- Yesterday evening, he wrote us: *I'll stay near my bed and hot tea tomorrow, sorry.*

- As you remember, we planned that Jonathan Krebs will introduce us to homotopy
- On Monday evening, he wrote us: *I'm sorry to disrupt the seminar schedule already at the first talk, but I have come down with a bad cold and am unsure if I'll be able to do my presentation . . .*
- Yesterday evening, he wrote us: *I'll stay near my bed and hot tea tomorrow, sorry.*
- I had to improvise something quickly

- In the end, in fact, something good came out of it (I hope)

- In the end, in fact, something good came out of it (I hope)
- As you remember, our original plan was to stick as close as possible to the book

- In the end, in fact, something good came out of it (I hope)
- As you remember, our original plan was to stick as close as possible to the book
- I also signalled that the book has both informal and formal presentation (at the beginning, Chapter 1 parallels Appendix A)

- In the end, in fact, something good came out of it (I hope)
- As you remember, our original plan was to stick as close as possible to the book
- I also signalled that the book has both informal and formal presentation (at the beginning, Chapter 1 parallels Appendix A)
- Moreover, we told you you can (and should) use the book as the basis for your presentation

- In the end, in fact, something good came out of it (I hope)
- As you remember, our original plan was to stick as close as possible to the book
- I also signalled that the book has both informal and formal presentation (at the beginning, Chapter 1 parallels Appendix A)
- Moreover, we told you you can (and should) use the book as the basis for your presentation
- I have quickly created over 100 slides for today to illustrate these two points

- In the end, in fact, something good came out of it (I hope)
- As you remember, our original plan was to stick as close as possible to the book
- I also signalled that the book has both informal and formal presentation (at the beginning, Chapter 1 parallels Appendix A)
- Moreover, we told you you can (and should) use the book as the basis for your presentation
- I have quickly created over 100 slides for today to illustrate these two points
- Hopefully, this will convince you that it's a viable plan—and creates a pattern for future lectures

§1.1, “Type Theory vs. Set Theory”, p. 17

From *Preliminaries*

Homotopy type theory is (among other things) a foundational language for mathematics, i.e., an alternative to Zermelo–Fraenkel set theory. However, it behaves differently from set theory in several important ways, and that can take some getting used to. . . . A set-theoretic foundation has two "layers": the deductive system of first-order logic, and, formulated inside this system, the axioms of a particular theory, such as ZFC.

Recap of ZF(C)

(appendix to my recent paper *Infinite Populations, Choice and Determinacy*, see there for references I used)

Zermelo-Fraenkel set theory ZF is formulated in the first-order language $\mathcal{L}_{\mathrm{ZF}}$ with one binary primitive $\in$ and numerous standard abbreviations like $x = \{y, z\}, \emptyset, x \cap y = \emptyset, \exists x \in y.\phi(x), \exists! x \in y.\phi(x), x \subseteq y$, etc. Axioms:

Axiom of Extensionality $\forall xy.(\forall z.z \in x \leftrightarrow z \in y) \rightarrow x = y.$

Axiom of Regularity $\forall x.x \neq \emptyset \rightarrow \exists y \in x.x \cap y = \emptyset.$

Axiom of Union $\forall x \exists y \forall z.z \in y \leftrightarrow \exists v \in x.z \in v.$

Axiom of Powerset $\forall x \exists y \forall z.z \in y \leftrightarrow \forall v \in z.v \in x.$

Axiom of Infinity $\exists x.\emptyset \in x \wedge \forall y.y \in x \rightarrow y \cup \{y\} \in x.$

Axiom scheme of Restricted Comprehension (also known as Specification or Separation) For any $\mathcal{L}_{\mathrm{ZF}}$ formula $\phi(x, \overline{z})$ with no occurrences of y,

$$\forall x \exists y \forall z.z \in y \leftrightarrow (z \in x \wedge \phi(x, \overline{z})).$$

