The Standard Model for Programming Languages: The Birth of a Mathematical Theory of Computation

Simone Martini
Department of Computer Science and Engineering, University of Bologna, Italy
INRIA, Sophia-Antipolis, Valbonne, France
`http://www.cs.unibo.it/~martini`
simone.martini@unibo.it

Abstract

Despite the insight of some of the pioneers (Turing, von Neumann, Curry, Böhm), programming the early computers was a matter of fiddling with small architecture-dependent details. Only in the sixties some form of "mathematical program development" will be in the agenda of some of the most influential players of that time. A "Mathematical Theory of Computation" is the name chosen by John McCarthy for his approach, which uses a class of recursively computable functions as an (extensional) model of a class of programs. It is the beginning of that grand endeavour to present programming as a mathematical activity, and reasoning on programs as a form of mathematical logic. An important part of this process is the *standard model* of programming languages – the informal assumption that the meaning of programs should be understood on an abstract machine with unbounded resources, and with true arithmetic. We present some crucial moments of this story, concluding with the emergence, in the seventies, of the need of more "intensional" semantics, like the sequential algorithms on concrete data structures.
The paper is a small step of a larger project – reflecting and tracing the interaction between mathematical logic and programming (languages), identifying some of the driving forces of this process.

to Maurizio Gabbrielli, on his 60th birthday

2012 ACM Subject Classification Social and professional topics → History of programming languages; Software and its engineering → General programming languages

Keywords and phrases Semantics of programming languages, history of programming languages, mathematical theory of computation

Digital Object Identifier 10.4230/OASIcs.Gabbrielli.2020.8

Funding *Simone Martini*: Research partially conducted while on sabbatical leave at the Collegium – Lyon Institute for Advanced Studies, 2018–2019. Partial support from French ANR project PROGRAMme.

Acknowledgements I am happy to thank Edgar Daylight for mentioning me Strachey's letter to the Computer Journal, and for the many critical reactions to the main thesis of this paper.

1 Introduction

Statements like "the Java programming language is Turing-complete", or "the C program '`int x=0; while 1 {x++;}`' is divergent", or even "the halting problem is undecidable for C programs", are common in most textbooks and in the practice of many computer scientists, despite the fact that they are *false*, for any actual implementation. The finiteness of any such implementation implies that the halting problem is decidable, being the system only a finite-state automaton; that the increment `x++` will produce after a finite time an overflow which will be caught by some hardware interrupt; and that any Java implementation cannot compute

functions requiring more resources than those of the universe. Of course, the statements become true when they are applied to a *model* of C or Java programs allowing for true unbounded arithmetic and unbounded (finite) resources. We propose to call *standard model* of programming languages this (informal) assumption, where programs are interpreted on an abstract machine with unbounded resources, and with true arithmetic. It is so "standard" that its use is almost never esplicitated in the literature (scientific, or in textbooks) – on the contrary, when *another* model is implied (for instance, one with finite arithmetic), that use is instead acknowledged and remarked.

We are so used to the fact that a finite computer program could be regarded as the infinite function it computes, that we tend to forget that this "double view" was not born together with the general purpose electronic computer – it is rather the result of a deliberate research agenda of the end of fifties and the early sixties of the last century, responding both to internal and external forces of the discipline that was going to be called, also thanks to that agenda, Computer *Science*. It is the beginning of that grand endeavour to present programming as a mathematical activity, and reasoning on programs as a form of mathematical logic. The paper will present some of the moments of this process, where the components of a mathematical theory of computations are introduced as the technical, formal elements of the informal standard model. Before delving into these developments, however, we cannot forget the founding fathers of the discipline, who already realised this double nature of computer programs, but whose insight did not pass, at that time, into the practice of computing.

2 The Pioneers

The two years from 1947 to 1949 are of particular interest for our story. First, it is when Herman Goldstine and John von Neumann wrote the second part [20] of a series of reports on the "mathematical and logical aspects of an electronic computing instrument". Its aim is to give an account of the "methods of coding and of the philosophy which governs it". A major methodological tool is the use of *flow diagrams* for expressing the dynamics of a computation. Flow diagrams are made of four distinct classes of boxes: operation, alternative, substitution, and assertion boxes. While boxes of the first two kinds contain operations that the machine will ultimately perform, the contents of an *assertion* box "are one or more relations." A box of this kind "never requires that any specific calculation be made, it indicates only that certain relations are automatically fulfilled whenever" the control reaches that point[1]. Assertion boxes are "logical expressions" that remain invariant during the computation – their only reason for being present in a diagram is that they are needed (or useful) in establishing that the diagram correctly renders "the numerical procedure by which the planner has decided to solve the problem", which is expressed in the "language of mathematics"[2]. For this reason, coding "has to be viewed as a logical problem and one that represents a new branch of formal logics." While the word "logic" (or "logical") does not necessarily refer to mathematical (formal, or symbolic) logic in the literature of that period[3], the reference to a "new branch

[1] Also substitution boxes do not specify any computation; they represent a change in *notation*, more specifically in the relation between the internal and the external notation, a significantly different perspective from the modern use of flow charts, see [29].

