PROPOSITIONS AS TYPES

ANSTEN KLEV

Abstract. Treating propositions as types allows for a unified presentation of logic and type theory. Both fields thereby gain in expressive and deductive power. This chapter introduces the reader to a system of type theory where propositions are types. The system will be presented as an extension of the simple theory of types. Philosophical and historical observations are made along the way. A linguistic example is given at the end.

Keywords—Propositions as types; Curry–Howard correspondence; type theory; logic; propositions; dependent types; donkey sentences.

After introducing the thesis that propositions are types and providing a sketch of its history, this article will develop that thesis for the propositions that can be formed by means of the operators of first-order predicate logic with identity. The presentation will proceed in three stages: firstly, pure implicational logic, secondly, full propositional logic, and thirdly, predicate logic.

Introduction. A type in the sense of type theory is a kind, sort, or category of objects. Types play the role of the domains of definition of functions, including propositional functions. In particular, any domain over which a quantifier ranges is a type. (The converse—that every type is a domain of quantification—does not follow.) Types in mathematics include the various number domains, such as $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, and complexes of these, such as $\mathbb{N} \times \mathbb{N}$, $\mathbb{N} + \mathbb{N}$, and $\mathbb{N} \to \mathbb{N}$. The set-theoretical universe, $\mathbb{V}$, taken for granted in set theory, is also a type. Computer scientists work with yet other types, such as booleans, lists, vectors, and trees. The analysis of natural language might require the use of types from the empirical realm. Examples are taxonomic units such as genera, families, orders, and classes.

A proposition is a content determiner, where the content in question is the content of a speech act, such as assertion, or a corresponding mental act, such as judgement. This role-theoretic characterization of propositions leaves open how propositions are to be understood as entities. A truth value, A, determines the content that A is the True. A truth condition, A, determines the content that A is satisfied. A set, A, of possible worlds determines the content that the actual world is an element of A.

This chapter is dedicated to the thesis that a proposition is a type, namely the type of its proofs. A proposition, A, in this sense determines the content that A is inhabited. The proofs, if any, that inhabit a type are its truthmakers. That a proposition, A, is true therefore means that A is inhabited. Proofs are here treated as objects, on a par with natural numbers, pairs of natural numbers, booleans, or pairs of booleans. Once proofs are thus treated, the stated thesis follows naturally from the fundamental principle that every object is an object of some type.

A type structure mirroring the structure of propositions must be more complex than the structure of the simple hierarchy of types. In particular, dependent types, unknown to simple type theory, are needed for representing quantified propositions. Dependent types together with the possibility of quantifying over proofs enhances the expressive and deductive power of a type theory. For example, they allow for a compositional analysis of donkey sentences and a proof of the type-theoretic Axiom of Choice (see below).

Some history. Curry and Feys (1958, ch. 9E) showed that, in combinatory logic, the inhabited function types are isomorphic to the theorems of constructive implicational logic. (The isomorphism was hinted at already by Curry (1934).) They also noted the parallel between lambda abstraction and function application, on the one hand, and implication introduction and implication elimination, on the other (ibid. 9F), as well as a relation between the rule of cut in sequent calculus and substitution in typed combinatory logic.

Howard (1980), in notes that were first circulated in 1969, extended these results to the fragment of Heyting Arithmetic that lacks the existential quantifier and disjunction, and he proved a normalization theorem for the resulting type theory. (Around the same time, De Bruijn treated propositions as types in his Automath proof assistant (see de Bruijn, 1980, 1995).) The extension to full Heyting Arithmetic—and more—was realized by the constructive type theory of Martin-Löf (1975). He employed natural deduction rather than sequent calculus, and he allowed types to depend on any other type, not only on the natural numbers. Constructive type theory will guide the current presentation.

