Topic 1: What is HoTT and why?

May 5, 2014

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.
$$(x, y) := \{\{x\}, \{x, y\}\},$$

2.
$$(x, y) := \{x, \{x, y\}\},$$

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

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
$$\mathrm{ind}_{\mathbb{N}}(A, x_0, f)(0) := x_0, \qquad \mathrm{ind}_{\mathbb{N}}(A, x_0, f)(s(n)) := f(n, \mathrm{ind}_{\mathbb{N}}(A, x_0, f)(n)).$$
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
$$0! := 1, \qquad (n+1)! := (n+1) \cdot n!,$$
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 $f(n, m) := (n+1) \cdot m$.

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

$$\prod_{x,y:A} (x = y \to y = x) \tag{1}$$

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:

Axiom 1 (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).