Wherefore thou art ... Semantics of Computation?

Furio Honsell
University of Udine, Italy

The power of digital simulation combined with the elementary simplicity of Universal Computational Models (e.g. Turing Machines, Church’s λ-calculus, Curry’s Combinatory Logic, cellular automata, ...) is apparently the Pythagorean dream made true, but because of the remoteness and gratuitousness of Computational Models it is also the original sin of Computing. In effect, the dynamics of token/symbol manipulation in such models is too idiosyncratic to be insightful. Hence, the more computers are used in life-critical applications, the more incumbent are digital woes due to potentially incorrect software. To achieve correct software, i.e. software which meets its specifications, we need to establish a formal correspondence between low level peculiarities and higher level conceptual understandings. This amounts to defining formal Semantics of programming languages [26] and addressing the related critical issue of adequacy of formalizations and encodings, which are ultimately irreducible to formalization [11, 12, 18].

I will try to outline a brief history of the quest for a Final Semantics in Computing.

The initial paradigm, since the 60’s, was that of Denotational Semantics: the meaning of a program is a function (an algorithm being a total function), whose behavior is captured by certain logical invariants called types or by observations. This approach used λ-calculus, which is a theory of functions, as the canonical computational model. Every language component received a functional interpretation. Categorically speaking, this approach is syntax directed and can be termed initial semantics, in that the interpretation function is an initial algebra-morphism which maps uniquely the algebraic structure of the syntax, of the programming language, to a set of abstract entities called denotations. The crucial property of the interpretation function is compositionality, namely the (algebraic) inductive structure of the syntax is reflected by the semantics, which therefore must feature a similar, but conceptually independent, algebraic structure. Hence this semantics is extensional and referentially transparent. Denotations are usually morphisms in suitable categories such as, possibly higher order, topological spaces, or domains [24, 21, 22, 25]. The added value of domains comes from the fact that they are endowed with an enriched structure. This allows for natural definitions of recursive objects, since all endomorphisms have fixed points and for approximations, and hence for new proof principles for reasoning on programs, such as Fixed Point Induction. Semantics is, ultimately, just an equivalence relation, in fact a congruence relation. The methodology of Program Synthesis through Program Equivalence capitalizes on this understanding of semantics. The drawbacks of Denotational Semantics are that it does not account for the dynamics of computation (Girard’s criticism [10]) and that it introduces non-standard objects which are not syntactically definable, causing the models not to be Fully Abstract [22], let alone Fully Complete.
Research in the semantics of $\lambda$-calculus led to establish also a correspondence between two, apparently unrelated, logical processes: computation and derivation, called Propositions-as-types and Proofs-as-Programs paradigm, pioneered by Curry and Howard. The question “what is the semantics of Computation?” then goes hand in hand with the question “what is the semantics of a Proof?”. Martin-Löf [19] and Girard [9, 10], developed extensively this analogy whereby the process of proof normalization becomes the canonical Computational Model. The practical outcome of this view is that from a proof of a specification one can extract a correct terminating program meeting that specification. This led to the program extraction from proofs paradigm and the development of proof editors like Coq and LF [6, 11, 12, 18].

A different strand of semantics arose from the study of Concurrent Systems and their dynamics as processes [20, 1, 8]. All the previous approaches dealt adequately only with terminating programs. But non-terminating programs are just as important as algorithms, even if they do not immediately compute a function. E.g. what functions, if at all, do the internet or an operating system compute? Circular and infinite objects, such as streams, are just as pervasive as initial datatypes [7, 13, 16]. Besides having an algebraic structure, the syntax of processes can be immediately endowed with the co-algebraic structure deriving from the operational behavior (transition systems). This provides a dual kind of semantics w.r.t domains, whereby the interpretation function can be construed as the unique final morphism mapping the co-algebra induced by the behavior on syntax into the final co-algebra [5]. This semantics can be viewed as model-oriented and has been termed Final Semantics. Also Final Semantics [23, 15] yields equivalence relations on processes, called strong extensionality [7] or bisimilarity, and provides original proof principles, such as the Co-induction Principles, for establishing it. This semantics, however, is not immediately compositional w.r.t the algebraic structure of the syntax, but it provides more easily fully complete and fully abstract models, which often arise as term models.

Since the distance between behavior and denotational semantics reduces, what is the point of Semantics, then? Semantics provides a kind of partita doppia, a duality, which enforces some kind of invariant. One can check the outcome in two conceptually entirely different ways, one bottom-up, algebraic, observational, denotational, initial, the other top-down, co-algebraic, intentional, behavioral, final. Think about propositional calculus truth values vs Tableaux semantics (proof search); or grammars and regular expressions vs recognizing automata.

A very significant leap forward was achieved by Girard [10] in the late 80’s when he succeeded in conceiving a denotational semantics for the dynamics of Computational Models. This approach, called Geometry of Interaction, was further developed by Abramsky [1, 4, 2] and many others leading to what is called Game Semantics. This semantics covers Linear Logic [9] as well as all the features of programming languages. It yields compositional equivalences, but the very evaluation process itself has a “denotational” counterpart in the semantics. Programs are not construed anymore as input-output functions but as strategies on moves, or operators on information flows.

A very simple but intriguing Universal Computational Model, along this lines, is that of pattern-matching automata, introduced by Abramsky [2] and inspired by Girard [10]. Combinatory Logic terms are interpreted as automata operating on a simple tree language.
At top level the automata model combinatorial reduction, but their operational behavior is explained in a bottom-up compositional fashion. Denotations have a dynamics which parallels, but also abstracts the idiosyncratic dynamics on the syntax, thus meeting Girard’s requirements. Evaluation amounts to normalization, in fact minimization, since the automaton normalizes to a minimal (strongly extensional) automaton [17].

References