Although the idea of identifying propositions with types was fully developed only in the late 1960s, its roots reaches deeper in the history of logic. Hilbert (1922), in setting out the programme that would later bear his name, proposed to treat proofs as objects. In a different context, proofs were treated as objects by Brouwer (1927), in his proof of the Bar Theorem, and by Heyting (1931), in his elucidation of intuitionistic propositional logic. The principle that every object is typed has obvious precursors in two doctrines fom Aristotle's Organon: the doctrine of the categories and the doctrine of genus predications. A genus predication is the appropriate answer to the question of what a thing is (*Topics* I.5), hence the genus of an object is very much like its type. Aristotle's categories have traditionally been understood as the highest genera, under which all other genera fall as specifications. Both Leibniz (e.g. *Generales inquisitiones* § 199 no. 6) and Bolzano (e.g. *Wissenschaftslehre* § 154 no. 4) associated the truth of a proposition with an object's instantiating a concept, thereby foreshadowing the parallel between the truth of a proposition and the inhabitation of a type. Finally, the parallel between logical operations on propositions and algebraic operations on sets or domains lies at the basis of the 19th century algebra of logic, as perfected by Schröder (1890, 1891).

Implicational logic. A formal definition of the simple hierarchy of types was first given by Carnap (1929). In the better known formulation of Church (1940), there are two ground types and an infinity of unary function types generated from these. The ground types are the type ι, of individuals, and the type o (omikron), of propositions. The rule of function type formation says that $A \to B$ is a type whenever A and B are types. We write $a : A$ to express that a is an object of type A. The result of substituting a for the variable x in b is written $b[a]$ or—when more precision is called for—$b[a/x]$. Postfixed square brackets, as in $a[x]$, will also be used as a reminder that certain variables might be free in the given expression.

Assume that $x : A$. In other words, let x be a variable ranging over the type A. Assume, moreover, that $b[a/x] : B$ whenever $a : A$, and that $b[x]$, as a term, contains no free variables other than x. An explicit definition of a unary function, $f : A \to B$, takes the form of a single equation, $f(x) =_{\text{def}} b[x]$. By virtue of this definition, $f(a)$ is the same object as $b[a]$, for every a of type A. Church's variable-binding lambda operator allows us to introduce the same function without the need to lay down an explicit definition. Namely, whenever x and b are as described, then

$\lambda x.b$ is stipulated to be the function from A to B such that $(\lambda x.b)(a) = b[a/x]$. We say that $\lambda x.b$ is constructed from b by means of lambda abstraction, or functional abstraction. Following a common notational convention, a sequence of lambda abstractions is indicated by a single lambda followed by the sequence of variables that it binds. For instance, instead of $\lambda x.\lambda y.x$, we write $\lambda xy.x$.

Curry observed, in effect, that the types that are inhabited by functions constructible solely by means of variables and lambda abstraction are isomorphic to the theorems of constructive implicational logic. Let us consider two examples. (i) Starting from the assumptions $x : A$ and $y : B$, we can prove

$$\lambda xy.x : A \to (B \to A).$$

The type here is isomorphic to the proposition $A \supset (B \supset A)$. (ii) Starting from the assumptions $x : A \to (B \to C)$, $y : A \to B$, and $z : A$, we can prove

$$\lambda xyz.xz(yz) : (A \to (B \to C)) \to ((A \to B) \to (A \to C)).$$

The latter type is isomorphic to the proposition

$$((A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C)).$$

In order to complete the correspondence, it remains to replace the ground types ι and o by types corresponding to atomic propositions. For instance, if we assume there to be a countable infinity of atomic propositions, we must stipulate ground types $P_0, P_1, P_2, \ldots$ Curry's observation holds independently of what is further assumed about these ground types.

The correspondence extends beyond the propositions to their proofs. A proof of an implication, $A \supset B$, is a function from A to B, hence an object of the function type $A \to B$. Declaring a variable, $x : A$, corresponds to assuming A, in the sense of natural deduction; binding a variable x of type A, corresponds to discharging the assumption A; functional abstraction corresponds to implication introduction; and function application corresponds to implication elimination.

$$\begin{array}{c|c||c|c}
\to\text{-introduction} & \supset\text{-introduction} & \to\text{-elimination} & \supset\text{-elimination} \\
\text{(Abstraction)} & & \text{(Application)} & \\
\dfrac{\begin{array}{c} x : A \\ | \\ b : B \end{array}}{\lambda x.b : A \to B} &
\dfrac{\begin{array}{c} A \\ | \\ B \end{array}}{A \supset B} &
\dfrac{c : A \to B \qquad a : A}{c(a) : B} &
\dfrac{A \supset B \qquad A}{B}
\end{array}$$

