Applied Logic - CS4860-2018-Lecture 1

Robert L. Constable

Abstract

For over two millennia logic developed as a distinct subject in philosophy and mathematics. It is now vitally important in computer science (CS) as well. Logic is essential to guarantee the correctness and safety of software systems on which society increasingly depends. A single coding error or misunderstanding in building a critical system component can and has resulted in disaster. Furthermore, research in artificial intelligence (AI) has shown how to provide substantial computer support for the highly precise logical reasoning needed in mathematics and computer science. With computer support, humans can achieve error free reasoning and quickly explore alternative proofs. This area of investigation has produced many interesting new research questions.

A new kind of software system, called a *proof assistant*, combines the imaginative ingenuity of humans with the inhumanly fast and flawless reasoning of machines to solve problems that arise in mathematics, software design and construction. Proof assistants have deepened our understanding of the fundamental concepts of logic, and they have helped build industrial software and solve open problems in mathematics. Efforts to improve the reasoning abilities of proof assistants now have both scientific and economic value. Research using proof assistants is leading to ever more surprising and unexpected results about the nature of human computer collaboration.

This Applied Logic course will cover the *basic logical core of proof assistants* and illustrate some important applications, both in fundamental research and applications in CS. We will also explore the impact of these tools on our philosophical understanding of logical concepts and on human-machine knowledge formation. Moreover, there is strong interest and substantial funding for deploying proof assistants more broadly in university education and eventually in secondary education as well. We have experience in this realm which we will discuss.

1 Course mechanics and topic outline

1. Logistics

   Textbook: Smullyan *First-Order Logic* [32].
   Lecture Notes
   Readings: articles, book (*Type Theory and Functional Programming* [33])
   Problem sets – 10 to 12

2. Outline of Topics

3. Propositional Calculus
   Smullyan account
   Constructive propositional calculus as a programming language
   Classical completeness
   Constructive completeness

4. First-Order-Logic
   Smullyan account
   Constructive version, iFOL
   Dependent types
   iFOL as a programming language
   Completeness
   Constructive completeness and *uniform validity*

5. Foundations of Mathematics – *Set Theory*
   Paradoxes
   Zermelo-Fraenkel Set Theory with the Axiom of Choice, *ZFC*
   cardinality
   continuum problem
   large cardinals
   ZFC based mathematics: number theory, real analysis, algebra

6. Foundations of Mathematics – *Type Theory*
   Principia Mathematica [39]
   Gödel's incompleteness theorem

7. Foundations of Computer Science – *Constructive Type Theory*
   shortcomings of set theory
   types in programming languages [20]
   impredicative type theories

8. Intuitionistic mathematics – Brouwer and Poincarè
   nature of the continuum of real numbers, $\mathbb{R}$
   free choice sequences

2 Content Overview

Logic has been an important element of mathematics for centuries and of computer science (CS) since its birth. Logic is in fact one of the oldest academic disciplines, going back to Aristotle's

philosophy in 350 BCE and to Euclid's geometry in 300 BCE, extended by Archimedes by mid 200 BCE. Logic played a major role in shaping mathematics as can be seen in Euclid's *Elements* and in the 20th century creation of *set theory* and *type theory* as languages for precisely expressing mathematical knowledge and for resolving questions about its foundational meaning. One of the founders of CS is Alan Turing who received his PhD at Princeton University [35, 36] under the guidance of Alonzo Church, one of the first internationally renown American mathematical logicians. Church's famous *lambda calculus* was the direct inspiration and basis for the programming language Lisp developed by John McCarthy [26, 25] in 1962, and still in use today. The lambda calculus is also the basis for new functional languages that followed Lisp such as Standard ML, OCaml, Haskell, $F^\sharp$ and others.

