When logic meets engineering: introduction to logical issues in the history and philosophy of computer science

Liesbeth de Mol, Giuseppe Primiero

When logic meets engineering: introduction to logical issues in the history and philosophy of computer science*

Liesbeth De Mol* and Giuseppe Primiero†
*CNRS – UMR8163 STL, Université de Lille 3, France
† Department of Computer Science, Middlesex University, UK

September 2, 2016

The birth, growth, stabilization and subsequent understanding of a new field of practical and theoretical enquiry is always a conceptual process including several typologies of events, phenomena and figures spanning often over a long historical period. This is especially true when the field in question is not uniquely identified by either the Academia, or the Laboratory, or the Industry. Computing is a paradigmatic case. So diverse and conflicting are its origins, that the debates on the nature of computer science have characterized its whole history. From its early beginnings onward, computing has been variously labelled as a technology, a science and as a form of mathematics. It has been said that computing is a discipline dealing with machines that compute (See [35]), with information processed by such machines (See [28]) or with the algorithms that direct the behaviour of such processes (See [31]). Today, when computers are so extensively present in our lives, one would expect that theoreticians and practitioners in the field of computing would have found, at least, some consensus on these questions. The opposite is true however and there is still much controversy on the scientific, engineering and experimental qualifications pertaining to the discipline.[1]

With the rise of the modern computer and the practices that surrounded it, came the realization that modern computing is as much a product of engineering as it is the result of formal and mathematical science. For instance, as Stan Ulam, an important figure in early computing practices of the late 40s and 50s, recounted:

*This is a preprint of the following paper: L. De Moland G. Primiero (2015), *When logic meets engineering: fundamental issues in the history and philosophy of computer science*, History and Philosophy of Logic, vol. 36, nr. 3, pp. 195–204.

[1] See [43] for a recent in-depth study of the past and ongoing debates on what computer science is.

> It is perhaps a matter of chance, that computer development became possible only by a confluence of at least two entirely different streams. One is the purely theoretical study of formal systems. The study of how to formalize a description of natural phenomena or even of mathematical facts. [...] The whole idea of proceeding by a given set of rules from a given set of axioms was studied successfully in this connection. The second stream is the technological development in electronics, which came at just the right time. (46, p. 94)

These two different aspects of computer science, moreover, are not strictly separated: logic and technology work together, from the lowest hardware level, governed by Boolean circuits and arithmetical operations in the stack memory; through the structure of assignment, sequencing, branching and iteration operations defining modern high-level programming languages; up to the equivalent abstract formulations of recursive definitions for algorithms. Accordingly, (the history of) computer science can be understood only by investigating the non-straightforward and non-linearly proceeding interactions between logic and engineering practices, which influenced each other and which received, moreover, further stimuli from external areas such as developments in business or the experimental sciences.

The logical roots of computer science are, at least, well-known. They can be traced back to the extensively studied debate on the foundations of mathematics from the end of the XIXth – early XXth century. The *Grundlagenkrisis* in mathematics notoriously brought the three foundationalist approaches to the fore: the logicist, the formalist and the intuitionist programmes.[2] The derivation of Russell's Paradox in Frege's *Begriffschrift* determined the collapse of the first of these programmes, which aimed at deriving all mathematics from purely logical notions. This drawback in the search for foundations meant that Hilbert's finitist and formalist programme was reinvigorated in its attack of problems such as consistency and decidability. It is within this context that the work by mathematicians such as Church, Kleene, Post and Turing has its origins. They each contributed in making the idea of calculation a central topic in logic, by proposing different formalizations of computability ([44]), effective calculability ([6]), generated set ([38])[3] and solvability ([39]).[4] These formalizations were entirely in the spirit of the formalist programme, in the sense that they allow '*to abstract from the meaning of the symbols and to regard the proving of theorems (of formal logic) as a game played with marks on paper according to a certain arbitrary*

[2]See respectively [3], [48] and [29] for a historical representation of these three programmes. For a collection of source texts on the foundations of mathematics, see [47].