Figure 1. Rules for implication

From Figure 1 one can see that the equation $(\lambda x.b)(a) = b[a/x]$, which defines λ, corresponds to the implication reduction, introduced by Prawitz (1965),

$$\dfrac{\dfrac{\begin{array}{c} A \\ \mathcal{D} \\ B \end{array}}{A \supset B} \qquad \begin{array}{c} \mathcal{D}' \\ A \end{array}}{B} \qquad \rightsquigarrow \qquad \begin{array}{c} \mathcal{D}' \\ A \\ \mathcal{D} \\ B \end{array}$$

Propositional logic. The next step is the extension to full propositional logic. Each logical connective—and in the following step, also each quantifier—corresponds to a type-forming operator. With each such operator there is associated a set (which might be empty) of introduction rules. An object of type A that is formed through the application of an introduction rule is called a canonical object of type A, or a canonical proof of A. For instance, $\lambda x.x$ is a canonical proof of $A \to A$, whereas the application $(\lambda x.x)(\lambda y.y)$ is not, where $x : A \to A$ and $y : A$.

A canonical proof of the conjunction $A \wedge B$ is a pair, $\langle a, b\rangle$, where a is a proof of A and b is a proof of B. This is an object of type $A \times B$, the product of A and B. Conjunction therefore corresponds to product types.

$\times$-introduction (Pairing)	$\wedge$-introduction	$\times$-elimination (Projections)	$\wedge$-elimination
$\dfrac{a : A \qquad b : B}{\langle a, b\rangle : A \times B}$	$\dfrac{A \qquad B}{A \wedge B}$	$\dfrac{c : A \wedge B}{\mathrm{p}(c) : A} \qquad \dfrac{c : A \wedge B}{\mathrm{q}(c) : B}$	$\dfrac{A \wedge B}{A} \qquad \dfrac{A \wedge B}{B}$

FIGURE 2. Rules for conjunction

The two coordinates of a pair may be recovered by means of projection functions, p and q, defined by the equations $\mathrm{p}(\langle a, b\rangle) = a$ and $\mathrm{q}(\langle a, b\rangle) = b$. These definitions correspond to the two conjunction reductions,

$$\dfrac{\dfrac{\overset{\mathcal{D}}{A} \qquad \overset{\mathcal{D}'}{B}}{A \wedge B}}{A} \rightsquigarrow \overset{\mathcal{D}}{A} \qquad\qquad \dfrac{\dfrac{\overset{\mathcal{D}}{A} \qquad \overset{\mathcal{D}'}{B}}{A \wedge B}}{B} \rightsquigarrow \overset{\mathcal{D}'}{B}$$

A canonical proof of the disjunction $A \vee B$ is a proof of A or a proof B together with information determining which of A or B the proof comes from. To be more specific, we say that such a proof is either $\mathrm{i}(a)$, where a is a proof of A, or $\mathrm{j}(b)$, where b is a proof of B. The disjunction $A \vee B$ therefore corresponds to the disjoint union, or sum, $A + B$, of A and B.

+-introduction	+-elimination
$\dfrac{a : A}{\mathrm{i}(a) : A + B} \qquad \dfrac{b : B}{\mathrm{j}(b) : A + B}$	$\dfrac{e : A + B \qquad \begin{matrix} x : A \\ \vert \\ c : C \end{matrix} \qquad \begin{matrix} y : B \\ \vert \\ d : C \end{matrix}}{\mathrm{D}(e, x.c, y.d) : C}$
$\vee$-introduction	**$\vee$-elimination**
$\dfrac{A}{A \vee B} \qquad \dfrac{B}{A \vee B}$	$\dfrac{A \vee B \qquad \begin{matrix} A \\ \vert \\ C \end{matrix} \qquad \begin{matrix} B \\ \vert \\ C \end{matrix}}{C}$

FIGURE 3. Rules for disjunction

The D-operator in the conclusion of +-elimination binds the variable x in its second argument and the variable y in its third argument. It is defined by the equations

$$\mathrm{D}(\mathrm{i}(a), x.c, y.d) = c[a/x]$$
$$\mathrm{D}(\mathrm{j}(b), x.c, y.d) = d[b/y],$$

corresponding to the two disjunction reductions. In effect, D glues together the two functions $\lambda x.c : A \to C$ and $\lambda y.d : B \to C$ into a single function $\lambda z.\mathrm{D}(z, x.c, y.d) : (A + B) \to C$. The binding of the variables x and y by D corresponds to the discharging of the assumptions A and B in an application of $\vee$-elimination.

