The Triumph of Types: *Principia Mathematica's* Impact on Computer Science

Robert L. Constable
Cornell University

Abstract

Types now play an essential role in computer science; their ascent originates from *Principia Mathematica*. Type checking and type inference algorithms are used to prevent semantic errors in programs, and type theories are the native language of several major interactive theorem provers. Some of these trace key features back to *Principia*.

This lecture examines the influence of *Principia Mathematica* on modern type theories implemented in software systems known as interactive *proof assistants*. These proof assistants advance daily the goal for which *Principia* was designed: to provide a comprehensive formalization of mathematics. For instance, the definitive formal proof of the Four Color Theorem was done in type theory. Type theory is considered seriously now more than ever as an adequate foundation for both classical and constructive mathematics as well as for computer science. Moreover, the seminal work in the history of *formalized mathematics* is the *Automath* project of N.G. de Bruijn whose formalism is type theory. In addition we explain how type theories have enabled the use of *formalized mathematics* as a practical programming language, a connection entirely unanticipated at the time of *Principia Mathematica's* creation.

1 Introduction

1.1 Background

Principia Mathematica of Whitehead and Russell [64] runs to one thousand nine hundred and seven pages in three volumes, much of it written in symbolic logic. Volume I was published in 1910, and all three volumes are still available new. It provides a systematic detailed development of topics in mathematics expressed in a rigorous (yet not completely formal) symbolic logic. It is a challenging book to read because of its length, notation, and conceptual novelty. My copy is the ninth impression of the 1927 second edition. The jacket asserts "No other book has had such an influence on the subsequent history of mathematical philosophy."

Indeed, *Principia Mathematica* (*PM*) is a monumental work on several accounts. It offers a vision for the role of logic in mathematics. It represented a prodigious single-minded effort by authors of exceptional stature in philosophy as well as mathematics. It exerted an influence on science and the philosophy of knowledge not foreseen at the time of its publication whose ramifications remain surprising and far reaching, as in mathematics knowledge management. It spawned an extensive investigation of type theories by many outstanding mathematicians, logicians, and computer scientists that continues to this day.

There is an excellent account of the early history of type theory in the book *Foundations of Set Theory* [24]. An account up to 1940 can be found in [41], and a more up to date perspective can be found in the book *A Modern Perspective on Type Theory: From its Origins until Today* [42]. One may consider 1940 to be another important date in the history of type theory since it was during that year that Church, one of the key players who studied type theory in depth, produced his account of *Simple Type Theory* [12] which is essentially the logical basis of the interactive theorem prover HOL [29, 33, 54, 34], today one of the most widely used proof assistants. Church's *Simple Type Theory* has played an influential role in subsequent

formalizations and extensions of type theory, largely because Church considered a foundational theory based on functions [11] instead of sets, and this work gave rise to the *lambda calculus* and *typed lambda calculus*, fundamental formalisms of computer science used in providing the semantics of programming languages and the computation systems of functional programming languages and proof assistants.

1.2 Types in logic and mathematics

The research of Frege [25, 21], Russell [58, 57, 64] and Whitehead [64] concerned the *design of a logical foundation for mathematics* free from the known paradoxes and able to support an extremely detailed comprehensive treatment of mathematics in a precise axiomatic logical language. *PM* was created in that context, intended to be *safe enough* to avoid paradox and *rich enough* to express all of the concepts of modern pure mathematics of its time in a language its authors regarded as *pure logic*.

For Russell and Whitehead, type theory was not introduced because it was interesting on its own, but because it served as a tool to make logic and mathematics safe. According to *Principia Mathematica* page 37: Type theory "only recommended itself to us in the first instance by its ability to solve certain contradictions. ... it has also a certain consonance with common sense which makes it inherently credible". This common sense idea was captured in Russell's definition of a type in his *Principles of Mathematics, Appendix B The Doctrine of Types* [58] where he says "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 within which x must lie if $\phi(x)$ is to be a proposition at all,...." It is interesting that later in computer science, types are used precisely in this sense: to define the *range of significance of functions* in programming languages.