[3]Post developed this notion and its formalization in terms of generated sets in 1921 and proved on its basis the (absolute) unsolvability of a particular decision problem for his normal systems. However, he did not submit the results to a journal. In 1941 he submitted an account of this work from the early 20s to the *American Journal of Mathematics*. The paper was rejected but a shortened version was finally accepted and published in 1943.

[4]These notions of course expanded on iteration and recursion, whose first definitions can be traced back to Bolzano (unnoticed), Cauchy and Weierstrass. See [1] for an extensive but accessible historical recollection of the notion of recursion as the foundation of computability.

set of rules' ([5]). Such formalizations were required to prove that there is in fact no finite method to solve Hilbert's *Entscheidungsproblem*, or other related decision problems. As such, these results, next to Gödel's incompleteness, broke Hilbert's dream of making mathematics void of *ignorabimus*. The fundamental problem of determining for any assertion of first-order calculus whether or not it is valid, Hilbert's *Entscheidungsproblem* (Decision Problem) in its original form, was proven (recursively) unsolvable by Church, who showed it depends on the recursive solvability of problems in the λ-calculus[5] and by Turing who showed its dependence on the decidability of decision problems for Turing machines, most notably the problem which we know today as the halting problem.[6,7] Similar problems were also proven unsolvable by Post already in the early 20s. Despite this strict link to effective calculability, the mere idea of using computations in mathematics was very much opposed by Hilbert, who considered the practical concerns of calculation removed from his interests.[8] Ironically, it were exactly the different formalist devices and techniques by which impossibility results were obtained, such as the universal Turing machine or the λ-calculus, that would also allow to provide (some of) the theoretical foundations of computer science.[9]

The third foundationalist programme also had an important and lasting influence on the theoretical foundation of computing. Brouwer's subject-based constructivist interpretation of mathematical truths, resulting in the formalization of Intuitionistic Logic by Kolmogorov and Heyting in the early '30s with the rejection of the Law of Excluded Middle, reflected more closely the algorithmic reconstruction of the rules for classical predicate logic. This approach matched the idea of *execution* of rules for a classical language. Later, the coupling of logic and computation was advanced further. The algorithmic operators S, K, I of the combinatorial calculus were defined as computationally equivalent representations by trees of any operation in the (untyped) version of the λ-calculus

[5]That is, his formalization of effective calculability (next to general recursive functions). More precisely, he proved that the problem to decide for any λ-defined formula whether or not it has a normal form is recursively unsolvable (Theorem XVIII of [6]). On the basis of this result, Church was able to show that the Entscheidungsproblem is unsolvable in any system of symbolic logic which is adequate to a certain portion of arithmetic and is ω-consistent (as a Corollary of Theorem XIX in [6].). In another short paper [7], he then showed that this result can be extended to first-order logic, hence proving the unsolvability of Hilbert's Entscheidungsproblem.

[6]More specifically, Turing proved that there exists no (Turing) machine which allows to decide for any Turing machine whether or not it is circular or circle-free. In Turing's terminology, circularity means that the machine never writes down more than a finite number of symbols (halting behavior). A non-circular machine is a machine that never halts and keeps printing digits of some computable sequence of numbers. On its basis, Turing then proved that also the problem to determine for any given machine whether or not it will ever print some symbol x cannot be computed by a Turing machine and showed that this problem can be reduced to first-order logic.

[7]For a comparative study of the different formalizations proposed by Church, Kleene, Post and Turing and their connection with decision problems, see [22].

[8]In an influential report on algebraic number theory known as *Zahlbericht* and published in 1897, Hilbert explicitly favours a more conceptual approach over a computational one (See for instance [8] for more details).

[9]See for instance the papers by Felice Cardone and Edgar Daylight in this volume.

(and hence to recursive functions). In this they constituted a further Turing complete language. The equivalence of the *type* of such operators to the axiom schemas

$$K : A \to (B \to A)$$

$$S : (A \to (B \to C)) \to ((A \to B) \to (A \to C))$$

and function application corresponding to Modus Ponens made this in turn equivalent to the implicational fragment of Intuitionistic Logic (and hence the typed λ-calculus), as suggested by Curry; this equivalence was later rediscovered by Howard for natural deduction, a correspondence known today as the Curry-Howard isomorphism, which influenced the construction of computational systems like Martin-Löf's Type Theory and (much later) the development of proof checkers and automated theorem provers, such as Coq and Isabelle/HOL. The family of strongly normalizing typed systems are today the basis for various functional programming languages, with guarantee of important properties such as termination (by strong normalization) and memory access consistency (by typing).