The subject of *automated reasoning* arose as a key subject in computer science starting with Newell, Shaw, and Simon in 1957 [28] and growing to be a major research area the 1980's [24, 6, 5, 9, 31]. This line of research eventually led to *proof assistants* which are now important and widely used tools in the heart of computer science [2]. They are used to precisely *define* the tasks that a software system is designed to accomplish and *logically prove* that the resulting code satisfies the precise specifications. One of the first modern proof assistants, Nuprl ("new pearl"), was built in CS at Cornell in 1984 [11]. It is still in use. Members of the Nuprl team went on to influence and help create one of the most widely used proof assistants today, Coq [4, 10], supported by the French government's INRIA research facility. Recent research at Cornell has used Coq to prove that the Nuprl logical rules are correct [1]. This is another landmark result about modern proof assistants.

Proof assistants have created a new kind of logical reasoning, just as chess playing computers such as Deep Blue created a new kind of chess playing [8]. An unaided human does not stand a chance at winning a game of chess against Deep Blue nor even against the best chess programs available on smart phones.[1] On the other hand, the combination of a human and a computer *chess assistant* can beat Deep Blue and the "smart chess phones". Why is that? What is the lesson behind this fact? Humans can provide what chess champion Kasparov calls "*strategic guidance*." They imagine creative new ways for how to win and find plans and moves that have never been explored before [8]. Machines working alone are much less good at creative imagination. The combination of human strategic guidance and the computer's ability to rapidly follow it is a powerful new mode of work and play. We will explore this mode.

This new *human-machine partnership* will be widely explored. Logicians, mathematicians, and computer scientists will use it to ensure that a software system will always work correctly. It applies to rigorously proving a new mathematical result or finding exactly the right theorem to prove. The futurist Kevin Kelly says that we will be paid in the future by *how well we work with machines*. This course will seek to broaden our horizons on how to work with a class of machine with a promising future, proof assistants. We will explore a subject with a distinguished history from ancient times and a very bright future in the second machine age, *applied logic*.

[1] By now, chess programs running on smart phones have grandmaster ratings and can beat almost any human on the planet.

2.1 Course Summary

The course will begin with an account of *propositional logic*, the typical starting point for all logic textbooks. We will use a small book by Raymond Smullyan entitled *First-Order Logic* [32], one of the most elegant and compact logic textbooks ever written. Propositional logic deals with the logical operators of *and*, as in $A\&B$, *or*, as in $A \vee B$, *implies*, as in $A \Rightarrow B$, and the logical constant $False$ which is sometimes also written as $\perp$. The logical operator *not* is written $\sim A$, and is defined as $A \Rightarrow \perp$. In the standard account of propositional logic, it is assumed that the propositional variables such as $A, B, C, D, ...$ have *truth values*, either *true* or $false$.[2] We will consider another interpretation in which we assign *evidence* to the propositional variables. The evidence is intended to provide a reason that we know the proposition to be "true."

We are interested in evidence because that is how humans come to understand ordinary propositions as well as mathematical propositions. Generalizing examples of evidence leads to the discovery of new mathematical results. For example, if we know that $x < y$ means that we can find a positive numerical witness $u > 0$ such that $x + u = y$, then we know that if $x < y$ and $y < z$, then we have evidence that $x < z$. This is because we have $x + u_1 = y$ and $y + u_2 = z$, so $x + u_1 + u_2 = z$. Therefore $x < z$ using $(u_1 + u_2)$ as the positive witness. This method of understanding logical arguments applies widely in mathematics as well as daily life. For example, in Euclidean Geometry we argue that if *figure A is congruent to figure B* and *B is congruent to C*, then *A is congruent to C*. It takes a little work to see this, defining what it means to be congruent. Intuitively we know that it means we can "place figure A on B" and "place figure B on C." We can then see a construction for placing A on figure C. We first place A on B. Then A comes along for the ride as we place B on C. This method of understanding goes beyond mathematics, for example evidence is critical in legal reasoning. To "prove" that a person violated the law, there must be evidence as in "being caught in the act" by a credible witness or by recording a car's speed at a point on the road. Legal training involves knowing the *logic of legal evidence*. Children learn the lessons of evidence early in home life and find strategies for ensuring "fair treatment" as in cutting the pie for desert: the child who cuts the pie into equal pieces is the last to choose his or her piece of pie. This "cake-cutting" topic in mathematics is extremely rich and applicable in creating *fair protocols* in computer science.