According to *PM*, statements of *pure mathematics* were inferences of pure logic. All commitments to "reality" (Platonic or physical) such as claims about infinite totalities (infinite classes), the interpretation of implication as a relation, the existence of Euclidean space, etc. were taken as hypotheses. At the time of *PM* it appeared that there would emerge settled agreement about the nature of pure inferences and their axiomatization. That was not to be.

At the very time that Russell was working on the design of his *theory of types* [58], say 1907-1908, another conception of logic was born in the mind of L.E.J. Brouwer [36, 63] *circa* 1907, a conception that would depart radically from the vision of Frege and Russell, just as they departed from Aristotle. By the early 1930's a mature expression of this new semantics emerged from the work of Brouwer, Heyting, and Kolmogorov; it is now called the *BHK semantics* for intuitionistic versions of formalisms originally developed based on truth-functional semantics. BHK semantics is also called the *propositions-as-types principle*.[1] By 1945 Kleene captured this semantics for first-order logic and Peano arithmetic in his notion of recursive *realizability* based on general recursive functions [43]. By 1968, a formal version of a comprehensive theory of types based on the *propositions as types principle* was implemented in the *Automath* theories of de Bruijn and his colleagues [20, 52]. Unlike Kleene's work, these theories did not take advantage of the computational interpretation of the logical primitives made possible by BHK, instead treating them formally as rules in the style of *PM*. Influenced by Automath, Scott [59] built on the computational interpretation of propositions as types using the lambda calculus in his 1970 sketch of a constructive type theory, a precursor to the theories of Martin-Löf.

1.3 The role of functions and analysis

I think it is fair to say that Frege, Russell, and Brouwer all agreed by 1907 that the notion of function is central to mathematics and logic, as in *propositional function* – though Brouwer was not interested in logic *per se*. The key difference was that for Brouwer a function is a mental construction and thus computable. For Frege and Russell a function is an abstract *logical notion* made clear in the axioms and principles of logic. So Brouwer provided an irreducible intuitive semantics while Frege, Russell, and de Bruijn provided

[1] Some computer scientists call this fundamental principle the "Curry-Howard isomorphism" even though it was not discovered first by either Curry or Howard and is not an isomorphism; however, Kleene, Curry, and Howard, stressed formal aspects of the principle. Recent accounts of this principle mention the contributions of twenty three logicians and computer scientists to its current formulation [61].

axioms and inference rules. Perhaps if Frege had developed his notion of the *sense* of a proposition more completely, some version of construction might have emerged.

The concept of a function was also critical in the development of computer science both in theoretical and applied work. By the 1960's computer scientists were trying to gain intellectual control over the process of programming and the design of programming languages that would better support that process. The notion of function was central to languages like Lisp [49] and to the topic that became a separate subfield, *functional programming* producing languages such as Haskell, ML and others. In the case of functional programming, the semantics was clearer than for programs with state and concurrent control. Moreover, the notion of type provided a basis for both a precise semantics and elegant programming logics. It was in this context that computer scientists and logicians created the type theories that are deeply connected to *Principia Mathematica* and serve now as comprehensive logical accounts of computing, computational mathematics, and programming logics.

As an illustration of the design issues, I will discuss later our efforts at Cornell to create one such type theory, *Computational Type Theory (CTT)* [15], very closely related to two others, the *Calculus of Inductive Constructions* (CIC) [19, 7] implemented in the Coq prover [7] and widely used, and *Intuitionistic Type Theory* (ITT) [46, 47] implemented in the Alf and Agda provers. All three of these efforts, but especially CTT and ITT, were strongly influenced by *Principia* and the work of Bishop [8, 9] presented in his book *Foundations of Constructive Analysis* [8].