But the relevance of these formal results in logic for later computing practices were certainly not evident, if not entirely disregarded, in the early days: the modern computer was not developed yet and the original context of those formal works was pure rather than applied mathematics. In this latter context, human and machine computational practices became more and more important because of, amongst others, advances in military research, requiring for example new firing tables for every type of new gunnery.[10] It is for instance well-known today that ENIAC, one of the first electronic and programmable computers, was the answer to the problems encountered at the Ballistic Research Lab at Aberdeen Proving Ground with the timely computation of ballistic tables: the combination of the computations of the differential analyzer – an analogue machine – and the teams of human computers could not cope with the demands of the military. They were too slow.[11] It was within this context of slow, error-prone human and machine computations that the first electronic and programmable computers such as ENIAC, the Baby Manchester machine, the EDSAC or the ACE and EDVAC designs were developed in the late 1940s.[12] These machines were real behemoths when compared to modern-day computers and access to

[10] In [37] the following description is given of firing tables: '*[D]uring the Wold War II period [t]he Army depended entirely on the accurate aiming of shells our guns fired at enemy targets. [...] The procedure was to aim first at enemy targets based on information provided in firing tables and, in the event the target was missed, to make corrections on information also provided by these tables [...] The information in the table was used directly by the gunner or was incorporated in the firing mechanism appended at the artillery equipment, anti-aircraft gun, or bomb sight*'. For a detailed study of calculatory practices before the rise of the modern computer, see [23].

[11] See [23, 37].

[12] There has been much debate within the history of computing about 'the first' computer. Today, historians consider this question no longer legitimate since much depends on how one defines 'computer' and adjectives such as 'stored-program' or 'general-purpose' which one often associates with it.

them was restricted to a selected number of people with diverse backgrounds: engineers, like Eckert and Mauchly, but also mathematicians or logicians like Turing, von Neumann or Curry. Partly thanks to the war effort, these people were forced to work together and disciplinary boundaries had to be crossed, especially between pure mathematics and engineering. Before that time, the connection between logic and digital circuitry had been made, amongst others, by Claude Shannon and Victor Shestakov who showed how to represent digital circuits by Boolean algebras.[13] Beyond this basic hardware level, though, the electronic programmable computer required a deeper reflection on the use of logic to control computations: on the one hand, programmability meant the possibility of use for a variety of purposes; on the other, the electronic nature of computers meant they were too fast for humans to follow the computation. As von Neumann explained (49, p. 2):

> [It is] necessary to consider carefully the ability of the computing mechanism to take our intention correctly. And the person controlling the machine must foresee where it can go astray, and prescribe in advance for all contingencies. To appreciate this, contemplate the prospect of locking twenty people for two years during which they would be steadily performing computations. And you must give them such explicit instructions at the time of incarceration that at the end of two years you could return and obtain the correct result for your lengthy problem! This dramatizes the necessity for high planning, foresight, and consideration of the logical nature of computation. *This integration of logic in the problem is a consequence of the high speed.* [m.i.]

Computers were born from the need of speed and precision in computations; and now logic was called for controlling (the correctness of) computations that were too fast for humans to check. One application of this requirement is the so-called stored-program idea which, roughly speaking, meant storing both instructions and data in the machine.[14] Another application was the development of the flowchart notation by von Neumann and Goldstine which relies heavily on logical terminology (for instance, the use of bound and free variables).

With the need for logical control over dynamically performed computations came also the need to develop communication means, feasible for both machine and human user. In the early days, such communications proceeded either through direct physical wiring (as in the case of the orginal ENIAC) or through a very primitive order code very close to the machine. As a result, 'programming' the machine, as we call it today, was an extremely laborious and

[13] See [41] and [42].

