

The New French School of Philosophy of Computing

Paula Quinon^{1*}

Warsaw University of Technology and University of Gothenburg
[paula.quinon@pw.edu.pl]

Abstract. This paper proposes the name *The New French School of Philosophy of Computing* for a community of French theoretical computer scientists and philosophers whose work, taken together, advances a distinctive conception of what computation is. Across several interlocking lines of work — Colson’s structural theory of primitive recursive algorithms, Berry and Curien’s sequential algorithms, Krivine’s classical realisability, Coquand’s calculus of constructions, Dowek’s undecidability results inside subrecursive systems, the programme of implicit computational complexity, and Girard’s Geometry of Interaction — a common philosophical sensibility emerges: the algorithm, not the function, is the primary mathematical object, and structural features of formalisms are genuine features of computation rather than artefacts of representation. I argue that these lines admit a candidate unifying framework in Girard’s *Geometry of Interaction*. The proposal is offered as the motivation for a special session of the History and Philosophy of Computing (HaPoC) at CiE 2026, where it is meant to be examined and contested.

Keywords: French philosophy of computing · constrained computation · primitive recursion · Curry–Howard correspondence · implicit computational complexity · Geometry of Interaction · intensionality

The objective of this paper is to disclose, define, and describe the specificity of the thought of a community of French theoretical computer scientists and philosophers of computing. Taken together, their work strikes me as more than a loose collection of research lines. For now, my sense that there is such a “thing” rests on a feeling, on the reading I have done for my own philosophical papers on the concept of computing, and on conversations with scholars I believe have contributed to the community. Understanding whether such a school really exists will require much more reading and many more conversations. I do not mean to imply that this is the only such case worth studying — scholars elsewhere may well recognise something similar in the communities they belong to, and those would be different investigations, with their own distinctive findings. The present paper is the presentation of the main motivation for the special session of the History and Philosophy of Computing (HaPoC) at CiE 2026, in which the

* Author.

thesis advanced here is meant to be examined and contested. Still, I want to take the risk and coin the name *The New French School of Philosophy of Computing*.

I will start by looking at several interlocking lines of work that, I think, strongly contribute to *the school's* shape. Each is presented here through its original technical content, with philosophical implications flagged either as already operative in the computer scientists' own work or as drawn out later by the philosophical community.

1 Primitive recursive algorithms as a domain in their own right

Colson (1991) developed a structural theory of primitive recursive algorithms in which primitive recursion is treated not, as in the mainstream recursion-theoretic tradition (Kleene 1952; Rogers 1967; Odifreddi 1989), as a propaedeutic to general recursion, but as an object of study with its own internal logic. Applying denotational semantics with the domain of lazy integers, Colson proved what is now known as the *ultimate obstinacy* property: “once an algorithm begins to consume an argument, it will never switch to another argument” (technical summary in Valarcher–Crolard 2011; cf. Colson 1991). The corollary is sharp: “if a primitive recursive derivation of $\min(x, y)$ is expressed faithfully in a programming language, then one of the two computations $\min(1, 1000)$ and $\min(1000, 1)$ will take at least 1000 steps” (Van den Dries 2003). Coquand (1992), in a paper whose title is literally *Une preuve directe du théorème d'ultime obstination*, supplied a constructive proof of the same result; David (2001) and Colson–Fredholm (1998) extended the analysis to mutual recursion and call-by-value variants. Valarcher (1996) and the subsequent work of Crolard, Polonowski & Valarcher (2011) developed the theory of primitive recursive algorithms further, including a representation theorem that establishes primitive recursive algorithms as mathematical objects with stable identity conditions.