[2] We will come back to this duality between the *specification* and the *implementation* (in the today's terminology, of course) at the end of Section 4.

[3] "Logical" is usually opposed to "physical", or "electronical", like in "logical design of digital circuits"; or in the very title of the series in which [20] appears: "mathematical and logical aspects of an electronic computing instrument."

of formal logics" is explicit. Assertions bring mathematical logic into the very notation for writing programs; moreover, references to other notions of formal languages are present in other places of [20], like the distinction between free and bound variables. "Formal logic" is for Goldstine and von Neumann at the core of programming, as the discipline where assertions may be written and proven.

An analogous use of assertions for "checking [the correctness of] a large routine" will be proposed by Alan Turing two years later. In [54][4] he lucidly describes a proof of correctness as consisting of three different steps. First, "the programmer should make assertions about the various states that the machine can reach." Then, "the checker [i.e., the one doing the proof] has to verify that [these assertions] agree with the claims that are made for the routine as a whole." "Finally the checker has to verify that the process comes to an end." Observe the clarity by which full correctness (in today's terms) is spelled out, the distinction between specification ("claims that are made for the routine as a whole") and implementation, and the interplay between these two, where assertions play their role.

Turing brings us back to year 1947, when he also writes a paper with some remarks on mathematical logic in computer programming. In the "Lecture on Automatic Computing Engine" [53] he has already clear that programming a computing machine could be done, in principle, with languages much more sophisticated than the machine instructions that were available at the time. Indeed, "in principle one should be able to communicate [with these machines] in any symbolic logic, provided that the machine were given instruction tables which would allow it to interpret that logical system." Like Goldstine and von Neumann, also Turing sees a bright future for mathematical logic (to be understood, in this case, as the discipline of artificial languages): "there will be much more practical scope for logical systems than there has been in the past."

Haskell B. Curry in those same years uses his logical apparatus in order to program the new computing machines. Discussing an inverse interpolation routine and its correctness, in [10] he introduces a notion of *type* for memory words: those containing instructions (*orders*), and those containing data (*quantities*). Starting from this, as reconstructed by [15], he builds a surprising, non-trivial mathematical theory of programs, containing theorems analogous to the "well-typed expressions do not go wrong"[5] of [38], and he uses them to define classes of program transformations and compositions which are "safe" for the intended operational semantics. The presence of mathematical logic is so explicit that Curry's reports will get a review on the Journal of Symbolic Logic[6].

A lesser-known contribution is the early work of Corrado Böhm, a few years later. His thesis at ETH Zurich[7] esplicitely connects the new computing machines to the mathematical, abstract analysis of Turing, up to the claim that "les calculatrices les plus évoluées sont universelles, au sens spécifié par M. Turing." Under this assumption, Böhm may assume that all the general purpose, stored-program computers "sont, au point de vue logico-mathématique, équivalentes entre elles," so that he may choose a specific type of computer (a three-address machine) without any loss of generality. These remarks, and the explicit

[4] See also [39], for a reprint of the original paper, and a commentary.

[5] "Suppose we have an initial type determination for the initial program", that is an assignment of types to words which assign type "order" to any instruction, and type "quantity" to any data, then the memory "word at the control location is always [*scil.*, at any time during execution] an order" [10], number 26.

[6] By G.W. Patterson, JSL 22(01), 1957, 102-103.

[7] Written under the direction of E. Stiefel and P. Bernays, submitted in 1952, and published in 1954 [6]; see also Knuth's [29].

connection to Turing's theory, that to most of us seem obvious, were not (yet) part of the standard background of the people working in the field of automatic computing machines. Of course, they were known to some of the other giants (von Neumann and his peers *in primis*; Böhm's advisor Paul Bernays is the probable source who mentioned Turing's work to him), but it will be only much later that Turing will become the iconic figure of father for computer science [11][8]. Another striking observation in Böhm's thesis is that a program is seen under a double interpretation: as a description of the behaviour of a computer, and as the description of a "méthode numérique de calcul." Without forcing the interpretation, we may read this as one of the first explicit references to the duality between an operational description of the behaviour of an (abstract) machine, and the numerical function that results from that sequence of operations.