[14] See [26] for a detailed discussion of the stored-program concept. This principle has led to attributing the invention of the modern computer to Turing, because his Universal Machine requires instructions to be treated as data and conversely. It is clear, though, from recent historical research, that the development of the idea of the electronic, general-purpose and stored-program computer is more complicated and cannot be attributed to Turing alone. See especially [10] and [25].

error-prone task and it became clear that much time could be gained if one could communicate with the machine in a 'language' that would abstract more from the hardware and allow to automate processes, e.g. calling a subroutine and returning to the main procedure.[15] Of course, this meant also the need for a computational method to 'translate' such language to machine language. In this way, the steady development of so-called high-level programming languages and compilers went hand-in-hand. The first compilers and languages were developed in the late 50s. Logic kept playing a crucial role: to give a few examples, Haskell B. Curry, who had also worked with ENIAC, developed in the late 40s a theory of program composition insisting on the significance of formal logic in this context;[16] Chomsky relied on Post's formal devices to define his hierarchy of languages which even today forms the foundation of compiler design;[17] McCarthy used notions coming from λ-calculus and recursive functions to define the LISP language.[18] Simplicity of programming and increasing computational power helped the commercialization of the computers, the emergence of the programmer's profession and the increasing academic acknowledgement of computer science.[19] These developments resulted in a range of problems which have been identified as a software crisis in the late 60s and (especially) early 70s by a selected group of people. Dijkstra, during his Turing award lecture in 1972, which had a profound impact on the community, used the term *software crisis* as follows (17, pp. 860–861):[20]

> [In the early days] one often encountered the naive expectation that, once more powerful machines were available, programming would no longer be a problem, for then the struggle to push the machine to its limits would no longer be necessary and that was all that programming was about wasn't it? But in the next decades something completely different happened: more powerful machines became available, not just an order of magnitude more powerful, even several orders of magnitude more powerful. But instead of finding ourselves in a state of eternal bliss with all programming problems solved, we found ourselves up to our necks in the *software crisis* [m.i.]! [...] The major cause is ... that the machines have become several orders of magnitude more powerful! To put it quite bluntly, as long as there were no machines, progamming was no problem at all; when we had a few weak computers, programming became a mild problem,

[15] For a discussion of the introduction of the so-called language metaphor in computer science, see [36].

[16] Amongst others, he connected this work to combinatory logic, lambda calculus and recursive functions. See [13] for a detailed discussion.

[17] See e.g. [4].

[18] See [34].

[19] See for instance [18].

[20] The 1968 NATO Software Engineering conference is the classic reference for the origin of the term 'software crisis'. As it has been argued in [24], one should be careful in overestimating the impact of this so-called crisis and the NATO conference. In fact, as he shows, *the idea of a 'software crisis' entered common use [....] following the 1972 Turing Award lecture [...] [of] Edsger Dijkstra*

> and now we have gigantic computers, programming has become an equally gigantic problem.[...] To put it in nother way: as the power of available machines grew by a factor of more than a thousand, society's ambition to apply these machines grew in proportion, and it was the poor programmer who found his job in this exploded field of tension between ends and means.

Typical problems that started arising were software failure, unreliability and malfunctioning, at least partly due to large software projects becoming too complex to manage. These problems were considered by some the result of the theory lagging behind the demands and expectations of society, a reflection which lead to the development of a new discipline called software engineering. Software engineering at that time, with a slightly more confined meaning than today, was referring to the use of formal methods within programming as a means to attack typical problems of this so-called crisis. At the same time it was also aimed at providing a more scientific status that a part of the community aspired to ascribe to the discipline by developing a solid theoretical methodology. Figures of the calibre of Dijkstra and Hoare defended the programs-as-proofs identity, with criteria of correctness and termination being paramount and to be proven in a logical or mathematical fashion.[21] New techniques were developed to integrate more logical approaches into programming methods.[22] For example, Dijkstra's method of 'structured programming' was developed to deal with, amongst others, problems of correctness;[23] Scott and Strachey developed denotational semantics for programming;[24] and de Bruijn aimed at formally verifying the whole of mathematics by writing AUTOMATH, a language also making use of types to induce the identity of theorems and output of an automated derivation.[25] But while these researches pointed at the primal role of logic in the design and construction of programs, the essential and delicate balance between the theoretical and practical aspects pertaining to computing was becoming pressing. The introduction of, for instance, the typed lambda-calculus into computer science was meant as a faithful modelling of well-specified computations in formally correct expressions. But the former, when intended as calculations actually executed on finite machines, operated by fallible programmers and users in a given social context, exceed the degree of precision of the latter by a much higher level of complexity. This position, counter-balancing the formalist view on correctness and validation with a more practical approach, was soon put forward by part of the computer science community. The reference to the social and multi-layered aspect of computational well-functioning (see e.g. [12]), as well as the practical impossibility to exclude essential aspects of computational malfunctions due to the physical nature of