Three philosophical payoffs follow from this. First, Colson's work takes primitive recursion seriously as a formalism with its own structure. The class admits both a denotational and an operational reading, and Colson uses denotational tools to prove an operational fact — that primitive recursive algorithms cannot freely choose which argument to examine first. Second, the result locates the source of inefficiency in the formalism itself. Primitive recursion forces an algorithm to commit to one argument before examining the other, and this commitment is built into the definitional scheme, not into any particular algorithm. The suboptimal complexity of $\min$ is structural, not contingent. Third, the result distinguishes the algorithm from the function. The function $\min$ is the same mathematical object whether computed by primitive recursion or by general recursion; what differs is the algorithm available to compute it. As I have argued elsewhere (Quinon 2025), Colson's theorem identifies a property of the algorithm, not of the function — a property invisible if one looks only at what is computed.

2 Sequential algorithms and the mathematics of computational structure

Berry and Curien show, by semantic means, that algorithms can be treated as mathematical objects in their own right. This is the same philosophical move Colson makes proof-theoretically, and the same one Moschovakis makes with recursors — three different routes to the insight that the algorithm is its own thing, in a certain sense “finer” than the function, worth studying directly. Berry and Curien (1982) gave “a sequential denotational semantics [an interpretation of programs as mathematical objects that distinguishes programs by the order in which they compute] for sequential programming languages, based on a new notion of sequential algorithm on the Kahn–Plotkin concrete data structures [data structures whose parts are revealed in a definite order]”, in which “an algorithm may be seen either as a concrete object — a ‘program’ in a simple output-driven language — or as an abstract object — the pair of a sequential function and of a computation strategy for it [the function says what is computed, the strategy says in what order the inputs are examined]” (Berry & Curien 1982, abstract). This was the first category in denotational semantics whose morphisms were “not (presented as) functions, but programs of some sort” (Curien 2022). Berry and Curien were hoping to solve a standard problem in semantics called full abstraction for PCF (Plotkin’s small functional language) — roughly, the problem of finding a denotational semantics that matches the language exactly, with no mathematical objects left over that the language cannot express. They did not succeed: Curien notes that “the model SA [sequential algorithms] was shown not to be amenable to a FA [fully abstract] model of PCF” [the SA model contained more mathematical objects than PCF contained programs, so the correspondence could not be established] (Curien 2022). They did, however, achieve full abstraction for a different language, CDS (Concrete Data Structures) — a language together with collaborators they designed so that its programs correspond exactly to the sequential algorithms in their model.

The philosophical contribution does not depend on the full abstraction result: it lies in the demonstration that a sequential algorithm “may be viewed as a sequential input-output function paired with a computation strategy” (Brookes, on Berry–Curien) — a mathematical object whose identity conditions are finer than the extensional equivalence of the function it computes.

3 Abstract machines and classical realisability

The philosophical interest of Krivine’s work is that the abstract machine — not the function it computes — is the primary mathematical object of the construction. Logical content and computational behaviour are jointly carried by the structure of the machine itself. As in the Colson and Berry–Curien cases, the algorithmic level of description is treated as mathematically fundamental rather than as an implementation of a function defined elsewhere.

Krivine’s work extends the Curry–Howard correspondence — a philosophically important identification of mathematical proofs with computer programs — into classical logic and set theory. The extension is consequential: the correspondence in its original form applied only to *intuitionistic logic*, where every existence proof must produce a concrete witness; classical principles such as the law of excluded middle, which permit reasoning by contradiction without producing witnesses, were thought to lack computational content. The boundary between constructive and non-constructive reasoning was therefore also taken to mark the boundary of what could be given computational meaning. Krivine’s “call-by-name lambda-calculus machine” (Krivine 2007) and his subsequent programme of classical realisability (Krivine 2001, 2003, 2009) show otherwise. He builds a framework, now called ‘classical realisability’, to explore the Curry–Howard (proof-program) correspondence in which “ ‘the programming’ side of the correspondence is an imperative low level language, based on a call-by-name lambda-calculus abstract machine [the Krivine machine, a mathematical model of a computing device that processes arguments only when needed], [while] the ‘proof’ side is usual formalization of Analysis in second-order logic [the standard axiomatic framework for classical mathematics]” (Streicher & Reus 1998, after Krivine). The technical achievement is summarised by Krivine himself: classical realisability “permits to extend the Curry–Howard correspondence between proofs and programs, to Zermelo–Fraenkel set theory and to build new models of ZF, called realisability models” (Krivine 2014).