2.2 Role of Computation

Mathematicians discovered early on, starting at least with Euclid, how to compute with geometric objects as well as with numbers and how to create evidence that constructions accomplished certain tasks. They could copy a line segment from one point to another, Euclid's Proposition 2. They could bisect an angle or construct a line segment perpendicular to another, both useful tasks. They extended the computational method to algebra and developed computations for finding solutions to equations, factoring polynomials, and so forth. The book by Harel [17] tells this story very well in describing the *spirit* of computer science. The centrality of human computation in understanding the logical operators was stressed by L.E.J. Brouwer [7] in the early 20th century, by 1907.

[2] Some logic books use 1 for *true* and 0 for $false$.

He noted that the basic logical operations as used in mathematics have computational meaning. Understanding these *intuitive mental constructions* is the key to understanding what it means for mathematical statements to be "true." He went on to argue forcefully that computation grounds *all* mathematical truths.

Brouwer also claimed that the contradictions and paradoxes that created a "foundational crisis" in mathematics in the mid 1800s were due to doomed attempts to justify non-computational methods for establishing *mathematical truth*, by adopting various "obviously true" logical principles with no computational meaning. These views provoked considerable controversy with other mathematicians such as Hilbert who believed that an axiomatic approach to truth was more general. Brouwer is known for rejecting the "law of excluded middle," the logical principle that every proposition is either *true* or *false*. Hilbert was very much against giving up this "obvious truth" that is so useful in most of mathematics. Nowadays, we see how to reconcile these two view points and deeply enrich the study of logic and mathematics. We will cover this issue in some detail because from the computer science point of view, Brouwer's idea and the compromise worked out in computer science, are quite clearly not only right, but very useful. The standard approach to mathematical truth that accepts *excluded middle*, was for various reasons called *classical logic*.[3]

Since Brouwer we have come to understand that the computational method is more general than the pure axiomatic method and can even be used to explain that method in terms of computing with "virtual evidence." We will examine this new idea in this course. Indeed, the course will stress the computational method and its explanations of the axiomatic method using evidence and virtual evidence. This fits very well into a course designed for both computer science students and mathematics students and also very informative for philosophy students as well. The approach based on evidence provides valuable information beyond simply knowing that a proposition is "true" (we have evidence) or is "false," meaning we can't possibly have evidence. It also leads to very useful versions of the completeness theorem for first-order logic that are the center piece of basically every introductory logic courses. We will study the "computational proofs of completeness," a new topic for any logic course.

2.3 Functional Programming Languages

We will use very small part of a popular functional programming language to compute with mathematical evidence. That will help us engage computers right at the start in doing logic. The programming language will precisely define how to compute with evidence and understand more deeply the *computational foundation of knowledge*. We will see that the *truth values*, *true* and *false* are a weak kind of evidence, but it is often too weak and uninformative to help understand computational issues. This programming language is at the core of modern proof assistants and is used by them to give useful computational value to logical proofs. Indeed these programming languages often use the initials ML, as in SML (for Standard ML) or OCaml (for Objective ML)

[3]This name is misleading because the mathematics of the classical Greeks was computational as well as axiomatic, unlike the later "axiomatic methods" which dropped the computational requirements. Brouwer claimed that this mistake of *ignoring the computational meaning* is what led to confusion and paradoxes in the new attempts to explain mathematical truth purely axiomatically.

or ClassicML, etc. The ML in these languages is derived from the very first example created by Robin Milner [4] and his research group at Edinburgh University in Scotland [15] where ML stood for "metalanguage." The proof assistants Coq [4], HOL [29], and Nuprl [11] are built on the "proof tactic" method from the Edinburgh system.