Axiom scheme of Replacement (also known as Collection) For any $\mathcal{L}_{\mathrm{ZF}}$ formula $\phi(x, y, \overline{z}, w)$ with no occurrences of v,

$$\forall w \overline{z}.(\forall x \in w \exists! y.\phi(x, y, \overline{z}, w)) \rightarrow \exists v \forall x \in w \exists y \in v.\phi(x, y, \overline{z}, w).$$

The Zermelo–Fraenkel set theory with Choice (ZFC) extends the above set of axioms with

Axiom of Choice (AC)

$$\forall w.(\forall x \in w.x \neq \emptyset \wedge \forall y \in w.x \neq y \rightarrow x \cap y = \emptyset) \rightarrow \exists c \forall x \in w.\exists! y \in x.y \in c).$$

Return to §1.1, “Type Theory vs. Set Theory”, p. 17

From *Preliminaries*

By contrast, type theory is its own deductive system: it need not be formulated inside any superstructure, such as first-order logic. Instead of the two basic notions of set theory, sets and propositions, type theory has one basic notion: types. Propositions are identified with particular types.

From *Preliminaries*

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

Introduction: Type theory, p.2

From *Introduction: Type theory*

Type theory was originally invented by Bertrand Russell , as a device for blocking the paradoxes in the logical foundations of mathematics that were under investigation at the time. It was developed further by many people over the next few decades, particularly Church who combined it with his λ-calculus. Although it is not generally regarded as the foundation for classical mathematics, set theory being more customary, type theory still has numerous applications, especially in computer science and the theory of programming languages .

From *Introduction: Type theory*

Per Martin-Löf , among others, developed a "predicative" modification of Church's type system, which is now usually called dependent, constructive, intuitionistic, or simply Martin-Löf type theory. This is the basis of the system that we consider here; it was originally intended as a rigorous framework for the formalization of constructive mathematics. In what follows, we will often use "type theory" to refer specifically to this system and similar ones, although type theory as a subject is much broader . . .

From *Introduction: Type theory*

In type theory, unlike set theory, objects are classified using a primitive notion of type, similar to the data-types used in programming languages. These elaborately structured types can be used to express detailed specifications of the objects classified, giving rise to principles of reasoning about these objects. To take a very simple example, the objects of a product type $A \times B$ *are known to be of the form* (a, b)*, and so one automatically knows how to construct them and how to decompose them. Similarly, an object of function type* $A \to B$ *can be acquired from an object of type B parametrized by objects of type A, and can be evaluated at an argument of type A.*

From *Introduction: Type theory*

This rigidly predictable behavior of all objects (as opposed to set theory's more liberal formation principles, allowing inhomogeneous sets) is one aspect of type theory that has led to its extensive use in verifying the correctness of computer programs. The clear reasoning principles associated with the construction of types also form the basis of modern computer proof assistants, which are used for formalizing mathematics and verifying the correctness of formalized proofs. . . .

From *Introduction: Type theory*

One problem in understanding type theory from a mathematical point of view, however, has always been that the basic concept of type is unlike that of set in ways that have been hard to make precise. We believe that the new idea of regarding types, not as strange sets (perhaps constructed without using classical logic), but as spaces, viewed from the perspective of homotopy theory, is a significant step forward. In particular, it solves the problem of understanding how the notion of equality of elements of a type differs from that of elements of a set.

From *Introduction: Type theory*

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

Introduction: “Comparing points of view on type-theoretic operations”, p.11

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

Table: Comparing points of view on type-theoretic operations

Returning to §1.1, “Type Theory vs. Set Theory”, p. 17

From *Preliminaries*

This leads us to another difference between type theory and set theory, but to explain it we must say a little about deductive systems in general. Informally, a deductive system is a collection of rules for deriving things called judgments. If we think of a deductive system as a formal game, then the judgments are the "positions" in the game which we reach by following the game rules. We can also think of a deductive system as a sort of algebraic theory, in which case the judgments are the elements (like the elements of a group) and the deductive rules are the operations (like the group multiplication). From a logical point of view, the judgments can be considered to be the "external" statements, living in the metatheory, as opposed to the "internal" statements of the theory itself.

