type theory

Contents

Context

Type theory

Deduction and Induction

Constructivism, Realizability, Computability

Foundations

1. Idea

Type theory is a branch of mathematical symbolic logic, which derives its name from the fact that it formalizes not only mathematical terms – such as a variable x, or a function f – and operations on them, but also formalizes the idea that each such term is of some definite *type*, for instance that the type $\mathbb{N}$ of a natural number $x:\mathbb{N}$ is different from the type $\mathbb{N} \to \mathbb{N}$ of a function $f:\mathbb{N} \to \mathbb{N}$ between natural numbers.

Explicitly, type theory is a formal language, essentially a set of rules for rewriting certain strings of symbols, that describes the introduction of types and their terms, and computations with these, in a sensible way.

What may seem like a triviality on first sight turns out to have deep implications:

1. ***foundations of mathematics***. On the one hand, logic itself is subsumed in the plain idea of operations on terms of types, by

observing that any type X may be thought of as the type of terms satisfying some proposition. Under this *propositions are types*-paradigm a proof of the proposition is nothing but a term of the corresponding type. This identification leads to a very fruitful identification of operations on types with logical operations.

2. ***programming language***. Since such a proof is constructive, the term witnessing it being a concrete implementation, and since type theory strictly works by rewriting rules, one may identify the construction of a term in type theory as a *program* whose output is a certain type. Under this proofs as programs-paradigm, type theory is a mathematical formalization of a *programming language*. (For instance, *Coq* and *Agda* are concrete machine implementations of such a language. They are used both in computer science, where the typing provides certified programming, and may one day be usable in industry.)

3. ***calculus for category theory***. On the other hand, if one regards, as is natural, any term $t:X$ to exist in a context Γ of other terms $x:\Gamma$, then t is naturally identified with a "map" $t:\Gamma \to X$, hence: with a *morphism*. Viewed this way the types and terms of type theory are identified, respectively, with the objects and morphisms of category theory. From this perspective, type theory provides a formal language for speaking about categories. Indeed, from this perspective type theory is formalization along the lines of the *Elementary Theory of the Category of Sets*.

These aspects naturally harmonize, involving, reflecting on, and subsuming internal logic of categories and categorical logic/categorical semantics in categories.

Moreover, when following the idea of type theory through seriously, it turns out to go well beyond these topics even: If all logical statements are to be witnessed by terms of the type that corresponds to the given proposition, then this should notably be true for what is maybe the most basic logical notion, that of

equality. Therefore it makes sense to demand for any two terms $x, y : X$ of a type X the existence of an *identity type* $\mathrm{Id}_X(x, y)$ which represents the proposition that x is equal to y, hence such that a term $p : \mathrm{Id}_X(x, y)$ is a proof of this fact. But this idea necessarily iterates, with the equality of two such proofs in turn being witnessed by a term of a second order identity type, and so on. Reflecting on this shows that the type-theoretic notion of equality resulting this way is not the traditional one, but is the notion of *homotopy equivalence* or *equivalence in an (∞,1)-category*. Type theory with such identity types properly implemented is thus called *homotopy type theory*. It is a calculus now for (∞,1)-category theory. See there for more details on this.

Notice that this is obtained not by adding something to the basic idea of type theory, but by removing something, namely the ad-hoc assumption of definite equality.

Similarly, while plain vanilla type theory formalizes intuitionistic logic/intuitionistic mathematics, it is possible to add by hand, if necessary for some reason, axioms such as the law of excluded middle to make type theory also describe classical logic. But by nature it is more general.

2. As a formal language for category theory

One way to look at type theory, from the point of view of a category theorist, is as a *syntax for describing the construction? of objects and morphisms in a category*. (An introduction and historical background is for instance in Taylor section 2.)

This interpretation can be called *categorical semantics*. More precisely, categorical semantics refers to an *adjunction* or *equivalence of categories* between type theories and categories

(category of contexts ⊣ internal language)

$$:(\mathrm{Con} \dashv \mathrm{Lan}):\mathrm{TypeTheories} \underset{\mathrm{Con}}{\overset{\mathrm{Lan}}{\leftarrow\!\!\rightarrow}} \mathrm{Categories}.$$

This is discussed in detail at *relation between type theory and category theory*.

There are many different versions of this adjunction, depending on what kind of type theory we consider (e.g. dependent, with identity types, etc.) and what structure we impose on the categories in question. In each case

- the left adjoint Con assigns to a type theory T the *category of contexts* of T (a review is for instance in Taylor section 2.8), which has structure (such as limits, colimits, etc.) derived from the operations in the type theory,
- the right adjoint Lan assigns to a category C a canonically defined type theory, called its *internal language* (a review is for instance in Taylor 99 section 7.6).

A *model* of a theory T in a category C is equivalently a functor $\mathrm{Con}(T) \to C$ or a morphism of type theories $T \to \mathrm{Lan}(C)$. This means that every T has a tautological model in $\mathrm{Con}(T)$, and for every category C there is a tautological model of $\mathrm{Lan}(C)$ in C. For the category theorist who is most accustomed to think about categories, it is natural to approach type theory by studying the structure of $\mathrm{Lan}(C)$ and how it is interpreted in C via this tautological model. We will do this in this section somewhat informally; in the next section we give a more formal definition of type theories.

A couple of side notes for experts:

- 1-categorical semantics is only relevant to *extensional* type theory; see the section on intensional type theory vs. extensional type theory below.
- The description given below is a somewhat simplified one, in that we identify objects of the category with single types,

rather than contexts. This is strictly speaking only okay in type theories with a dependent sum operation, which allows us to reinterpret every context as a single type. An alternative approach which avoids this problem is to use cartesian multicategories.

The internal language of a category

Given a category $\mathcal{C}$, we may speak about its ***internal language*** as a type theory (see e.g. in Taylor, section 2.8).

There is a whole page on internal logic, but here our goal is to exhibit it as a particular type theory, to help explain the meaning of type-theoretic notions. The syntactic constructs corresponding to objects and morphisms are called *types* and *terms*, respectively. The types correspond to objects (with various subtleties), while the terms denote morphisms by using *variables* to indicate domains.