Negation is defined in terms of implication and absurdity, $\neg A =_{\text{def}} A \rightarrow \bot$. The proposition $\bot$ is uninhabited as a type, hence it has no introduction rules. Absurdity elimination is the principle of *ex falso quodlibet.* This corresponds in type theory to the postulate that for every type C, there is a function $\lambda x.\text{ex}_C(x) : \bot \rightarrow C$.

Predicate logic. Predicate logic, when considered as an interpreted formalism, takes for granted domains over which the predicates and quantifiers range. Any such domain is a type. As a stepping stone for what follows, let us assume that we have certain ground types to serve as domains of quantification. For the development of mathematics within type theory, the most important ground type is the type $\mathbb{N}$ of natural numbers. It is remarkable that $\mathbb{N}$ can be treated just as a logical constant, determined by its introduction and elimination rules. There are two introduction rules:

$$0 : \mathbb{N} \qquad \frac{n : \mathbb{N}}{\text{s}(n) : \mathbb{N}}$$

The elimination rule is at once a scheme for defining functions on $\mathbb{N}$ by recursion and the principle of inference by induction. Those seeking to apply type theory in linguistics might find it necessary to assume empirical ground types, such as a type of human beings or a type of birds. Being types, these are also propositions, according to the thesis that propositions are types. Recall that a proposition is a content determiner. For example, the proposition $\mathbb{N}$ determines the content that $\mathbb{N}$ is inhabited. More generally, a type A, considered as a proposition, determines the content that A is inhabited.

Given a type, A, and objects a and b of type A, we may form the identity proposition $\mathbf{Id}_A(a, b)$. For example, $\mathbf{Id}_{\mathbb{N}}(0, 2)$ is a proposition, and if H is the type of human beings and Tom and Harry are two objects of this type, then $\mathbf{Id}_H(\text{Tom}, \text{Harry})$ is a proposition. Notice that $\mathbf{Id}$ is a ternary relation: it takes as first argument a type A, and as second and third arguments objects of type A.

For every object $a : A$ there is a canonical proof $\text{r}(a) : \mathbf{Id}_A(a, a)$. In other words, the proposition $\mathbf{Id}_A(a, a)$ is true for every object a of type A. In yet other words, the identity relation over A, $\mathbf{Id}_A$, is a reflexive relation.

The $\mathbf{Id}$-operator is a novel kind of type former, since it operates, not only on types, but also on objects: it takes a type, A, and objects, a and b, of type A to yield a type, $\mathbf{Id}_A(a, b)$. By letting variables take the place of a and b here, we form what is called a family of types, $\mathbf{Id}_A(x, y)$, over A. That $B[x]$ is a unary family of types over A just means that $B[a]$ is a type whenever $a : A$. Let A be a type, and let $B[x]$ be a unary family of types over A. That $C[x, y]$ is a binary family of types over (A, B) means that $C[a, b]$ is a type whenever $a : A$ and $b : B[a]$. The characterization of an n-ary type family proceeds by induction.

A family of types is the type-theoretical correlate of a propositional function. Having thus made sense of what a propositional function is type-theoretically, we have begun to extend the correspondence between propositions and types to the language of predicate logic.

Let us first complete our presentation of the identity types, $\mathbf{Id}_A(a, b)$. In $\mathbf{Id}$-elimination (Figure 4), $C[y, z]$ is a binary propositional function over A. The rule says that if the relation defined by $C[y, z]$ is reflexive, then $C[a, b]$ is true whenever a and b are identical. For example, provided the relation of y's knowing z's telephone number is reflexive, we may infer that Cicero knows Tully's telephone number. Since $C[y, z]$ can be any reflexive relation, the two $\mathbf{Id}$-rules taken together say that identity is the smallest reflexive relation on A.