The far-sight of these pioneers should not obfuscate the fact that the role of logic in the early days of the digital computing machines (both their design and their programming) was modest, if not absent (e.g., [11, 14]). Programming those machines was more a technological affair, strictly coupled to the technology of the different computers. Despite the genial recognition by Turing that "any symbolic logic" could be used as a programming language, it is only during the fifties, and with graduality, that programming started to be perceived as a "linguistic activity" [42]. As for the correctness of programs, Knuth's recollection [28] is that at the end of the fifties "the accepted methodology for program construction was [...]: People would write code and make test runs, then find bugs and make patches, then find more bugs and make more patches, and so on. We never realized that there might be a way to construct a rigorous proof of validity." "The early treatises of Goldstine and von Neumann," and of Turing, and Curry, "which provided a glimpse of mathematical program development, had long been forgotten." In summary, while there are important relations with other parts of mathematics, like numerical analysis (e.g., Jim Wilkinson's backward error analysis), or automata theory (from von Neumann, to Kleene's regular events, to Rabin and Scott), or cybernetics, at the macro scale little happens on the explicit border between logic and the new field of computing, which at that same time struggled to be recognised as an autonomous scientific discipline[9]. In this process, the availability of computer-independent ("universal," in the terminology of the time) programming languages allowed the expression of *algorithms*

[8] Describing the relations between Turing's work – and especially the notion of Turing machine, – the modern digital computer, and computer programming, is well outside the scope of this paper. For some of the relations between Turing and von Neumann, see Stanley Frankel's letter to Brian Randell, quoted in [45]. An argument on the equivalence of Turing machines and McCulloch and Pitts's neuron nets "supplied with an infinite blank tape," can be found in von Neumann's [58]. A balanced review of the actual impact of Turing on computer science may found in Section 5 of Liesbeth De Mol's entry on Turing machines for the Stanford Encyclopedia of Philosophy [13], from which we quote the following. "Recent historical research shows also that one should treat the impact of Turing machines with great care and that one should be careful in retrofitting the past into the present." Only "in the 1950s then the (universal) Turing machine starts to become an accepted model in relation to actual computers and is used as a tool to reflect on the limits and potentials of general-purpose computers by both engineers, mathematicians and logicians." See also [22, 11].

[9] A struggle that was going to be long. The first *Computer Science* department of the US was established in 1962 at Purdue University; Samuel D. Conte, first Head of that department, will recall in a 1999 Computerworld magazine interview: "Most scientists thought that using a computer was simply programming – that it didn't involve any deep scientific thought and that anyone could learn to program. So why have a degree? They thought computers were vocational vs. scientific in nature" (quoted in Conte's obituary at Purdue University, 2002). Next computer science departments to be established would be those at the University of North Carolina at Chapel Hill, in 1964, and at Stanford in 1965. Still in 1967, Perlis, Newell and Simon (all of them will receive the Turing award; Simon will also be a Nobel laureate in Economics) feel the need of a letter to Science [41] to argue "why there is such a thing like computer science".

in a machine neutral way, thus making algorithms and their properties amenable to a formal study. Programming languages themselves were treated as an object of study, starting from the formal definition of Algol's syntax [1, 2]. For an entrance ticket to "science", however, a complete "theory" was needed, encompassing also semantics of programming languages and their use in program development.

3 Towards a General theory

The construction of a "mathematical theory of computation," (or a "mathematical science of computation") is the explicit goal of John McCarthy, starting from 1961 (when he was still at MIT) and covering his first period at Stanford, where he moved in 1963. In [35][10] he sketches an ambitious plan for a general theory, based on mathematical logic, that could serve as a foundation for computation, in such a way that "it is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last." After having dismissed numerical analysis (for being too narrow for a general theory), computability theory (for focussing on undecidability, instead of positive results, and for being uninterested on properties of algorithms), and finite automata theory (for being of no use in the treatment of computers, because they have too many states), it proceeds to list the "practical" (*sic*) results of the theory. The first is "to develop a universal programming language": Algol is a good step in the right direction, but it has several "weaknesses", among which the impossibility to describe different kinds of data[11]. Second, "to define a theory of the equivalence of computation processes," to be used to define and study equality preserving transformation. Let us esplicitate what McCarthy leaves unsaid: once we have an accepted *model* for the behaviour of a program, we may study under which transformations of (the syntactical presentation of) the program the behaviour remains invariant (in the model). A third goal goes in the same direction: "To represent computers as well as computations in a formalism that permits a treatment of the relation between a computation and the computer that carries out the computation." The models for both the program and its executing agent should be expressed in the same conceptual framework, in the same theory, so that one may express relations among the two, like the fact that (the model of) the computer that carries out the computation is "sound" with respect to (the model of) the program. Contrasting the lack of interest of recursive function theory for the positive results and for the properties of algorithms, a fourth goal is "to give a quantitative theory of computation [...] analogous to Shannon's measure of information." The paper does not elaborate further on this point; we may probably ascribe to this goal the development of computational complexity. Finally, a last purpose of a general theory would be "to represent algorithms by symbolic expressions in such a way that significant changes in the behavior represented by the algorithms are represented by simple changes in the symbolic expressions." Once again the point will not be taken up again in the rest of the paper, besides explaining that it is relevant for programs that "learn from experience". For our purposes, it suffices to stress the reference to the availability of (a model of) the behaviour of a program and to the interplay between the syntactic representation of the algorithm and that behaviour.