Types, terms, and variables

- In category theory a morphism f in $\mathcal{C}$ with domain B and codomain A is in symbols
$$B \xrightarrow{f} A \ .$$
- In the internal language of the category the same is a *term* $f(x)$ of *type* A where x is a *free variable* of *type* B, which in symbols is given by
$$x : B \vdash f(x) : A \ .$$

We may think of the *free variables* here as being placeholders for all the generalized elements $U \xrightarrow{x} B$ of B. Then the assertion $x : B \vdash f(x) : A$ indicates that with $B \xrightarrow{f} A$ given we may send $U \xrightarrow{x} B$ to the composition $(U \xrightarrow{x} B \xrightarrow{f} A) = (U \xrightarrow{f(x)} A)$.

So the notation $x : B \vdash f(x) : A$ is a direct reflection of the description

of the morphism f under the Yoneda embedding $C \hookrightarrow \mathrm{Func}(C^{\mathrm{op}}, \mathrm{Set})$. Since the Yoneda embedding is a full and faithful functor, this is indeed an entirely equivalent characterization of the morphism f.

Evaluation

Generally, composition of morphisms in the category

$$C \xrightarrow{f} B \xrightarrow{g} A \quad = \quad C \xrightarrow{g \circ f} A$$

corresponds to *substitution* in type theory of a term for a free variable: the morphisms f and g are interpreted as terms $f(x)$ and $g(y)$ of type B and A respectively, where x and y are variables of type C and B respectively. The composite morphism $g \circ f$ is the term $g(f(x))$ of type A where x is again a variable of type C.

In symbols this is written as:

$$\frac{x:C \vdash f(x):B \qquad y:B \vdash g(y):A}{x:C \vdash g(f(x)):A}$$

Here the horizontal bar indicates that we have written down a *rule*, the rule that the judgement on the bottom is valid whenever the judgements on the top are valid.

What is an identity morphism $A \xrightarrow{f \;=\; \mathrm{Id}_A} A$ in category theory is a term representing the function $f(x) = x$ in type theory, namely the variable x itself regarded as a term: x is a term of type A whenever x is a variable of type A.

In symbols:

$$\frac{}{x:A \vdash x:A}$$

Type constructors

What sorts of additional syntactical constructions you allow on types and terms corresponds to the structure of the category $\mathcal{C}$ in which the semantics is intended to occur.

For example, if our semantic categories have binary products, then the syntax of the type theory includes a *type constructor* $\times$ allowing us to build a new product type $A \times B$ from two given types A and B.

It will also have *term constructors* allowing us to build, for example, a term $\langle a, b \rangle$ of type $A \times B$ from any given terms a of type A and b of type B, and to build terms $\pi_1(z)$ and $\pi_2(z)$ from any term z of type $A \times B$, with rules that say that $\pi_1\langle a, b\rangle = a$, $\pi_2\langle a, b\rangle = b$, and $\langle \pi_1(z), \pi_2(z)\rangle = z$.

Note the great advantage of the type-theoretic formalism: the notation (and thought process) can be very set-theoretic, but because the terms a and b can denote morphisms with arbitrary domain (i.e. generalized elements), this really describes the full universal property of a categorical cartesian product.

Dependent types

An important extension of type theory involves *dependent types* : types which are a "function" of the *elements* of some other type. For instance, the type $D(m)$ of "days of the month" is a function of the element m of the type M of months, since different months have different allowable collections of days.

In category theory language a *type* $C(x)$ *dependent on* an element x of type A is (again) a morphism

$$\begin{array}{c} C \\ \downarrow p \\ A \end{array}$$

thought of as an A-indexed family of objects/types – a bundle of

objects. In type theory language this is written

$$x:A \vdash C(x):\mathrm{Type}$$

and read "*for each x of type A there is a type $C(x)$*" or "*for each x of type A there is a $C(x)$ of type* Type."

Here the variable x is again a placeholder for generalized elements, but now the $\vdash$ denotes not postcomposition with a morphism, but pullback along a morphism: for every generalized element $U \xrightarrow{x} A$, we have the pullback $x^*C := C(x)$ in

$$\begin{array}{ccc} C(x) & \to & C \\ \downarrow & & \downarrow p \\ U & \xrightarrow{x} & A \end{array}$$

This $C(x)$ is the type (relative to the domain of definition U) that is the "value" of the dependent type C at the parameter value x of type A (which also has domain of definition U).

If we have a morphism $h:A \to D$ regarded as a term $a:A \vdash h(a):D$ (rather than as a generalized element of D), then the corresponding pullback functor on overcategories

$$h^*:\mathcal{C}/D \to \mathcal{C}/A \ .$$

represents the "reindexing" or "substitution" operation: a dependent type $y:D \vdash C(y):\mathrm{Type}$ gives rise to a dependent type $x:A \vdash C(h(x)):\mathrm{Type}$.

Now the left adjoint of this pullback functor always exists, and is given by postcomposition with h. This sends a morphism $p:C \to A$ (representing a dependent type $x:A \vdash C(x):\mathrm{Type}$) to the morphism $C \xrightarrow{h(p)} D$. Now suppose in particular that $D = *$ is the terminal object. Then this operation takes C with all its fibers $C(x)$ and

regards it as an independent type, i.e. an object of the category $\mathcal{C}$, consisting of the "disjoint union" of all these fibers. In the type theory, this operation is called the dependent sum and written

$$\sum_{x\,:\,A} C(x)$$

This is another *type constructor* that constructs the new type $\sum_{x:A} C(x)$ from the dependent type $x{:}A \vdash C(x)$.