The J-operator in the conclusion of $\mathbf{Id}$-elimination (Figure 4) binds the variable x in its fourth argument. It is defined by the equation $\text{J}(a, a, \text{r}(a), x.d) = d[a/x]$. In effect, J extends the unary function $\lambda x.d$ to a ternary function taking as its

$$\begin{array}{c|c||c|c}
\mathbf{Id}\text{-intro} & = \text{-intro} & \mathbf{Id}\text{-elim} & = \text{-elim} \\
\dfrac{a : A}{\mathrm{r}(a) : \mathbf{Id}_A(a,a)} & a = a & \dfrac{c : \mathbf{Id}_A(a,b) \qquad \begin{array}{c} x : A \\ | \\ d[x] : C[x,x] \end{array}}{\mathrm{J}(a,b,c,x.d) : C[a,b]} & \dfrac{a = b \qquad C[x,x]}{C[a,b]}
\end{array}$$

FIGURE 4. Rules for identity

arguments triples (a, b, c), where $c : \mathbf{Id}_A(a, b)$. The binding of the variable x by J corresponds to the binding of x within the derivation of the minor premiss, $C[x, x]$, in an application of =-elimination. This elegant elimination rule for the identity predicate, =, of predicate logic is due to Martin-Löf (1971). A rule of **Id**-elimination corresponding to the more familiar principle of the indiscernibility of identicals was given by Paulin-Mohring (1993).

It remains to treat the quantifiers. Just as the identity operator, **Id**, the quantifiers are always restricted to a type. Whenever A is a type and $B[x]$ is a family of types over A, then $(\forall x : A)B$ and $(\exists x : A)B$ are propositions.

A proof of $(\forall x : A)B$ is a function, c, which for every $a : A$ yields a proof $c(a)$ of $B[a]$. For instance, $\lambda x.\mathrm{r}(x)$ is a proof of $(\forall x : A)\mathbf{Id}_A(x, x)$, the law of identity on A. Notice that the type of the function value, $(\lambda x.\mathrm{r}(x))(a)$, depends on the argument, a. For instance, the type of $\lambda x.\mathrm{r}(x)$ applied to 0 is $\mathbf{Id}_{\mathbb{N}}(0, 0)$, whereas the type of the same function applied to 1 is $\mathbf{Id}_{\mathbb{N}}(1, 1)$. Regarded as a type, $(\forall x : A)B$ is the dependent function type, $(\Pi x : A)B$. If B does not depend on A, then the rules for $(\Pi x : A)B$ are just the rules for $A \to B$.

$$\begin{array}{c|c||c|c}
\Pi\text{-intro} & \forall\text{-intro} & \Pi\text{-elim} & \Pi\text{-elim} \\
\dfrac{\begin{array}{c} x : A \\ | \\ b[x] : B[x] \end{array}}{\lambda x.b : (\Pi x : A)B} & \dfrac{B[x]}{(\forall x : A)B} & \dfrac{c : (\Pi x : A)B \qquad a : A}{c(a) : B[a/x]} & \dfrac{(\forall x : A)B}{B[a/x]}
\end{array}$$

FIGURE 5. Rules for the universal quantifier

In an application of Π-introduction, just as in an application of ∀-introduction, the variable x may not occur free in any open assumptions. The binding of x by λ in Π-introduction corresponds to the binding of x within the derivation of the premiss $B[x]$ in an application of ∀-introduction.

A canonical proof of $(\exists x : A)B$ is a pair $\langle a, b\rangle$, where $a : A$, and $b : B[a]$. The type of the second coordinate, b, thus depends on the first coordinate, a. Regarded as a type, $(\exists x : A)B$ is the dependent product type, $(\Sigma x : A)B$. In Σ-elimination (Figure 6), $C[z]$ is a family of types over $(\Sigma x : A)B$. This rule therefore generalizes the usual ∃-elimination, where a proposition, C, takes the corresponding place. This generalization, which was first realized by Martin-Löf (1975), allows for the definition of projection functions and, consequently, for a proof of the Axiom of Choice (ibid.). Also +-elimination and **Id**-elimination allow for a similar generalization, which turns the rule into a principle of inference by induction: if $C[z]$ can be shown to be true of every canonical object of type A, then the elimination rule allows us to infer that $C[z]$ is true of an arbitrary $a : A$.

