Introduction to "Interactive models of computation and program behaviour"

Pierre-Louis Curien (PPS, CNRS and Université Paris 7)

May 14, 2012

Since the mid-eighties of the last century, a fruitful interplay between computer scientists and mathematicians has led to much progress in the understanding of programming languages, and has given new impulse to areas of mathematics such as proof theory or category theory. Two of the authors of the present volume (Krivine and Curien) designed independently, at around the same time (1985), interpreters for the lambda-calculus, which both turned out to have important consequences. The lambda-calculus, besides being one of the formalisms capturing the notion of computable function, is by far the best understood core programming language. It underlies many modern programming languages, like LISP, ML, Haskell.

- Curien's device (developped in collaboration with Cousineau and Mauny), called the Categorical Abstract Machine (CAM) [1], served as the basis for the compiler of the French dialect of ML, the language CAML – a language which is well-suited for teaching computer programming, and for prototyping various pieces of software.
- Krivine's device, called Krivine Abstract Machine (KAM) [5], is at the heart of his subsequent work on the extraction of computational contents from mathematical axioms and statements.

In the CAM acronym, "categorical" stands for the connection between the lambda-calculus and cartesian closed categories. As a matter of fact, lots of structuring thoughts and results have come from the triangle formed by the languages of proofs, categories, and programs, respectively. Melliès' contribution to this volume recounts the latest state of the art on this correspondence, which has learned a lot from the rise of Girard's linear logic, from 1986 on [3]. Categories are particularly good at capturing some invariants in an algebraic way. We just mentioned cartesian closed categories which capture the invariance of λ-terms under the two basic equalities in this theory, β and η, in terms of the universal constructions of categorical product and internal homspace. In his contribution, Melliès places this role of categories in context, by recalling the role of categories in capturing, say, the invariants of knot theory.

In the rest of this introduction, we give some background on the lambda-calculus and its dynamics. Lambda-calculus comes in two flavours: untyped and typed. Melliès' contribution in this volume is of the second flavour, while the other two contributions build on untyped terms.

The λ-calculus is a language of terms built with only three operations:

1. variables: x is a λ-term (think of an identifier in a program, or of a variable in a function $f(x)$);
2. application: if M, N are terms, then MN is a term (think of the application $f(a)$ of a function f to an argument a);
3. abstraction: if M is a term, then $\lambda x.M$ is a term.

One may also wish to add constants, as Krivine does a lot in his contribution to this volume.

Above, λ is the only non-familiar symbol for the general mathematician (or the high school student). It makes explicit on which parameter we want the term M to depend. Think of an expression like $x^2 + 3mx + 4$, where m is a parameter, and x is the unknown – a difference of status that one might want to stress by writing $x \mapsto x.x^2 + 3mx + 4$, or $\lambda x.x^2 + 3mx + 4$ in the notation of the λ-calculus.

The next important thing to know about the λ-calculus is its dynamics: λ-terms are programs, and hence should be executable. The theoretical model for this is by successive transformations, or *rewritings*, of the term, using again and again the following unique rule, called β-reduction:

$$(\lambda x.M)N \to M[x \leftarrow N]$$

(for example, $(\lambda x.x^2 + 3mx + 4)(5 + y) \to (5 + y)^2 + 3m(5 + y) + 4)$. This is the only computational rule of the *pure* λ-calculus, i.e., without constants. Generally, when constants are added, corresponding rules are given (for example, for an addition constant $+$, one has, say, $3 + 4 \to 7$).

The β-rule can be applied to any subterm of our working term, and hence gives rise to a number of possible rewriting paths. For example, if $N = (\lambda x.P)Q$, then we can reduce $(\lambda x.M)N$ either to $M[x \leftarrow N]$ or to $(\lambda x.M)N'$, where $N' = P[x \leftarrow Q]$. The first key theorem of the λ-calculus is the confluence theorem, which says that no matter which paths are used, they can be made to converge to a same term: if M rewrites (in a number of steps) to M' and to M'', then there exists a term M''' which can be reached by rewriting both from M' and from M''.

We also mentioned a further equality above, the η rule, which is the following:

$$\lambda x.Mx = M$$

where x is chosen so as not to appear free in M. (The notions of free and bound variables are rather straightforward, e.g. x is bound and y is free in $\lambda x.xy$.)

This rule is quite different from β. Its primary purpose is to assert that (in the untyped λ-calculus) every term is a function. Thus, the most interesting way to look at this equality is to orient it from right to left: this is called η-expanding). Curien and Herbelin's contribution sheds some light on η-expansion, and gives it some computational meaning.