4 Termination as a structural property

The philosophical interest of Coquand’s work is that properties of computational processes — that they terminate, that the underlying logic is consistent, that we can mechanically check whether two expressions agree — are made to follow from how definitions are built rather than from anything checked after the fact. The structure of the formalism itself enforces these properties. Coquand’s *Calculus of Constructions* (Coquand 1985 thesis; Coquand & Huet 1988) — a formal system in which mathematical proofs and computer programs are written as the same kind of object, and the foundation of modern proof assistants such as Coq and Agda — is strongly normalising, meaning, in Casinghino’s gloss, that “there are no infinite reduction sequences from well-typed expressions” (Casinghino 2022): every computation in the system terminates. This single property has two consequences that matter philosophically. First, the calculus is consistent as a logic, which “makes it an attractive target for the formalization of mathematics” (Casinghino 2022). Second, “it implies that there is an algorithm to check whether two expressions are β -convertible [whether one can be transformed into the other by computation steps]. Thus, type checking is decidable and CC provides a practical basis for programming languages” (Casinghino 2022). Termination is therefore not verified externally on a case-by-case basis but follows from the typing discipline itself: ‘termination of recursive functions is an important property in

proof assistants based on dependent type theories; it implies consistency and decidability of type checking” (Barthe, Grégoire & Pastawski 2010).

In *the school’s* broader picture, I take this to be the same pattern visible in the previous sections. Colson’s primitive recursion fixed the order in which arguments could be consumed. Berry and Curien’s sequential algorithm carried the order of input examination as part of its mathematical identity. Krivine’s abstract machine carried logical content in its instructions. Coquand’s type system carries the guarantee of termination. In each case, how the computational objects are constructed determines what they can and cannot do — and the formalism, in shaping admissible constructions, shapes admissible computations.

5 Undecidability inside subrecursive systems

The philosophical interest of Dowek’s work is that undecidability does not require Turing-completeness [the property of being able to compute every function that a Turing machine can compute]. It can already arise in systems that are much weaker — systems expressive enough only to represent the primitive recursive functions. Dowek (1993) proved this for the *pattern matching problem*: the problem of deciding, given two expressions, whether one can be rewritten into the form of the other. He showed that pattern matching “is undecidable in all the calculi in which primitive recursive functions can be fairly represented” (Dowek 1993), including Girard’s system F [a typed lambda calculus that captures higher-order polymorphism] and Gödel’s system T [a typed lambda calculus with primitive recursion over natural numbers]. The proof works by reducing Hilbert’s tenth problem — a famously undecidable problem about Diophantine equations — to the pattern matching problem. The broader philosophical significance of Dowek’s work on the changing relationship between proof and computation has been discussed by Arana (2017).

6 Implicit computational complexity

The philosophical interest of implicit computational complexity (ICC) is that it asks whether efficiency — how fast a program runs, how much memory it uses — is best understood as an external constraint imposed on computational processes from the outside, or as an intrinsic feature of certain ways of constructing functions. The standard way to talk about complexity assumes the first: one fixes a machine model, runs a program, counts steps. ICC pursues the second. The programme, developed in particular by Baillot, Marion, Leivant, and their collaborators (Leivant & Marion 1993; Bonfante, Marion & Moyen 2011; Baillot 2010), “aims at studying computational complexity without referring to external measuring conditions or particular machine models, but instead by considering restrictions on programming languages or logical principles implying complexity properties” (Baillot 2010). Equivalently, ICC “characterizes programs by constraints on the way in which they are constructed, without reference to a spe-

cific underlying machine model or to explicit bounds on computational resources” (standard survey formulation).

The philosophical move is structurally the same as the one made by Colson at the level of computability and by Coquand at the level of termination: a property classically thought to require external verification (a running clock, a memory monitor) is internalised into the syntax of the formalism. Efficiency becomes a feature of how a function is defined, not of how its execution is observed.

