Semantics and syntax, between computer science and mathematics

Pierre-Louis Curien

April 2022

Abstract

This text recounts my scientific itinerary from the late 1970's up to now, as I view it today, as well as the context in which it took place. The views expressed here are of course quite personal, and extremely partial in regard of the global landscape of research on programming languages in France and in the world. My research takes place mostly on the theoretical end of the spectrum of computer science. As a matter of fact, my scientific journey is now mainly taking place in homotopical algebra and higher category theory, with an eye on their recently unveiled links with type theory.

This text is dedicated to the memory of
Gilles Kahn (1946-2006), Martin Hofmann (1965-2018), and Maurice Nivat (1937-2017).

1 How I became a computer scientist

As a student, I had a hesitation, as for my future, between languages (the real ones) and mathematics. This brought me all the way to the Marquises islands where I had the plan to become acquainted with Polynesian languages. This was not serious, though (but the human experience was very much worth!). When I came back, I went to visit Maurice Gross (a pioneer in computational linguistics), who recommended me to enroll in the "DEA d'Informatique Théorique" at Université Paris VII (1976-1977). I followed his advice. I remember my passion for the courses of Dominique Perrin and Luc Boasson on codes (à la Schützenberger) and on formal languages, respectively. My memoir was undertaking a compilation/comparison/classification of Dutch works on biology-inspired formal languages (L-systems, as I recall). When I presented it for the oral examination, I understood that it had been received well enough when Maurice Nivat began to say "tu" to me. I also remember that a fellow student had as a starting point for his memoir a long paper of Dana Scott (Data types as lattices), which did not appeal to me at all... I had no idea of λ-calculus.

Then, something happened. I was about to accept a thesis grant from Université Paris VII, when a young researcher named Gérard Berry tried (and finally succeeded) to get in contact with me and mentioned me the possibility to come to Sophia-Antipolis to do my PhD there with him. When I heard this, I immediately made a connection with my hang-gliding activities of the time: what a wonderful place to work at, with so many flying sites around! I received Maurice's benediction to accept Gérard's offer, and I headed towards the South, with my wing on the roof of my Combi

VW. And then I learned λ-calculus! Working with Gérard Berry proved magical rightaway. He had a visionary approach to the full abstraction problem for PCF, and shared his intuitions with me. They were so sharp that I had no difficulty in "mathematising them" in a few months time. But let me introduce the subject.

2 Various flavours of semantics

In the early 1970's, a brand new field of research called *denotational semantics* arose from the encounter between a logician (Dana Scott) and a computer scientist (Christopher Strachey) in Oxford.

- Strachey was working "like a physicist": he assumed that equations such that $D = \{\star\} + (A \times D)$ (for lists), or the really challenging $D = D \to D$ (needed for giving meaning to untyped programs) had solutions, and he developped both a notation for and a thorough description of the mathematical meaning of programs in terms of set-theoretic functions [41]. A notation was introduced for this purpose: $[\![M]\!]$ for the meaning of the program (or piece of program) M, with the understanding that M lives in the syntax, and $[\![M]\!]$ is an element of some set of denotations, as they were called. Typically, a program P from Nat (the type of natural numbers) to Nat has as denotation a (partial, see below) function from $\mathbb{N}$ to $\mathbb{N}$ (the natural numbers that we learn at school). Organising the set of denotations of programs yields a notion of model. The word is taken from logic, where we have a set $\mathcal{M}$ of individuals (the model), and where predicates and formulas (the syntax) are interpreted as subsets of $\mathcal{M}$.

- Scott had the mathematical framework to solve these equations, in the setting of complete partial orders and continuous functions between them [50, 51]. He arrived there by taking inspiration from works in recursion theory (Myhill-Shepherdson [44], Platek [45]) that highlighted the partial order on the set of partially defined functions between any two sets X and Y: if f, g are two such functions, we set $f \leq g$ if g extends f in the sense that g coincides with f on the domain of definition of f (and hence the domain of definition of g contains the domain of definition of f). Scott's reading of this key example of a complete partial order was that f, g represent partial knowledge of the result of a computation as it proceeds. A recursively defined function is modelled by successive approximations. We know nothing (denoted by $\bot$) before the computation starts; before the first unfolding of the recursive definition, we might know its value for some arguments (typically if the text of the program starts with "if 0 then 3 else ..."), and after each unfolding, the information about the program increases (or not!).

On the other hand, a number of researchers, like McCarthy or Landin, had enriched the toolkit of the theory of programming languages with methods to describe formally the execution of programs, by means of interpreters, or what would be called a bit later structural *operational semantics* in seminal notes of Gordon Plotkin [47]. Gilles Kahn later advocated *natural semantics*, which is big-step (the rules describe directly the result of the computation, as opposed to small-step semantics which describes computation step by step), and easier to implement in logic programming style.

What occupied me in the first years of my career was the correspondence between these two kinds of semantics. I'll come to that in the next section. For the time being, let me briefly complete the landscape of semantics as it emerged in roughly the same period of time.

Quite linked with operational semantics is *axiomatic semantics*, with Floyd, Dijkstra and Hoare as founding fathers. Here, one aims at establishing properties of programs by means of logical

assertions that hold at the beginning of an execution (the preconditions) and change according to formal rules associated with each instruction in the code of the program, allowing to predict and prove that the program will satisfy a desired logical assertion (the postcondition) when the program's tasks will be completed, or, in case of non-termination, that the program will respect some invariants (typically liveness assumptions, or fairness assumptions in a concurrent setting).

Denotational, operational, and axiomatic semantics are the three main approaches, but the frontiers are not so clear between them, and this is fortunate. In particular, as we shall see, some denotational models are very much "syntactic". Some uses of axiomatic semantics techniques are so refined that they are quite close to the full description of the meaning of a program. Conversely, "rough" kinds of denotational semantics may be used to prove properties of programs, very much in the style of axiomatic semantics: this was advocated by Patrick Cousot and Radhia Cousot, who founded the nice framework called *abstract interpretation*.

Let me finish this very brief overview with another flavour of semantics, called *algebraic semantics*, which flourished a lot in France thanks to the enthusiastic efforts of Maurice Nivat. Here, the meaning of the program remains a syntactic object (typically an infinite tree) that makes all the potential of a finite piece of code explicit. The approach was based on the formalism of recursive program schemes, while denotational semantics was very much based on Church's λ-calculus. In recent years, these formalisms have been brought together: λ-calculus is equivalent to so-called higher-order recursive program schemes, which are themselves equivalent to automata with stacks of stacks...

3 The full abstraction problem

The full abstraction problem was formulated in an influential "manifesto" of Scott [52], relatively to a toy language called PCF, that served as a backbone for a lot of the works on denotational semantics. It suffices here to say that we have a core typed deterministic functional language (i.e. allowing to declare functions and to apply them), together with a collection of observable types (with associated observable values, such as 3 or "true"), and an operational semantics for closed programs (i.e. without non-locally defined identifiers) of observable type. One writes $M \to^* v$ to denote the fact that the evaluation of M terminates with an observable value v. Most denotational models satisfy the so-called *computational adequacy* properly, which states that $M \to^* v$ if and only if $[\![M]\!] = v$ (in particular, observable values receive their "standard" interpretation $[\![v]\!] = v$).

Thus, there is a perfect match between the syntax and the semantics *at observable types.* Computational adequacy is usually proved by a variant of the so-called computability/realisability method, which has a long history in logic as a tool to prove consistency of logical systems and is also instrumental in proving termination properties in typed λ–calculi.