There are more practical models of the dynamics of the λ-calculus than the one given by the notion of β-reduction, that are formalised through an *abstract machine*, like the CAM or the KAM (cf. above). While we refer to the respective papers for their precise description, we just mention here that they share a common structure. Computation proceeds by successibely rewriting triples, or *states*, or *processes*, as Krivine calls them, of the form

$$(\text{term}, \text{environment}, \text{stack})$$

The environment is there to avoid actually performing the (costful) substitution $M[x \leftarrow N]$. Typically, starting from

$$((\lambda x.M)N, \text{empty}, \text{empty})$$

(empty environment and empty stack), we reach (roughly – we only want to give an idea here)

$$(M, [x \leftarrow N], \text{empty}) \ .$$

where $[x \leftarrow N]$ has now the meaning of storing the value N for x. Later on, when we reach a variable, we consult its value in the environment: so if we reach a state whose first two components are x and $[\ldots, x \leftarrow N, \ldots]$, then the machine proceeds by replacing x by N in the first component (variable look-up). Formally, the machine will thus proceed from

$$(x, [\ldots, x \leftarrow N, \ldots], S)$$

to

$$(N, [\ldots, x \leftarrow N, \ldots], S)$$

The third component serves to store the context of a computation. Typically, if the actual β-reduction is applied to a subterm $(\lambda x.P)Q$ of M – a situation that we can write formally as $M = C[(\lambda x.P)Q]]$, where C is a context, i.e. a λ-term with a hole, which is filled here with $(\lambda x.P)Q$ – , then the abstract machine will lead us, typically, from

$$(M, \text{empty}, \text{empty})$$

to

$$((\lambda x.P)Q, \text{empty}, C[]) \ .$$

For example, in Krivine abstract machine, if C has the form $([]N_1)N_2$, and hence $M = (((\lambda x.P)Q)N_1)N_2$, then starting from $(M, \text{empty}, \text{empty})$, one reaches $(((\lambda x.P)Q)N_1, \text{empty}, []N_2]$, and then $((\lambda x.P)Q, \text{empty}, [[]N_1]N_2])$ (and then, cf. above, $(P, [x \leftarrow Q], [[]N_1]N_2])$). Thus we use the stack to store the context, which is accumulated gradually. Read $[[]N_1]N_2]$ as the context obtained from the context $[]N_2$ by placing in its hole the context $[]N_1$, which results in the context $C = ([]N_1)N_2$, which in English spells as "apply to N_1, and then to N_2.

The reader will see two different simplified versions of such abstract machines in this volume:

- In Krivine's paper, the environment component is omitted. The machine is extended to deal with various new instructions that are added to the λ-calculus.
- In Curien and Herbelin's paper (see in particular section 5.2), in order to stress the duality between the term and the environment, the framework is adapted so as to avoid the use of a stack.

Also, both in Curien-Herbelin's and in Krivine's papers, the computing devices, or the programs, receive a natural interpretation in a two-player's game:

- In Curien-Herbelin's paper, this idea is so basic that it has guided the first author in the design of a generalization of the λ-calculus (first described in [2]) that encompasses various common extensions of the λ-calculus. We explain briefly the idea. The reader may easily check that a λ-term P in normal form (i.e., which cannot be further reduced) is made of bricks of the form

  $$\lambda x_1.(\lambda x_2.\ldots.(\lambda x_m.(\ldots(yP_1)\ldots P_n))\ldots)$$

  where the P_i's are themselves (hereditarily) of this form. Such a brick is called a head normal form, and y, which is the most important information in the brick, is called the head variable. The brick can itself be divided in two "moves":

  - a move "$\lambda x_1.(\lambda x_2;\ldots.(\lambda x_m$" made by a player called *Opponent* (or *attacker*, or *context*),
  - and a move y made by the other player, called *Player* (or *defendent*, or *program*).

  The Opponent's move reads as a question: "what is the head variable of P?", while the Player's move reads as an answer to this question. Then the Opponent may pick up one of the i's, and ask the same question relative to P_i, etc...

  Potentially infinite normal forms as above in the λ-calculus are called Böhm trees.

- In Krivine's paper, it is shown that all the proofs (in fact, all the realisers, see below for this notion) of an arithmetic formula of the form

  $$\exists x \forall y.\ f(x, y) = 0$$

  behave like the following stategy: the defendent (or proof) plays an $x = m_0$, the attacker then tries a $y = n_0$. If $f(m_0, n_0) = 0$, then the attacker failed in disproving the formula, and the game is over. Otherwise, the play goes on: the player chooses a new m_1, and so on. Of course it is not through a single play of this kind that the formula can be fully proved. But the fact that the attacker looses in all possible plays characterises the validity of the formula.