From *Preliminaries*

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

From *Preliminaries*

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

From *Preliminaries*

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

From *Preliminaries*

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

From *Preliminaries*

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

From *Preliminaries*

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

From *Preliminaries*

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

From *Preliminaries*

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

From *Preliminaries*

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

From *Preliminaries*

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

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

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

From *Preliminaries*

Because judgments cannot be put together into more complicated statements, the symbols “:” and “$\equiv$” bind more loosely than anything else.Thus, for instance, “$p : x = y$” should be parsed as “$p : (x = y)$”, which makes sense since “$x = y$” is a type, and not as “$(p : x) = y$”, which is senseless since “$p : x$” is a judgment and cannot be equal to anything.

From *Preliminaries*

Similarly, "$A \equiv x = y$" can only be parsed as "$A \equiv (x = y)$", although in extreme cases such as this, one ought to add parentheses anyway to aid reading comprehension. Moreover, later on we will fall into the common notation of chaining together equalities — e.g. writing $a = b = c = d$ to mean "$a = b$ and $b = c$ and $c = d$, hence $a = d$" — and we will also include judgmental equalities in such chains. Context usually suffices to make the intent clear.

From *Preliminaries*

Judgments may depend on assumptions of the form $x : A$, where x is a variable and A is a type. For example, we may construct an object $m + n : \mathbb{N}$ under the assumptions that $m, n : \mathbb{N}$. Another example is that assuming A is a type, $x, y : A$, and $p : x =_A y$, we may construct an element $p^{-1} : y =_A x$. The collection of all such assumptions is called the context; from a topological point of view it may be thought of as a "parameter space". In fact, technically the context must be an ordered list of assumptions, since later assumptions may depend on previous ones: the assumption $x : A$ can only be made after the assumptions of any variables appearing in the type A.

From *Preliminaries*

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

From *Preliminaries*

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

From *Preliminaries*

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

From *Preliminaries*

Aside from some fairly obvious rules (such as the fact that judgmentally equal things can always be substituted for each other), the rules of type theory can be grouped into type formers. Each type former consists of a way to construct types (possibly making use of previously constructed types), together with rules for the construction and behavior of elements of that type. In most cases, these rules follow a fairly predictable pattern . . .

From *Preliminaries*

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

From *Preliminaries*

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

From *Preliminaries*

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

§1.2, “Function Types”, p. 21

- Of course, ordinary function types are, as we are going to discuss next week, degenerate examples of dependent function types (Π-types).
- Thus, this part of informal discussion does not correspond to anything in the formal presentation. It is just meant to explain the broader audience how functions work in a type-theoretical setting, as opposed to set-theoretical one.
- Hopefully, this is material all of you are are familiar with, thanks to ThProg, FPP or SemProg, for example. I just keep it here as a reminder, to see how things work in the terminology and notation of this book, and to recall α-, β- and η-conversion for the purpose of defining judgemental equality . . .

From *Preliminaries*

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

From *Preliminaries*

Given a function $f : A \to B$ and an element of the domain $a : A$, we can apply the function to obtain an element of the codomain B, denoted $f(a)$ and called the value of f at a. It is common in type theory to omit the parentheses and denote $f(a)$ simply by $f\ a$, and we will sometimes do this as well.

From *Preliminaries*

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

$$f(x) :\equiv \Phi \tag{0.1}$$

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

From *Preliminaries*

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

From *Preliminaries*

If we don't want to introduce a name for the function, we can use λ-abstraction. Given an expression Φ of type B which may use $x : A$, as above, we write $\lambda(x : A).\, \Phi$ to indicate the same function defined by (0.1)*. Thus, we have*

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

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

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

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

From *Preliminaries*

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

From *Preliminaries*

Another equivalent notation is

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

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

From *Preliminaries*

Now a λ-abstraction is a function, so we can apply it to an argument $a : A$. We then have the following computation rule, which is a definitional equality:

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

where Φ' is the expression Φ in which all occurrences of x have been replaced by a.