[21] See for instance [15] and [30]. The story and origin of the extended Curry-Howard isomorphism is still partly unclear and deserves an analysis on its own.

[22] See the paper by Maarten Bullynck in this volume.

[23] See [16].

[24] See [40].

[25] See [11].

the processes involved (see e.g. [19]) were considered.[26]

Despite tensions between those insisting on the role of logic and those less convinced of its applicability have been recurring throughout the history of computer science, the need for formal methods is higher than ever. One major reason, present already in the above quotes by von Neumann and Dijkstra, is that the more ambitious society becomes in applying computation, the lesser control we have over what is happening inside the (networks of) machines. As a result, automatic certification relying heavily on formal methods is becoming increasingly important, especially in the context of cyber-physical and safety-critical computational systems like in avionics and the autonomous automotive industries, i.e. in applications where computations are no longer taken in isolation, but rather as elements in sensitive connection to humans.[27] Hence, it is clear that the relations between formal logic, engineering practices and physical machinery characterize some fundamental issues within computer science and its history: tensions and convergences which one needs to reflect upon to understand the nature of the discipline.

Until recently, however, besides few historical and philosophical contributions,[28] not much attention was devoted to the complexity of this topic. One reason for this is that in order to study the historical and philosophical influence of formal methods in computer science, one should also engage with the technology: to understand in what sense, for instance, Post production systems have played a role in the history of compiler design, or how a formal system like Hoare logic is the basis of systems used today to reason about programs in terms of states of the store and the heap. The need of technical understanding to write a proper history of computer science was recently at the centre of a debate within the community of historians of computing. In the 2014 Kailath lecture at Stanford titled *Let's not dumb down the history of computer science*, Donald Knuth, a very well-known computer scientist, explained his regret for the so-called professionalization of the history of computing which has implied an increasing neglect of technical content for the sake of more socially, institutional, politically and/or industry oriented histories. This talk resulted in lively discussions and finally a short piece [27] in the *Communications of the ACM*. A similar argument holds for the philosophical community as well. Computing and engineering at large have always been a significant source of inspiration for

[26]The complex formulation of notions of formal correctness, reliable design, effective debugging and so on are all still central issues in the academic and industrial development of mainframe and especially software systems. For a philosophical, rather than strictly technical, categorization and definition of the problem of computational errors, see e.g. [21] and [20]. For the argument on the practical impossibility of program correctness, see for example the paper by Selmer Bringsjord in this volume. For the evolution of computational systems in relation to the user, see for example the paper by Graham White in this volume.

[27]By way of example, consider the recent development of a formally verified C compiler, part of the CompCert project (See `http://compcert.inria.fr/motivations.html`): '*[This compiler is not] [f]or non-critical, "everyday" software [where] bugs introduced by the compiler are negligible compared to those already present in the source program [but for] safety-critical or mission-critical software, where human lives, critical infrastructures, or highly-sensitive information are at stake.*'.

[28]See e.g. [33], [2], [32], [9], [45].

philosophical research. Such research has been directed either to the conceptual analysis of important and largely appealing themes, for example incompleteness and complexity; or it has focused on non-technical, often ethically oriented topics. This has attracted some interest from computer science practitioners, who have indulged in technically-aware reflections on their discipline, but a common field recognised by all parties, where philosophers dare to be really technically prepared is still missing.