[10] The preliminary version is from 1961; the final 1963 version contains a new section on the "relations between mathematical logic and computation."

[11] Note, en passant, that the word "type" is still not used in this context, for a collection of homogeneous values; see [32].

The technical contents of the paper, of course, cannot match the *grandeur* of these goals. The paper introduces a theory of higher order, partial computable functions on arbitrary domains, defined by composition, conditional statements, and general recursion. Moreover, a uniform way to define new data spaces is presented – data space constructors are the Cartesian product, the disjoint union, and the power set, each of them equipped with its canonical maps, which are used to define functions on the new spaces from functions on the base spaces. Base data spaces could be taken as frugal as the single "null set", since natural numbers could be defined from it. The goal is to provide general, abstract mechanisms, instead of choosing an arbitrary new palette of primitive types.

The picture sketched in [35] is further developed in [34] (to "be considered together with my earlier paper"), which contains a first attempt for an epistemology for (the still unborn) computer science[12], and introduces again, after the pioneers, the problem of program correctness: "Instead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer program. For this to be possible, formal systems are required in which it is easy to write proofs." Of course, one needs first a semantics for the behaviour of a program. This is an important contribution of the paper: programs expressed as sequences of assignments and conditional go to's are given meaning as recursive functions acting on the set of current values of the variables (the "state vector"). Each statement of the program corresponds to such a recursive function, the meaning of a program being obtained by a suitable composition of these functions, in a *compositional* approach which will be the cornerstone of formal program semantics. Moreover, the proof technique of "recursion induction," already introduced in [35], is extended, so that it could be applied directly to programs without first transforming them to recursive functions. This last contribution is particularly relevant for our story: the program is implicitly understood as a representative for its meaning, so that one may argue on the program (a finite, textual object) for obtaining results on its model (an infinite function over the set of possible data).

McCarthy will not develop the formal semantics introduced in his two papers, and the creation of a mathematical semantics for programs is usually credited to Robert Floyd [18][13] and Tony Hoare [23]. Both are lucid on the need of a machine independent meaning of programs, and the need of formal theories for reasoning on programs. For Hoare, "computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning." *Sola scripta* are normative for the behaviour of a program: a model for the semantics is implicit and implied.

In those same years, the construction of an *explicit* semantic model of a programming language is the goal of Christopher Strachey, after ideas of Peter Landin [31]. Starting with [50], presented at a working conference in 1964, and especially with the (then unpublished) notes of a course at Copenhagen in 1967 [51], Strachey presents a full-blown account[14] of a mathematical semantics of a programming language, introducing the notions of "abstract store" and "environment," for modelling assignments and side-effects. Commands are interpreted

[12] "What are the entities with which the science of computation deals? What kinds of facts about these entities would we like to derive? What are the basic assumptions from which we should start?"

[13] "An adequate basis for formal definitions of the meanings of programs [...] in such a way that a rigorous standard is established for proofs about computer programs." "Based on ideas of Perlis and Gorn." "That semantics of a programming language may be defined independently of all processors [...] appear[s] to be new, although McCarthy has done similar work for programming languages based on evaluation of recursive functions."

[14] See, for instance, Figure 1 of [51], page 17 of the reprinted version.

as functions from stores to stores. The basic semantical domains for values are not discussed, but in a later monograph Strachey will esplicitely refer to "abstract mathematical objects, such as integers," in such a way that "when we write a statement such as `x := Sin(y + 3)` in ALGOL 60, what we have in mind is the mathematical functions sine and addition. It is true that our machines can only provide an approximation to these functions but the discrepancies are generally small and we usually start by ignoring them. It is only after we have devised a program which would be correct if the functions used were the exact mathematical ones that we start investigating the errors caused by the finite nature of our computer."

We cannot treat in this paper how mathematical logic and programming languages interacted from the end of sixties – a process much less linear than we may think from today's perspective. Even the relations between the notion of "type" in the two fields are not as straightforward as they may seem [32, 33]. An important chapter of that story would be that of *logic programming languages* (starting from Colmerauer in 1970 and Kowalski in 1974), where logic enters as a first-class actor (see [36]).