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

From *Preliminaries*

Continuing the above example, we have

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

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

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

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

Use of this equality is often referred to as η-conversion or η-expansion.

From *Preliminaries*

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

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

as

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

From *Preliminaries*

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

From *Preliminaries*

As an example, consider $f : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$ *defined as*

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

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

From *Preliminaries*

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

Use of this equality is often referred to as α-conversion.

From *Preliminaries*

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

From *Preliminaries*

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

From *Preliminaries*

The idea of currying is to represent a function of two inputs $a : A$ and $b : B$ as a function which takes one input $a : A$ and returns another function, which then takes a second input $b : B$ and returns the result. That is, we consider two-variable functions to belong to an iterated function type, $f : A \to (B \to C)$. We may also write this without the parentheses, as $f : A \to B \to C$, with associativity to the right as the default convention.

From *Preliminaries*

Then given $a : A$ and $b : B$, we can apply f to a and then apply the result to b, obtaining $f(a)(b) : C$. To avoid the proliferation of parentheses, we allow ourselves to write $f(a)(b)$ as $f(a, b)$ even though there are no products involved. When omitting parentheses around function arguments entirely, we write $f\,a\,b$ for $(f\,a)\,b$, with the default associativity now being to the left so that f is applied to its arguments in the correct order.

From *Preliminaries*

Our notation for definitions with explicit parameters extends to this situation: we can define a named function $f : A \to B \to C$ *by giving an equation*

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

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

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

which may also be written as

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

From *Preliminaries*

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

From *Preliminaries*

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

From *Preliminaries*

For instance, using a representation of sets as trees, we can directly encode Russell's paradox .
To avoid the paradox we introduce a hierarchy of universes

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

where every universe $\mathcal{U}_i$ is an element of the next universe $\mathcal{U}_{i+1}$. Moreover, we assume that our universes are cumulative, that is that all the elements of the i^{th} universe are also elements of the $(i+1)^{\text{st}}$ universe, i.e. if $A : \mathcal{U}_i$ then also $A : \mathcal{U}_{i+1}$. This is convenient, but has the slightly unpleasant consequence that elements no longer have unique types, and is a bit tricky in other ways that need not concern us here . . .

From *Preliminaries*

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

From *Preliminaries*

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

From *Preliminaries*

To model a collection of types varying over a given type A, we use functions $B : A \to \mathcal{U}$ whose codomain is a universe. These functions are called families of types (or sometimes dependent types); they correspond to families of sets as used in set theory.

From *Preliminaries*

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

From *Preliminaries*

A more trivial (but very important) example of a type family is the constant type family at a type $B : \mathcal{U}$, which is of course the constant function $(\lambda(x : A).\, B) : A \to \mathcal{U}$. As a non-example, in our version of type theory there is no type family "$\lambda(i : \mathbb{N}).\, \mathcal{U}_i$". Indeed, there is no universe large enough to be its codomain. Moreover, we do not even identify the indices i of the universes $\mathcal{U}_i$ with the natural numbers $\mathbb{N}$ of type theory (the latter to be introduced . . .).

Appendix “Formal Type Theory”, beginning p. 425

From *Appendix A*

One can develop mathematics in set theory without explicitly using the axioms of Zermelo–Fraenkel set theory . . . [the book develops] mathematics in univalent foundations without explicitly referring to a formal system of homotopy type theory.