The property of full abstraction, which we introduce now, allows to formulate a correspondence between syntax and semantics for arbitrary terms, not necessarily closed nor of observable type. Its formulation relies on the notion of contextual (or operational) equivalence, which is the following:

$$M =_{op} N \quad \Leftrightarrow \quad (\forall C \;\; C[M] \to^* v \;\Leftrightarrow\; C[N] \to^* v) \; ,$$

where C ranges over contexts (programs with a hole) such that both $C[M]$ (C in which the whole is filled with M) and $C[N]$ are closed and of observable type. We can think of these contexts as experiments; then M, N are operationally equal if there is no experiment that can distinguish them. Then the model is called *fully abstract* with respect to L if for all M, N (of the same type):

$$[\![M]\!] = [\![N]\!] \quad \Leftrightarrow \quad M =_{op} N \; .$$

The left-to-right direction is called *adequacy* (or "abstraction") and is an easy consequence of computational adequacy. The other direction is the hard one, and corresponds to the *full* of full abstraction or the *complète* of "complète adéquation" (the French term used for full abstraction). A nice explanation of why this is the hard side of the equivalence comes from the theory of intersection types [13] (systematised in Abramsky's "domains in logical form" by [2]), which recasts the denotational side of the correspondence as the side of "proofs", and the operational side as the side of "models". Then, under these glasses, fullness/completeness goes in the right direction: if $\models M = N$ (i.e., if $M =_{op} N$), then $\vdash M = N$ (i.e., $[\![M]\!] = [\![N]\!]$).

The *full abstraction problem* consisted in constructing a fully abstract model of PCF. Milner exhibited one such model as a quotient (mimicking/accounting for the contextual equivalence) of a sort of "algebraic semantics", and he even showed that this model was the unique extensional fully abstract model (i.e. made of functions, unlike the model discussed in the next section) [42]. On the other hand, Plotkin showed that Scott's continuous model of PCF (the only one available at the time) is not fully abstract, but he showed that it becomes fully abstract for an extension of PCF with "parallel or" (actually, originally, "parallel if", see [16] for the more natural variant with parallel or). From there on, two choices were open for investigations on full abstraction:

1. vary/"synthesise" the language to fit an intended model,
2. vary the model to fit an intended language.

Plotkin's result was a result of the first kind. The quest for an answer to question (2) for the original language PCF is what constituted the (loosely formulated) full abstraction problem. The (implicit) goal was to adjust Scott's continuous model to recast Milner's model without reference to the syntax, but also without resorting to a quotient. Since Plotkin had identified parallel or as the culprit, Berry engaged in restricting Scott continuous functions to a subclass of so-called *stable functions* [6]. To discuss these notions, it is good to think of input and output data as tree-like structures whose nodes or/and edges are decorated by elementary pieces of data. Continuous functions are defined to be those functions f for which any single piece of the output $f(x)$ may be computed using a finite part of the input x, and one can take it to be minimal. The stability condition requires that, for a fixed x and a fixed piece of $f(x)$, such a minimal input is unique, and thus minimum. Let us see why it excludes *por* (parallel or), which is such that that $por(\text{true}, \bot) = \text{true}$ and $por(\bot, \text{true}) = \text{true}$ (recall that $\bot$ is a symbol for an undefined value). (One endows $\mathbf{B} = \{\bot, \text{true}, \text{false}\}$ with the partial order defined by $\bot \leq \text{true}$ and $\bot \leq \text{false}$.) We have hat both $(\text{true}, \bot)$ and $(\bot, \text{true})$ are minimal, hence we have not uniqueness (relative to the input $(\text{true}, \text{true})$). A program implementing $por(a, b)$ should be able to output the value true as soon as either a or b is true, and, supposing that a and b are the outputs of two programs P and Q, it would necessitate computing P and Q in parallel, which a sequential language like PCF does not allow.

But stability is not enough. A more subtle example was given by Berry and is known as Gustave's function (a nickname given to Gérard Berry because of an overloading of Gérard's at Inria, which was IRIA at the time), or Berry-Kleene function (as Kleene had encountered a variant of this function too):

$$\begin{aligned} BK(\text{true}, \text{false}, \bot) &= \text{true} \\ BK(\text{false}, \bot, \text{true}) &= \text{true} \\ BK(\bot, \text{true}, \text{false}) &= \text{true} \end{aligned}$$

The reader can check that this function is stable, but again, it could be computed only by running the computation of the three arguments in parallel.

The next candidate was then the notion of *sequential function*, introduced by Vuillemin [53] (and also Milner [42]), and then generalised by Kahn and Plotkin [34] to the framework of concrete

data structures. But the definition did not scale up to higher-types such as (Nat $\to$ Nat) $\to$ Nat. More precisely, if A, B are concrete data structures, then the set of sequential functions from A to B does not give rise to a concrete data structure.

Then Berry had a dream: it should be possible to model the sequential computational behaviours, and only them, by moving from a traditional framework of appropriate ordered sets and *functions* to a setting retaining more than the ordinary input-output behaviour of programs. For example, the following two schedules for the logical *and* (partially specified below) should be given different interpretations in the model.

$$\begin{array}{l}\text{if } b = \text{true then if } a = \text{true then true}\\ \text{if } a = \text{true then if } b = \text{true then true}\end{array}$$

Starting from this luminous intuition, I made the dream into a model (Gérard helped me in smoothing its presentation, while Maurice Nivat followed the work with benevolence and suggested the name "sequential algorithm"), and Gérard made it into a programming language called CDS, to which I gave its first interpreter. I recall this in the next section.

4 Sequential algorithms

A concrete data structure (originally called information matrix in [34]) has a collection C of cells and a collection V of values. Datas x are represented as sets of pairs (c, v) such that $c \in C$ and $v \in V$, called states. Hence a data is made of elementary bricks which consist of a cell c filled with some value v (where a cell holds at most one value). Some cells must be filled before others can be filled, just like a node of a tree can be drawn only if its ancestors have been drawn before, but we shall not go into that level of detail.

For example, the booleans can be represented with one cell ? and two values true and false. The triplet $(\text{true}, \text{false}, \bot)$ may be represented as $\{(?_1, \text{true}), (?_2, \text{false})\}$: there are three cells corresponding to the three coordinates a, b, c, and we spell out that a holds the value true, that b holds the value false, and (implicitly, by not mentioning it) that c is not filled.

Cells (resp. values) of a function type have the form xc' (resp. "valof c" and "output v'"). (Here, x, c refer to the input type, while c', v' refer to the output type.) Let f be a data of function type, called *sequential algorithm*: $(xc', \text{output } v') \in f$ expresses (roughly) that $(c', v') \in f(x)$, and $(xc', \text{valof } c) \in f$ expresses that, at input x, in order to compute the value of the output cell c', f needs, or waits for, the value of cell c. As an example, the meaning of

$$\begin{array}{rcl} A & = & \text{if } b = \text{true then}\\ & & \quad \text{if } a = \text{true then true}\end{array}$$

of type $Bool_1 \times Bool_2 \to Bool_\epsilon$ is

$$\begin{array}{l}\{(\emptyset ?_\epsilon, \text{valof } ?_2)\\ (\{(?_2, \text{true})\} ?_\epsilon, \text{valof } ?_1)\\ (\{(?_2, \text{true}), (?_1, \text{true})\} ?_\epsilon, \text{output true})\}\end{array}$$

In CDS, computation proceeds in a lazy, stream-like way. If an expression a of the language, of any type, is entered for evaluation, the interpreter of the language prompts the user for a request c and returns the value v of this cell in (the meaning of) a, and then prompts the user for a new request c_1, etc...

Now we are ready to explain the dynamics of function application, as illustrated by the two figures APPLICATION 1 and APPLICATION 2: the outer box, when receiving a request c',

consults the input state y stored in a local table, which stands for the part of the input x read so far (initially, the internal table is the empty set). The function is then asked the question yc'. If may either answer (cf. above) "valof c" or "output v'". In the first case, control is transfered to the argument x, because the value of cell c in x is requested. When c answers with some v, then the internal table is updated and become $y' = y \cup \{(c, v)\}$, allowing for the more informed question $y'c'$ to be asked. In this way a loop is formed between f and x, witnessed by a sequence $c, v, c_1, v_1, \ldots$ of alternating "moves". The loop terminates when f has received enough information from x, i.e., when the internal table y has become large enough, so that f presented with yc' answers with "output v'", which yields v' as answer to the initial request c' to $f(x)$.