Bishop believed that Brouwer had formulated a core truth about mathematics but that he pushed his views too far ahead for contemporary understanding and focused too much on attacking "classical" methods; in so doing he alienated colleagues unnecessarily. Bishop showed by example that constructivizing the standard notions of mathematics allowed him to develop a very large part of analysis in such a way that it is *computationally meaningful and yet readable as ordinary analysis*. All of his theorems were readable as mainstream ("classical") mathematics, and yet all had computational meaning, and indeed in most cases "numerical meaning" [9]. The success of his program posed a major challenge to logicians: to formalize the theory and logical language he used. Bishop's logic appeared to involve a constructive set theory, where sets carried their own equality relation. So considerable effort was spent in trying to find the right formulation. One of the contenders was the type theory, ITT, formulated by Martin-Löf in which types came equipped with a notion of equality specific to the objects of the type, another was intuitionistic set theory IZF [51, 26].

1.4 Types in programming

The notion of type is a central organizing concept in the design of programming languages, both to define the *data types* and also to determine the range of significance of procedures and functions.[2] Types feature critically in reasoning about programs as Hoare noted in his fundamental paper on data types [38]. The role of types in programming languages is evident in Algol 60 [65] and its successors such as Pascal and Algol 68 (where types were called *modes*). One of the most notable modern examples is the language ML, standing for MetaLanguage, designed by Milner as an integral part of the *Edinburgh LCF* mechanized *Logic for Computable Functions* [30, 60]. This ML programming language with its remarkably effective *type inference algorithm* and its recursive data types is widely taught in computer science. It also provides the metalanguage for several of the major interactive proof assistants in service today, such as Agda [10], Coq [7], HOL [29], Isabelle [54], MetaPRL [37], Nuprl [15], and Twelfth [55].

Type systems also play an important role in understanding program termination. For example, in some formalizations of type theory [5], a typed term A must be *strongly normalizing*, that is, it reduces to a value regardless of the order of reducing subterms. Some modern types, such as intersection, were introduced for the purpose of characterizing strong normalization.

[2] This use matches Russell's definition of a type as the *range of significance* of a propositional function.

1.5 Types in programming logics and computer-assisted reasoning

It is also the case that the subject of *automated reasoning* got its start with attempts of Newell, Shaw, and Simon to automatically prove elementary theorems of *Principia* using their *Logic Theorist* [53] program – another contribution to computer science which like that of Hoare, Milner, and Scott was recognized by its highest honor, the Turing Award.

In the early 1980's my colleagues and I designed *Computational Type Theory* (CTT) and implemented it as the native logic of the Nuprl ("new pearl") interactive theorem prover [15] and later in MetaPRL [37] as well. The theory CTT extended Per Martin-Löf's *Intuitionistic Type Theory* (ITT) [46, 47, 48], and it employed his novel semantic method.[3] This theory also realized a dream since at least 1971 [14] to create a formal theory of constructive mathematics that is a combination programming language and logic.[4]

During the design process we also studied *Principia Mathematica* (*PM*) in some detail – acquiring a copy of the entire three volume second printing and reading widely about it and drawing inspiration from it. Like Martin-Löf we concluded that the notion of orders from *Principia* was a clean way to avoid the known paradoxes associated with overly inclusive collections and to include "large objects" like categories. We found Russell's argument in *Principia* Volume I that there could not be a single type of all propositions completely compelling.

We built the Nuprl interactive theorem prover in 1984 as an extension of λPRL which used the tactic mechanism of Milner's *Edinburgh LCF*; LCF is a formalization of Scott's theory of computing with *partial recursive functions* [60]. Nuprl-5 is in use today, implementing a significant extension [3] of the 1984 theory based on the work of over twenty Cornell PhD students as well as a number of students and researchers elsewhere, including from Bundy's automated reasoning group at Edinburgh University.

Nuprl and CTT have contributed over the years several ideas that have influenced modern implemented type theories and their applications in both mathematics and computer science. Here I will report on some of those ideas with connections to *PM*. Our current technical understanding of *PM* and its relationship to Nuprl owes a great deal to the work of Kamareddine, Laan, and Nederpelt [42].

1.6 Design Choices *circa* 1980

By the 1970's when computer scientists and logicians worked to formulate a comprehensive logical account of programming logics and computational mathematics, there were many more options in the design space than in 1907, and the case for a comprehensive foundational language for "pure classical mathematics" seemed to have been resolved in favor of set theory (in one form or another).