7 Geometry of Interaction as candidate unifying framework

I propose to read the lines surveyed above as broadly unified by Girard’s *Geometry of Interaction* — a programme that takes the same idea each of these lines explores in its own way (the algorithm is a mathematical object in its own right, finer than the function it computes) and tries to give it a single mathematical setting. Girard (1989, 1995) introduced *Geometry of Interaction* as a semantics “that accounts for the dynamics of cut-elimination” (Seiller, *Interaction Graphs: Graphings*) — meaning, in plain terms, a semantics that takes seriously the process by which a proof is reduced to its normal form, rather than treating the proof and its normal form as semantically identical. The mathematical content is captured in the slogan that “the interpretation of a proof π and its normal form ρ are not equal, but one has a way of computing the interpretation of the normal form ρ from the interpretation of the proof π ” (Seiller, *ibid.*). This is the same philosophical move encountered throughout §§1–6: the dynamic process of computation, not just its result, is what the mathematics is about. Subsequent developments through Seiller’s interaction graphs and graphings “[were] introduced as a general, uniform, construction of dynamic models of linear logic [a logic that tracks resources rather than only truth], encompassing all Geometry of Interaction (GoI) constructions introduced so far” (Seiller, research page).

The *school’s* shared concern with the structure of computation is pursued institutionally through Naibo’s proof-theoretic work in the ANR projects GoA (Naibo, 2024–2027) and DySCo (Seiller, 2022–2027), which offer a candidate framework for articulating, in mathematically precise terms, what that concern amounts to. The 2023 Dagstuhl meeting *Formalising the Notion of Algorithm* brought together abstract state machines, recursors, and graphing-based models around precisely this question. The unification proposed here is editorial rather than settled: it is a thesis to be examined in the session, not a fact about how the field already sees itself.

References

1. Baillot, P. (ed.): Proceedings International Workshop on Developments in Implicit Computational Complexity (DICE 2010). EPTCS 23 (2010). <https://doi.org/10.4204/EPTCS.23>