From *Appendix A*

One can develop mathematics in set theory without explicitly using the axioms of Zermelo–Fraenkel set theory . . . [the book develops] mathematics in univalent foundations without explicitly referring to a formal system of homotopy type theory. Nevertheless, it is important to have a precise description of homotopy type theory as a formal system in order to, for example,

- *state and prove its metatheoretic properties, including logical consistency,*
- *construct models, e.g. in simplicial sets, model categories, higher toposes, etc., and*
- *implement it in proof assistants like* Coq *or* Agda*.*

From *Appendix A*

Even the logical consistency of homotopy type theory, namely that in the empty context there is no term $a : \mathbf{0}$*, is not obvious: if we had erroneously chosen a definition of equivalence for which* $\mathbf{0} \simeq \mathbf{1}$*, then univalence would imply that* $\mathbf{0}$ *has an element, since* $\mathbf{1}$ *does. Nor is it obvious that, for example, our definition of* $\mathbb{S}^1$ *as a higher inductive type yields a type which behaves like the ordinary circle.*

From *Appendix A*

There are two aspects of type theory which we must pin down before addressing such questions. Recall from the Introduction that type theory comprises a set of rules specifying when the judgments $a : A$ *and* $a \equiv a' : A$ *hold—for example, products are characterized by the rule that whenever* $a : A$ *and* $b : B$, $(a, b) : A \times B$. *To make this precise, we must first define precisely the syntax of terms—the objects* $a, a', A, \ldots$ *which these judgments relate; then, we must define precisely the judgments and their rules of inference—the manner in which judgments can be derived from other judgments.*

From *Appendix A*

[W]e present two formulations of Martin-Löf type theory, and of the extensions that constitute homotopy type theory.

- *The first presentation (Appendix A.1) describes the syntax of terms and the forms of judgments as an extension of the untyped λ-calculus, while leaving the rules of inference informal.*
- *The second (Appendix A.2) defines the terms, judgments, and rules of inference inductively in the style of natural deduction, as is customary in much type-theoretic literature.*

Preliminaries of Appendix A, p. 426–427

From *Appendix A*

[W]e presented the two basic judgments of type theory. The first, $a : A$, asserts that a term a has type A. The second, $a \equiv b : A$, states that the two terms a and b are judgmentally equal at type A. These judgments are inductively defined by a set of inference rules described in Appendix A.2.

From *Appendix A*

To construct an element a of a type A is to derive a : *A; in the book, we give informal arguments which describe the construction of a, but formally, one must specify a precise term a and a full derivation that a* : *A.*

From *Appendix A*

However, the main difference between the presentation of type theory in the book and in this appendix is that here judgments are explicitly formulated in an ambient context, or list of assumptions, of the form

$$x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n.$$

An element $x_i : A_i$ of the context expresses the assumption that the variable x_i has type A_i. The variables $x_1, \ldots, x_n$ appearing in the context must be distinct. We abbreviate contexts with the letters Γ and Δ.

From *Appendix A*

The judgment $a : A$ *in context* Γ *is written*

$$\Gamma \vdash a : A$$

and means that $a : A$ *under the assumptions listed in* Γ*. When the list of assumptions is empty, we write simply*

$$\vdash a : A$$

or

$$\cdot \vdash a : A$$

where $\cdot$ *denotes the empty context. The same applies to the equality judgment*

$$\Gamma \vdash a \equiv b : A$$

From *Appendix A*

However, such judgments are sensible only for well-formed contexts, a notion captured by our third and final judgment

$$(x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n)\ \text{ctx}$$

expressing that each A_i is a type in the context $x_1 : A_1, x_2 : A_2, \ldots, x_{i-1} : A_{i-1}$. In particular, therefore, if $\Gamma \vdash a : A$ and Γ ctx, then we know that each A_i contains only the variables $x_1, \ldots, x_{i-1}$, and that a and A contain only the variables $x_1, \ldots, x_n$.

From *Appendix A*

In informal mathematical presentations, the context is implicit. At each point in a proof, the mathematician knows which variables are available and what types they have, either by historical convention (n is usually a number, f is a function, etc.) or because variables are explicitly introduced with sentences such as "let x be a real number". We discuss some benefits of using explicit contexts in Appendices A.2.4 and A.2.5.

