Forcing as a program transformation

Alexandre Miquel
Laboratoire d'Informatique du parallélisme
École Normale Supérieure de Lyon
46, allée d'Italie.
69364 Lyon Cedex 07, France
E-mail: alexandre.miquel@ens-lyon.fr

***Abstract*—This paper is a study of the forcing translation through the proofs as programs correspondence in classical logic, following the methodology introduced by Krivine in [12], [14]. For that, we introduce an extension of (classical) higher-order arithmetic suited to express the forcing translation, called $\mathrm{PA}\omega^+$. We present the proof system of $\mathrm{PA}\omega^+$—based on Curry-style proof terms with call/cc—as well as the corresponding classical realizability semantics. Then, given a poset of conditions (represented in $\mathrm{PA}\omega^+$ as an upwards closed subset of a fixed meet semi-lattice), we define the forcing translation $A \mapsto (p \Vdash A)$ (where A ranges over propositions) and show that the corresponding transformation of proofs is induced by a simple program transformation $t \mapsto t^*$ defined on raw proof-terms (i.e. independently from the derivation). From an analysis of the computational behavior of transformed programs, we show how to avoid the cost of the transformation by introducing an extension of Krivine's abstract machine devoted to the execution of proofs constructed by forcing. We show that this machine induces new classical realizability models and present the corresponding adequacy results.**

I. Introduction

Forcing has been introduced by Cohen [3], [4] in order to build models of set theory that fulfil the negation of continuum hypothesis, thus proving its independence w.r.t. the axioms of ZFC. (The relative consistency of continuum hypothesis was already proved by Gödel [7] introducing constructible sets.) Since then, the theory of forcing has been widely investigated, and it now constitutes a standard tool of model theory.

On the other hand, the method of forcing has received much less attention in proof theory, especially in the perspective of analysing the computational contents of proofs by forcing. One reason for this is that the correspondence between proofs and programs (a.k.a. the Curry-Howard correspondence) was for a long time limited to intuitionistic logic and to constructive mathematics, so that proof-theoretic analyses of forcing [8], [1], [5] could only be carried out indirectly—through negative translations of classical logic to intuitionistic logic—and in logical frameworks whose proof-theoretic strength is way below the strength of Zermelo-Fraenkel set theory.

Following the discovery [9] of a connection between classical reasoning principles and control operators—which led to the extension of the Curry-Howard correspondence to classical logic—Krivine introduced the theory of *classical realizability* [13], which is a reformulation of Kleene's realizability [10] in which the computational contents of classical proofs can be analysed directly, and not through a negative translation.

Recently, Krivine showed how to combine forcing with classical realizability [12], [14], and discovered the existence of a simple program transformation (defined on classical λ-terms) that turns any Curry-style proof-term t of a formula A (in PA2/PA3) into a classical realizer of the formula $p \Vdash A$ in the suitable realizability model (where p is an arbitrary condition). From this, he deduced a method to build a realizer of a theorem whose proof depends on an axiom that can be forced using a suitable set of conditions.

The aim of this paper is to present and study (a variant of) this program transformation in higher-order arithmetic, using a fully typed setting. For that, we shall present an extension of higher-order arithmetic with classical proof terms, called $\mathrm{PA}\omega^+$, and define the forcing relation $p \Vdash A$ in this framework. However, the forcing relation has to be designed carefully throughout the hierarchy of finite types, since we want that the corresponding transformation on typing derivations can be lifted at the level of Curry-style proof terms, that contain much less information.

Finally, we shall study the computational behavior of transformed classical λ-terms, and we will deduce from this analysis a way to internalize forcing into Krivine's Abstract Machine (KAM) by introducing a special execution mode for proofs-by-forcing, in which the originating proof does not need to be transformed anymore. We shall also present the realizability models coming with this new abstract machine, together with the corresponding adequacy results.