The issue of avoiding paradox was also fresh on the minds of the theory architects because in 1972 Girard had discovered a contradiction in Martin-Löf's 1971 theory of types in which he had postulated a universal type, $Type$, with $Type \in Type$. This is now known as *Girard's Paradox* [17, 39]. In response to this paradox, Martin-Löf introduced a predicative [22] hierarchy of universes, $\mathbb{U}_i$. These correspond to the orders in *Principia*.

At Cornell a key challenge for us was to reason about computation in all of its many forms, to deal with intensional notions such as computational complexity and program structure. However, it was also critical to interface to applications and pure mathematics. Applications provide urgent goals and mathematics informs good computation in numerical analysis, scientific computing, computational geometry, and a myriad of other subjects. Moreover, as programming logics [44, 32] developed, they touched more concepts and methods from logic. All of this activity brought to the forefront the possibility of a unified computational semantics

[3] According to this method, a type was defined by first giving its canonical (irreducible) values in a computation system and saying when two such values are equal. Martin-Löf's method built in the notion that any term t' that reduces to a canonical value t is equal to that value in its type. His method led us to the notion that a type is like the structured sets of Bishop and could be defined as a *partial equivalence relation* over a set of terms of a computation system [4].

[4] Before knowing of the work of Martin-Löf, I had been implementing a typed programming logic, called V3, using Kleene *realizability*. The generality and elegance of Martin-Löf's methods and the ease of implementing them in Lisp and ML, changed the course of our work on computational type theory at Cornell.

for logic, mathematics, and computing.

2 Type Theory Design Principles

We will examine the design issues that faced the logicians and computer scientists who created and implemented the type theories that are now the native languages of several modern proof assistants mentioned above. These proof assistants are rapidly advancing the goals of *Principia*, at the rate of thousands of theorems per year, creating a new field called *mathematics knowledge management* (MKM). These are not the only considerations in the design space, but they illustrate issues closely related to *Principia Mathematica* and the context in which it arose.

2.1 Sets versus Types

For a computer scientist, sets are not a universal data type. Church's efforts [11] to base a comprehensive foundation for mathematics on *functions rather than sets* led him and his students to the *lambda calculus* [12] and to his simple theory of types (STT) [13]. This in turn led McCarthy [49] and his students to define the programming language Lisp where functions, atoms, and lists are the major data types. Lisp is one of the first functional programming languages.

It makes sense to provide *function types*, $atype \rightarrow btype$ in any programming language, rather than viewing functions as sets of ordered pairs. The same is true for numbers, arrays, lists, sequences, trees, graphs, etc. – they are defined as data types, not encoded as sets. Thus from the beginning, types were more appropriate for practical computation than sets. It became natural to think of Set as one among many (data) types. Aczel [1, 2] has shown how to define sets as types and provide a model of Constructive ZF set theory (CZF) inside ITT. His semantics makes it possible to consider finite and infinite sets as elements of a type Set at any universe $\mathbb{U}_i$ in ITT and CTT. The MetaPRL implementation of CTT includes CZF sets.

On the other hand, it is useful to encode types as sets for some purposes, for example, in order to make comparisons with set theory. The writings of Howe [40, 23] demonstrate the value of defining types as sets in order to relate the types of a classical theory such as HOL to the types of a constructive theory.

2.2 Computational versus Axiomatic Semantics

Computation provides a meaningful account of large parts of mathematics and is essential in a theoretical foundation for computer science. In contemporary terms, computational meaning is given explicitly by *reduction rules*. Our ability as humans to execute small instances of this semantics, and on that basis *grasp it fully in principle*, directly connects this approach to what Brouwer called mental constructions. This computational meaning is captured well by our intuitive and practical grasp of small step reductions as in Plotkin's *structured operational semantics* [56]. Not only that, but our ability to implement this semantics in software and check the correct performance of computations with simple programs provides a concrete realization of the meaning and *a basis for computer assistance with complex intellectual tasks*.