From *Appendix A*

We write $B[a/x]$ *for the substitution of a term a for free occurrences of the variable* x *in the term B, with possible capture-avoiding renaming of bound variables... The general form of substitution*

$$B[a_1, \ldots, a_n / x_1, \ldots, x_n]$$

substitutes expressions $a_1, \ldots, a_n$ *for the variables* $x_1, \ldots, x_n$ *simultaneously.*

From *Appendix A*

To bind a variable x in an expression B means to incorporate both of them into a larger expression, called an abstraction, whose purpose is to express the fact that x is "local" to B, i.e., it is not to be confused with other occurrences of x appearing elsewhere. . . . Various notations are used for binding, such as $x \mapsto B$, $\lambda x.\, B$, and $x\,.\, B$, depending on the situation. We may write $C[a]$ for the substitution of a term a for the variable in the abstracted expression, i.e., we may define $(x.B)[a]$ to be $B[a/x]$.

From *Appendix A*

As discussed . . . changing the name of a bound variable everywhere within an expression ("α-conversion") does not change the expression. Thus, to be very precise, an expression is an equivalence class of syntactic forms which differ in names of bound variables.

From *Appendix A*

One may also regard each variable x_i of a judgment

$$x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \vdash a : A$$

to be bound in its scope, consisting of the expressions $A_{i+1}, \ldots, A_n$, a, and A.

A.1 and A1.1, “The first presentation”, p. 427–428

From *Appendix A.1*

The objects and types of our type theory may be written as terms using the following syntax, which is an extension of λ-calculus with variables $x, x', \ldots,$ primitive constants $c, c', \ldots,$ defined constants $f, f', \ldots,$ and term forming operations

$$t ::= x \mid \lambda x.\, t \mid t(t') \mid c \mid f$$

The notation used here means that a term t is either a variable x, or it has the form $\lambda x.\, t$ where x is a variable and t is a term, or it has the form $t(t')$ where t and t' are terms, or it is a primitive constant c, or it is a defined constant f. The syntactic markers 'λ', '(', ')', and '.' are punctuation for guiding the human eye.

From *Appendix A.1*

We use $t(t_1, \ldots, t_n)$ as an abbreviation for the repeated application $t(t_1)(t_2)\ldots(t_n)$. We may also use infix notation, writing $t_1 \star t_2$ for $\star(t_1, t_2)$ when $\star$ is a primitive or defined constant.
Each defined constant has zero, one or more defining equations. There are two kinds of defined constant. An explicit defined constant f has a single defining equation

$$f(x_1, \ldots, x_n) :\equiv t,$$

where t does not involve f. For example, we might introduce the explicit defined constant $\circ$ with defining equation

$$\circ(x, y)(z) :\equiv x(y(z)),$$

and use infix notation $x \circ y$ for $\circ(x, y)$. This of course is just composition of functions.

From *Appendix A.1*

The second kind of defined constant is used to specify a (parameterized) mapping $f(x_1, \ldots, x_n, x)$*, where* x *ranges over a type whose elements are generated by zero or more primitive constants. For each such primitive constant* c *there is a defining equation of the form*

$$f(x_1, \ldots, x_n, c(y_1, \ldots, y_m)) :\equiv t,$$

where f *may occur in* t*, but only in such a way that it is clear that the equations determine a totally defined function. The paradigm examples of such defined functions are the functions defined by primitive recursion on the natural numbers. We may call this kind of definition of a function a total recursive definition. In computer science and logic this kind of definition of a function on a recursive data type has been called a definition by structural recursion.*

From *Appendix A.1*

Convertibility $t \downarrow t'$ *between terms* t *and* t' *is the equivalence relation generated by the defining equations for constants, the computation rule*

$$(\lambda x.\, t)(u) :\equiv t[u/x],$$

and the rules which make it a congruence with respect to application and λ*-abstraction:*

- *if* $t \downarrow t'$ *and* $s \downarrow s'$ *then* $t(s) \downarrow t'(s')$*, and*
- *if* $t \downarrow t'$ *then* $(\lambda x.\, t) \downarrow (\lambda x.\, t')$*.*

The equality judgment $t \equiv u : A$ *is then derived by the following single rule:*

- *if* $t : A$*,* $u : A$*, and* $t \downarrow u$*, then* $t \equiv u : A$*.*

Judgmental equality is an equivalence relation.