The E-operator in the conclusion of Σ-elimination (Figure 6) binds the variables x and y in its second argument. It is defined by the equation $\mathrm{E}(\langle a, b\rangle, xy.d) =$

Σ-introduction

$$\frac{a : A \qquad b : B[a]}{\langle a, b\rangle : (\Sigma x : A)B}$$

Σ-elimination

$$\frac{c : (\Sigma x : A)B \qquad \begin{array}{c} x : A, y : B \\ | \\ d[x,y] : C[\langle x, y\rangle] \end{array}}{\mathrm{E}(c, xy.d) : C[c]}$$

∃-introduction

$$\frac{B[a]}{(\exists x : A)B}$$

∃-elimination

$$\frac{(\exists x : A)B \qquad \begin{array}{c} B[x] \\ | \\ C \end{array}}{C}$$

Figure 6. Rules for the existential quantifier

$d[a/x, b/y]$, corresponding to the reduction rule for the existential quantifier. In effect, E turns the binary function $\lambda xy.d$ into a unary function on $(\Sigma x : A)B$. The binding of the variables x and y by E corresponds to the binding of x in the derivation of the minor premiss C, and the discharge of the assumption B, in an application of ∃-elimination. In an application of Σ-elimination, the variable x may be free in B, but not in any other assumption, and y may not be free in any open assumptions.

Example. The expressive power of the type theory presented here is well illustrated by the compositional analysis of donkey sentences offered by Sundholm (1986) and Ranta (1995). The classical donkey sentence, "Every farmer who owns a donkey beats it", is formalized in standard first-order predicate logic as

$$\forall x \forall y\, (\mathrm{Farmer}(x) \wedge \mathrm{Donkey}(y) \wedge \mathrm{Owns}(x,y) \supset \mathrm{Beats}(x,y))$$

This formalization is not compositional, since it contains no existential quantifier corresponding to the indefinite noun phrase "a donkey". For the type-theoretic formalization we stipulate two ground types, Farmer and Donkey, and two families of types, $\mathrm{Owns}(x,y)$ and $\mathrm{Beats}(x,y)$, over (Farmer, Donkey). Consider the type

(F) $$(\Sigma x : \mathrm{Farmer})(\Sigma y : \mathrm{Donkey})\mathrm{Owns}(x,y).$$

Reading the Σ's here as existential quantifiers, we would understand (F) as the proposition that there is a farmer who owns a donkey. A type may, however, also be understood as the meaning of a noun phrase, just as $\mathbb{N}$ is the meaning of the noun phrase "natural number". An object of the type (F) is a triple, (a, b, c), where a is a farmer, b is a donkey, and c is a truthmaker of the proposition that a owns b. The type expression (F) may therefore be regarded as a formalization of the noun phrase "farmer who owns a donkey". Letting the type (F) be the domain of a universal quantification, we obtain a formalization of the quantifier "every farmer who owns a donkey",

$$(\forall z : (\Sigma x : \mathrm{Farmer})(\Sigma y : \mathrm{Donkey})\mathrm{Owns}(x,y))$$

From any triple, (a, b, c), in the domain of quantification here we can extract the farmer, a, by means of the first projection, p, and a donkey, b, that a owns by means of the second projection, q. The donkey sentence can therefore be formalized as

$$(\forall z : (\Sigma x : \mathrm{Farmer})(\Sigma y : \mathrm{Donkey})\mathrm{Owns}(x,y))\, \mathrm{Beats}(\mathrm{p}(z), \mathrm{q}(z))$$

The formalization has been achieved by means of the resources already available in constructive type theory. No novel notions were needed.