Given the computer scientist's goal of providing a foundational account of programming logics, computational mathematics, as well as computing and information science (informatics), it is natural to see whether a precise implementable account of Plotkin's structured operational semantics provides a semantic basis for such a theory. Martin-Löf' [47] discovered that it does; his ITT82 makes this case. ITT82 also leads to a philosophical view based on *actual computing* that is congruent with the computer scientist's experience of computation and at the same time an instance of Brouwer's ideas about the most basic mental constructions – those that Bishop identified as fundamental for numerically meaningful mathematics. Following Bishop, we can avoid dealing with other intuitionistic principles for the time being and focus on those needed to support modern computational mathematics and informatics [45, 62]. CTT's extensions of ITT82 show how far we can go in this direction. In particular we can include elements of concurrent and distributed computing and

computation on streams and other co-inductive structures. Perhaps in the future, some of Brouwer's less clear intuitionistic principles will be seen as instances of precise new methods of computing and reasoning about those new methods.

Computational content is present in the assertions of the constructive type theories, and one of the features of CIC, CTT, and ITT is that implementing the proof machinery makes it possible to find this content and execute it. This discovery has led to a new proof technology for *extracting computational content* from assertions. That technology has proven to be very effective in practice, and there is a growing literature on this subject which we do not explore here, but there are references in survey articles such as [3].

The semantical basis of *Principia Mathematica* is clearly neither computational nor set theoretic. It is based in our philosophical understanding of logical primitives in terms of their truth values, and like *Begriffsschrift*, *Principia* treats these principles *axiomatically*. As a result, the notion of function is treated axiomatically as well. What is striking about ITT, and thus also about CTT and CIC as well, is that the Brouwer-Heyting-Kolmogorov interpretation provides computational meaning to the logic. Thus *Principia Mathematica* set the stage for the emergence of the propositions as type principle, the foundation of the constructive proof assistants, by bringing logic and types together. To some logicians and computer scientists, this principle is one of the deepest of the subject.

2.3 Extensional versus Intensional Equality

The types of programming languages have not been defined as mathematical objects in the normal sense because a programmer's definition does not require for every type a precise notion of equality on its elements. For instance, equality of functions is left unspecified, and for floating point numbers it is not sound programming practice to decide a branch point based on exact equality of reals. Also it is not required to know whether two syntactically different type definitions are equal, e.g. two different recursive data types. The definition of type needed by the programmer and compiler writer is concerned with compatibility of the data formats and machine representation of the data. Beeson [6] discusses "presets" as types without an associated notion of equality; *and programmers tend to think of types as "structured presets."*

In mathematics, the definition of types clearly requires a notion of equality. We see this in Bishop's writings where he uses the term "set" rather than "type." His notion of equality is not the conventional set theoretic one because for Bishop each set comes with its own equality, as he says on page 13 of *Foundations of Constructive Analysis* [8], " Each set A will be endowed with a relation of equality. This relation is a matter of convention, except that it must be an *equivalence relation*." It is this concept of equality that Martin-Löf partially adopts in ITT and we fully adopted in CTT by defining the *quotient type*, $A//E$, which associates the equivalence relation E with type A. This quotient type is very important for capturing the style of Bishop's mathematics and for treating the algebraic notion of "quotient structures" in a computational manner. For example we use quotients to define a computationally sensible notion of integers mod p, $\mathbb{Z}_p$ and other such concepts.

Function equality is not extensional in *programming logics* with function types because that would make type checking undecidable. However in the 1982 version of ITT, equality is *extensional* in the sense that two functions f_1 and f_2 on a type $A \to B$ are equal iff for all a of type A, $f_1(a) = f_2(a)$ *in* B. This means that equality is not a decidable relation on functions. Because the dependent types are defined with functions as components, the equality of types in ITT82 and CTT is also not decidable. Consequently, membership in a type, say $a \in A$, is not decidable either. Thus there is no *type checking algorithm* for ITT82 and CTT. In mathematics it is common to use extensional equality, but in programming it is common to use *intensional or structural equality*; so two functions are equal if their syntactic form is the same. Indeed, we expect equal functions in a programming type to have the same computational behavior, say the same cost (computational complexity), but extensionally equal algorithms might have very different computational cost on some inputs, indeed the seminal paper on computational complexity by Hartmanis and Stearns [35] is titled "On the computational complexity of algorithms." They do not say "of *functions*."