I presented this then brand new model in 1978 at the Spring School on λ-calculus in La Châtre (the 6th Ecole de Printemps d'Informatique Théorique – an annual series of convivial one week meetings launched by Maurice Nivat in 1973). I presented the operational semantics of the language CDS in 1982 in a joint French-US workshop held in Fontainebleau, and I vividly remember Gilles Kahn stopping me when I was on the very edge of writing stuff, not on a slide, but on the table of the projector itself. The language CDS was developped during a period of several years in the 1980's in Sophia-Antipolis by a team directed by Berry (Matthieu Devin, Francis Montagnac and Annie Ressouche). Gérard Huet had called it encouragingly the "language of the years 1990". Unfortunately, the compiler technology of the time was poor, and also the underlying ideas were somehow ahead of their time (a widely shared enthusiasm for interactive models of computation arose only later, with the advent of linear logic, game semantics, ludics,... – see below). And Berry turned into new adventures (Esterel).

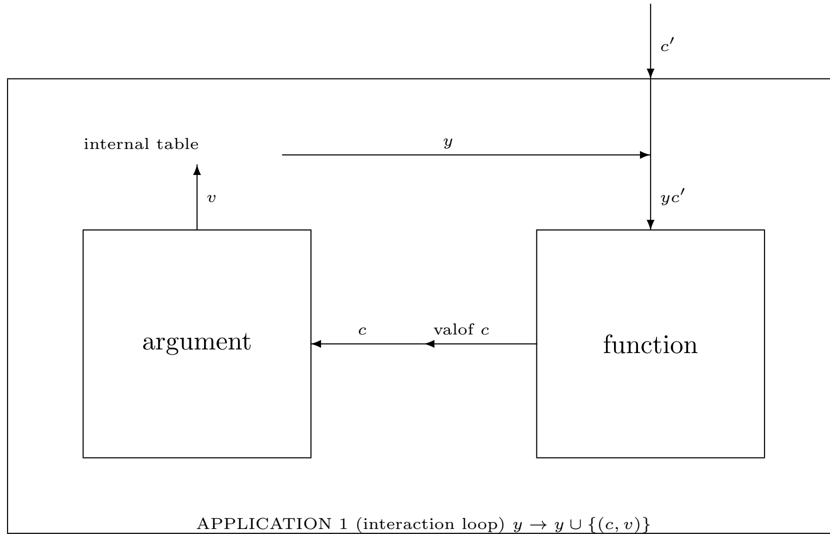
Sequential algorithms did provide a fully abstract model, not for PCF, but for extensions of PCF with control primitives, as offered in the language CDS, or in PCF extended with an operation *catch* [12]. The model is effective in the sense that operational equivalence classes can be effectively enumerated, and there is even a finite number of them when the base types are themselves finite (such as the booleans).

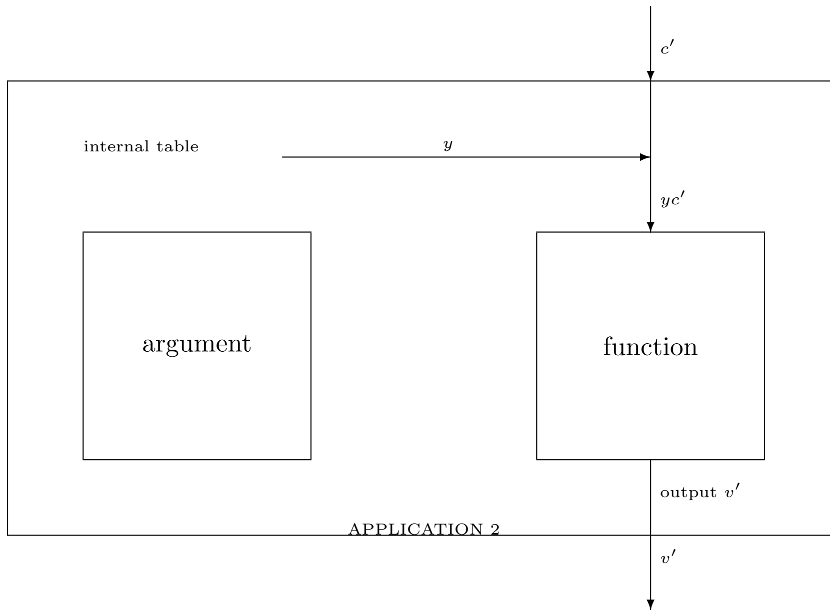
This full abstraction result was not our initially intended goal: we wanted a fully abstract model of PCF for short. But a later result by Loader [39] sort of settled the question negatively: he showed that the operational equivalence for PCF is not effective. As a matter of fact, the models of PCF given in 1994 by Abramsky, Jagadeesan, and Malacaria (AJM), and by Hyland and Ong (H0) [30, 3] offer syntax-free presentations of *term-models* (which we could call an algebraic semantics), and the fully abstract model of PCF is obtained from them by a rather "radical" quotient, called "extensional collapse", which gives little more information than Milner's original term model construction of the fully abstract model.

But the importance lies not in the goal, but in the research that it triggered. The AJM and the HO models were partly inspired by our early work on sequential algorithms, partly by a tradition of dialogue games in logic, and partly by linear logic (see below). Like in CDS, they feature interactive computation, where cells and values become an opponent and a player, and where what we called states are now called strategies. Collectively, they opened a new research area, called *game semantics*, mainly based in the United Kingdom. Game semantics (in the HO presentation) turned out to be extremely successful in providing a sharp classification of various programming language features such as control, references, non-determinism,..., each of these corresponding to suitable restrictions or liberalisations on the allowed strategies.

Last but not least, we discovered later that Kleene had the same experience of a need to record internal information in addition to the plain underlying functions, and as a matter of fact he essentially built the Berry-Curien sequential algorithms at lower-order types [35]. In his flourished vocabulary, he modelled higher-order recursive computations as "machines" communicating via

"oracles", or "envelopes" that are handed by the sender and opened by the receiver.





Before closing this section, I'd like to say that the influence I received from Gilles Kahn goes far beyond his joint work with Plotkin, which was the basis for our constructions. His joint work on coroutines with David MacQueen [33], as well as the so-called Kahn networks [32], were an inspiring source for us (and especially helped me to conceive the interpreter of CDS sketched above).

I'd also like to mention that, parallel to, and complementing the semantic study of stability and sequentiality, Berry had "double-checked" their relevance by showing a syntactic stability theorem

and a syntactic sequentiality theorem within the λ-calculus [7]. In the same vein, stability and sequentiality have been investigated in the realm of recursive schemes and term rewriting systems by Berry and Lévy [10] and by Huet and Lévy [29], respectively. The common author of these two papers, Jean-Jacques Lévy, was the author of a monument: the theory of optimal reduction in the λ-calculus, described in his Thèse d'Etat [38].

5 The categorical abstract machine

My work on sequential algorithms had an interesting side-effect. The model was too baroque to fit into the definitions that were available at the time. Berry and I had to learn a bit of category theory (we knew from Plotkin that we should!), and soon we were able to formulate the model of sequential algorithms in this language. Then, viewing the categorical interpretation of PCF as a *compilation* into a syntax of *categorical combinators*, with the help of Guy Cousineau and Michel Mauny, I designed the *Categorical Abstract Machine* [15], which was taken as the basis of the first implementation of the language Caml (ancestor of the present language OCaml). "Caml" stands for the compression of "Categorical Abstract Machine" (CAM) and "Meta Language" (ML). The language ML was originally conceived to develop proof tactics in the early theorem prover LCF developed by Robin Milner and his research group at the University of Edinburgh in the 1970s.

Let me explain this unexpected transition from sequential algorithms to the CAM in some more detail. The language CDS was actually not based on the λ-calculus directly, but on constructions directly derived from the categorical model: composition, pairing, projections, curryfication, evaluation. I soon realised that these primitives, independently of the particular model of sequential algorithms, provided a language of combinators, different from the classical combinatory logic of Curry et al. And I began to play with those combinators. My first PhD student, Thérèse Hardin, wrote her PhD thesis on their confluence properties [25].