Precisely to tackle such need for a history and philosophy of computing that also engages with actual computational and programming practices, the DHST Commission for the History and Philosophy of Computing (HaPoC) was founded in 2013.[29] One of the aims of the commission is to organize regular meetings, providing an open platform for historians, philosophers, computer scientists, logicians, programmers, mathematicians (and all other figures involved by the field at large) to discuss across their own disciplinary boundaries and to offer the open environment required to reflect on all facets of computing. The present collection of papers, which resulted from several HaPoC events, aims exactly at this kind of reflection, by bringing to the fore the problem of bridging the gap between formal methods and practices of computing. We approach various apparently distinct issues concerning computability at large, correctness, software design and implementation, program semantics and human-computer interaction, with each contribution being commonly characterised in a double way: first, each author plays with the combination of historical background and philosophical insight we consider essential in exploring a technical and theoretical relevant issue in computing; second, every contribution insists on the relevance of logic and formal methods as the counterpart to the engineering practice, constituting the double face of the discipline. The arrangement of the papers in this volume also reflects a historically aware presentation of facts and topics.

We have already briefly pointed out above how the most common lore, which traces the history of computing back to the role of Turing, is largely a simplification. The logical roots of computer science are to be contextualised in a larger set of research fields and figures, each contributing specific and very crucial results to the field as we know it today. Certainly Turing deserves a prominent position. As historians and history-aware philosophers of computing, however, it is of the greatest relevance to understand how, why and by whom Turing came to be recognised very often as *the* father of the discipline. This is the task that Edgar Daylight is set to approach: with a historical analysis that stretches over results in logic including not only Turing, but recasts of his results by Kleene, Rosenbloom, Markov, he is able to identify a particular group of actors – including Booth, Carr, Gill and Gorn – who were looking for a more theoretical foundation of computation and found it in (recast versions of) the work of Turing.

The two decades across the '60s and the '70s have been often identified as

[29]The website of the commission can be found at `www.hapoc.org`. For a more detailed discussion of the need for more technical content within the history and philosophy of computing, see [14].

a turning point for the development of computing as an academic and scientific discipline: many of the research methods currently at its core were initiated at this time, both at the low-level of machines, the higher-level of programs and the communications between them. The software crisis and the problematic confrontation with implementations solicited the development of new programming paradigms and semantic theories of programming like the denotational, axiomatic and event-based ones, aimed at a mathematical theory independent from particular implementations and allowing, for instance, to prove program correctness or equivalence. The problems associated with connecting the formal approaches with the practice of computing became apparent also in this context. Maarten Bullynck reflects on the case of the computation of a list of primes to discuss stepwise, structured and formally verified programming and considers the limitations imposed by user-machine interaction in implementations for systems such as the MULTICS and the ILLIAC IV. In what could be seen as a parallel exploration of the limitations of logical approaches to the semantics of programs, Felice Cardone reconstructs the historical and conceptual evolution of the principle of continuity from recursion in the 50s, through Scott's denotational semantics at the end of the '60s and Dijkstra's work in the '70s: in this formal analysis, continuity is identified and explained as the principle that qualifies performed computations as finitary, and hence bounded by the time-related constraints of mathematical computation to be executed by machines, a notion that will have large conceptual consequences in modern computing, e.g. for concurrency.

The mentioned debate on the notion and theoretical possibility of formal verification is at the core of the duality between logic and engineering in computer science. In open contrast with practitioners like Dijkstra and Hoare who understood programs as instances of proofs and insisted on the need to prove their correctness by logical means, the highly debated and influential paper [19] claimed the impossibility of such a request, in view of the physical, non-purely theoretical nature of computational objects. The debate on program verification has spanned for decades, and has never been really closed, still generating conflicts of ideas today. In his contribution, Selmer Bringsjord re-opens the debate at a different level, by attacking the very core of Fetzer's argument, namely its logical consistency, claiming it is a self-refuting position on the basis that it is construed on the very same fallibility that the original attributes to computing. Hence, once again, logical correctness and physical implementation of computation (in humans or in machines) are opposed and compared, in what seems to reinforce the dual nature of this field. This relation between the logic of the machine and the logic of the human is at the core of the analysis of the final contribution: Graham White explores – with the help of many historical examples – how the various levels of abstraction from hardware on, are controlled by languages that are meant to accommodate the human user's intention and her understanding of the computation to be performed, and how such relation moves also in the opposite direction, with improvements in hardware and software to force accommodations by the user.