2.4 Predicative versus Impredicative Types

In the book *Foundations of Set Theory* [24] on page 58 in discussing impredicativity [22], the authors bring up concerns about impredicative definitions; they say on page 178 "*The fourth argument, and certainly the strongest one, refers to the nonconstructive character of impredicatively introduced objects. We can hardly be said to have a clear idea of a totality if the membership of a certain object in this totality is determinable only by reference to the totality itself. ... it might occasionally give rise to antinomies and will in any case cause great difficulties in the construction of models that would prove the consistency of the system.*" This viewpoint proved prescient when Girard discovered a paradox in Martin-Löf's first impredicative version of ITT. However, it also proved true that Girard was able to introduce a new method of proof which established the consistency of an impredicative type theory [27, 28] that eventually formed the basis for the *Calculus of Constructions* (CoC) [18] implemented in the first version of the Coq prover. Coquand proved the consistency of CoC using an extension of Girard's method.

The Coq and Nuprl provers were youthful contemporaries circa 1984-88; they were the two main experimental interactive proof assistants for versions of type theory based on constructive logic and a computational semantics. From the start there were interesting contrasts. CTT adopted the predicative universes of ITT73 while CoC was impredicative. CTT adopted extensional function equality from ITT82 while CoC used intensional equality. CTT's computation system included the entire *untyped lambda calculus*, including the Y-combinator, and is thus a *Turing complete* programming language, while CoC is based on a class of total functions and was thus *subrecursive*. Unlike ITT82, CTT extensively exploited the use of the Y combinator to create efficient programs. More striking however is that the definition of the CoC existential quantifier was not constructive in the standard sense and could not be strengthened without creating a contradiction. In addition, the inductive data types did not justify a recursion combinator for computation in constrast to the recursive types of CTT [15, 50]. These limitations motivated the design of the richer CIC logic [19, 7] with its *predicative* hierarchy of sets and inductive types similar to CTT's recursive types, and thus its ties to *PM*.

2.5 Effectively Computable, Turing Computable, and Subrecursive Computation Systems

Brouwer's notion of computability is not formal and not axiomatic. It is intuitive and corresponds to what is called *effective computability*. The *Church/Turing Thesis* claims that all effectively computable functions are computable by Turing machines (or any equivalent formalism, e.g. the untyped λ-calculus). There is no corresponding formalism for *Brouwer Computable*. However, I believe that this notion can be captured in intuitionistic logics by leaving a Turing complete computation system for the logic *open-ended* in the sense that new primitive terms and rules of reduction are possible. This method of capturing effective computability may be unique to CTT in the sense that the computation system of CTT is open to being "Brouwer complete" as a logic. We have recently added a primitive notion of *general process* to formalize distributed systems whose potentially nonterminating computations are not entirely deterministic because they depend on asynchronous message passing over a network which can only be modeled faithfully by allowing unpredictable choices.

2.6 The Issue of Partial Computable Functions

A salient feature of Turing complete computation systems is that they must include *nonterminating computations*, and thus *partial computable functions*. While ITT82 allows such functions in its computation system, it does not provide rules for reasoning about them. CTT provides a tentative *constructive logic for computable functions* based on [16]. Edinburgh LCF [30] provides classical rules for fixed point induction based on [60]. Neither of these theories provides a fully adequate account of *partial computable functions*, and finding a workable theory remains a perplexing subject of theoretical and practical interest. This problem was not foreseen in the early days of type theory, and its resolution seems to require understanding the *range of significance of a partial computable function* – a good challenge for 21st century type theory.

References

[1] Peter Aczel. The type theoretic interpretation of constructive set theory. In A. MacIntyre, L. Pacholski, and J. Paris, editors, *Logic Colloquium '77*. North Holland, 1978.