But I also realised quickly that categorical combinators could be executed directly in a form akin to a small-step operational semantics. I shared my preliminary thoughts at a small informal workshop organised by Berry in Sophia-Antipolis in 1983, and this gave rise to a remarkable and rare convergence of cultures: from Gérard Huet, I learned de Bruijn's nameless notation for the λ-calculus (which turned out to be a natural intermediate between λ-calculus and categorical combinators). On the other hand, Guy Cousineau presented his work on implementing ML's static binding in Le-Lisp (a dialect of Lisp then developped at IRIA by Jerôme Chailloux). Implicitly, his encoding was doing the same job as the compilation into categorical combinators. All this converged rapidly to yield the CAM. Guy and myself were joined by Michel Mauny (whom we cosupervised), and we presented our work at the conference Functional Programming and Computer Architecture in Nancy in 1985.

I next introduce categorical combinators (no need to learn category theory for this!), and sketch how the CAM works. The syntax of categorical combinators is the following:

$$f ::= id \mid f \circ f \mid \langle f, g \rangle \mid Fst \mid Snd \mid \Lambda(f) \mid ev$$

and their abstract machine litteral counterparts are given by the following table:

$$\left.\begin{array}{l} id \\ \circ \\ \langle \\ , \\ \rangle \\ Fst \\ Snd \\ \Lambda \\ ev \end{array}\right\} \leadsto \left\{\begin{array}{ll} \texttt{SKIP} & \text{“do nothing”} \\ ; & \text{“code concatenation”} \\ \texttt{PUSH} & \text{“save the current environment”} \\ \texttt{SWAP} & \text{“exchange the environment with the top of the stack”} \\ \texttt{CONS} & \text{“pair the environment with the top of stack”} \\ \texttt{CDR} & \text{“access the tail of the environment”} \\ \texttt{CAR} & \text{“access the head of the environment”} \\ \texttt{CLOSURE} & \text{“form a pair of the body of a function and of its static environment”} \\ \texttt{EV} & \text{“apply a closure to an (evaluated) argument”} \end{array}\right.$$

The CAM has three components, found in many other abstract machines: the code, the environment and the stack. The environment has the structure of a (binary) tree (becoming a graph when recursion is added). More precisely, environments are declared by the following syntax:

$$s ::= () \mid (s, s) \mid \Lambda(f)s,$$

where () stands for the empty environment (used initially when evaluating (the compilation of) a closed program), while (s, t) builds the tree structure, and $\Lambda(f)s$ is a *closure*. Here are the transitions of the machine:

$$\begin{array}{lcl} \langle \texttt{SKIP}; C \mid s \mid S\rangle & \to & \langle C \mid s \mid S\rangle \\ \langle \texttt{CDR}; C \mid (s_1, s_2) \mid S\rangle & \to & \langle C \mid s_1 \mid S\rangle \\ \langle \texttt{CAR}; C \mid (s_1, s_2) \mid S\rangle & \to & \langle C \mid s_2 \mid S\rangle \\ \langle \texttt{PUSH}; C \mid s \mid S\rangle & \to & \langle C \mid s \mid s \cdot S\rangle \\ \langle \texttt{SWAP}; C \mid s_1 \mid s_2 \cdot S\rangle & \to & \langle C \mid s_2 \mid s_1 \cdot S\rangle \\ \langle \texttt{CONS}; C \mid s_2 \mid s_1 \cdot S\rangle & \to & \langle C \mid (s_1, s_2) \mid S\rangle \\ \langle \Lambda(C); C' \mid s \mid S\rangle & \to & \langle C' \mid \Lambda(C)s \mid S\rangle \\ \langle ev; C' \mid (\Lambda(C)s, t) \mid S\rangle & \to & \langle C; C' \mid (s, t) \mid S\rangle \end{array}$$

A few years later (end of the 1980's), Jean-Jacques Lévy had a nice idea. Rather than compiling the λ-calculus into a calculus of categorical combinators (which are sort of “low-level”, even if I synthesised them from very abstract mathematics!), he proposed to “import” some of their benefits in the syntax of the λ-calculus. We developped this idea while I was on a sabbatical visit at DEC SRC in Palo Alto (1989-90), together with Martín Abadì and Luca Cardelli, and we named our calculus the $\lambda\sigma$-calculus, or λ-calculus with *explicit substitutions*. For a progression (and contrast), we give, successively, the syntax of the λ-calculus, of its nameless version (the aforementioned De Bruijn notation), and of the $\lambda\sigma$-calculus (which has a two-sorted syntax of terms M and substitutions s):

$$\begin{array}{ll} M ::= x \mid \lambda x.M \mid MM & \\ M ::= n \mid \lambda.M \mid MM & (n \text{ natural number, } n \neq 0) \\ M ::= 1 \mid \lambda.M \mid MM \mid M[s] & s ::= id \mid \Uparrow \mid M \cdot s \mid s \circ s, \end{array}$$

(where 1 and $\uparrow$ are new names for *Snd* and *Fst*). The main interest of $\lambda\sigma$-calculus is that it provides a finite (unlike De Bruijn's notation) variable-free syntax to account precisely and rigorously for the subtle capture-avoiding substitution mechanism in the λ-calculus. Let us contrast how computation proceeds in the first and in the third syntax. In traditional λ-calculus, a term of the form $(\lambda x.M)N$ (think of a procedure, with body M, called with N as argument) rewrites to $M[x \leftarrow N]$. In the

$\lambda\sigma$-calculus, this is replaced by $(\lambda.M)N) \to M[N \cdot id]$, and the actual substitution of N in M is carried out stepwise, traversing the structure of M. This makes the $\lambda\sigma$-calculus into an intermediate language, suitable to prove correctness properties of compilers (see e.g. [26]).

This experience with explicit substitutions was instrumental to me when I worked on the semantics of Martin-Löf's dependent type theory (see next section), again using the abstractions of category theory. In the dependent setting, substitution not only goes into terms, but also into types, for example $List(n)$ is the type of lists of length n, and in type theory you can write $List(P)$, where P is a term of type Nat. It turns out that substitution in types is interpreted by means of a categorical abstraction called Grothendieck fibration, the definition of which involves some universal property. But universal properties in mathematics are defined only up to isomorphism. And as a result, I noticed that the interpretations of $A[s][t]$ and of $A[s \circ t]$ (for a type A and for substitutions s, t that type-check together) are not equal, but only related by a (canonical) isomorphism (or coercion), while in the syntax of type theory $A[s][t]$ and of $A[s \circ t]$ (without explicit substitutions) are exactly the same type. This mismatch gives rise to the same dilemma as in Section 3:

1. vary the type theory to fit the original intended model,
2. vary the notion of model to fit the original theory.

I followed the first path in [17] by designing a syntax of type theory with explicit substitutions *and coercions*, while (independently) Martin Hofmann followed the second one by providing a construction that starting from a model yields another model that is "strict", i.e., in which $A[s][t]$ of $A[s \circ t]$ are interpreted by the same object. Many years later, the two of us, joined by Richard Garner, were able to place these results in a global picture [20], allowing us to give a new proof of my main *coherence result* in [17], namely that the interpretation of the original type theory in the original model is after all well-defined. More precisely, what is interpreted in the original model is not a (typed) term, but a proof of typing of this term. The coherence theorem asserts that all the possible typing proofs of a given term receive the same interpretation, and hence that we can call it the meaning of the term.

6 Curry-Howard

But let us return to the early 1980's, which saw a capital convergence between logicians (proof theorists) and computer scientists, happening (lucky me!) at my doorsteps. As far as my memories go, in a rather short time interval (one or two years), the following events had a direct impact on (and not only on) me.