With this collection we hope to strengthen the bridge between the commu-

nity of historians and philosophers of logic with computing. It is essential that both areas better understand and appreciate how computing and the related machinery represent the evolving state of formal logic; and how the latter has been a crucial, although not unique element, in the evolution of the former.

References

[1] Adams, R., 2011. *An early History of recursive Functions and Computability*, Boston, Massachusetts: Docent Press.

[2] Aspray, B., 1990. *John von Neumann and the origins of modern computing*, Massachusettes: MIT Press.

[3] Carnap, R., 1931. 'Die logizistische Grundlegung der Mathematik', *Erkenntnis*, **2**, 91–105.

[4] Chomsky, N., 1959. 'On certain formal properties of Grammars', *Information and Control*, **2**, 137–167.

[5] Church, A., 1933. 'A set of postulates for the foundation of logic (second paper)', *Annals of mathematics*, **34**, 839–864.

[6] Church, A., 1936. 'An unsolvable problem of elementary number theory', *American Journal of Mathematics*, **58**, 345–363.

[7] Church, A., 1936. 'A note on the Entscheidungsproblem', *The Journal of Symbolic Logic*, **1**, 40–41.

[8] Corry, L., 2008. 'Number crunching vs. number theory: computers and FLT, from Kummer to SWAC (1850–1960) and beyond', *Archive for the history of Exact Sciences*, **62**, 393–455.

[9] Davis, M., 2001. *Engines of Logic: Mathematicians and the Origin of the Computer*, New York: W.W. Norton & Co.

[10] Daylight, E., 2014. 'A Turing tale', *Communications of the ACM*, **57**, 36–38.

[11] de Bruijn, 1968. *AUTOMATH, a language for mathematics*, T.H. Report 66-WSK-05, Department of Mathematics, Eindhoven University of Technology.

[12] De Millo, R., Lipton, R. and Perlis, A., 1979. 'Social Processes and Proofs of Theorems and Programs', *Communications of the ACM*, **22**, 271–280.

[13] De Mol, L., Carlé, M., Bullynck, M., 'Haskell before Haskell: an alternative lesson in practical logic of the ENIAC, *Journal of Logic and Computation*, in print, doi: 10.1093/logcom/exs072.

[14] De Mol, L. and Primiero, G., 2014. 'Facing Computing as Technique: Towards a History and Philosophy of Computing', *Philosophy & Technology*, 1–6.

[15] Dijkstra, E.W., 1968. 'A constructive approach to the problem of program correctness', *BIT Numerical Mathematics*, **8**, 174–186.

[16] Dijkstra, E.W., 1972. 'Notes on Structured Programming', in: Dahl, O.j., Dijkstra, E.W. and Hoare, C.A.R., eds., *Structured Programming*, London: Academic Press Ltd., pp. 1–82.

[17] Dijkstra, E. W., 1972. 'The humble programmer', *Communications of the ACM*, **15**, 859–866.

[18] Ensmenger, N. 2010. *The computer boys take over*, Massachusettes: MIT Press.

[19] Fetzer, J., 1988. 'Program Verification: The Very Idea', *Communications of the ACM*, **37**, 1048–1063.

[20] Floridi, L., Fresco, N. and Primiero, G., 2015. 'On Malfunctioning Software', *Synthese*, **192**, 1199–1220.

[21] Fresco, N. and Primiero, G., 2013. 'Miscomputation', *Philosophy and Technology*, **26**, 253–272.

[22] Gandy, R., 1988. 'The confluence of ideas in 1936', in: Herken, R., ed., *The Universal Turing machine*, Oxford: Oxford University Press, pp. 55–111.

[23] Grier, D. A. 2007. *When computers were human* Princeton: Princeton University Press.

[24] Haigh, T., 2010. 'Dijkstra's crisis: The end of Algol and the beginning of software engineering: 1968-72', in: *Workshop on the history of software, European styles*, Lorentz Center, University of Leiden.

[25] Haigh, T., 2014. 'Actually, Turing did not invent the computer', *Communications of the ACM*, **57**, 36–41.