The programming language we use (in a very elementary way) is OCaml [27], but we will also use a simpler mathematical syntax to write basic OCaml programs. For example, the *identity function* in OCaml, is written as $fun\ x \rightarrow x$. This is a syntax close to Church's original *lambda calculus* definition of a function, written $\lambda x.x$. The name "lambda calculus" comes from Church's decision to signify a function definition by writing the Greek letter lambda (λ) followed by the name of the argument (or input) to the function followed by the "body" of the function definition. The body typically uses the input but is not required to, as in the constant function $\lambda x.0$. We will take the integers as numerical values, 0,1,-1,2,-2,.... *The exceptional power and enduring value of this simple programming language comes from the fact that functions are data as well as operations on data.* So we can compute with the function $\lambda x.\lambda y.x$ which takes an input value x and returns as output a constant function that produces that value on any input for y. Initially to make it clear that there is an *operator* for applying a function, we will write $ap(f; a)$ to denote the *application* of the function f to the input value a. We can write $ap(f; 0)$ as the application of the function f to the numerical constant 0. If we write $ap((\lambda x.\lambda y.x); 0)$ the result will be the function $\lambda y.0$ which is the *constant function* that returns 0 on any input. So we can compute $ap((\lambda y.0); 1)$ to the value 0.

Notice that the name of the input in the identity function, the x in $\lambda x.x$. is arbitrary. We could use any variable name such as $\lambda y.y$, $\lambda z.z$, $\lambda in.in$, and so forth. All of these expressions define the abstract idea of the *identity function*. We will consider them *equal as functions* even though they are definitely not equal as *syntactic expressions*. We give this kind of equality the name *alpha equality,* following Church. In logic it is very important to understand this distinction. We need syntax to write down mathematical expressions or expressions of programming languages. But the mathematical objects we are defining, in the above case the identity function, usually does not depend on the syntax. In the case of numerical expressions, the story is a bit different. We want 0,1,2,... to be the standard (decimal) names for the natural numbers, say as opposed to the Roman numerals, I,II, III, IV, V, VI,.... There are other examples we will see where we adopt *standard names*, as for the logical operators.

Types Another critical feature of the OCaml programming language, and of the entire ML family of languages, is that data is classified into *types*. The integers are a type of the language, written as *int*. The function $fun\ x \rightarrow x + 1$ computes the successor function on integers, so $ap(\lambda(x.x + 1); 1)$ reduces or computes to the integer value 2. We say that the function has the type $int \rightarrow int$, telling us that it maps any integer to another integer. If we try to reduce the term $ap(\lambda(x.x + 1); \lambda(y.y))$ we would get the expression $\lambda(y.y) + 1$ which does not "make sense" in the normal way. The plus operator, +, is defined as an operation on integers, so 10 + 5 makes sense and reduces to 15, but $\lambda(y.y) + 1$ *does not make sense*, and if we try to further evaluate it, the computation "gets stuck."

[4]Robin was a good friend of Cornell CS who won the Turing Award for this work, and we sometimes think of ML as Milner's Language.

Boolean values Another atomic OCaml data type is the type of *Booleans*. The type is called *bool* and the values are spelled out as $true$ and $false$. With this type we can treat logic as Boolean Algebra. OCaml has the standard Boolean operators with odd looking names. If a and b are Boolean expressions, then so are the following expressions: a && b for "and," $a \parallel b$ for "or," and $not\ a$, for negation. There is no built in "implication" operator, but it can be defined as an infix function, $a\ imp\ b$ means $(not\ a) \parallel b$. We can learn a lot about propositional logic by seeing how OCaml evaluates expressions such as $true \parallel true$, $true \parallel false$, $false \parallel true$ and so forth.

The type of Booleans help us describe the computational features of other types. For example, we know that we can decide whether two integers are the same. We can reduce them to their decimal form and then compare those decimals digit by digit. Thus we know that the equality relation $n = m$ between two integer expressions n and m is such that we can compute the expression to a boolean value. We know that $n = m$ will compute to $true$ exactly when the two numbers are the same. Otherwise the value is $false$. In OCaml, these facts are made explicit by the procedure that does this computation and reduces $n = m$ to either the Boolean value $true$ or $false$.