4 The Standard Model

The previous section sketched the gradual proposal of (several) semantics for programming languages that could abstract from specific processors and their limitations. While general semantical theories of programming languages had limited impact outside the research communities, the natural approach to a program as the description of a computation happening on an abstract (and largely unspecified) computational device had a major impact. True arithmetic and (in principle) unbounded resources is all that is required of such an abstract processor, that in the Introduction we identified as the *standard model*: the naturalness of assuming these simple hypotheses, gave the standard model the momentum it needed to establish itself as a permanent fixture in programming language theory *and* education[15]. Several reasons cooperate for the establishment of such standard model. One is certainly the need to validate computing as a science – -a mathematical theory is always the entrance ticket to science. Several successes of the use of mathematics into computing were already present at the end of the sixties, like the theory of deterministic parsing (LL and LR), the application of formal language theory, complexity theory, and of course numerical analysis. Despite the large body of results (and relevant problems) in these areas, it is mathematical logic that, as we have seen, dominate the field of programming languages, up to the view (or dream) of McCarthy that logic will be for computation what analysis has been for physics. But a theory of program correctness (which, as we have seen, is deeply connected to the emergence of the standard model) is needed also by other reasons, internal to the

[15] This is not to say that other perspectives were absent. In a 1965 letter to the Computer Journal [49] Strachey proves the undecidability of the halting problem for his Algol-like language CPL, assuming (without saying it!) the standard model. Among the reactions to the letter, the one by W.D. Maurer (27 August 1965) clearly doesn't share the "common ground": Strachey's "letter was of particular interest to me because I had, several months ago, proved that it is indeed possible to write such a program," evidently by assuming a finite automaton as a processor. We find different views also among the founders of the mathematical theory of programs. In the same year of Strachey's [52], Edsger W. Dijkstra writes: "We are considering finite computations only; therefore we may restrict ourselves to computational processes taking place in a finite state machine – although the possible number of states may be very, very large – and take the point of view that the net effect of the computation can be described by the transition from initial to final state." "One often encounters the tacit assumption that [...] the inclusion of infinite computations leads to the most appropriate model." "However, we know that the inclusion of the infinite computation is not a logically painless affair." "It seems more effective to restrict oneself to finite computations." [16].

discipline, and probably more important. As an applied discipline – at the end, programs will be used in the real world by real people – computing needs a way to ensure that what it delivers satisfies the requirements on its use. McCarthy's remark on the relations between physics and mathematics seems to suggest that the model here is structural engineering, where mathematical physics laws and empirical knowledge are used together to understand, predict, and calculate the stability, strength and rigidity of structures for buildings. The mathematical theory of computation and its standard model are instrumental for reaching an analogous standard of rigor, so that "when the correctness of a program, its compiler, and the hardware of the computer have all been established with mathematical certainty, it will be possible to place great reliance on the results of the program, and predict their properties with a confidence limited only by the reliability of the electronics," as Hoare writes in his landmark paper on program correctness [23][16]. In this, computing has a big advantage over structural engineering – only the very last layer of the deployment of a system ("the reliability of the electronics") is left out of reach of the formal approach. Since all levels in the hierarchy of a computing system are of the same, *abstract* nature, all levels could be subject (at least conceptually) to the same analysis. When a formally proved chain of compilers will be availabile, a proof that a model of the higher level program satisfies a certain condition, transfers automatically to a proof that a model of the low level program satisfies some other condition, also obtained automatically from the higher level one. No concrete, no iron, no workmanship is involved.

In this context, the standard model is to programming languages what movement without friction is to mechanics. And this is why it is so important. It is not that it implies Turing-completeness that matters, but its simplicity and the fact that, indeed, one may do some mathematics with it. The analogy with the Galilean effort for physics is illuminating – no bodies of different masses reach the ground at the same time when they actually fall from the leaning tower of Pisa, like there is no true unbounded arithmetic inside any laptop; or there is no true isochronous pendulum in nature. Yet, you do not understand a single bit of mechanics if you don't abstract away friction, and don't approximate to small oscillations.

The standard model comes with three features, deeply connected among them. First, *compositionality* – the meaning of a complex construct is obtained from the meaning of its constituents, by composing them in a way that only depends on the construct under consideration. Second, *extensionality* – two constructs with the same input-output behaviour (on their intended domains) have also the same meaning. Finally, *referential transparency* – "if we wish to find the value of an expression which contains a sub-expression, the only thing we need to know about the sub-expression is its value. Any other features of the sub-expression, such as its internal structure [...] are irrelevant to the value of the main expression" [51][17].

Other subtle ingredients are present, but mostly hidden, the most important one being *continuity*, which is the abstract characterization of the *finitary character* of the operations which occur during a computation[18]. In recursive function theory, continuity was implicit in Kleene's first recursion theorem [27], and then exploited in a beautiful set of results, whose apex are the Rice-Shapiro and Myhill-Shepherdson theorems (see [46]), of the second half

[16] Verification of a concrete system, of course, is not the same as verifying its (mathematical) model. A full literature exists on the limitations of formal verification, e.g. [21, 17, 12, 56].