[26] Haigh, T., Priestley, M. and Rope, C., 2014. 'Reconsidering the stored-program concept', *IEEE Annals of the history of computing*, **36**, 4–17.

[27] Haigh, T., 2015. 'The tears of Donald Knuth', *Communications of the ACM*, **58**, 40–44.

[28] Hartmanis, J. and Lin, H., 1992. 'What is computer science and engineering', in: Hartmanis, J. and Lin, H., eds., *Computing the Future: A Broader Agenda for Computer Science and Engineering*, Washington, D.C.: National Academy Press, pp. 163–216.

[29] Heyting, A., 1931. 'Die intuitionistische Grundlegung der Mathematik', *Erkenntnis* **2**, 106–115.

[30] Hoare, T., 1969. 'An axiomatic basis for computer programming', *Communications of the ACM*, **12**, 576–580.

[31] Knuth, D., 1974. 'Computer Science and its Relation to Mathematics', *American Mathematical Monthly*, **81**, 323–343.

[32] MacKenzie, D. 2001. *Mechanizing Proof – Computing, Risk and Trust*, Massachusettes: MIT Press.

[33] Mahoney, M.S. 1988. 'The history of computing in the history of technology', *IEEE Annals of the History of Computing*, **10**, 113–125.

[34] McCarthy, J., 1960. 'Recursive functions of symbolic expressions and their computation by machine, Part I', *Communications of the ACM*, **3**, 184–195.

[35] Newell, A., Perlis, A. J., and Simon, H. A., 1967. 'Computer science', *Science*, **157**, 1373–1374.

[36] Nofre, D., Priestley, M. and Alberts, G., 2014. 'When Technology Became Language: The Origins of the Linguistic Conception of Computer Programming, 19501960', *Technology and Culture*, **55**, 40–75.

[37] Polachek, H. 1997. 'Before the ENIAC', *IEEE Annals of the History of Computing*, **19**, 25–30.

[38] Post, E. L., 1965. 'Absolutely unsolvable problems and relatively undecidable propositions - Account of an anticipation', in: Davis, M., ed., *The Undecidable. Basic papers on undecidable propositions, unsolvable problems and computable functions*, New York: Raven Press, pp. 340–433.

[39] Post, E. P., 1936. 'Finite Combinatory Processes - Formulation 1', *The Journal of Symbolic Logic*, **1**, 289–291.

[40] Scott, S. and Strachey, C., 1971. *Toward a mathematical semantics for computer languages*, Oxford Programming Research Group Technical Monograph. PRG-6.

[41] Shannon, C., 1938, A Symbolic Analysis of Relay and Switching Circuits, *Transactions of the American Institute of Electrical Engineers*, **57**, 713–723.

[42] Shestakov, V.I., 1941. Algebra of Two Poles Schemata (Algebra of A-Schemata), *Journal of Technical Physics*, **11**, 532–549 (in Russian).

[43] Tedre, M. 2015. *The science of Computing. Shaping a discipline*, Boca Rato: CRC Press.

[44] Turing, A. M., 1937. 'On computable numbers with an application to the Entscheidungsproblem', *Proceedings of the London Mathematical Society*, **42**, 230–265.

[45] Turner, R. 2014, The Philosophy of Computer Science, *The Stanford Encyclopedia of Philosophy* (Winter 2014 Edition), Edward N. Zalta (ed.), urlhttp://plato.stanford.edu/archives/win2014/entries/computer-science/.

[46] Ulam, S. 1980. 'Von Neumann: The Interaction of Mathematics and Computing', in: J. Howlett, N. Metropolis and G.-C. Rota, eds., *A History of Computing in the Twentieth Century. Proceeding of the International Research Conference on the History of Computing, Los Alamos, 1976*, New York: Academia Press, pp. 93–99.

[47] van Heijenoort, J., 1967. *From Frege to Gödel: A source book in Mathematical Logic 1879–1931*, Cambridge, MA: Harvard University Press.

[48] von Neumann, J., Die formalistische Grundlegung der Mathematik. *Erkenntnis* 2, pp. 116–121, 1931.

[49] von Neumann, J. 1948. 'Electronic methods of computation', *Bulletin of the American Academy of Arts and Sciences*, **1**, 2–4.