From *Appendix A.1.1*

We postulate a hierarchy of universes denoted by primitive constants

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

The first two rules for universes say that they form a cumulative hierarchy of types:

- $\mathcal{U}_m : \mathcal{U}_n$ *for* $m < n$,
- *if* $A : \mathcal{U}_m$ *and* $m \leq n$, *then* $A : \mathcal{U}_n$,

and the third expresses the idea that an object of a universe can serve as a type and stand to the right of a colon in judgments:

- *if* $\Gamma \vdash A : \mathcal{U}_n$, *and* x *is a new variable,*[1] *then* $\vdash (\Gamma, x : A)$ ctx.

[1] By "new" we mean that it does not appear in Γ or A.

From *Appendix A.1.1*

In the body of the book, an equality judgment $A \equiv B : \mathcal{U}_n$ between types A and B is usually abbreviated to $A \equiv B$. This is an instance of typical ambiguity, as we can always switch to a larger universe, which however does not affect the validity of the judgment.
The following conversion rule allows us to replace a type by one equal to it in a typing judgment:

- *if $a : A$ and $A \equiv B$ then $a : B$.*

A.2, “The second presentation”, p. 431–433

From *Appendix A.2*

In this section, there are three kinds of judgments

$$\Gamma \text{ ctx} \qquad \Gamma \vdash a : A \qquad \Gamma \vdash a \equiv a' : A$$

which we specify by providing inference rules for deriving them.

From *Appendix A.2*

A typical inference rule has the form

$$\frac{\mathcal{J}_1 \qquad \cdots \qquad \mathcal{J}_k}{\mathcal{J}}\ \textsc{Name}$$

It says that we may derive the conclusion $\mathcal{J}$, provided that we have already derived the hypotheses $\mathcal{J}_1, \ldots, \mathcal{J}_k$. (Note that, being judgments rather than types, these are not hypotheses internal to the type theory in the sense of §1.1; they are instead hypotheses in the deductive system, i.e. the metatheory.) On the right we write the NAME *of the rule, and there may be extra side conditions that need to be checked before the rule is applicable.*

From *Appendix A.2*

A derivation of a judgment is a tree constructed from such inference rules, with the judgment at the root of the tree. For example, with the rules given below, the following is a derivation of $\cdot \vdash \lambda x.\, x : \mathbf{1} \to \mathbf{1}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot \text{ ctx}}\,\text{ctx-EMP}}{\vdash \mathbf{1} : \mathcal{U}_0}\,\mathbf{1}\text{-FORM}}{x{:}\mathbf{1} \text{ ctx}}\,\text{ctx-EXT}}{x{:}\mathbf{1} \vdash x : \mathbf{1}}\,\text{Vble}}{\cdot \vdash \lambda x.\, x : \mathbf{1} \to \mathbf{1}}\,\Pi\text{-INTRO}$$

From *Appendix A.2*

A context is a list

$$x_1{:}A_1\,,\, x_2{:}A_2\,,\, \ldots\,,\, x_n{:}A_n$$

which indicates that the distinct variables $x_1, \ldots, x_n$ are assumed to have types $A_1, \ldots, A_n$, respectively. The list may be empty. We abbreviate contexts with the letters Γ and Δ, and we may juxtapose them to form larger contexts.

From *Appendix A.2*

The judgment Γ ctx *formally expresses the fact that* Γ *is a well-formed context, and is governed by the rules of inference*

$$\frac{}{\cdot \text{ ctx}}\ \text{ctx-EMP} \qquad \frac{x_1{:}A_1, \ldots, x_{n-1}{:}A_{n-1} \vdash A_n : \mathcal{U}_i}{(x_1{:}A_1, \ldots, x_n{:}A_n) \text{ ctx}}\ \text{ctx-EXT}$$

with a side condition for the second rule: the variable x_n *must be distinct from the variables* $x_1, \ldots, x_{n-1}$.

