REALIZABILITY IN CLASSICAL LOGIC

Jean-Louis Krivine

Panoramas et Synthèses

Numéro 27

2009

SOCIÉTÉ MATHÉMATIQUE DE FRANCE
Publié avec le concours du Centre national de la recherche scientifique

Panoramas & Synthèses
27, 2009, p. 199–231

REALIZABILITY IN CLASSICAL LOGIC

by

Jean-Louis Krivine

Abstract. – We build a framework, now called "classical realizability", to explore the Curry-Howard (proof-program) correspondence.

The "programming" side of the correspondence is an imperative low level language, based on a call-by-name lambda-calculus abstract machine. The "proof" side is usual formalization of Analysis in second order logic.

In this way, we can get programs for all classical proofs of theorems in Analysis. For some of them, we solve the "specification problem": find the common behavior of all these programs, for a given theorem.

We show that the axiom of dependent choice (essential for Analysis) corresponds to very simple programming instructions: clock and signature.

***Résumé* (Réalisabilité en logique classique).** – On construit un cadre, appelé maintenant « réalisabilité classique », pour explorer la correspondance de Curry-Howard (preuves-programmes).

Le côté « programmation » de la correspondance est un langage impératif de bas niveau, basé sur une machine lambda-calcul en appel par nom. Le côté « preuve » est la formalisation habituelle de l'Analyse en logique du second ordre.

On obtient ainsi des programmes pour toutes les preuves classiques de theorèmes d'Analyse. Pour certains d'entre eux, on résout le problème de la « spécification » : trouver le comportement commun de tous ces programmes, pour un théorème donné.

On montre que l'axiome du choix dépendant (qui est essentiel pour l'Analyse) correspond, en programmation, à des instructions très simples : l'horloge et la signature.

Introduction

The essential aim is to explore the *Curry-Howard correspondence*: we want to associate a program with each mathematical proof and to consider each theorem as a

2010 Mathematics Subject Classification. – 03B40, 68N18.
Key words and phrases. – Lambda-calculus, Curry-Howard isomorphism, proof-program correspondence, dependent choice axiom.

Lessons in Marseille-Luminy, may 2004 (last revision: june 27, 2005).

specification, i.e. to find the common behavior of the programs associated with every proof of this theorem. It is a very difficult and very interesting problem, which I call the "specification problem" in what follows.

In the first place, we must give the programming language in which we write these programs, and also explain in which way they are executed. This is done in the first section, it is the "program side" of the correspondence. As we shall see, this programming frame is very similar to usual *imperative* programming. As the theory develops, many usual and important notions of imperative and system programming will naturally appear, such as: storage and call-by-value for datas, pointers, signature of files, system clock, boot, ...

Then we must give a computational content to each logical rule and each axiom. We do this by means of *elementary instructions*. The instructions for the rules of intuitionistic propositional logic have been found long time ago, at the very discovery of the Curry-Howard correspondence; the programming language was Church's lambda-calculus. Then the instructions for intuitionistic *second order logic* were obtained and the programming language was still the same.

But mathematical proofs are not done in intuitionistic logic, not even in pure classical logic. We need *axioms* and the usual axiom systems are:

1. Second order classical Peano arithmetic with the axiom of dependent choice; this system is also called "Analysis".
2. Zermelo-Frænkel set theory with the axiom of choice.

In this paper, we consider the first case. We shall give new elementary instructions for the lacking axioms, which are: the excluded middle, the axiom of recurrence and the axiom of dependent choice. It is necessary, for that, to define an extension of lambda-calculus. We notice that some of these instructions are *incompatible with β-reduction*. Therefore the execution strategy is deterministic and is given in the form of a weak head reduction machine.

The same machine is used for Zermelo-Frænkel set theory. The instructions associated with the axioms of ZF are given in [**4**]. The full axiom of choice remained an open problem until recently (may 2005). The new instructions necessary for this axiom and also for the continuum hypothesis will be given in a forthcoming paper.