Now by the universal property of the pullback, an element

$$z : \sum_{x\,:\,A} C(x)$$

of this sum type, i.e. a morphism

$$z{:}U \to C$$

determines a morphism $x \coloneqq p(z){:}U \to A$, i.e. a term $x{:}A$, along with a section y of x^*C, i.e. a term $y{:}C(x)$.

$$\begin{array}{ccc} U & \overset{z}{\to} & C \\ \downarrow^{\mathrm{Id}} & & \downarrow p \\ U & \xrightarrow{x\,:\,=\,p\,(z\,)} & A \end{array}$$

Thus, we can think of $C = \sum_{x:A} C(x)$ as the the *type of pairs* (x, y) such that $x{:}A$ and $y{:}C(x)$. This is reflected in the type-theoretic rules for the dependent sum.

Similarly, the right adjoint to the pullback functor is, if it exists, the *dependent product* operation , which sends the dependent type $p{:}C \to A$ to the type

$$\Pi_{x:A} C(x)$$

regarded as the *type of functions* f such that for any $x \in A$, we have $f(x) \in B(x)$. This right adjoint exists in any locally cartesian closed category $\mathcal{C}$.

Logic versus type theory in categorical semantics

How does type theory relate to logic? Well, *propositional logic* is just the type theory whose semantic categories are *posets*. In this case, the types $P, Q, \ldots$ are usually called *propositions*, and the existence of a (necessarily unique) term of type Q, having a free variable of type P, is just the assertion that $P \leq Q$ (or, in more logical language, "P implies Q"). The type constructor for binary products is usually written $\wedge$ and called "and," the type constructor for binary coproducts is usually written $\vee$ and called "or," and so on. The term constructors are generally called *inference rules*, since they allow us to infer new theorems from old ones.

Now, it turns out that there are (at least) two ways to reconcile propositional logic (the type theory of posets) with type theory of more general categories, producing predicate logic.

Logic over type theory

In the first approach, which can be described as *typed predicate logic* or *logic over type theory*, we keep the propositions separate from the types. (Since, as we have seen, propositional logic is a specific kind of type theory, this means we really have two interacting type theories. However, in this case we generally reserve "type" for the second kind of type as distinguished from the "propositions.")

In this case, the syntax has collections of types and terms, together with constructors, and also rules for forming propositions out of types and terms, and inference rules for forming implications between propositions. The types and terms form the underlying type theory of the logic, and the propositions 'depend' on these.

For instance, given two terms x, y of type A, we can often form a proposition $x = y$ which asserts that x and y are equal. Other important "proposition constructors" are the quantifiers $\exists x$ and $\forall x$, where x is a variable associated to a type (not a proposition). This can concisely be formalized as a pure type system with one sort for types and another sort for propositions, such that propositions are allowed to depend on types, but not conversely.

The natural home for the semantics of typed predicate logic turns out to be an *indexed poset*: a category C together with a functor $P : C^{\mathrm{op}} \to \mathrm{Pos}$. This is often described equivalently as a category E of propositions that is fibred over the category C of terms, and whose fibers are posets. (Thus, an alternative way of thinking of propositional logic is as the 'logic' of a poset fibred over the trivial one-object category, which corresponds to the fact that the propositions do not contain or depend on typed terms.) The ordinary type theory happens in C as described above, and a proposition ϕ with a free variable x of type A is interpreted by an element $[\phi]$ of the poset $P(A)$ (the fiber over A). The prototypical indexed poset is $P : \mathrm{Set}^{\mathrm{op}} \to \mathrm{Pos}$ sending each set to the poset of its subsets, with an evident generalization to subobjects in any category; thus we think of $[\phi]$ as "the set of all $x \in A$ such that $\phi(x)$ is true." Another way of describing this setup is as the *subobject fibration* $\mathrm{cod} : \mathrm{Sub}(C) \to C$.

Just as the allowed constructions on types are reflected in the structure of the semantic category, the allowed constructions on propositions here are reflected in the structure of the semantic posets $P(A)$. For instance, if we allow conjunction $\wedge$ of propositions, then each $P(A)$ must be a meet-semilattice. The action of the functor P on morphisms, usually written $f^* : P(Y) \to P(X)$, is used to model the substitution of the term represented by f for the variable of a proposition, multiple variables and substitutions being interpreted by means of finite products, as in Lawvere theories. In that case, the functor $\pi^* : P(X) \to P(X \times Y)$ interprets the adding of an unused variable to a context. The left and right adjoints to f^*,

when they exist, describe the semantics of the two quantifiers; thus we write them as $\exists_f \dashv f^* \dashv \forall_f$. The functors $\exists_\pi$ and $\forall_\pi$ interpret the traditional existential and universal quantifiers.

The internal logic of various sorts of categories are most naturally regarded as the typed predicate logic associated to the "poset of subobjects" functor $\mathrm{Sub}: C^{\mathrm{op}} \to \mathrm{Pos}$, and the requisite levels of structure on C induce the required semantic structure on both C and Sub. For instance, if C is regular, then each $\mathrm{Sub}(X)$ is a meet-semilattice and the adjoints $\exists_f$ exist, while if C is a Heyting category, then each $\mathrm{Sub}(X)$ is a Heyting algebra and both adjoints $\exists_f$ and $\forall_f$ exist. See also internal logic. However, not all indexed posets in which one wants to apply type theory are constructed from subobjects in some category; see for instance tripos.

Propositions as types

The second approach to reconciling type theory with logic is to blur the distinction between types and propositions; this is called the "propositions as types" paradigm. Instead of requiring that a proposition ϕ be interpreted as merely true or false (that is, a truth value or equivalently a subsingleton), we allow it to be interpreted by any set (that is, any object of the semantic category). One way to think of this is that $[\![\phi]\!]$ is the set of *proofs*, or *reasons*, why ϕ is true; it is inhabited iff ϕ is true, but a true statement may have many distinct proofs (although, for technical reasons, this is not the case in naive categorical models of classical logic). Thus, for instance, instead of asserting that $\phi \Rightarrow \psi$, we consider the *type* $\phi \to \psi$ of all proofs that ϕ implies ψ, which is inhabited just when ϕ actually does imply ψ. Similarly, the quantifiers $\exists$ and $\forall$ become identified with the dependent type constructors Σ and Π.