From *Appendix A.2*

Note that the hypothesis and conclusion of ctx-EXT *are judgments of different forms:*

- *the hypothesis says that in the context of variables* $x_1, \ldots, x_{n-1}$*, the expression* A_n *has type* $\mathcal{U}_i$*;*
- *while the conclusion says that the extended context* $(x_1{:}A_1, \ldots, x_n{:}A_n)$ *is well-formed.*

(It is a meta-theoretic property of the system that if $x_1{:}A_1, \ldots, x_n{:}A_n \vdash b : B$ *is derivable, then the context* $(x_1{:}A_1, \ldots, x_n{:}A_n)$ *must be well-formed; thus* ctx-EXT *does not need to hypothesize well-formedness of the context to the left of* x_n*.)*

From *Appendix A.2*

The fact that the context holds assumptions is expressed by the rule which says that we may derive those typing judgments which are listed in the context:

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

As with ctx-EXT*, the hypothesis and conclusion of the rule* Vble *are judgments of different forms, only now they are reversed: we start with a well-formed context and derive a typing judgment.*

From *Appendix A.2*

The following important principles, called substitution and weakening, need not be explicitly assumed. Rather, it is possible to show, by induction on the structure of all possible derivations, that whenever the hypotheses of these rules are derivable, their conclusion is also derivable.

Such rules are called admissible.

From *Appendix A.2*

For the typing judgments these principles are manifested as

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x{:}A, \Delta \vdash b : B}{\Gamma, \Delta[a/x] \vdash b[a/x] : B[a/x]}\ \mathrm{Subst}_1$$

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, \Delta \vdash b : B}{\Gamma, x{:}A, \Delta \vdash b : B}\ \mathrm{Wkg}_1$$

and for judgmental equalities they become

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x{:}A, \Delta \vdash b \equiv c : B}{\Gamma, \Delta[a/x] \vdash b[a/x] \equiv c[a/x] : B[a/x]}\ \mathrm{Subst}_2$$

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, \Delta \vdash b \equiv c : B}{\Gamma, x{:}A, \Delta \vdash b \equiv c : B}\ \mathrm{Wkg}_2$$

From *Appendix A.2*

In addition to the judgmental equality rules given for each type former, we also assume that judgmental equality is an equivalence relation respected by typing.

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

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

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash A \equiv B : \mathcal{U}_i}{\Gamma \vdash a : B}$$

$$\frac{\Gamma \vdash a \equiv b : A \qquad \Gamma \vdash A \equiv B : \mathcal{U}_i}{\Gamma \vdash a \equiv b : B}$$

From *Appendix A.2*

Additionally, for all the type formers below, we assume rules stating that each constructor preserves definitional equality in each of its arguments; for instance, along with the Π-INTRO *rule, we assume the rule*

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, x{:}A \vdash B : \mathcal{U}_i \qquad \Gamma, x{:}A \vdash b \equiv b' : B}{\Gamma \vdash \lambda x.\, b \equiv \lambda x.\, b' : \prod_{(x:A)} B}\ \Pi\text{-INTRO-EQ}$$

However, we omit these rules for brevity.

From *Appendix A.2*

We postulate an infinite hierarchy of type universes

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

Each universe is contained in the next, and any type in $\mathcal{U}_i$ *is also in* $\mathcal{U}_{i+1}$*:*

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

From *Appendix A.2*

We shall set up the rules of type theory in such a way that $\Gamma \vdash a : A$ *implies* $\Gamma \vdash A : \mathcal{U}_i$ *for some* i*. In other words, if A plays the role of a type then it is in some universe. Another property of our type system is that* $\Gamma \vdash a \equiv b : A$ *implies* $\Gamma \vdash a : A$ *and* $\Gamma \vdash b : A$*.*

Suggestion

- Next time: products (dependent and finite) and coproducts

Suggestion

- Next time: products (dependent and finite) and coproducts
- Later: booleans, natural numbers, W-types and identity types