Further reading. Introductions to constructive type theory suitable for the philosophical reader include Martin-Löf (1984), Nordström et al. (1990), Ranta (1995), and Klev (2018). Mathematical readers might prefer (The Univalent Foundations Program, 2013). A wide-ranging discussion of the philosophical foundations of constructive type theory can be found in (Martin-Löf, 1993).

Another type theory that relies on the identification of propositions with types is the Calculus of Constructions (Coquand and Huet, 1988). In this type theory, the Π-operator may be applied to the type of propositions. This allows for the explicit definition of all the other logical operators. For example, $A \vee B$ is explicitly defined as $(\Pi X : \mathbf{prop})(A \to X) \to ((B \to X) \to X)$ (cf. ∨-elimination in Figure 3 above).

References

Brouwer, L. E. J. (1927). Über Definitionsbereiche von Funktionen. *Mathematische Annalen*, 97:60–75.

de Bruijn, N. G. (1980). A survey of the project AUTOMATH. In Hindley, J. R. and Seldin, J. P., editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, London.

de Bruijn, N. G. (1995). Types in mathematics. *Cahiers du Centre de Logique*, 8:27–54.

Carnap, R. (1929). *Abriss der Logistik.* Springer, Vienna.

Church, A. (1940). A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68.

Coquand, T. and Huet, G. (1988). The calculus of constructions. *Information and Computation*, 76:95–120.

Curry, H. B. (1934). Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20:584–590.

Curry, H. B. and Feys, R. (1958). *Combinatory Logic.* Volume 1. North-Holland, Amsterdam.

Heyting, A. (1931). Die intuitionistische Grundlegung der Mathematik. *Erkenntnis*, 2:106–115.

Hilbert, D. (1922). Neubegründung der Mathematik. *Abhandlungen aus dem mathematischen Seminar der Hamburgischen Universität*, 1:157–177.

Howard, W. A. (1980). The formulae-as-types notion of construction. In Seldin, J. P. and Hindley, J. R., editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, London.

Klev, A. (2018). A brief introduction to constructive type theory. In Rahman, S., McConaughey, Z., Klev, A., and Clerbout, N., editors, *Immanent Reasoning*, pages 17–55. Springer, Cham.

Martin-Löf, P. (1971). Hauptsatz for the intuitionistic theory of iterated inductive definitions. In Fenstad, J. E., editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 179–216. North-Holland, Amsterdam.

Martin-Löf, P. (1975). An intuitionistic theory of types: Predicative part. In Rose, H. E. and Shepherdson, J. C., editors, *Logic Colloquium '73*, pages 73–118. North-Holland, Amsterdam.

Martin-Löf, P. (1984). *Intuitionistic Type Theory.* Bibliopolis, Naples.

Martin-Löf, P. (1993). Philosophical aspects of intuitionistic type theory. Transcript of a lecture course given at Leiden University in the autumn semester of 1993. `https://pml.flu.cas.cz/`.

Nordström, B., Petersson, K., and Smith, J. (1990). *Programming in Martin-Löf's Type Theory.* Oxford University Press, Oxford.

Paulin-Mohring, C. (1993). Inductive definition in the system Coq – rules and properties. In Bezem, M. and Groote, J.-F., editors, *Typed Lambda Calculi and Applications*, pages 328–345, Berlin.

Prawitz, D. (1965). *Natural Deduction.* Almqvist & Wiksell, Stockholm.

Ranta, A. (1995). *Type-Theoretical Grammar.* Oxford University Press, Oxford.

Schröder, E. (1890). *Vorlesungen über die Algebra der Logik. Erster Band.* Teubner, Leipzig.

Schröder, E. (1891). *Vorlesungen über die Algebra der Logik. Zweiter Band.* Teubner, Leipzig.

Sundholm, B. G. (1986). Proof theory and meaning. In Gabbay, D. and Guenthner, F., editors, *Handbook of Philosophical Logic*, volume III, pages 471–506. Reidel, Dordrecht.

The Univalent Foundations Program (2013). *Homotopy Type Theory: Univalent Foundations of Mathematics.* `http://homotopytypetheory.org/book`, Institute for Advanced Study, Princeton.

Institute of Philosophy, Czech Academy of Sciences, Prague, Czechia