- When the time came for me to defend my Thèse d'Etat, Maurice Nivat suggested to invite Jean-Yves Girard to be member of my jury. This was my first encounter with a logician, and, to say the truth, with logic... (Again, Maurice was orienting me discreetly and timely towards an interesting new territory.)
- At a certain point, we (this included Girard, Huet, Cousineau, Coquand – then a young student –, and me) had a kind of joint working group between logicians and computer scientists at the university Paris VII, and, in particular, we started to avidly read a paper by Böhm and Berarducci [11], which explained how to encode some recursive data structures such as natural numbers presented according to Peano axioms (0 is a natural number, and if n is a natural number, so is $n+1$), or lists (cf. Section 2), as formulas in second-order propositional logic, and how to manipulate them using "system F", or polymorphic λ-calculus. These are

two names for the same thing: system F was introduced by Girard in his Thèse d'Etat [21] as a language of proofs for this logic (and he used it to prove its consistency), and completely independently from a computer science perspective by John Reynolds [48], who was interested in such things as data independence and parametricity.

- Shortly after we had established these contacts, Girard got interested in denotational semantics and proposed the first model for system F, for which he reinvented Berry's notion of stable functions. He arrived to stability from a completely different angle. He had played for about a decade on a rather conceptual approach to studies on the complexity of proofs, and had introduced a theory (dilators – dilatateurs in French) in which pullback preserving functors (a categorical version of (conditionally) meet preserving functions) play a central role. This is exactly, in a different language, the idea of stable functions. For example, coming back to *por* (cf. Section 3), it does not hold that $por((\text{true},\bot)\wedge(\bot,\text{true})) = por(\text{true},\bot)\wedge por(\bot,\text{true})$, since
$$\begin{array}{l} por((\text{true},\bot)\wedge(\bot,\text{true})) = por(\bot,\bot) = \bot \\ por(\text{true},\bot)\wedge por(\bot,\text{true}) = \text{true}\wedge\text{true} = \text{true}. \end{array}$$

- Girard gave an enormously influential master course on Proofs and types at Université Paris VII, which gave rise to an eponymous book [22].

These two views of the same language (second item above) are part of a whole picture known as the *Curry-Howard correspondence*: λ-terms can be seen as programs, or as providing a language of (intuitionistic) proofs (expressed in the so-called style of natural deduction). This correspondence had been identified earlier (in the late 1960's) [28], but the realisation of its importance had to wait for more than a decade...

Under this correspondence (which one should not take too strictly, see Section 8), a type reads as a formula, and a program P of type A reads as a proof of A. The dictionary associates product ($\times$) and function types ($\rightarrow$) with conjunction ($\wedge$) and implication ($\Rightarrow$), respectively. Indeed, on one hand, a data of a product type is a pair of data of the component types, and, on the other hand, a conjunction $A\wedge B$ is proved by providing a proof of A and a proof of B. The story for implication goes as follows. Suppose you want to prove a formula B, and that you have established (via a proof π) as a lemma that $A\Rightarrow B$ holds: then any proof π_1 that you can provide for A will give you a proof of B (that we can denote as $\pi(\pi_1)$) by using the lemma.

The correspondence relates also the dynamics: program execution corresponds to the process of *cut elimination*, by which a proof is transformed into a proof "without lemmas", i.e., in which every use of an assumption that has been proved separately via a proof π is replaced by that proof π (this process is called "inlining" in the terminology of programming languages).

Two major projects, both deeply guided by this dictionary, started in the mid 1980's: the calculus of constructions of Coquand and Huet [14] and Girard's linear logic [23].

The calculus of constructions is an extension of both Martin-Löf's type theory (featuring dependent types, cf. Section 5) and of polymorphic λ-calculus, to which inductive types were then added (work of Christine Paulin-Mohring) to form the *calculus of inductive constructions*. The latter served as the kernel of an interactive proof assistant named Coq (a very French symbol!), first released in 1989. In the following decades, Coq has been used for large-scale developpements, both for software certification (certification of a compiler of the language C, led by Xavier Leroy), and for certification of mathematical proofs (certification of the four colour theorem, of Feit-Thompson's theorem – a key part of the classification of finite groups –, led by Georges Gonthier). Thierry Coquand, Gérard Huet, Christine Paulin-Mohring, Bruno Barras, Jean-Christophe Filliâtre, Hugo

Herbelin, Chetan Murthy, Yves Bertot, and Pierre Castéran received the 2013 ACM Software System Award for their role in creating and developping Coq. The adventure continues: the project Deepspec (federating driving forces mainly located on the East Coast of the USA) aims at pushing the certification chain even further towards (and including) the hardware; a proof assistant directly inspired by Coq, called Lean, is attracting rapidly a lot of attention in the mathematical community...

Let us turn to the other "revolution" of the mid 1980's: *linear logic* arose as a side effect of Girard's work on his model of system F. By analysing this model, and by cutting the unessential features of the underlying partial orders, he arrived at a bare notion that he called coherence space, and there he observed a key phenomenon: the function space $A \to B$ (or $A \Rightarrow B$, depending on which side of the mirror you want to look at) decomposes as $(!A) \multimap B$, and he immediately realised the logical significance of these two constructions, which he turned into connectives, that he called the *linear implication* ($\multimap$) and the *exponential* (or "bang"), respectively. The decomposition is accompanied by an explanation in terms of resources: when proving our above lemma that A implies B, we may use the assumption A a number of times (or even not use it at all, and then you have actually a proof of B which you have *weakened* to a proof of $A \Rightarrow B$). This multiple use is governed by the exponential (which can be seen as a modal operator). The linear function space / implication is constrained to use its argument exactly once (e.g. the function $x \mapsto x + 4$ is linear, while $x \mapsto x + x$ and $x \mapsto 3$ are not). Finally, the equation $(A \Rightarrow B) = ((!A) \multimap B)$ reads as: a function that uses its argument several times is the same thing as a linear function whose argument has been made duplicable, and hence can pretend to have been used exactly once.

This decomposition had lots of consequences.

- Negation (in fact linear negation) becomes involutive – a feature that was banished in intuitionnistic logic, since identifying A and the negation of the negation of A amounts to admit that all formulas A are "decidable" (i.e., A, or its negation, holds). This ability to regain duality is again due to the modality "bang" (the intuitionistic negation of A is encoded as the linear negation of $!A$, and remains non involutive).

- Proofs in linear logic can be presented traditionally in natural deduction style, but they also enjoy a description in terms of graphs called proof nets. In subsequent works, Girard developed a program called geometry of interaction (GoI), where the dynamics of cut elimination (rewriting of proof nets) is replaced by a circulation of tokens in the proof net (which remains fixed). This interactive interpretation of computation is in tune with our early work on the language CDS (reported in Section 4). As a matter of fact, numerous links between the GoI, game semantics and abstract machines have been made explicit, by Vincent Danos, Laurent Regnier and Patrick Baillot, notably. Applying Ockham's razor, Girard has pursued this program in his *ludics*, where connectives are defined interactively: types are behaviours, which are collections of "untyped programs" (that he calls designs, and play the role of states or of strategies, cf. Section 4) that "succeed" when confronted or executed against a collection of "counter-programs".

- The formula-as-resource paradigm led to variants of linear logic characterising some time complexity classes, starting with polynomial time. The idea is to have the number of steps of cut elimination (or of GoI execution) controlled by the logic itself, imposing restrictions on the use of the magical "bang" (which Girard nicely described as the ever renewable dollar in your pocket). The framework of ludics, which develops a distinctive paradigm of formulas located in (and competing for) space, opened the way to give a proof-theoretical account of space complexity as well.

- The connectives of linear logic, while initially designed according to the formula-as-resource prism, can also be classified according to polarity: there are passive (or reversible, or negative) and active (or irreversible, or positive) connectives. This led to the development of polarised logics (intuitionistic, linear, classical), and to the investigation of new facets of the Curry-Howard correspondence, where the polarities read as lazy versus eager types.