We end this introduction with a short hint on types and on the logical background of the λ-calculus. Originally, the λ-calculus served to describe formally languages of formulas, for example $A \Rightarrow B$ is represented by the term $(\Rightarrow A)B$, where $\Rightarrow$ is a constant. But a tighter connection, known as the Curry-Howard isomorphism, arises when λ-terms are used to denote not formulas, but *proofs* of formulas: then we restrict attention to λ-terms denoting proofs, that are called *typed* λ-terms. We illustrate this here only with a trivial example: $\lambda x.x$ denotes the identity function, but it also denotes a proof of $A \Rightarrow A$, for any A. In typed λ-calculus, terms come with a type (which can be explicit, or can be reconstructed from type informations in the term, or can be inferred). More precisely, typed λ-terms are formalised as so-called judgements of the form

$$x_1 : A_1, \ldots, x_n : A_n \vdash M : A$$

where $x_1, \ldots, x_n$ include the free variables of M. From there, the categorical reading (alluded to above) is easy: M is (interpreted as) a *morphism* from $A_1 \times \ldots \times A_n$ to A. In other words, formulas correspond to objects and λ-terms correspond to morphisms in a suitable category (with products). This is very much the starting point of Melliès' contribution to this volume.

There are less typed λ-terms than untyped ones: in other words, not all λ-terms are typable. A second key theorem of (typed) λ-calculus is that typed terms M terminate, i.e. there is no infinite reduction sequence

$$M \rightarrow M_1 \rightarrow \ldots \rightarrow M_n \rightarrow \ldots$$

Since some λ-terms (like $(\lambda x.xx)(\lambda x.xx)$) do not terminate, not all λ-terms are typable.

Also, in some sense, when remaining in the untyped realm, there are more "types" (and more "typed terms"). While in typed λ-calculi there is a fixed, extrinsic syntax of types (or formulas), in untyped λ-calculus one can *define* types intrisically (or internally) as sets of (untyped) terms "behaving the same way". More precisely, and equivalently, a type U is given by *any* set of contexts, and the terms "of that type", or *realizing* that type, are the terms M such that

$$\text{for any } C[] \in U, C[M] \in \perp\!\!\!\perp,$$

where $\perp\!\!\!\perp$ is a fixed set of terms (actually, of processes – read $C[M]$ as a state (M, C) in Krivine stack-free abstract machine, cf. above). Saying that $C[M]$ belongs to $\perp\!\!\!\perp$ formalises the fact M and $C[]$ "get along" well, or *socialise*, as Girard would say. The notion of "term of type A" (above) and "term realizing (the interpretation of) A" are related by inclusion of the former concept in the latter one (a result which is called Adequation by Krivine). The set $\perp\!\!\!\perp$ can itself vary (and it does vary in the different applications of the idea described in Krivine's paper), but then we get a different whole model.

These models are called realizability models. The general idea goes back to Kleene (see e.g. [4]), whose motivation was to connect formally logic and recursive function theory. Krivine makes an intensive use of the flexibility offered by the whole framework, with the aim of associating λ-terms, that is, computer programs, with mathematical statements: every theorem is a specification, all its realisers behave according to this specification, and in many cases this behaviour can be described in illuminating terms. The flexibility is also offered by the possibility of adding insightful new constants to realise different axioms, the most challenging being the axiom of choice.

Just as there are untyped and typed λ-calculi, there are typed and untyped approaches to games and strategies. We mentioned two untyped approaches above. In the final section of his contribution, Melliès constructs a game model of linear logic, by assigning games to types, and typed strategies (i.e., strategies in these games) to terms. His approach permits to display the algebraic structure of the underlying game semantics, i.e., of the game interpretation of proof systems and programming languages.

References

[1] G. Cousineau, P.-L. Curien, and M. Mauny, The categorical abstract machine, Science of Computer Programming 8, 173-202 (1987).

[2] P.-L. Curien, Abstract Böhm trees, Mathematical Structures in Computer Science 8(6), 559-591(1998).

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

[4] S.C. Kleene, Realizability: a retrospective survey, in Proc. Cambridge Summer School in Mathematical Logic, Lecture Notes in Mathematics 337, 95-112, Springer (1973).

[5] J.-L. Krivine, A call-by-name lambda-calculus machine, Higher-Order and Symbolic Computation 20, p.199-207 (2007).