Terms, stacks, processes, execution

We denote by QP the set of *closed* λ-terms built with some set of constants which are called *instructions*; one of these instructions is denoted by cc. We shall denote the application of t to u by $(t)u$ or tu; the application of t to n arguments $u_1,\ldots,u_n$ is denoted by $(t)u_1\ldots u_n$ or $tu_1\ldots u_n$. Therefore, we have $tuv = (t)uv = (tu)v$ with this notation.

Elements of QP are called *proof-like terms*.

Let $\mathsf{L} \supset \mathsf{QP}$ be the set of *closed* λ-terms built with a new constant k and a set $\Pi_0 \neq \varnothing$ of new constants called *stack constants*.

A closed λ-term of L of the form $\mathsf{k}t_1 \dots t_n\pi_0$ with $n \in \mathbb{N}$, $t_1, \dots, t_n \in \mathsf{L}$ and $\pi_0 \in \Pi_0$ is called a *continuation*.

A λ_c-*term* is, by definition, a closed λ-term $\tau \in \mathsf{L}$ with the following properties:

- each occurrence of k in τ appears at the beginning of a subterm of τ which is a continuation;
- each occurrence of a stack constant in τ appears at the end of a subterm of τ which is a continuation.

Remark. – *Proof-like terms are therefore λ_c-terms which do not contain the symbol k or, which amounts to the same thing, which do not contain any stack constant.*

The set of λ_c-terms is denoted by Λ_c. In what follows, we almost always consider only λ_c-terms; so, they will be called simply "terms" or "closed terms".

Lemma 1. – i) $QP \subset \Lambda_c$;
ii) *If $tu \in \Lambda_c$ and $u \notin \Pi_0$, then $t \in \Lambda_c$ and $u \in \Lambda_c$;*
iii) *If $tu \in \Lambda_c$ and $u \in \Pi_0$, then tu is a continuation;*
iv) *If $\lambda x\, t, u \in \Lambda_c$, then $t[u/x] \in \Lambda_c$;*
v) *If $t_1, \dots, t_n \in \Lambda_c$ ($n \in \mathbb{N}$) and $\pi_0 \in \Pi_0$, then $(\mathsf{k}\, t_1 \dots t_n)\pi_0 \in \Lambda_c$.*

Proof. – i) Trivial.

ii) Consider an occurrence of k (resp. of a stack constant π_0) in t or u; it is therefore in tu. Since $tu \in \Lambda_c$, this occurrence is at the beginning (resp. at the end) of a continuation $\mathsf{k}t_1 \dots t_n\pi_0 = (\mathsf{k}t_1 \dots t_n)\pi_0$ which is a subterm of tu. Now $u \neq \pi_0$, so that this subterm is not tu itself; thus, $\mathsf{k}t_1 \dots t_n\pi_0$ is a subterm of t or u.

iii) We have $u = \pi_0 \in \Pi_0$ and this occurrence of π_0 is at the end of a subterm of $t\pi_0$ which is a continuation $(\mathsf{k}t_1 \dots t_n)\pi_0$. Therefore, this subterm is $t\pi_0$ itself.

iv) Consider an occurrence of k (resp. π_0) in $t[u/x]$; thus, it is in $\lambda x\, t$ or u. But $\lambda x\, t, u \in \Lambda_c$, so that this occurrence is at the beginning (resp. at the end) of a continuation $\mathsf{k}t_1 \dots t_n\pi_0$ which is a subterm of $\lambda x\, t$ or u. If this continuation is a subterm of u, it is also a subterm of $t[u/x]$. If it is a subterm of $\lambda x\, t$, then this occurrence of k (resp. π_0) in $t[u/x]$ is at the beginning (resp. at the end) of the subterm $\mathsf{k}t_1[u/x] \dots t_n[u/x]\pi_0$, which is a continuation.

v) Trivial. □

If $t_1, \dots, t_n \in \Lambda_c$ ($n \in \mathbb{N}$) and $\pi_0 \in \Pi_0$, the sequence $\pi = (t_1, \dots, t_n, \pi_0)$ is called a *stack* and is denoted by $t_1.t_2 \dots t_n.\pi_0$; the set of stacks is denoted by Π. The continuation $\mathsf{k}t_1 \dots t_n\pi_0$ is denoted by k_π. If $t \in \Lambda_c$ and $\pi = t_1.t_2 \dots t_n.\pi_0 \in \Pi$, then the stack $t.t_1.t_2 \dots t_n.\pi_0$ is denoted by $t.\pi$. Thus, the dot is an application of $\Lambda_c \times \Pi$ in Π.