Contribution of the paper: This work is largely inspired by the methodology introduced by Krivine in [12], [14]. The author's own contributions are the following:

- A reformulation of the forcing translation in higher-order arithmetic (rather than in PA2/PA3), and the design of an expressive type system ($\mathrm{PA}\omega^+$) in which the transformation preserves typability on proof-terms. (In [12], [14], well-typed terms are only mapped to realizers.)
- Some simplifications in the program transformation presented by Krivine. In particular, we get rid of the two extra instructions χ and χ' used in [12], [14].
- A new abstract machine, the KFAM, deduced from a computational analysis of the program transformation. This abstract machine extends Krivine's with a special execution mode devoted to the evaluation of proofs by forcing, thus removing the need of the transformation.
- The classical realizability models induced by the KFAM,

as well as the corresponding adequacy results.

II. An extension of higher-order arithmetic

Throughout this paper, we work in a presentation of higher-order arithmetic called PAω^+, that is basically an extension of (Curry-style) system $F\omega$. As for system $F\omega$, system PAω^+ is stratified into three syntactic categories: *kinds*, *higher-order terms* (that correspond to mathematical objects, including propositions) and *(Curry-style) proof-terms*.

A. Kinds

Kinds (notation: τ, σ, etc.) of PAω^+ are given by the BNF:

Kinds $\qquad \tau, \sigma \quad ::= \quad \iota \quad | \quad o \quad | \quad \tau \to \sigma$

Here, ι denotes the kind of *individuals*, o the kind of *propositions*, and $\tau \to \sigma$ is the kind of *functions* from τ to σ.

B. Higher-order terms

1) Definition: We assume given an infinite set of variables (notation: x^τ, y^τ, z^τ, etc.) for every kind τ. Higher-order terms (notation: M, N, etc.) of PAω^+ are 'simply kinded' λ-terms enriched with extra constructions to represent arithmetic operations and logical constructions. Formally:

Definition 1 (Higher-order terms): — Higher-order terms of all kinds are inductively defined as follows:

(Lambda-calculus)

- If x^τ is a variable of kind τ, then x^τ is a term of kind τ.
- If x^τ is a variable of kind τ and if M is a term of kind σ, then $\lambda x^\tau . M$ is a term of kind $\tau \to \sigma$.
- If M is a term of kind $\tau \to \sigma$ and if N is a term of kind τ, then MN is a term of kind σ.

(Arithmetic constructions)

- The constant 0 ('zero') is a term of kind ι.
- The constant s ('successor') is a term of kind $\iota \to \iota$.
- For every kind τ, the constant rec_τ ('recursor') is a term of kind $\tau \to (\iota \to \tau \to \tau) \to \iota \to \tau$.

(Logical constructions)

- If M and N are terms of kind o, then $M \Rightarrow N$ is a term of kind o.
- If M is a term of kind o possibly depending on a variable x of kind τ, then $\forall x^\tau M$ is a term of kind o.
- If M and M' are terms of kind τ, and if N is a term of kind o, then $\langle M = M' \rangle N$ is a term of kind o. This ternary construction represents an *equational implication* whose meaning will be given below (section II-B3).

In what follows, we shall write $FV(M)$ the set of free variables of M, and $M\{x^\tau := N\}$ the term obtained by replacing in the term M (of some kind σ) all the free occurrences of the variable x^τ with the term N (of kind τ).

2) Propositions: We call a *proposition* any term A of kind o, preferring the letters A, B, C, etc. to denote them. We use the standard second-order encodings [6] to represent negation, conjunction, disjunction, existential quantification and Leibniz equality.

3) Equational implication: The intuitive meaning of the proposition $\langle M = M' \rangle A$ is:

$$\langle M = M' \rangle A \qquad \equiv \qquad \begin{cases} A & \text{if } M \text{ equals } M' \\ \top & \text{otherwise.} \end{cases}$$

(Here, $\top$ denotes the proposition proved by any proof-term, that will be formally defined in section II-E3.) As suggested by its name, the equational implication $\langle M = M' \rangle A$ is provably equivalent to the implication $M =_\tau M' \Rightarrow A$, where the symbol $=_\tau$ stands for Leibniz equality (see above).

In practice, the proposition $\langle M = M' \rangle A$ carries over the same logical contents as the proposition $M =_\tau M' \Rightarrow A$, but with more compact proof terms. While this compact form of an implication is not strictly needed to define the forcing translation, it helps to make the translation more understandable at the level of proof terms. However, the presence of this extra construction has a cost on the type system of PAω^+, since it makes the typing judgment $\mathcal{E}; \Gamma \vdash t : A$ not only depend on a typing context Γ, but also on an equational theory $\mathcal{E}$.