[2] Peter Aczel. The type theoretic interpretation of constructive set theory: Inductive definition. In *Logic, Methodology and Philosophy of Science VII*, pages 17–49. Elsevier Science Publishers, 1986.

[3] Stuart Allen, Mark Bickford, Robert Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and Evan Moran. Innovations in computational type theory using Nuprl. *Journal of Applied Logic*, 4(4):428–469, 2006.

[4] Stuart F. Allen. A Non-type-theoretic Definition of Martin-Löf's Types. In Gries [31], pages 215–224.

[5] Henk P. Barendregt. *Handbook of Logic in Computer Science*, volume 2, chapter Lambda Calculi with Types, pages 118–310. Oxford University Press, 1992.

[6] Michael J. Beeson. *Foundations of Constructive Mathematics*. Springer-Verlag, 1985.

[7] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[8] E. Bishop. *Foundations of Constructive Analysis*. McGraw Hill, NY, 1967.

[9] E. Bishop. Mathematics as a numerical language. In *Intuitionism and Proof Theory*, pages 53–71. North-Holland, NY, 1970.

[10] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda – a functional language with dependent types. In Christian Urban Makarius Wenzel Stefan Berghofer, Tobias Nipkow, editor, *LNCS 5674, Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[11] Alonzo Church. A set of postulates for the foundation of logic. *Annals of mathematics, second series*, 33:346–366, 1932.

[12] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:55–68, 1940.

[13] Alonzo Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematical Studies*. Princeton University Press, Princeton, 1941.

[14] Robert L. Constable. Constructive mathematics and automatic program writers. In *Proceedings of the IFIP Congress*, pages 229–233. North-Holland, 1971.

[15] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986.

[16] Robert L. Constable and Scott F. Smith. Computational foundations of basic recursive function theory. *Journal of Theoretical Computer Science*, 121:89–112, December 1993.

[17] Thierry Coquand. An analysis of Girard's paradox. In *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science*, pages 227–236. IEEE Computer Society Press, June 1986.

[18] Thierry Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[19] Thierry Coquand and Christine Paulin. Inductively defined types. In Grigori Mints Per Martin-Lf, editor, *Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag, 1988.

[20] N. G. de Bruijn. The mathematical language Automath: its usage and some of its extensions. In J. P. Seldin and J. R. Hindley, editors, *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61. Springer-Verlag, 1970.

[21] Michael Dummett. *Frege Philosophy of Mathematics*. Harvard University Press, Cambridge, MA, 1991.

[22] Solomon Feferman. Predicativity. pages 590–624, 2005.

[23] Amy P. Felty and Douglas J. Howe. Hybrid interactive theorem proving using Nuprl and HOL. In William McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 351–365. Springer, July 13–17 1997.

[24] A. A. Fraenkel and Y. Bar-Hillel. *Foundations of Set Theory*, volume 67 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 2nd edition, 1958.

[25] Gottlob Frege. Begriffsschrift, a formula language, modeled upon that for arithmetic for pure thought. In J. van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, pages 1–82. Harvard University Press, Cambridge, MA, 1967.

[26] Harvey Friedman. Set theoretic foundations for constructive analysis. *Annals of Math*, 105:1–28, 1977.

[27] J-Y. Girard. Une extension de l'interpretation de Gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In *2nd Scandinavian Logic Symposium*, pages 63–69. Springer-Verlag, NY, 1971.

[28] J-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*, volume 7 of *Cambridge Tracts in Computer Science*. Cambridge University Press, 1989.

[29] Michael Gordon and Tom Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, 1993.

[30] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, NY, 1979.

[31] D. Gries, editor. *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1987.

[32] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. Foundations of Computing Series. MIT Press, Cambridge, 2000.

[33] John Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998.

[34] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Clarendon Press, Oxford, 2009.

[35] Juris Hartmanis and R. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematics Society*, 117:285–306, 1965.