In this case, the semantics involved is the more general *codomain fibration* $p: C^{\to} \to C$, whose fibres are the slice categories C/A. If we want to take the point of view of "proof irrelevance," meaning that we only care whether something is true rather than how many

proofs it has, then we can think of the semantics as living in the "poset reflections" $\mathrm{pos}(C/A)$ of these slice categories (in which all parallel morphisms are identified). Note also that the $\mathrm{pos}(C/A)$ is equivalent to the poset of subobjects of A in the free exact completion of C, so this can also be regarded as doing "logic over type theory" with semantics valued in free exact completions.

Syntactic categories and free models

As mentioned above, there are two equivalent ways to describe formally the semantics of a given type theory (possibly with logic) in a category. There is an adjunction (which is at least sometimes an equivalence):

$$\mathrm{typetheories} \underset{\mathrm{Lan}}{\overset{\mathrm{Con}}{\rightleftarrows}} \mathrm{categories}$$

in which

- the right adjoint Lan (sometimes called "semantics") assigns to a category its internal type theory whose types and terms (and propositions, if present) are the objects and morphisms (and subobjects) of the category, while
- the left adjoint Con (sometimes called "syntax") builds the syntactic category of a type theory, whose objects, morphisms, and subobjects are the types (or contexts), terms, and propositions of the type theory.

Thus, if T is a type theory and C a category with corresponding structure, it is equivalent to give a structure-preserving functor $\mathrm{Con}(T) \to C$, or to give a translation of type theories $T \to \mathrm{Lan}(C)$. Either one is called a "model" of T in C. For more details on the construction of Con, see syntactic category, and for more details on Lan, see internal logic. For a description of the adjunction/equivalence, see relation between type theory and category theory.

By the way, it should be noted that there are various technical difficulties in making this precise. For instance, categories of any sort form a 2-category (or something more, if they are higher categories themselves), so we have to either make type theories into a 2-category as well, or consider strict categorical structures that form a 1-category. Also, there is a bit of a mismatch in that *substitution* in type theory is usually "implicit," which implies that it is strictly associative, but the corresponding categorical operation of pullback is not generally strictly associative. For this reason, various people have defined technical intermediaries between type theories and categories, which mostly boil down to a category equipped with a split fibration replacing its codomain fibration. These go by names like *comprehension category*, *category with attributes*, or *contextual category*; see categorical model of dependent types.

3. Syntax of type theory

It's hard to give a universal definition of "a type theory," but the following very general setup covers most cases.

Generally, a type theory is formulated by the rules called *natural deduction*, which declare the nature of each kind of type by a 4-step rule:

1. ***type formation* rules**, which say on which basis a new type can be introduced
2. ***term introduction* rules**, which say how that new type can be inhabited by terms
3. ***term elimination* rules**, which say how from a term of the new type one gets terms of other types
4. ***computation rules*** which constrain the result of combining term introduction with term elimination.

Note that in general, the following definitions are mutually recursive.

- A ***typing declaration*** is something of the form $t:A$. We say that t is a ***term*** (of type A) and that A is the ***type***. In some type theories, there is a fixed collection of allowable types, while in others the types are themselves terms belonging to some other type (often called Type).
- A ***context*** is a list of typing declarations, in which each term is a fresh variable (i.e. one not occurring to the left of its typing declaration). If the list of types is not fixed, then one requires that each type occurring in a context be well-formed relative to the sub-context appearing to its left. In other words, for $\Gamma, x:A$ to be a valid context, the judgment (see below) $\Gamma \vdash A:\mathrm{Type}$ must be derivable.
- A ***judgment*** or *hypothetical judgment* is symbols of the form $\Gamma \vdash \mathcal{J}$, where Γ is a valid context – a sequent. Different type theories allow different things in the place of $\mathcal{J}$, but the most common are *typing declarations* and *equalities* between terms of the same type. For example, the judgment
$$x:N, y:N \vdash x + y:N$$
asserts that any two natural numbers have a sum, which is also a natural number. Similarly,
$$x:N, y:N, z:N \vdash (x + y) + z = x + (y + z):N$$
asserts that natural number addition is associative.
- A ***rule*** asserts that if some given list of judgments are valid, then so is another one of a specified form derived from them. Of course, to be interesting such rules must contain "meta-variables" which range over contexts, types, or terms. Rules are generally written in the following form:
$$\frac{\Gamma_1 \vdash t_1:A_1 \quad \ldots \quad \Gamma_n \vdash t_n:A_n}{\Delta \vdash s:B}.$$
This is to be read as a rule asserting that if $\Gamma_1 \vdash t_1:A_1$ through $\Gamma_n \vdash t_n:A_n$ are valid judgments, then so is $\Delta \vdash s:B$.

A given type theory is determined by its collections of types, judgments, and rules. Rules can of course be classified in various ways; here are some of the most common.

Structural rules

Structural rules say essentially that variables can be substituted, reordered, and ignored in appropriate ways. For instance, there is an "exchange" structural rule:

$$\frac{\Gamma, x:A, y:B, \Delta \vdash ?}{\Gamma, y:B, x:A, \Delta \vdash ?}.$$

which asserts that variables in the context can be reordered. (In the presence of dependent types, there is a restriction here that B cannot depend on x.)

Some type theories, such as linear type theory related to linear logic, omit some of the structural rules, but most of the time the structural rules are taken for granted.

Type-forming rules

Most of the most interesting rules involve forming new types. For instance, we may want to assert that if A and B are types then so is $A \times B$. It may not appear that we have a kind of judgment meaning "A is a type," but we can solve this by treating every *type* as being itself also a *term* of a type such as Type (which is sometimes written $*$). Thus, for instance, the product-forming rule is written

$$\frac{\Gamma \vdash A:\mathrm{Type} \qquad \Gamma \vdash B:\mathrm{Type}}{\Gamma \vdash A \times B:\mathrm{Type}}.$$

It then comes with attendant rules for forming terms of type $A \times B$, such as:

$$\frac{\Gamma \vdash A:\mathrm{Type} \qquad \Gamma \vdash B:\mathrm{Type}}{\Gamma, x:A, y:B \vdash \langle x, y\rangle : A \times B}$$

and for extracting the original terms out, such as

$$\frac{\Gamma \vdash A:\mathrm{Type} \qquad \Gamma \vdash B:\mathrm{Type}}{\Gamma, t:A \times B \vdash \pi_1(t):A} \qquad \frac{\Gamma \vdash A:\mathrm{Type} \qquad \Gamma \vdash B:\mathrm{Type}}{\Gamma, t:A \times B \vdash \pi_2(t):B}$$

and the obvious rules saying that $\pi_1\langle x, y\rangle = x$ and $\pi_2\langle x, y\rangle = y$ and $\langle \pi_1(t), \pi_2(t)\rangle = t$.