The world of semanticists was changed radically by the advent of linear logic. People began to realise that the linear decomposition of the function space is present in most of the models considered so far. In the case of sequential algorithms, this was observed by Lamarche (unpublished), and by myself (elaborating on his observations) [18]. (The AJM variant of) game semantics was first developped as a model of (a fragment of) linear logic.

But the logic underlying game semantics is not quite linear logic: negation is not involutive, as it corresponds to two cumulated "actions" on the game: exchange the role of player and opponent, and add a unit of time at the beginning of the game (at least in the sequential versions of games) to maintain its (negative) polarity. The fine study of the rich relations betweeen game semantics and linear logic has been undertaken by Paul-André Melliès, culminating in his habilitation dissertation [40].

But the influence of linear logic extended far beyond the community of theoreticians on either side of the Curry-Howard correspondence. It has pervaded the research community working on programming language design and implementation. For example, linear types may allow for compiler optimisations, as was initially observed by Yves Lafont [37]. Much more space (and bibliographical work) would be needed to tell that story in more detail.

Before closing this section, I want to mention another important milestone that occurred in 1990, when Tim Griffin [24] found the logical counterpart of call-cc, a control operator present in the language Scheme (and in some other functional programming languages), which is ... classical reasoning! This was sort of breaking the monolithic slogan that classical logic "was no good for computation". Griffin's discovery was then explained and discussed under different angles: by Girard using polarities, by Michel Parigot who developped a carefully crafted extension of the λ-calculus called $\lambda\mu$-calculus as a language for classical logic in natural deduction style (with a particular discipline on cut elimination). I'll come back on this thread in the next section. Let me just mention here that Griffin's work was also the starting point of Jean-Louis Krivine's ambitious and successful programme of *classical realisability*, aimed at extracting the computational content of all axioms of set theory, and of mathematical proofs in general [36].

7 Dualities

I have encountered "complementary pairs" of various kinds as my career unrolled:

- memory *cell* (or location or register) versus its actual contents or *value*; in object-oriented style, record field names versus their values, method names versus their actual definition (cf. the bricks (c, v) of concrete data structures);
- *input* and *output*, or (in the language of proof theory) hypotheses and conclusions;
- *sending* and *receiving* messages (in process calculi);
- a *program* and its *context* (the libraries of your program environment – or the larger program of which the program under focus is a subpart, or a module) (cf. the operational equivalence of Section 3); the programmer and the computer; two programs that call each other;

- call-by-name and call-by-value.

In a joint work with Hugo Herbelin [19], we found a striking convergence between the second and the last item in this list. In his PhD thesis, Hugo had proposed a language for classical proofs in sequent calculus style. In *sequent calculus*, connectives are not described via introduction rules and elimination rules like in natural deduction, but via right and left introduction rules. Hugo also identified the operational counterpart of the left introduction rule for implication. Let me try to explain this. When a call-by-name interpreter is faced with an expression MN, it will concentrate on M until M "becomes" (i.e., M rewrites to) some $\lambda x.P$, and then will proceed with the body P, keeping the task of evaluating N for later. Thus, if the "stack of things to do when arriving at MN" (the context) is E, the context right after moving the focus from MN to M (as suggested above) consists of the pair of N and E, written as $(N \cdot E)$ ("put N on top of E"). More formally, recalling from Section 3 that a context is a term with a hole in it, our global program writes as $E[MN]$. If we write E' for $N \cdot E$, our change of focus from MN to M is captured by the equality $E[MN] = E'[M]$ (read from left to right). And now comes the logic. We consider a context as a "coterm" whose type is the type of its hole, and this type is "on the left" (see the definition of sequents below), on the side of assumptions: a context expects an expression to fill its hole. In contrast, a term M of type A produces a value of type A "on the right", on the side of the conclusions (in classical logic, there are multiple conclusions). With this in mind, let us now analyse the types involved in our equality $E[MN] = E'[M]$: the context E expects a term of type B, while E' expects a term of type $A \to B$. But E' is a pair of a term of type A (on the right) and, as we have just seen, of a context expecting a term of type B. This matches perfectly with the left introduction rule for implication in sequent calculus, which is of the form

$$\text{"if } (\ldots \vdash A, \ldots) \text{ and } (\ldots, B \vdash \ldots)\text{, then } (\ldots, A \to B \vdash \ldots)\text{"}.$$

(A sequent is a pair of two lists Γ and Δ of formulas, written as $\Gamma \vdash \Delta$, with the intention that the conjunction of the formulas in Γ entails the disjunction of the formulas in Δ.)

In our work, Hugo and I continued the program started by Hugo by providing a syntax (in the form of a binding construct called $\tilde{\mu}$) allowing to control the change of focus from one assumption to another in a sequent, dual to the μ of Parigot's $\lambda\mu$-calculus, which served to switch between conclusions. Equipped with this symmetric syntax, we were able to exhibit a striking symmetry between call-by-name and call-by-value, according to whether one priviledges rewriting the central term below to the right or to the left, respectively:

$$c_1[\alpha \leftarrow \tilde{\mu}x.c_2] \quad \leftarrow \quad \langle \mu\alpha.c_1 \mid \tilde{\mu}x.c_2 \rangle \quad \to \quad c_2[x \leftarrow \mu\alpha.c_1]$$

(in rewriting theory, such a situation of a term rewriting in two different ways is called a critical pair). Slowly, $\langle \mu\alpha.c_1 \mid \tilde{\mu}x.c_2 \rangle$ reads as follows: c_1 (resp. c_2) represents the proof of a sequent with some occurrence of a formula A among its conclusions (resp. among its assumptions), $\mu\alpha.c_1$ (resp. $\tilde{\mu}x.c_2$) adds a focus on this conclusion (resp. this assumption), and the construct $\langle - \mid - \rangle$ represents an application of the cut rule: if $\ldots_1, A \vdash \ldots_2$ and $\ldots_1 \vdash A, \ldots_2$ (with focus on the respective highlighted occurrences of A), then $\ldots_1 \vdash \ldots_2$. (We invite the reader to check the soundness of this rule and of the above left introduction rule for implication with respect to the intended "conjunction of $\ldots_1$ implies disjunction of $\ldots_2$" interpretation.) We called our calculus the $\overline{\lambda}\mu\tilde{\mu}$-calculus.

Later, Guillaume Munch-Maccagnoni enriched the picture by showing the role of polarities (cf. Section 6) in controlling the crucial critical pair above, and provided notions of models for this polarised version of the $\overline{\lambda}\mu\tilde{\mu}$-calculus.

8 The homotopical era

This section is dedicated to the memory of Vladimir Voevodsky (1966-2017),
and of Jean-Louis Loday (1946-2012) and Patrick Dehornoy (1952-2019).

The turn of the year 2010 saw the emergence of a new era for type theory, when the interpretation of types as (topological) spaces, and in particular of the predicate of equality (called the identity type in Martin-Löf type theory) in terms of paths in spaces has been actively pursued. If x, y are of type A, then there is a type $x =_A y$, called identity (or equality) type over A, and a term p of type $x =_A y$ is read as a path from x to y in (the interpretation of) A. And then one can form, for two distinct terms p, q of type $x =_A y$, a type $p =_{x=_A y} q$, and a term of this type reads as a homotopy (or continuous deformation) from the path p to the path q (which share their endpoints x and y). Continuing in this way we get so-called higher structures. Technically, it has been proved independently by Lumsdaine and by van den Berg and Garner that every type defines a syntactic weak ∞-groupoid.

Also, Martin-Löf had associated with identity types (as with other type constructors) its constructors (the terms $\mathit{refl}(a)$ of type $a =_A a$, for a type A and a term a of type A – reflexivity!) and its induction principle, which says that in order to build a function taking triplets (x, y, p) of terms x, y of type A and p of type $x =_A y$ to some type C (more generally a type $C(x, y, p)$ depending on x, y, p), it is enough to define it for the special triplets $(x, x, \mathit{refl}(x))$. In homotopical terms, this is linked to the observation that a path with variable ends can be shrinked to a point.