[17] For this use of the expression "referential transparency", Strachey quotes Quine [44] (it is in §30), who in turn refers to Whitehead and Russell's *Principia Mathematica*.

[18] See [8] for a historical reconstruction of the relevance of continuity in early programming language semantics.

of of the fifties. Programming language semantics has a problem analogous to those solved by Kleene's recursion theorem – how to give meaning to (multiple) recursive definitions. In his PhD thesis at MIT, Morris [40] uses the fixed-point combinator of the λ-calculus to link recursion in programming languages to recursive function theory. He shows that the fixed-point operator, applied to a recursive definition, gives the *least* solution with respect to a certain operational order. It was then Dana Scott to put continuity under spotlight [47], and to make it one of the cornerstone of the denotational semantics approach (also to solve those recursive equations *between domains* which are needed in Strachey's approach.)

One of the most important characteristics of a good semantic definition is that it allows for multiple concrete realisations ("implementations") – it must not anticipate those choices that should only be made when the language is implemented. One possible approach to the definition of a language, in fact, could be to define the meaning through a particular interpreter: "This used to be quite common: languages would be "as defined by" some particular compiler" [48]. In this way, however, all the details of that "particular compiler" are needed in order to understand a program. Even more important, to what level of detail is this canonical implementation normative? Is the computation time of a program part of its definition? Is the reporting of errors? The difference between definition and implementation is crucial in the literature we have cited in the previous section. The proposed models are a possible result of the difficult quest for the happy medium between exactness and flexibility, in such a way as to remove ambiguity, still leaving room for implementation (see [19]).

Finally, the explicit availability of models (and especially of the standard model) allows for a clear separation between specification (normative, expressed in the explicit or implicit model), and implementation (that it be abstract or concrete has little importance in this context). Programs are not only abstract mathematical objects living in the theory of computation; nor are only textual, concrete objects embodied in a processor, and thus living in the physical world. The specification defines their *function*[19], while implementation realises that function – it is only in the interplay among these two aspects that programs get their ontology as technical artifacts [55].

5 More Intensional Models

What we have called the standard model is in reality a plurality of abstractions, depending on the language which is modelled. They all share the fact that the numerical functions on the integers are the true arithmetical ones, and that computation happens on an (abstract) processor with unlimited resources (in storage, and time). Coping with real languages required, since the beginning, the introduction of several complications, for instance to deal with side effects (which needed environments and stores), or unrestricted jumps (which required continuations to be used). Still, much effort was in ensuring that the "internal structure" of a program didn't influence its meaning: two different algorithms for the same functions (with the same side-effects, if any), should give rise to the same semantics. This is, after all, what extensionality is about. But this is also an important simplification, or abstraction, that at some point one needs to overcome – real movement happens with friction. Moreover, the fact that the domains involved in the semantics should have "the usual mathematical properties" is something that called for a subtler investigation.

One of the first questions to be tackled, was how to characterise (express, study) *sequentiality*, an important, intensional aspect of certain computations. For this, some notion of "event" seemed essential, to be the elementary building block to serialize. In one of

[19] See, for instance, [30, 57]

the first papers attempting to model parallel programming, Gilles Kahn [24] described a network of sequential processes, communicating through unbounded queues; together with Dave MacQueen, he constructed an operational model for such processes, which, in response to a request, consume data to produce output [25]. Is there a reasonable mathematical (denotational) model for such processes?

Almost at the same time, Gérard Berry studied bottom-up computations, where recursion is understood as a production of "events" [3]; he soon discovered [4] that computation in λ-calculus is intrinsically sequential[20]. Are there models built only with sequential functions, thus "closer" to the operational behaviour[21]?

The answer to these questions came in a series of contributions by Kahn, Berry, Plotkin, and Pierre-Louis Curien (in several configurations as co-authors). The outcome were the notions of *concrete data structure*, *concrete domain*, and *sequential algorithms*. Sequential algorithms [5] are (Kahn-Plotkin [26]) sequential functions, equipped with a strategy for their computation, expressed in a demand-driven way, as in [25][22]. The model of sequential algorithms is "intensional": there are programs with the same input-output behaviour which are separated in the model. It is not a surprise that a similar notion of intensionality is found in Glynn Winskel's *event structures* [59], developed almost at the same time than sequential algorithms, and which may be seen as a generalization of concrete domains (which already contains a notion of incompatibility between elements)[23].

From there, semantics of programming languages strived to cope *also* with intensional phenomena, trying, at the same time, to not abandon the power and simplicity of extensional reasoning.

But telling that story would require another paper. Which I shall write for Maurizio's seventieth birthday.