Universes

Of course, this raises the question—what is the type of Type? We don't strictly need it to have one—nothing says that everything has to be a term of some type. But it is also sometimes convenient to write $\mathrm{Type} = \mathrm{Type}_0$ and introduce a hierarchy of additional "universes," so that $\mathrm{Type}_0:\mathrm{Type}_1$, $\mathrm{Type}_1:\mathrm{Type}_2$, and so on. A technique called "universe polymorphism" means that usually we can forget about the indices and just treat "Type" as a single entity to which everything belongs, unless we do perverse things to try to get paradoxes.

Dependent types

As suggested above, we can have types which depend on terms, and type constructors which apply to these. For instance, we can have a rule of dependent product formation:

$$\frac{\Gamma, x:A \vdash B(x):\mathrm{Type}}{\Gamma \vdash \Pi_{x:A} B(x):\mathrm{Type}}$$

Note that in the case when B is independent of x, this includes a "function type" $A \to B$. Similarly, we have dependent sums $\Sigma_{x:A} B(x):\mathrm{Type}$, which in the non-dependent case include ordinary products $A \times B$.

The original dependent type theory was Martin-Löf dependent type theory.

Propositions

As mentioned above, one way to deal with logic over type theory is to represent a proposition simply by a type, regarded as the type of all its proofs, or of all reasons why it is true. A different way is to introduce a separate type Prop, perhaps living at the same "level" as Type, and allow propositions to depend on types, in the same way that types depend on types. The same sorts of type constructors, but acting on propositions, then implement the logical connectives and quantifiers. For instance, the analogue of dependent product formation becomes a rule of universal quantification:

$$\frac{\Gamma, x:A \vdash B(x):\mathrm{Prop}}{\Gamma \vdash \forall_{x:A} B(x):\mathrm{Prop}}$$

and similarly Σ becomes $\exists$.

In either case, asserting that a proposition is "true" is the same as asserting that it is inhabited, i.e. exhibiting a term of that type. Thus, we don't need to introduce a new kind of judgment for logic; we can continue to use the same sorts of judgments of the form "t is a term of type A," only now A can be a proposition and t a proof or reason why A is true. In particular, the *axioms* of a logical theory can also be formulated as term-forming rules.

Additional dependencies

It is also possible to have types depending on propositions, propositions depending on propositions, kinds depending on types, etc. etc. See, for instance, pure type systems and the calculus of constructions.

4. Type-theoretical foundations

From a foundational point of view, type theory can also be regarded as the language in which mathematics is written. This has several aspects, notably *syntax* (the language) and *semantics* (what it means).

Syntax of type-theoretical foundations

At the most basic level, what we do when we do mathematics is *manipulate symbols according to specified rules*. Just as in chess the rules state that a knight moves like *so* and not like *so*, in mathematics the rules state that a quantifier can be eliminated like *so* and not like *so*. The actual rules of the game of mathematics are extremely complicated, but the idea of foundations is to derive them from a much simpler list of fundamental rules. Type theory says that these fundamental rules are a *calculus of terms*, and that each term comes equipped with a *type*. Thus, the rules define one or more *types*, and one of the judgments one can make (that is, one of the "moves" of the game) is of the form "t is a well-formed term of type A. This corresponds to the syntax described above.

If we include enough type constructors, then we can use type theory as a foundation for much of mathematics. Instead of building mathematical objects out of sets as in foundational set theories such as ZFC or ETCS, we build mathematical objects out of types. The presence of dependent types, with sums and products, is usually quite convenient for this purpose. That is, instead of defining a group to be a set equipped with (among other things) a function $G \times G \to G$, we could interpret a group as a *type* G equipped with (among other things) a *term* $m(x, y):G$ with free variables $x:G$ and $y:G$.

Type theory versus set theory

Alternately, we could change our terminology so that what we have been calling "types" are instead called "sets". However, in order for this to accord with the common usages of "set", we need to include enough type constructors that our types can mimic the behavior of sets, and in particular be "extensional" and have "quotient types". See the section on *Extensional vs Intensional type theory*, below.

On the other hand, type theory is, among other things, a

convenient language for formulating first-order logical theories, and among these theories are foundational set theories such as ZFC and ETCS. For instance, ZFC has two types Set and Prop, proposition-forming rules saying that if x:Set and y:Set then $(x = y)$:Prop and $(x \in y)$:Prop, the usual rules of logical inference and a collection of axioms. The same with ETCS, which it is convenient to write with three types Set, Function, and Prop.

Especially when we intend a theory like ZFC or ETCS as a foundation for all of mathematics, it is convenient to call the type-theoretic language in which these theories are written the "meta-language" or "meta-theory," while ZFC/ETCS is the "object language" or "object theory." On the other hand, for a more complex and powerful type theory with many type-constructors, which is suitable to serve as a foundation for mathematics itself, it is natural to say that this type theory is *itself* the *object-theory* in a meta-theory having meta-types such as Type, Term, and Judgment.

Thus, words like "type" and "set" and "class" are really quite fungible. This sort of level-switch is especially important when we want to study the mathematics *of* type theory, i.e. the mathematical theory of manipulating symbols according to the rules of type theory, analogous to the mathematical theory of moving pieces around on a chessboard according to the usual rules. When we study type theory in this way, we are *doing* mathematics, just like when we're doing group theory or ring theory or whatever. It's just that our objects of study are called "types", "terms", and so on. However, what we do in this mathematical theory *can*, like any other area of mathematics, be formalized in any particular chosen foundation, be it ZFC or ETCS or a type theory at a higher level. Now the type theory is itself the "object-theory" and ZFC is the "meta-theory"!