C. The congruence $M \cong_\mathcal{E} M'$

We call an *equational theory* $\mathcal{E}$ any finite list of the form

$$\mathcal{E} \;\equiv\; M_1 = M'_1; \ldots; M_k = M'_k$$

where for all $i \in [1..k]$, M_i and M'_i are (open) higher-order terms of the same kind τ_i. (For simplicity, we assume that the equations $M = M'$ are non oriented.) Given two equational theories $\mathcal{E}$ and $\mathcal{E}'$, we write $\mathcal{E} \subseteq \mathcal{E}'$ when all equations of $\mathcal{E}$ also appear in $\mathcal{E}'$ (not necessarily in the same order).

Every equational theory $\mathcal{E}$ induces a relation $M \cong_\mathcal{E} M'$ between higher-order terms M and M' of the same kind. Formally, the family of relations $\cong_\mathcal{E}$ (where $\mathcal{E}$ ranges over all equational theories) is inductively defined from the rules given in Fig. 1. We easily check that:

Proposition 1 (Monotonicity and congruence):

1) If $M_1 \cong_\mathcal{E} M_2$ and $\mathcal{E} \subseteq \mathcal{E}'$, then $M_1 \cong_{\mathcal{E}'} M_2$.
2) For all equational theories $\mathcal{E}$, the relation $M \cong_\mathcal{E} M'$ is a congruence.

D. The proof-system of PAω^+

1) Proof-terms and typing contexts: Raw proof-terms (notation: t, u, etc.) of PAω^+ are pure λ-terms enriched with a constant $\mathtt{c\!c}$ (call/cc, for *call with current continuation*):

Proof-terms $\qquad t, u \quad ::= \quad x \quad | \quad \lambda x . t \quad | \quad tu \quad | \quad \mathtt{c\!c}$

(Here, x, y, z, etc. denote *proof-variables* that shall not be confused with higher-order variables x^τ, y^σ, z^ρ, etc. we introduced in II-B.) The set of free variables of a proof-term t is written $FV(t)$ and the corresponding operation of substitution, written $t\{x := u\}$, is defined as expected.

Typing contexts (notation: Γ, Γ', etc.) are finite ordered lists of the form

Typing contexts $\qquad \Gamma \quad ::= \quad x_1 : A_1, \;\ldots,\; x_n : A_n$

where $x_1, \ldots, x_n$ are pairwise distinct proof-variables and where $A_1, \ldots, A_n$ are arbitrary propositions. Given a typing

Reflexivity, symmetry, transitivity and base case

$$\frac{}{M \cong_{\mathcal{E}} M} \qquad \frac{}{M \cong_{\mathcal{E}} M'}\ {\scriptstyle (M=M')\in\mathcal{E}}$$

$$\frac{M \cong_{\mathcal{E}} M'}{M' \cong_{\mathcal{E}} M} \qquad \frac{M \cong_{\mathcal{E}} M' \quad M' \cong_{\mathcal{E}} M''}{M \cong_{\mathcal{E}} M''}$$

Context closure

$$\frac{M \cong_{\mathcal{E}} M'}{\lambda x^\tau . M \ \cong_{\mathcal{E}}\ \lambda x^\tau . M'} \qquad \frac{M \cong_{\mathcal{E}} M' \quad N \cong_{\mathcal{E}} N'}{MN \ \cong_{\mathcal{E}}\ M'N'}$$

$$\frac{A \cong_{\mathcal{E}} A' \quad B \cong_{\mathcal{E}} B'}{A \Rightarrow B \ \cong_{\mathcal{E}}\ A' \Rightarrow B'} \qquad \frac{A \cong_{\mathcal{E}} A'}{\forall x^\tau A \ \cong_{\mathcal{E}}\ \forall x^\tau A'}$$

$$\frac{M_1 \cong_{\mathcal{E}} M_1' \quad M_2 \cong_{\mathcal{E}} M_2' \quad A \cong_{\mathcal{E};M_1=M_2} A'}{\langle M_1 = M_2\rangle A \ \cong_{\mathcal{E}}\ \langle M_1' = M_2'\rangle A'}$$