Every term $\tau \in \Lambda_c$ is either an application, or an abstraction, or a constant *which is an instruction* (indeed, this term can be neither k, nor a stack constant, by definition of λ_c-terms). From lemma 1, it follows that, for every $\tau \in \Lambda_c$, we are in one and only one of the following cases:

i) τ is an application tu with $u \notin \Pi_0$ (and therefore $t, u \in \Lambda_c$);

ii) τ is an abstraction $\lambda x\, t$;
iii) τ is an instruction (particular case: $\tau = \mathsf{cc}$);
iv) τ is an application tu with $u \in \Pi_0$, i.e. $\tau = \mathsf{k}_\pi$ for some stack π.

A *process* is an ordered pair (τ, π) with $\tau \in \Lambda_c$, $\pi \in \Pi$. It is denoted by $\tau \star \pi$; τ is called the *head* or the *active part* of the process. The set of all processes will be denoted by $\Lambda_c \star \Pi$.

We describe now the execution of processes, which is denoted by $\succ$. We give the rule to perform *one execution step* of the process $\tau \star \pi$. It depends on the form (i), ..., (iv) of τ given above. In the four rules that follow, $t, u, \lambda x\, v$ denote elements of Λ_c; π, ρ denote stacks.

i) $tu \star \pi \succ t \star u.\pi$ (push);
ii) $\lambda x\, v \star u.\pi \succ v[u/x] \star \pi$ (pop);
iii) $\mathsf{cc} \star t.\pi \succ t \star \mathsf{k}_\pi.\pi$ (store the stack);
the rule for other instructions will be given in due time;
iv) $\mathsf{k}_\pi \star t.\rho \succ t \star \pi$ (restore the stack).

We say that the process $t \star \pi$ *reduce to* $t' \star \pi'$ (notation $t \star \pi \succ t' \star \pi'$) if we get $t' \star \pi'$ from $t \star \pi$ by means of a finite (possibly null) number of execution steps.

Remark. – *By lemma 1, we can check that the four execution rules give processes when applied to processes.*

Truth values and types. – Consider an arbitrary set of processes, which is denoted by $\perp\!\!\!\perp$ and which we suppose cc-*saturated*; it means that:

$$t \star \pi \in \perp\!\!\!\perp, t' \star \pi' \succ t \star \pi \Rightarrow t' \star \pi' \in \perp\!\!\!\perp.$$

$\mathscr{P}(\Pi)$ is called the *set of truth values*. If $U \subset \Pi$ is a truth value and $t \in \Lambda_c$, we say that t *realizes* U (notation $t \Vdash U$) if $(\forall \pi \in U)\, t \star \pi \in \perp\!\!\!\perp$. The set $\{t \in \Lambda_c;\ t \Vdash U\}$ will be denoted by $|U|$. Thus, we have $|\bigcup_{i\in I} U_i| = \bigcap_{i\in I} |U_i|$.

The truth value $\varnothing$ (resp. Π) is called *true* (resp. *false*) and denoted by $\top$ (resp. $\bot$). Thus, we have $|\top| = \Lambda_c$; $t \in |\bot| \Leftrightarrow t \star \pi \in \perp\!\!\!\perp$ for every stack $\pi \in \Pi$.

Whenever U, V are truth values, we define the truth value:

$(U \to V) = \{t.\pi;\ t \Vdash U,\ \pi \in V\}$; we put $\neg U = (U \to \bot)$.

We shall sometimes use the following notation, when V is a truth value and $A \subset \Lambda_c$:

$$(A \to V) = \{t.\pi; t \in A,\ \pi \in V\}.$$

For example, $\{t\} \to V$ is the truth value $\{t.\pi;\ \pi \in V\}$ if $t \in \Lambda_c$ and $V \subset \Pi$.

With this notation, $U \to V$ is the same as $|U| \to V$, for $U, V \subset \Pi$.