Here are some blog discussions about the difference between type theory and set theory:

- one

- two
- three

Semantics of type-theoretical foundations

Now, intuitively, we generally think of a type A as denoting some "collection" of "things", and a term $t:A$ as indicating a "particular one" of those things. In order for this to make sense, the type theory has to exist in some metatheory (which might or might not be formalized) having a notion of "set" to specify the relevant "collections of things". In particular, there must be a set of types, and for each type there is a set of terms which can be judged to be of that type. The judgment rules for propositions then become the study of formal logic; we say that a proposition is "provable" or is a "theorem" if it can be judged to be true.

Now, a *model* of this theory (in the category of sets) assigns a set $[A]$ (in the meta-theoretic sense) to every type A and a function of appropriate arity to every term, in a way so that the rules and axioms are satisfied. Thus, for instance, a model of Peano arithmetic consists of a set $[N]$, an element $[0] \in [N]$, a function $[s]:[N] \to [N]$, and so on. Likewise, a model of the type theory of ZFC (here the levels get confusing) consists of a set $[\mathrm{Set}]$, a function $[\ \in\ ]:[\mathrm{Set}] \times [\mathrm{Set}] \to [\mathrm{Prop}]$, and so on.

One can then prove, under certain hypotheses, various things about the relationship between syntax and semantics, such as:

- The *Soundness Theorem*: if φ is a proposition which is provable from the axioms of a theory, then the corresponding statement $[\varphi]$ in any model is actually true (in the sense of the metatheory). Equivalently, if a theory has at least one model, then it doesn't prove a contradiction.
- The *Completeness Theorem*: if $[\varphi]$ is true in every model of a theory, then φ is provable in that theory. Equivalently, if a theory doesn't prove a contradiction, then it has at least one

model.

- The (first) *Incompleteness Theorem*: if a theory doesn't prove a contradiction, then there exist statements φ such that neither φ nor $\neg\varphi$ is provable in the theory.
- Corollary to the completeness and incompleteness theorems: if a theory doesn't prove a contradiction, then it has more than one model.

The "certain hypotheses" is where we get into the difference between *first-order* and *higher-order*. We say that a type theory is *higher-order* if it involves type constructors such as function-types B^A (intended to represent the "type of all functions $A \to B$") or power-types PA (intended to represent the "type of all subtypes of A). Otherwise it is *first-order*. (We have to deal with Prop specially in first-order logic. If we actually have a type Prop, then the theory should be higher-order, since Prop $\cong P1$; thus in first-order logic we take Prop to be a "kind" on the same level as Type, which doesn't participate in type operations.) We say "second-order" if we never iterate the power-type operation.

> I don't buy your argument that Prop must be treated specially; perhaps I don't understand what you're saying, but I'll pretend that I do. First, I don't see the relevance of your premise, that Prop makes things higher-order because it is a power type. You might as well say that 1 makes things higher-order because $1 \cong P0$. What really makes things higher order is the ability to form *arbitrary* power types or function types, not the existence of one or two special cases. And second, I don't agree with the conclusion, that Prop can't participate in type operations. It's true that many type theories *do* treat Prop specially and forbid its participation in type operations, but allowing it to participate in first-order type operations like $\times$ and $+$ is not

The Soundness Theorem is true for all theories, but *the Completeness Theorem is true only for first-order theories*. The

Incompleteness Theorem as stated above is true for higher-order theories, but the corollary fails since the completeness theorem does. In particular, a higher-order theory can sometimes be *categorical* in the logician's sense: having exactly one model (at least, up to isomorphism). The second-order version of Peano Arithmetic has this property. (At this level, there is little fundamental difference between first-order and higher-order theories; they each have advantages and disadvantages. However, when we move up to the metalevel and talk about the term calculus itself, we always get a first-order theory. This is why some people believe that first-order logic is the only truly "foundational" logic.)

Term models

One usually proves the Completeness Theorem by building a "tautological" model out of the theory itself. That is, for each type A we simply take the set $[A]$ to be the set of terms of type A with no free variables (or "ground terms"). However, without modification, this naive idea fails for two reasons.

First of all, there might not be enough ground terms. Some of the axioms of the theory might assert that there exists something with some property, without there being a corresponding term constructor actually producing something with that property. This is obviously the case for the usual version of ZFC, which has no term constructors at all (hence no ground terms at all!) but lots of axioms that assert the existence of things. This problem is easily remedied, however, by introducing new constant terms or term constructors into the language.

The second problem is that we may not know how to define all the necessary relations on the ground terms in order to have a model. Suppose, for instance, we have a couple of ground terms t_1 and t_2 in some augmented version of ZFC; how can we tell whether $t_1 \in t_2$ should hold in our tautological model? Certainly if the axioms of the theory imply $t_1 \in t_2$, then it should hold, and if they imply

$t_1 \not\in t_2$, then it shouldn't, but they might not imply either one. The usual way to remedy this is to enumerate all such statements and one by one decide arbitrarily whether to make them true or false in the model we're constructing.

This works, but the model we get (though small, even countable, and concrete) isn't really *canonical*; we had to make a bunch of arbitrary choices. In particular, this means we can't prove the completeness theorem this way, since the statements true in this model will no longer be *precisely* those that are derivable in the theory.

In the case of Peano Arithmetic, we can avoid introducing new constant terms and obtain a model which is "canonical" and in fact the "smallest" in some sense: it consists of the terms $s(s(...(s(0))...))$, which can of course be identified with "the" natural numbers in the meta-theory. But this doesn't work for most other theories. Suppose, for instance, that we augment ZF with term constructors for all of its existence axioms. Let φ be a sentence independent of ZF; then our term-constructor for the axiom of separation gives us a term $\{\varnothing | \varphi\}$. Does the relation $\varnothing \in \{\varnothing | \varphi\}$ hold in the term model? We have to make an arbitrary choice.