The full realisation of this new dictionary gave rise to *homotopy type theory* (HoTT), culminating into the collective book [31] which cristallised during the special year on Univalent foundations at the Institute of Advanced Study (2012-2013) organised by Vladimir Voevodsky, Thierry Coquand, and Steve Awodey. We highlight below some of the main innovations offered by homotopy type theory.

- A new way to look at how types are "big" or "complex", by classifying them according to how much non-trivial higher structure they have. In particular, in the HoTT setting, types are not propositions in general, as propositions have no higher structure at all: it is reduced to the existence ("true") or inexistence ("false") of an element of that type, therefore they are very particular kinds of types.

- A new whole provision of inductive types, named *higher inductive types* (HIT), which allow to encode lots of familiar objects from topology, like spheres, tori... For example, just like natural numbers are given by a 0 and a successor function, circles are "given" by a point *base* and a path *loop* of type *base* = *base*. One can then use an induction principle to define type-theoretically maps from the circle into some space A (i.e., to "draw" a circle in A). This is the beginning of a whole area that emerged very rapidly (in less than a decade), called *synthetic homotopy theory*. Its achievements so far, beyond providing elegant new proofs of elementary results in algebraic topology that trade the explicit use of continuity against logical induction principles, have been to place some more advanced theorems (like Blakers-Massey theorem [5]) into a wider context. In computer science, HIT's have been used to describe quotient types of various kinds.

- A strong axiom called *univalence*, proposed by Voevodsky after he "synthesised" it out of the simplicial model of type theory. This axiom is to the universe of all types what the axiom of extensionality is to the function type. So let us discuss function extensionality first. This axiom says that if two terms M, N of type $A \to B$ are such that for all x of type A we have

a term $p(x)$ of type $Mx =_B Nx$, then we have $M = N$ (i.e., the axiom gives us a term *function–extensionality*(p) of type $M =_{A \to B} N$). Univalence is similarly providing a way to get terms of type $A =_U B$ (where U is the universe, i.e. a "big" type of all "small" types). It says (simplifying slightly) that if we have two terms f, g of type $A \to B$, $B \to A$, respectively, such that there are terms ϵ, η of type $g \circ f = id$ and $f \circ g = id$, then we have $A =_U B$ (i.e., the axiom gives us a term *univalence–axiom*(f, g, ϵ, η) of type $A =_U B$). This axiom is "magical", as it allows to "instantaneously" transfer algebraic structure from one type to another isomorphic (or more property homotopy equivalent) one, and makes the extremely powerful principle of induction associated with identity types (recalled above) available in this context.

A community of hundreds of enthusiastic young scientists working on these ideas, either coming from mathematics or from computer science, has grown in almost no time in many corners of the world, including in France (Paris, Nantes).

The two trends of homotopy type theory (and synthetic homotopy theory) and of formalising mathematics in Coq or Lean (cf. Section 6) have yet to somehow merge. Here are some obstacles or challenges for such a convergence.

- The homotopical extension of type theory (notably univalence) has raised some questions on how to maintain, in the homotopical setting, the whole architecture of Coq, which is based on proofs "that compute": e.g., if a term M of type $A \to \mathrm{Nat}$ is given, where A is, say, a type of moduli of spaces (i.e., its elements are spaces of some kind), then we would like to extract from it an actual program that computes this number MX for a given space X of type A. An answer to these concerns has been given by cubical type theory, developed by Coquand and his coauthors.

- Although the "higher structure attitude" has permeated quite a number of branches of mathematics beyond algebraic topology, like algebraic geometry or representation theory, many areas of mathematics do not have (at least at present) natural meaningful analogs of "homotopy" or "deformation".

- A number of mathematicians express their interest (or even need) for theorem provers (interactive or automated) *that would support their usual way of reasoning*, which includes non-constructive arguments, which can create some cultural gap with the computer scientists attached to the constructive nature of a software like Coq, built on top of the very constructive calculus of constructions (cf. Section 6).

- Homotopy type theory is sort of "too much higher-order". Since every type is endowed with a structure of weak ∞-groupoid, this raises a chicken-and-egg problem: in order to define or formalise any kind of algebraic structure in HoTT, say, by generators and (ordinary) relations, one is immediately confronted to the cascade of having to specify *at the same time* all the higher coherences to be associated with such relations, that can only hold weakly. Understanding how to "bootstrap" this properly is an active subject of research.

As for myself, I have followed a path (!) that is sort of "parallel" to that trend, by "foraging" in related fields such as operad theory or higher dimensional rewriting, using my "syntactic toolbox" and my own "obsessions", among which are coherence issues of various sorts (cf. Section 5). Like for much of what I have done before, there was no predefined plan. I like to call this serendipity (like in, say, the movie Notting Hill). I had the chance to have Jean-Louis Loday and Patrick Dehornoy

as friends: they communicated their enthusiasm for algebraic operads, and braids, respectively, to me, and this opened new horizons... They left us much too early, and I miss them badly.

Coming back to types and to the main theme of this contribution, I had also the chance to get to know Vladimir Voevodsky on a daily basis, when I organised in the spring 2014 a trimester at the IHP (Institut Henri Poincaré) on the Semantics of proofs and certified mathematics, together with Hugo Herbelin and Paul-André Melliès: we had occasions to visit exhibitions ... or IKEA; we had conversations about irrational phenomena...

9 The French touch in semantics of programs and proofs

The tutelar figures of Gérard Huet, Gilles Kahn, Jean-Jacques Lévy, Gérard Berry, and of Jean-Yves Girard and Jean-Louis Krivine, have shaped a "French identity" around the whole field of programs and proofs. I benefited a lot from knowing them, both scientifically and on the personal level, and – crossing the Channel – I would add Robin Milner, Gordon Plotkin and Martin Hyland to the list. A "cousin" of mine is Glynn Winskel, whose thesis work [54] was contemporary to mine. He explored event structures, one facet of which is to be to concrete data structures what stability is to sequentiality, while they also played an important and successful role in the development of concurrency theory. A "young brother" of mine is Thomas Streicher, whose work on the foundations of type theory turned out to have a great impact. As for the next generation (here "generations" are often counted by units of a few years – typically the time of a PhD thesis!), Thierry Coquand, Thomas Ehrhard, Yves Lafont, Vincent Danos, Laurent Regnier, Hugo Herbelin, Paul-André Melliès – just to quote those in or from France with whom I interacted most – have very much enriched the scene: differential or probabilistic extensions of denotational semantics, links with knot theory,.... As for the international scene, among the great scientists I had the fortune to collaborate with, I have a special thought for Martin Hofmann, whose tragic disparition on Mount Nikko Shirane in Japan in 2018 has put our community into a deep mourning. He was smart and generous. And the next generation, and the generation next to the next generation... I would not like to call this a school, as the motivations and background of every single researcher in this (open) list are so different. But certainly a common spirit, in which computational/syntactic aspects are stressed, even in the most (seemingly) abstract mathematical structures coming into play.

References

[1] M. Abadi, L. Cardelli, L. P.-L. Curien, and J.-J. Lévy, Explicit substitutions, Journal of Functional Programming 1(4), 375-416 (1992).

[2] S. Abramsky, Domain theory in logical form, Annals of Pure and Applied Logic 51, 1-77 (1991).

[3] S. Abramsky, R. Jagadeesan, and P. Malacaria, Full abstraction for PCF, Information and Computation 163, 409-470 (2000). (Manuscript circulated since 1994.)

[4] R. Amadio and P.-L. Curien, Domains and Lambda-Calculi, Cambridge University Press (1998).

[5] M. Anel, G. Biedermann, E. Finster, and A. Joyal, A generalized Blakers-Massey theorem, Journal of Topology 13(4), 1521–1553 (2020).

[6] G. Berry, Stable models of typed lambda-calculi, Proceedings of ICALP 1978, Springer Lecture Notes in Comp. Science 62 (1978).

[7] G. Berry, Modèles complètement adéquats et stables des lambda-calculs typés, Université Paris VII (Thèse de Doctorat d'Etat) (1979).