[36] A. Heyting, editor. *L. E. J. Brouwer. Collected Works*, volume 1. North-Holland, Amsterdam, 1975. (see On the foundations of mathematics 11-98.).

[37] Jason Hickey, Aleksey Nogin, et al. MetaPRL — A modular logical environment. In David Basin and Burkhart Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 2003.

[38] C. A. R. Hoare. Notes on data structuring. In *Structured Programming*. Academic Press, New York, 1972.

[39] Douglas J. Howe. The computational behaviour of Girard's paradox. In Gries [31], pages 205–214.

[40] Douglas J. Howe. Semantic foundations for embedding HOL in Nuprl. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 85–101. Springer-Verlag, Berlin, 1996.

[41] Fairouz Kamareddine, Twan Laan, and Rob Nederpelt. Types in logic and mathematics before 1940. *Bulletin of Symbolic Logic*, 8(2):185–245, 2002.

[42] Fairouz Kamareddine, Twan Laan, and Rob Nederpelt. *A Modern Perspective on Type Theory: From its Origins until Today*. Kluwer Academic Publishers, Boston, 2004.

[43] S.C. Kleene. On the interpretation of intuitionistic number theory. *Ann. Hist. Comput.*, 3:52–67, 1981.

[44] Dexter C. Kozen and Jerzy Tiuryn. Logics of programs. In van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 789–840. North Holland, Amsterdam, 1990.

[45] G. Kreisel. *Foundations of intuitionistic logic*, volume Logic Methodology and Philosophy of Science, pages 198–210. Stanford University Press, Stanford, CA, 1962.

[46] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, pages 73–118. North-Holland, Amsterdam, 1973.

[47] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[48] Per Martin-Löf. An intuitionistic theory of types. In Giovanni Sambin and Jan M. Smith, editors, *Twenty-Five Years of Constructive Type Theory*, volume 36 of *Oxford Logic Guides*, pages 127–172, Oxford, 1998. Clarendon Press.

[49] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[50] P.F. Mendler. Recursive types and type constraints in second-order lambda calculus. In Gries [31], pages 30–36.

[51] J. Myhill. Constructive set theory. *The Journal of Symbolic Logic*, 40:347–382, 1975.

[52] R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer. *Selected Papers on Automath*, volume 133 of *Studies in Logic and The Foundations of Mathematics*. Elsevier, Amsterdam, 1994.

[53] A. Newell, J.C. Shaw, and H.A. Simon. Empirical explorations with the logic theory machine: A case study in heuristics. In *Proceedings West Joint Computer Conference*, pages 218–239, 1957.

[54] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[55] Frank Pfenning and Carsten Schürmann. Twelf — a meta-logical framework for deductive systems. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206, Berlin, July 7–10 1999. Trento, Italy.

[56] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Aarhus University, Aarhus University, Computer Science Department, Denmark, 1981.

[57] Bertrand Russell. Mathematical logic as based on a theory of types. *Am. J. Math.*, 30:222–62, 1908.

[58] Bertrand Russell. *The Principles of Mathematics*. Cambridge University Press, Cambridge, 1908.

[59] D. Scott. Constructive validity. In D. Lacombe M. Laudelt, editor, *Symposium on Automatic Demonstration*, volume 5(3) of *Lecture Notes in Mathematics*, pages 237–275. Springer-Verlag, New York, 1970.

[60] D. Scott. Lattice theoretic models for various type-free calculi. In *Proceedings 4th International Congress in Logic and Methodology and Philosophy of Science*, pages 157–87. North-Holland, Amsterdam, 1972.

[61] M.H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomoprhism*. Elsevier, 2006.

[62] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, Amsterdam, 1996.

[63] Walter P. van Stigt. *Brouwer's Intuitionism*. North-Holland, Amsterdam, 1990.

[64] A.N. Whitehead and B. Russell. *Principia Mathematica*, volume 1, 2, 3. Cambridge University Press, 2nd edition, 1925–27.

[65] N. Wirth and C.A.R. Hoare. A contribution to the development of ALGOL. *Communications of the ACM*, 9:413–432, 1966.