(It *is* true that any *given* model of ZF contains a minimal model?, i.e. a smallest transitive set which is a model of ZF. However, different models of ZF have different minimal models, and the construction of the minimal model is not "syntactic" in this sense.)

The slicker categorial approach described above using categories of contexts does produce a really canonical model, but only with an expanded notion of "model": instead of each $[A]$ being a set, we take it to be an object of some fixed category $\mathcal{S}$ with enough structure. We can then build a much more "tautological" model because we have the freedom to build the category $\mathcal{S}$ along with the model. In the resulting model, the true statements are *precisely* the statements provable in the theory, and it's even initial among all models of the theory in the appropriate sort of category.

5. Extensional vs Intensional

There is an important distinction between *extensional* type theories and *intensional* ones. The meanings of these words is probably clearest when dealing with function types, such as an exponential Y^X, but also arises in respect to quotient types and identity types.

Extensional and intensional function types

A function type Y^X is said to be ***extensional*** if whenever $f, g : X \to Y$ are functions such that $f(x) = g(x)$ for all $x \in X$, then in fact $f = g$ as elements of Y^X. This clearly corresponds to the modeling of function types by function sets in the set-theoretic semantics, or more generally by exponential objects in the categorical semantics discussed above. The uniqueness clause of the defining assertion of an exponential object, i.e. that any map $Z \times X \to Y$ factors through a *unique* map $Z \to Y^X$, precisely models this "extensionality" property. Thus, the standard categorical semantics is most closely allied to type theories which have such an "extensionality" axiom.

By contrast, suppose that X and Y are interpreted by data types in some programming language, and Y^X is interpreted by some type of computable functions from Y^X. Of course, many differently coded functions can have the same "extensional behavior," i.e. the same output for any given input, but we may still want to distinguish between these functions because they may not share other properties (such as running time or complexity). Thus, this type Y^X is not extensional—equality of functions, as elements of Y^X, is "implementation-sensitive," a finer measure than mere equality on all inputs. We say instead that these function types are ***intensional***.

In type theory, extensional function-types generally come with both a "β-rule," which specifies the computational behavior of a λ-abstraction (i.e. $(\lambda x\,.\,t)(y) = t[y/x]$), and an "$\eta$-rule," which

specifies that a λ-abstraction is determined by its behavior (i.e. $f = (\lambda x \,.\, f(x))$). In the categorical semantics, the first specifies the existence of factorizations, while the second requires them to be unique. In intensional type theory, we generally keep the β-rule (it is certainly natural from a computational standpoint) but discard the η-rule. Thus, one natural sort of semantics for intensional type theory is valued in a category with "weak exponentials," i.e. objects which satisfy the existence but not uniqueness property of an exponential (and similarly for dependent type theory with Π-types and weak dependent products).

Quotient types and exact completion

Intensional type theory is also popular among adherents of constructive mathematics and especially predicative mathematics, because of its computational content. Per Martin-Löf's original dependent type theory is often presented from this perspective.

When viewing intensional type theory as a foundation for mathematics (rather than, say, a syntax for reasoning about computer programs), it is natural to view the types as representing presets, rather than sets. This is in line with the classical constructivist viewpoint that "a set is defined by a collection of things together with an equality relation." Note that in intensional type theory, the "equality" between terms is free to be the "syntactic" equality, which is entirely computable and preserved by everything in sight. In particular, if we adopt the viewpoint of propositions as types, then "the axiom of choice is trivially valid" for functions between types (i.e. presets) since to assert that something exists is to give an element of a sum type, which is exactly to give a witness and thereby a way to choose such a thing.

If we then define "sets" to be types equipped with equality relations (sometimes called setoids), then the sets will have more familiar properties, such as existence of extensional exponentials (obtained by equipping the intensional exponentials with an extensional equality relation), as well as the existence of quotient

sets. (The existence of quotient types is often assumed in extensional type theory, but not in intensional type theory.) In categorical terms, the syntactic category of an intensional type theory has only weak exponentials (resp. dependent products), but that is sufficient to ensure that its free exact completion has actual exponentials (resp. dependent products). Note also that free exact completions always also validate COSHEP, since every object of the starting category (here the category of types) is projective. This matches the above observations about the axiom of choice.

Identity types

> Quick comment: Even in internal type theory, one needs identity types to validate COSHEP. Type theory without identity types is very strange (the category of contexts may not have all pullbacks, and not every morphism need be a display morphism).
>
> Mike Shulman: I'm not sure that that's so strange. At least, not to the type theorists. (-: Categorically, I think of display maps as being fibrations in some model-category-type structure, which makes sense to me, although in semantics valued in an honest higher-category I would expect every morphism to be equivalent to a display morphism.
>
> Actually, don't you need *extensional* identity types in order

(to be written…)

Higher-categorical semantics

- homotopy type theory

(to be written…)

6. Particular type theories

The following particular type theories are important enough to (potentially) have pages of their own.

- simple type theory
- simply typed lambda calculus
- Martin-Löf dependent type theory
- pure type system
- the calculus of constructions
- the internal logic of various kinds of categories, including the Mitchell-Benabou language of an elementary topos
- the internal logic of a 2-topos?
- the internal logic of an (∞,1)-topos
- homotopy type theory
- cubical type theory

7. Related concepts

- ***type theory***, logic
  - logical framework, meaning explanation
  - natural deduction, calculus of constructions
  - dependent type theory, Martin-Löf dependent type theory
  - intensional type theory, extensional type theory, observational type theory
  - calculus of inductive constructions
  - modal type theory
  - computational type theory
  - geometric type theory
  - linear type theory
  - two-level type theory
  - canonical form, normal form, beta-reduction