References

1 John W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In *Proceedings Int. Conf. on Information Processing, UNESCO*, pages 125–132, 1959.

2 John W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language ALGOL 60. *Commun. ACM*, 3(5):299–314, 1960.

[20] The function "parallel or" (or "parallel if"):

$$por(x,y) = \begin{cases} tt & \text{if } x = tt \text{ or } y = tt \\ ff & \text{if } x = y = ff \\ \bot & \text{otherwise} \end{cases}$$

is not λ-definable, although it is clearly computable: simultaneously evaluate x and y (e.g., by dovetailing) until one of the two terminates. The parallel or is also a continuous function present in the Scott models for the λ-calculus, and hence it is a "spurious" element of these models, being undefinable by syntactic means. Gordon Plotkin re-discovered the same result for Milner's PCF [43].

[21] And are there models where any element is definable, thus excluding the parallel or? Definability, as it was soon discovered, is related (indeed, for PCF it is equivalent) to full-abstraction [37] (which we do not treat here; see [9].)

[22] The detailed story of the discovery of sequential algorithms is told by Stephen Brookes [7] in the introduction to the journal version (1993) of the technical report (1978) by Kahn and Plotkin [26] on concrete domains and sequential functions.

[23] See Cardone [8] for the relevance of the notion of continuity in this context, and for some of the relations of event structures to Scott's theory and to Carl Adam Petri's analysis of concurrency.

3 Gérard Berry. Séquentialité de l'évaluation formelle des lambda-expressions. In B. Robinet, editor, *Program Transformations*, 3eme Colloque International sur la Programmation, pages 67–80, Paris, 1978. Dunod.

4 Gérard Berry. Modèles complètement adéquats et stables des lambda-calculs typés. Thèse de Doctorat d'État, Université Paris VII, 1979.

5 Gérard Berry and Pierre-Louis Curien. Sequential algorithms on concrete data structures. *Theor. Comput. Sci.*, 20:265–321, 1982.

6 Corrado Böhm. Calculatrices digitales. Du déchiffrage des formules logico-mathématiques par la machine même dans la conception du programme. *Annali di matematica pura e applicata*, IV-37(1):1–51, 1954.

7 Stephen D. Brookes. Historical introduction to "Concrete Domains" by G. Kahn and Gordon D. Plotkin. *Theor. Comput. Sci.*, 121(1&2):179–186, 1993.

8 Felice Cardone. Continuity in semantic theories of programming. *History and Philosophy of Logic*, 26(3):242–261, 2015.

9 Felice Cardone. Games, full abstraction and full completeness. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2017 edition, 2017.

10 Haskell B. Curry. On the composition of programs for automatic computing. Technical Report Memorandum 10337, Naval Ordnance Laboratory, 1949.

11 Edgar Daylight. Towards a historical notion of 'Turing — the father of computer science'. *History and Philosophy of Logic*, 36(3):205–228, 2015.

12 Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, 1979.

13 Liesbeth De Mol. Turing Machines. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2019 edition, 2019.

14 Liesbeth De Mol, Maarten Bullynck, and Edgar G. Daylight. Less is more in the fifties. Encounters between logical minimalism and computer design during the 1950s. *IEEE Annals of the History of Computing*, 2018.

15 Liesbeth De Mol, Martin Carlé, and Maarten Bullyinck. Haskell before Haskell: an alternative lesson in practical logics of the ENIAC. *Journal of Logic and Computation*, 25(4):1011–1046, 2015.

16 Edsger W. Dijkstra. A simple axiomatic basis for programming language constructs. *Indagationes Mathematicae*, 36:1–15, 1974.

17 James H. Fetzer. Program verification: The very idea. *Commun. ACM*, 31(9):1048–1063, 1988.

18 Robert W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19:19–32, 1967.

19 Maurizio Gabbrielli and Simone Martini. *Programming Languages: Principles and Paradigms*. Undergraduate Topics in Computer Science. Springer, 2010.

20 Hermann Goldstine and John von Neumann. Planning and coding of problems for an electronic computing instrument. Technical Report Part II, Volume 1-3, Institute of Advanced Studies, 1947.

21 Solomon W. Golomb. Mathematical models: Uses and limitations. *IEEE Transactions on Reliability*, R-20(3):130–131, 1971.

22 Thomas Haigh. Actually, turing did not invent the computer. *CACM*, 57(1):36–41, 2014.

23 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

24 Gilles Kahn. The semantics of a simple language for parallel processing. In Jack L. Rosenfeld, editor, *Information Processing 74, Proceedings of IFIP Congress*, pages 471–475. North-Holland, 1974.

25 Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *Information Processing 77, Proceedings of IFIP Congress*, pages 993–998. North Holland, 1977.