2. Barthe, G., Grégoire, B., Pastawski, F.: CIC^ω : Type-based termination of recursive definitions in the Calculus of Inductive Constructions. In: LPAR 2006. LNCS, vol. 4246, pp. 257–271. Springer (2006). https://doi.org/10.1007/11916277_18
3. Berry, G., Curien, P.-L.: Sequential algorithms on concrete data structures. *Theoretical Computer Science* 20, 265–321 (1982). [https://doi.org/10.1016/S0304-3975\(82\)80002-9](https://doi.org/10.1016/S0304-3975(82)80002-9)
4. Bonfante, G., Marion, J.-Y., Moyen, J.-Y.: Quasi-interpretations: a way to control resources. *Theoretical Computer Science* 412(25), 2776–2796 (2011). <https://doi.org/10.1016/j.tcs.2011.02.007>
5. Casinghino, C.: Strong Normalization for the Calculus of Constructions. *arXiv:2210.11240* (2022). <https://arxiv.org/abs/2210.11240>
6. Colson, L.: About primitive recursive algorithms. *Theoretical Computer Science* 83, 57–69 (1991). [https://doi.org/10.1016/0304-3975\(91\)90039-5](https://doi.org/10.1016/0304-3975(91)90039-5)
7. Colson, L., Fredholm, D.: System T, call-by-value and the minimum problem. *Theoretical Computer Science* 206, 301–315 (1998). [https://doi.org/10.1016/S0304-3975\(98\)00011-5](https://doi.org/10.1016/S0304-3975(98)00011-5)
8. Coquand, T.: Une théorie des constructions. Thèse de troisième cycle, Université Paris VII (1985).
9. Coquand, T.: Une preuve directe du théorème d’ultime obstination. *Comptes rendus de l’Académie des sciences, série I* 314, 389–392 (1992).
10. Coquand, T., Huet, G.: The calculus of constructions. *Information and Computation* 76, 95–120 (1988). [https://doi.org/10.1016/0890-5401\(88\)90005-3](https://doi.org/10.1016/0890-5401(88)90005-3)
11. Crolard, T., Polonowski, E., Valarcher, P.: Extending the loop language with higher-order procedural variables. In: 2nd International Workshop on Higher-Order Rewriting (HOR 2004) / WRS (2005).
12. Crolard, T., Polonowski, E., Valarcher, P.: A Representation Theorem for Primitive Recursive Algorithms. *Fundamenta Informaticae* 107(4), 313–330 (2011). <https://doi.org/10.3233/FI-2011-405>
13. Curien, P.-L.: Sequential algorithms “from the source” (talk slides) (2022). <https://curien.galene.org/presentations/SA-friends-2022.pdf>
14. David, R.: On the asymptotic behaviour of primitive recursive algorithms. *Theoretical Computer Science* 266(1–2), 159–193 (2001). [https://doi.org/10.1016/S0304-3975\(00\)00165-1](https://doi.org/10.1016/S0304-3975(00)00165-1)
15. Dowek, G.: The undecidability of pattern matching in calculi where primitive recursive functions are representable. *Theoretical Computer Science* 107, 349–356 (1993). [https://doi.org/10.1016/0304-3975\(93\)90175-S](https://doi.org/10.1016/0304-3975(93)90175-S)
16. Girard, J.-Y.: Geometry of interaction I: Interpretation of System F. In: *Logic Colloquium ’88*, pp. 221–260. North-Holland (1989).
17. Girard, J.-Y.: Geometry of interaction III: Accommodating the additives. In: *Advances in Linear Logic*, pp. 329–389. Cambridge University Press (1995).
18. Krivine, J.-L.: Typed lambda-calculus in classical Zermelo–Fraenkel set theory. *Archive for Mathematical Logic* 40(3), 189–205 (2001). <https://doi.org/10.1007/s001530000057>
19. Krivine, J.-L.: Dependent choice, “quote” and the clock. *Theoretical Computer Science* 308(1–3), 259–276 (2003). [https://doi.org/10.1016/S0304-3975\(02\)00776-4](https://doi.org/10.1016/S0304-3975(02)00776-4)
20. Krivine, J.-L.: A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation* 20(3), 199–207 (2007). <https://doi.org/10.1007/s10990-007-9018-9>
21. Krivine, J.-L.: Realizability in classical logic. In: *Interactive models of computation and program behaviour. Panoramas et synthèses* 27, pp. 197–229. SMF (2009).
22. Leivant, D., Marion, J.-Y.: Lambda calculus characterizations of poly-time. *Fundamenta Informaticae* 19(1/2), 167–184 (1993). <https://doi.org/10.1007/BFb0037112>

23. Quinon, P.: Intensional Differences Between Programming Languages: A Conceptual and Practical Analysis. *Philosophies* 10(6), 129 (2025). <https://doi.org/10.3390/philosophies10060129>
24. Seiller, T.: Interaction Graphs: Graphings. *arXiv:1405.6331* (2014). <https://arxiv.org/abs/1405.6331>
25. Streicher, T., Reus, B.: Classical logic, continuation semantics and abstract machines. *Journal of Functional Programming* 8(6), 543–572 (1998). <https://doi.org/10.1017/S0956796898003141>
26. Valarcher, P.: Contribution à l'étude des algorithmes primitifs rékursifs. PhD thesis, Université Paris 7 (1996).
27. Van den Dries, L.: Generating the greatest common divisor, and limitations of primitive recursive algorithms. *Foundations of Computational Mathematics* 3(3), 297–324 (2003). <https://doi.org/10.1007/s10208-002-0061-y>
28. Arana, A.: The changing practices of proof in mathematics: Gilles Dowek, Computation, Proof, Machine. *Metascience* 26(1), 131–135 (2017). <https://doi.org/10.1007/s11016-016-0150-1>