    - Bishop set, predicative topos
    - coercion
    - intrinsic and extrinsic views of typing
    - bidirectional typechecking
- relation between category theory and type theory
- relation between type theory and topology
- 2-type theory, 2-logic
- (∞,1)-type theory, (∞,1)-logic
    - homotopy type theory
- directed homotopy type theory
  opetopic type theory
- computer science
    - data type

type, type theory

- propositions as types, inductive type, W-type
- empty type, unit type
- pointed type

dependent type, dependent type theory, Martin-Löf dependent type theory

- dependent sum type, dependent product type

homotopy type, homotopy type theory

- identity type
- homotopy n-type, connected type
- higher inductive type
- geometric homotopy type, cohesive homotopy type

8. References

The concept of typing in the foundations of mathematics is implicit in Gottlob Frege's work and, inspired by that, appears explicitly for the first time in

- Bertrand Russell, appendix of *The principles of mathematics* 1903 (online)

where it has the famous passage

> Every propositional function $\phi(x)$ — so it is contended - has, in addition to its range of truth, a range of significance, i.e. a range in which x must lie if $\phi(x)$ is to be a proposition at all, whether true or false. This is the first point in the theory of types; the second point is that ranges of significance form types, i.e. if x belongs to the range of significance of $\phi(x)$, then there is a class of objects, the type of x, all of which must also belong to the range of significance of $\phi(x)$, however ϕ may be varied;

Also

- Bertrand Russell, Alfred Whitehead, *Principia Mathematica*, 1910

and later work by Russell, where it is used to prevent paradoxes of set theory such as the liar's paradox (via a "ramified hierarchy" of types) and Russell's paradox (via an "extensional hierarchy" of types). This then evolved into the "simple type theory":

- Bertrand Russell, *Mathematical Logic as Based on the Theory of Types*, American Journal of Mathematics, Vol. 30, No. 3 (Jul., 1908), pp. 222-262
- Alonzo Church, *A Formulation of the Simple Theory of Types*, The Journal of Symbolic Logic Vol. 5, No. 2 (Jun., 1940), pp. 56-68 (JSTOR)

This is reviewed for instance in

- Jean van Heijenoort, *From Frege to Gödel*, A Source Book in

Mathematical Logic, 1879-1931

The introduction of identity types in "intuitionistic type theory" is due to

- Per Martin-Löf, *An intuitionistic theory of types: predicative part*, In Logic Colloquium (1973), ed. H. E. Rose and J. C. Shepherdson (North-Holland, 1974), 73-118. (web)

The development of that to homotopy type theory followed insights by (Hofmann-Streicher 98) and others and was laid out in

- Univalent Foundations Project, *Homotopy Type Theory – Univalent Foundations of Mathematics*, 2013

A survey of the history of type theory is in

- Stanford Encyclopedia of Philosophy, *Type theory*

Of course there is also discussion of formalized types originating in computer science as *data types*, see there for references.

Surveys of and introductions to type theory include

- Per Martin-Löf, *Intuitionistic type theory*, Studies in Proof Theory 1, Bibliopolis, Naples, 1984 (pdf)
- Thomas Streicher, *Investigations into intensional type theory*, habilitation 2003, pdf;
- Per Martin-Löf, *On the Meanings of the Logical Constants and the Justifications of the Logical Laws*, Nordic Journal of Philosophical Logic, 1(1): 11–60, 1996, (pdf, pdf)
- Rober Constable, *The Triumph of Types: Creating a Logic of Computational Reality* (pdf)
- Herman Geuvers, *Introduction to type theory*, Lecture notes 2008 (pdf)
- N. G. de Bruijn, *On the roles of types in mathematics*, Cahier du Centre de Logique (vol 8) (pdf)

- Bart Jacobs, *Categorical Logic and Type Theory*, Studies in Logic and the Foundations of Mathematics 141, North Holland, Elsevier (1999)
  Categorical Type Theory PhD. Thesis, University of Nijmegen (1991) (ps pdf)
- Andrej Bauer, *Spartan Type Theory*, UniMath School, 2017 (pdf)
- wikipedia: *type theory*, *intuitionistic type theory*, typed lambda calculus, Curry-Howard correspondence, Per Martin-Löf

Textbook accounts in from the point of view of programming languages include

- Bengt Nordström, Kent Petersson Jan M. Smith, *Programming in Martin-Löf's type theory*, Oxford University Press 1990 (pdf)
- Robert Harper, *Practical Foundations for Programming Languages*

Further discussion of type theory in the context of programming languages includes

- Simon Thompson, *Type theory and functional programming*, book page, pdf
- Jean-Yves Girard, *Proofs and Types*, (book page with download links).
- Benjamin Pierce, *Types and Programming Languages*, book page

Discussion aimed at foundations include

- Paul Taylor, *Practical Foundations of Mathematics*, Number 59 in Cambridge Studies in Advanced Mathematics. Cambridge University Press (1999) (webpage)
- Paul Taylor, *Foundations for computable topology* (web)

Work leading up to homotopy type theory includes

- Martin Hofmann, Thomas Streicher, *The groupoid interpretation of type theory*, in Sambin, Giovanni (ed.) et al., Twenty-five years of constructive type theory. Proceedings of a congress, Venice, Italy, October 19–21, 1995. Oxford: Clarendon Press. Oxf. Logic Guides. 36, 83-111 (1998) ps.gz
- Thomas Streicher, *Identity Types and Weak Omega-Groupoids* 2006 pdf.gz

Formalization of parts of mathematics in type theory is discussed in

- Thierry Coquand, *ForMath*

for homological algebra:

- Thierry Coquand, Arnaud Spiwack, *Towards constructive homological algebra in type theory* (pdf)

Thoughts about type theory and metaphysics are in

- Per Martin-Löf, *A path from logic to metaphysics*, talk at *Nuovi problemi della logica e della filosofia della scienza*, Jan 1990 (pdf)

For a type theory suitable for non-cartesian monoidal categories see

- Michael Shulman, *A practical type theory for symmetric monoidal categories*, (arXiv:1911.00818)

Further online resources include

- Type theory summer school Göteborg 2005, (materials)
- type-theory questions on MathOverflow
- type-theory questions on CS StackExchange
- types mailing list
- types-announce mailing list

Last revised on February 15, 2020 at 11:38:14. See the history of this page for a list of all contributions to it.