[8] G. Berry and P.-L. Curien, Sequential algorithms on concrete data structures, Theoretical Computer Science 20, 265-321 (1982).

[9] G. Berry and P.-L. Curien, The kernel of the applicative language CDS: theory and practice, Proceedings of the French-US Seminar on the Applications of Algebra to Language Definition and Compilation, Cambridge University Press, 35-87 (1985).

[10] G. Berry and J.-J. Lévy, Minimal and optimal computations of recursive programs, Journal of the ACM 26 (1), 148–175 (1979).

[11] C. Böhm and A. Berarducci, Automatic synthesis of typed lambda-programs on term algebras, Theoretical Computer Science 39, 135-154 (1985).

[12] R. Cartwright, P.-L. Curien, and M. Felleisen, Fully abstract semantics for observably sequential languages, Information and Computation 111 (2), 297-401 (1994).

[13] M. Coppo and M. Dezani, A new type assignment for lambda-terms, Archiv. Math. Logik 19, 139-156 (1978).

[14] T. Coquand and G. Huet, The calculus of constructions, Information and Computation 76 (2/3), 95–120 (1988) (preceded by a conference paper: constructions: A higher order proof system for mechanizing mathematics, European Conference on Computer Algebra, Linz, in 1985, Linz, Lecture Notes in Computer Science 203, 151-184 (1985)).

[15] G. Cousineau, P.-L. Curien, and M. Mauny, The categorical abstract machine, Science of Computer Programming 8, 173-202 (1987) (shorter conference version in the Proceedings of . Functional Programming and Computer Architecture 1985, Lecture Notes in Computer Science 201).

[16] P.-L. Curien, Categorical combinators, sequential algorithms and functional programming, Pitman (1986) (revised edition, Birkhäuser (1993)).

[17] P.-L. Curien Substitution up to isomorphism, Fundamenta Informaticae 19 (1-2), 51-86 (1993).

[18] P.-L. Curien, On the symmetry of sequentiality, Proceedings of Mathematical Foundations of Programming Semantics 1993, Lecture Notes in Computer Science 802, 29-71 (1994).

[19] P.-L. Curien and H. Herbelin, The duality of computation, Proceedings of International Conference on Functional Programming 2000, Montréal, ACM Press (2000).

[20] P.-L. Curien, R. Garner and M. Hofmann, Revisiting the categorical interpretation of dependent type theory, Theoretical computer Science 546, 99-119 (special issue in honor of G. Winskel's 60th birthday) (2014).

[21] J.-Y. Girard, Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur, Université Paris VII, Doctorat d'Etat (1972).

[22] J.-Y. Girard (with the collaboration of Y. Lafont and P. Taylor), Proofs and types, Cambridge University Press (1989).

[23] J.-Y. Girard, Linear logic, Theoretical Computer Science 50, 1-102 (1987).

[24] T. Griffin, A formulae-as-types notion of control, Proceedings ACM Principles of Programming Languages, ACM Press, 47-58 (1990).

[25] T. Hardin, Résultats de confluence pour les règles fortes de la logique combinatoire catégorique et liens avec les lambda-calculs, PhD thgesis, Univ. Paris VII (1987).‘

[26] T. Hardin, L. Maranget, and B. Pagano, Functional back-ends within the Lambda-sigma calculus, Proceedings of International Conference on Functional Programming 1996, Philadelphia, ACM Press, 25–33 (1996).

[27] H. Herbelin, Séquents qu'on calcule: de l'interprétation du calcul des séquents comme calcul de λ-termes et comme calcul de stratégies gagnantes, PhD thesis, University Paris 7 (1995).

[28] W. Howard, The formulas-as-types notion of construction, in Curry Festschrift, Hindley and Seldin (eds.), 479-490, Academic Press (1980) (manuscript circulated since 1969).

[29] G. Huet, J.J. Lévy, Computations in orthogonal term rewriting systems I and II, In Computational Logic, J.A. Robinson anniversary volume, MIT Press, 395–414 and 415-443 (1991).

[30] M. Hyland and L. Ong, On full abstraction for PCF, Information and Computation 163, 285-408 (2000). (Manuscript circulated since 1994.)

[31] Homotopy Type Theory: Univalent Foundations of Mathematics, Collective book `https://homotopytypetheory.org/book` (2013).

[32] G. Kahn, The semantics of a simple language for parallel programming, Proceedings of IFIP Congress 1974, 471-475 (1974).

[33] G. Kahn and D. MacQueen, Coroutines and networks of parallel processes, Proceedings of IFIP Congress 1977, Toronto, 993–998, North Holland (1977).

[34] G. Kahn and G. Plotkin, Concrete domains, Böhm Festschrift, Theoretical Computer Science 121, 187-277 (1993) (and in French TR IRIA-Laboria 336 in 1978).

[35] S. Kleene, Recursive functionals and quantifiers of Ffnite types revisited I, II, III, and IV, in Proc. General Recursion Theory II, Fenstad et al. eds, North-Holland (1978), Proc. of the Kleene Symposium, Barwise et al. eds, North-Holland (1980), Proc. Patras Logic Symposium, North Holland (1982), and Proc. Symposia in Pure Mathematics 42 (1985), respectively.

[36] Jean-Louis Krivine, Realizability in classical logic, in Interactive models of computation and program behaviour. Panoramas et synthèses, Société Mathématique de France 27, 197-229 (2009).

[37] Y. Lafont, The linear abstract machine, Theoretical Computer Science 59, 157-180 (1988).

[38] J.-J. Lévy, J.-J., Réductions correctes et optimales dans le λ-calcul, Thèse d'Etat, Université Paris VII (1978).

[39] R. Loader, Finitary PCF is not decidable, Theoretical Computer Science 266, 341–364 (2001).

[40] P.-A. Melliès, Une étude micrologique de la négation, Habilitation thesis, Université Paris Diderot (2017).

[41] R. Milne and C. Strachey, Theory of programming language semantics (2 volumes), Chapman & Hall (1977).

[42] R. Milner, Fully abstract models of typed lambda-calculi, Theoretical Computer Science 4, 1-23 (1977).

[43] G. Munch-Maccagnoni, Focalisation and classical realisability, Proceedings of Computer Science Logic 2009, Lecture Notes in Computer Science 5771, 409–423 (2009).

[44] J. Myhill, J. and J. Shepherdson, Effective operations on partial recursive functions, Zeitschrift für Mathematische Logik und Grundlagen der Mathematik 1, 310-317 (1955).

[45] R. Platek, Foundations of recursion theory, PhD thesis, Stanford University (1966).

[46] G. Plotkin, LCF as a programming language, Theoretical Computer Science 5, 223-257 (1977).

[47] G. Plotkin, A structural approach to operational semantics, Aarhus University report DAIMI FN-19, (1981) (available at `https://homepages.inf.ed.ac.uk/gdp/publications/sos_jlap.pdf`).

[48] J. Reynolds, Towards a theory of type structure, Proceedings of Colloque sur la Programmation 1974, Paris , Lecture Notes in Computer Science 19, 408–423 (1974).

[49] J. Reynolds, Types, abstraction and parametric polymorphism, Proceedings IFIP Congress 1983, 513–523, North Holland (1983).

[50] D. Scott, Continuous lattices, Proceedings Toposes, Algebraic Geometry and Logic, Springer Lecture Notes in Mathematics 274, 97-136 (1972).

[51] D. Scott, Data types as lattices, SIAM Journal of Computing 5, 522-587 (1976).

[52] D. Scott, A type-theoretical alternative to ISWIM, CUCH, OWHY,x Theoretical Computer Science 121, 411-440 (1993) (manuscript circulated since 1969).

[53] J. Vuillemin, Syntaxe, sémantique et axiomatique d'un langage de programmation simple, Thèse d'Etat, Université Paris VII (1974).

[54] G. Winskel, Events in computation, PhD thesis, University of Edinburgh (1980).