26 Gilles Kahn and Gordon D. Plotkin. Concrete domains. *Theor. Comput. Sci.*, 121(1&2):187–277, 1993. Reprint of the IRIA-LABORIA rapport 336 (1978).

27 Stephen C. Kleene. *Introduction to Metamathematics*. Van Nostrand, New York, 1959.

28 Donald E. Knuth. Robert W Floyd, *in memoriam*. *SIGACT News*, 34(4):3–13, December 2003.

29 Donald E. Knuth and Luis T. Pardo. The early development of programming languages. In N. Metropolis, J. Howlett, and Gian-Carlo Rota, editors, *A History of Computing in the Twentieth Century*, pages 197–273. Academic Press, New York, NY, USA, 1980.

30 Peter Kroes. Engineering and the dual nature of technical artefacts. *Cambridge Journal of Economics*, 34(1):51–62, 2010.

31 Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6:308–320, 1964.

32 Simone Martini. Several types of types in programming languages. In Fabio Gadducci and Mirko Tavosanis, editors, *HAPOC 2015*, number 487 in IFIP Advances in Information and Communication Technology, pages 216–227. Springer, 2016.

33 Simone Martini. Types in programming languages, between modelling, abstraction, and correctness. In Arnold Beckmann, Laurent Bienvenu, and Nataša Jonoska, editors, *CiE 2016: Pursuit of the Universal*, volume 9709 of *LNCS*, pages 164–169. Springer, 2016.

34 John McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.

35 John McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, volume 35 of *Studies in Logic and the Foundations of Mathematics*, pages 33–70. Elsevier, 1963. A preliminary version presented at the Western Joint IRE-AIEE-ACM 1961 Computer Conference, pp. 225–238. ACM, New York, NY, USA (1961).

36 Dale Miller. Reciprocal influences between proof theory and logic programming. *Philosophy & Technology*, 2019. doi:10.1007/s13347-019-00370-x.

37 Robin Milner. Processes: a mathematical model of computing agents. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, number 80 in Studies in the Logic and the Foundations of Mathematics, pages 157–174, Amsterdam, 1975. North-Holland.

38 Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

39 Francis Lockwood Morris and Cliff B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, 6:139–143, 1984.

40 James H. Morris. *Lambda-calculus models of programming languages*. PhD thesis, MIT, 1968.

41 Allen Newell, Alan J. Perlis, and Herbert A. Simon. Computer science. *Science*, 157(3795):1373–1374, 1967.

42 David Nofre, Mark Priestley, and Gerard Alberts. When technology became language: The origins of the linguistic conception of computer programming, 1950–1960. *Technology and Culture*, 55:40–75, 2014.

43 Gordon D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255, 1977.

44 Willard Van Orman Quine. *Word and Object*. MIT Press, 1960.

45 B. Randell. On Alan Turing and the origins of digital computers. *Machine Intelligence*, pages 3–20, 1972.

46 Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw Hill, New York, 1967.

47 Dana Scott. Outline of a mathematical theory of computation. In *Proc. Fourth Annual Princeton Conference on Information Sciences and Systems*, pages 169–76, Also Tech. Mono. PRG-2, Programming Research Group, University of Oxford., 1970.

48 Joseph E. Stoy. *Denotational semantics: The Scott-Strachey approach to programming language theory*. MIT Press, 1977.

49 Christopher Strachey. An impossible program. *The Computer Journal*, 7(4):313, 1965.

50 Christopher Strachey. Towards a formal semantics. In T.B. Jr. Steel, editor, *Formal Language Description Languages for Computer Programming*, pages 198–220. North-Holland, 1966.

51 Christopher Strachey. Fundamental concepts in programming languages. International Summer School in Computer Programming; Copenhagen. Reprinted in Higher-Order and Symbolic Computation, 13, 11–49, 2000, August 1967.

52 Christopher Strachey. The varieties of programming language. Technical Report PRG-10, Oxford University Computing Laboratory, 1973.

53 Alan M. Turing. Lecture to L.M.S. Feb. 20 1947. In Turing archive, AMT/B/1, 1947.

54 Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 70–72. University Mathematical Laboratory, Cambridge, 1949.

55 Raymond Turner. Programming languages as technical artefacts. *Philosophy and Technology*, 27(3):377–397, 2014.

56 Raymond Turner. *Computational Artifacts*. Springer, 2018.

57 Raymond Turner and Nicola Angius. The philosophy of computer science. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy (Winter 2014 Edition)*. Stanford University, 2017. URL: `http://plato.stanford.edu/archives/win2014/entries/computer-science`.

58 John von Neumann. Rigorous theories of control and information. Published in *Theory of Self-Reproducing Automata*, A. W. Burks (Ed.), University of Illinois Press, 1966, pages 42–56, 1949.

59 Glyn Winskell. *Events in Computation*. PhD thesis, University of Edinburgh, 1981.