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 λ-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 in a suitable constructive logic 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 combinatory 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

[1] S. Abramsky, "Retracing some paths in Process Algebra", CONCUR, LNCS, Springer 1996.

[2] S. Abramsky, "A Structural Approach to Reversible Computation", *Theoretical Computer Science* 347(3), 2005.

[3] S. Abramsky, "Two Puzzles About Computation", in S.B. Cooper, J. van Leeuwen (eds.), *Alan Turing: his work and impact*, Elsevier 2013, pp. 53-57.

[4] S. Abramsky, M. Lenisa, "Linear realizability and full completeness for typed lambda-calculi", *Annals of Pure and Applied Logic* 134(2-3), 2005.

[5] P. Aczel, "Final Universes of Processes", MFPS, LNCS, Springer 1993.

[6] T. Coquand, G. Huet et al., *The Coq Proof Assistant*, http://coq.inria.fr

[7] M. Forti, F. Honsell, "Set Theory with Free Construction Principles", *Annali della Scuola Normale Superiore di Pisa - Classe di Scienze, Sér. 4* 10(3), 1983.

[8] F. Gadducci, "Graph rewriting for the pi-calculus", *Mathematical Structures in Computer Science* 17(3), 2007.

[9] J.Y. Girard, "Linear logic", *Theoretical Computer Science* 50, 1987.

[10] J.Y. Girard, "Geometry of interaction I: interpretation of system F", *Logic Colloquium*, North-Holland 1989.

[11] R. Harper, F. Honsell, G.D. Plotkin, "A framework for defining logics", *Journal of ACM* 40(1), 1993.

[12] F. Honsell, "25 years of formal proof cultures: some problems, some philosophy, bright future", LFMTP, ACM 2013.

[13] F. Honsell, M. Lenisa, "Conway games, algebraically and coalgebraically", *Logical Methods in Computer Science* 7(3), 2011.

[14] F. Honsell, M. Lenisa, "Unfixing the Fixpoint: The Theories of the λY-Calculus", *Computation, Logic, Games, and Quantum Foundations*, LNCS, Springer 2013.

[15] F. Honsell, M. Lenisa, R. Redamalla, "Coalgebraic semantics and observational equivalences of an imperative class-based OO-language", Computational Metamodels, ENTCS 104, Elsevier 2004.

[16] F. Honsell, M. Lenisa, R. Redamalla, "Categories of coalgebraic games", MFCS, LNCS, Springer 2012.

[17] F. Honsell, M. Lenisa, I. Scagnetto, "The Theory of Automatic Combinators", draft, 2015.

[18] F. Honsell, L. Liquori, I. Scagnetto, "$L^{ax}F$: Side Conditions and External Evidence as Monads", MFCS, LNCS, Springer 2014.

[19] P. Martin-Löf, *Intuitionistic type theory*, Bibliopolis, 1984.

[20] R. Milner, *A Calculus of Communicating Systems*, LNCS, Springer 1980.

[21] G.D. Plotkin, "Call-by-Name, Call-by-Value and the lambda-Calculus", *Theoretical Computer Science* 1(2), 1975.

[22] G.D. Plotkin, "LCF Considered as a Programming Language", *Theoretical Computer Science* 5(3), 1977.

[23] J.J.M.M. Rutten, "Universal coalgebra: a theory of systems", *Theoretical Computer Science* 249(1), 2000.

[24] D.S. Scott, "Continuous Lattices", in F.W. Lawvere (ed.), *Toposes, Algebraic Geometry and Logic,* LNM 274, Springer 1972.

[25] D.S. Scott, "Domains for denotational semantics", ICALP, LNCS, Springer 1982.

[26] T.B. Steel Jr. (ed.), Formal Language Description Languages for Computer Programming, North Holland, 1966.