OCaml also has the type *unit* which has only one element denoted $\star$. We can also define the type *void* in OCaml. It is a type with no elements and corresponds to the logical notion $False$ which is a type with no elements or a proposition with no possible evidence. Early on we will study the remarkable fact that some of the basic type constructors of OCaml represent logical operators on propositions. Moreover, *all of the OCaml types have a clear mathematical interpretation.*

First-Order Logic The title of Smullyan's text book is *First-Order Logic*. This logical system is the heart of modern logic because it is expressive enough to formulate the standard theories used in modern mathematics, e.g. elementary number theory (sometimes called Peano Arithmetic - PA), real analysis, and Zermelo, Fraenkel set theory with the axiom of choice (ZFC). ZFC set theory is widely believed to be expressive enough to rigorously define all the concepts of modern mathematics. However, it is not a sufficient basis for computational theories and thus not a good foundation for computer science. There is a version of First-Order Logic that can express computational ideas well. Mathematicians call it *intuitionistic First-Order Logic*, a "mouth full" indeed. If we abbreviate First-Order Logic as FOL, then the intuitionistic version is abbreviated as iFOL. This version of the logic can be seen as a programming language in which it is possible to express a computational theory of numbers, both natural numbers and real numbers. We will study this theory and its role in computing. It brings logic and computer science together in one theory, and it ties that theory to some of the deepest philosophical issues in the foundations of mathematics and epistemology. The story of this logic involves some of the most colorful and influential mathematicians of the last century – Brouwer, Hilbert, Poincaré, Herbrand, Krönecher, Cantor, and others.

One of the central theorems about first-order logic is due to Gödel in his doctoral dissertation. It is the *completeness theorem* that shows that every valid formula of this logic is provable. Smullyan gives a very elegant proof of this theorem and explores its consequences. We will study this fundamental result carefully. We will also sketch a proof of the analogue of this result for iFOL and show how that proof applies to FOL as well. Gödel is most well known for his deep *incompleteness theorem*. This is one of the most celebrated theorems in logic, and we will look

at it in its simplest form, for a theory called **Q**. We will also look at Kleene's abstract form of Gödel's theorem presented in his widely read logic book, *Introduction to Meta-Mathematics* [22]. This proof uses the unsolvability of the halting problem, a result known to all computer science students.

Type Theory A major alternative foundational theory for mathematics is *type theory* which was first presented in work of Bertrand Russell [30, 39]. Constructive versions of this theory are adequate as a foundation for computer science as well as mathematics. These constructive type theories have been implemented in proof assistants [11, 21, 4] and are having a significant impact on computer science research and education. We will look at the core ideas in this subject and show how programs can be "extracted" from constructive proofs [3]. Many of the ideas in constructive type theory were developed by L.E.J. Brouwer, the Dutch mathematician who created the intuitionistic philosophy of mathematics [7, 19, 38, 37]. These ideas appear to be steadily gaining force over the years and are now well integrated into three of the major proof assistants in use today. We will explore some of Brouwer's ideas about logic and computability and show how they are being used today in very practical ways.

Philosophy Brouwer's ideas have had a significant impact on the philosophy of mathematics [12, 13, 18, 34, 23, 16, 14],and we will devote parts of several lectures to explore the philosophical issues he raised. His bold ideas are likely to lead to lively class discussion as they have for over a hundred years among mathematicians and philosophers. The ideas are considerably less controversial in computer science where several of them have been embraced, explored, and deeply integrated into using proof assistants and programming languages. To give the flavor of Brouwer's ideas it is interesting to note that he does not believe that the famous "law of excluded middle," is valid in all areas of mathematics, although it has been used in logic since Aristotle and Plato. This law simply says that "every proposition P is either true or false." Symbolically it is usually written $P \vee \sim P$.

I agree with Brouwer, and will try to persuade you to have an open mind on this topic despite the fact that it is a law of the first logical system we will study, the (classical) propositional calculus. In that simple setting the law can be defended.

References

[1] Abhishek Anand and Vincent Rahli. Towards a formally verified proof assistant. In *International Conference on Interactive Theorem Proving*, pages 95–197, 2014.

[2] Henk Barendregt and Herman Geuvers. Proof-assistants using dependent type systems. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1149–1238. Elsevier, 2001.

[3] J. L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions of Programming Language Systems*, 7(1):53–71, 1985.

[4] 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.

[5] W. Bibel. *Automated Theorem Proving*. Vieweg, Braunschweig, 1982.

[6] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, 1979.

[7] L.E.J. Brouwer. Intuitionism and formalism. *Bull Amer. Math. Soc.*, 20(2):81–96, 1913.

[8] Erik Brynjolfsson and Andrew McAfee. *The Second Machine Age*. W.W. Norton and Company, New York, 2014.

[9] Alan Bundy. *The Computer Modeling of Mathematical Reasoning*. Academic Press, New York, 1983.

[10] Adam Chlipala. An introduction to programming and proving with dependent types in Coq. *Journal of Formalized Reasoning (JFR)*, 3(2):1–93, 2010.

[11] 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.

[12] Michael Dummett. The philosophical basis of intuitionistic logic. In H.E. Rose J. Shepherdson, editor, *Logic Colloquium '73*, pages 5–40, Amsterdam, 1975. North Holland.

[13] Michael Dummett. *Elements of Intuitionism*. Oxford Logic Series. Clarendon Press, 1977.

[14] Kurt Gödel. On intuitionistic arithmetic and number theory. In M. Davis, editor, *The Undecidable*, pages 75–81. Raven Press, 1965.

[15] 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.

[16] Johan Georg Granström. *Treatise on Intuitionistic Type Theory*. Springer, 2011.

[17] D. Harel. *Algorithmics: The Spirit of Computing*. Addison-Wesley, Reading, MA, 1987.

[18] A. Heyting. *Intuitionism, An Introduction*. North-Holland, Amsterdam, 1966.

[19] A. Heyting, editor. *L. E. J. Brouwer Collected Works*, volume 1. North-Holland, Amsterdam, 1975.

[20] C. A. R. Hoare. Recursive data structures. *International Comput. Inform. Sci.*, 4(2):105–132, 1975.

[21] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq proof assistant : A tutorial : Version 6.1. Technical report, INRIA-Rocquencourt, CNRS and ENS Lyon, August 1997.

[22] S. C. Kleene. *Introduction to Metamathematics*. D. Van Nostrand, Princeton, 1952.

[23] S. C. Kleene and R. E. Vesley. *Foundations of Intuitionistic Mathematics*. North-Holland, 1965.

[24] D.W. Loveland. *Automated Theorem Proving: A Logical Basis*. North-Holland, New York, 1978.

[25] 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.

[26] J. McCarthy et al. *Lisp 1.5 Users Manual*. MIT Press, Cambridge, MA, 1962.

[27] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml*. O'Reilly, Beijing, Cambridge, 2014.

[28] 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.

[29] 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.

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

[31] Jörg Siekmann and Graham Wrightson. *Automation of Reasoning*, volume 1 of *Classical Papers on Computational Logic*. Springer-Verlag, New York, 1983.

[32] Raymond M. Smullyan. *First-Order Logic*. Dover Publications, New York, 1995.

[33] Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.

[34] Anne Sjerp Troelstra. *Metamathematical Investigation of Intuitionistic Mathematics*, volume 344 of *Lecture Notes in Mathematics*. Springer-Verlag, 1973.

[35] A. M. Turing. On computable numbers, with an application to the Entscheidungs problem, a correction. In *Proceedings London Math Society*, volume 43.

[36] A. M. Turing. Computability and λ-definability. *Journal of Symbolic*, 2:153–63, 1937.

[37] Mark van Atten. *On Brouwer*. Wadsworth Philosophers Series. Thompson/Wadsworth, Toronto, Canada, 2004.

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

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