β-reduction, η-reduction and recursion

$$\frac{}{(\lambda x^\tau . M)N \cong_{\mathcal{E}} M\{x^\tau := N\}}$$

$$\frac{}{\lambda x^\tau . Mx \ \cong_{\mathcal{E}}\ M}\ {\scriptstyle x^\tau \notin FV(M)}$$

$$\frac{}{\mathrm{rec}_\tau\, M\, M'\, 0 \ \cong_{\mathcal{E}}\ M}$$

$$\frac{}{\mathrm{rec}_\tau\, M\, M'\, (s\, N) \ \cong_{\mathcal{E}}\ M'\, N\, (\mathrm{rec}_\tau\, M\, M'\, N)}$$

Computationally equivalent propositions

$$\frac{}{\forall x^\tau \forall y^\sigma A \ \cong_{\mathcal{E}}\ \forall y^\sigma \forall x^\tau A} \qquad \frac{}{\forall x^\tau A \ \cong_{\mathcal{E}}\ A}\ {\scriptstyle x^\tau \notin FV(A)}$$

$$\frac{}{A \Rightarrow \forall x^\tau B \ \cong_{\mathcal{E}}\ \forall x^\tau (A \Rightarrow B)}\ {\scriptstyle x^\tau \notin FV(A)}$$

$$\frac{}{\langle M = M\rangle A \ \cong_{\mathcal{E}}\ A} \qquad \frac{}{\langle M = M'\rangle A \ \cong_{\mathcal{E}}\ \langle M' = M\rangle A}$$

$$\frac{}{\langle M = M'\rangle\langle N = N'\rangle A \ \cong_{\mathcal{E}}\ \langle N = N'\rangle\langle M = M'\rangle A}$$

$$\frac{}{A \Rightarrow \langle M = M'\rangle B \ \cong_{\mathcal{E}}\ \langle M = M'\rangle(A \Rightarrow B)}$$

$$\frac{}{\forall x^\tau \langle M = M'\rangle A \ \cong_{\mathcal{E}}\ \langle M = M'\rangle \forall x^\tau A}\ {\scriptstyle x^\tau \notin FV(M,M')}$$

Fig. 1. Inference rules of the relation $M \cong_{\mathcal{E}} M'$

$$\frac{}{\mathcal{E};\Gamma \vdash x : A}\ {\scriptstyle (x:A)\in\Gamma} \qquad \frac{\mathcal{E};\Gamma \vdash t : A}{\mathcal{E};\Gamma \vdash t : A'}\ {\scriptstyle A \cong_{\mathcal{E}} A'}$$

$$\frac{\mathcal{E};\Gamma, x : A \vdash t : B}{\mathcal{E};\Gamma \vdash \lambda x . t : A \Rightarrow B}$$

$$\frac{\mathcal{E};\Gamma \vdash t : A \Rightarrow B \qquad \mathcal{E};\Gamma \vdash t : A}{\mathcal{E};\Gamma \vdash tu : B}$$

$$\frac{\mathcal{E}, M = M';\Gamma \vdash t : A}{\mathcal{E};\Gamma \vdash t : \langle M = M'\rangle A} \qquad \frac{\mathcal{E};\Gamma \vdash t : \langle M = M\rangle A}{\mathcal{E};\Gamma \vdash t : A}$$

$$\frac{\mathcal{E};\Gamma \vdash t : A}{\mathcal{E};\Gamma \vdash t : \forall x^\tau A}\ {\scriptstyle x \notin FV(\Gamma)} \qquad \frac{\mathcal{E};\Gamma \vdash t : \forall x^\tau A}{\mathcal{E};\Gamma \vdash t : A\{x := N^\tau\}}$$

$$\frac{}{\mathcal{E};\Gamma \vdash \text{œ} : ((A \Rightarrow B) \Rightarrow A) \Rightarrow A}$$

Fig. 2. Deduction/typing rules of system PAω^+

context $\Gamma \equiv x_1 : A_1,\ \ldots,\ x_n : A_n$, we write $\mathrm{dom}(\Gamma) = \{x_1; \ldots; x_n\}$ and $FV(\Gamma) = FV(A_1) \cup \cdots \cup FV(A_n)$.

2) Typing rules: The proof system of system PAω^+ is based on a typing judgment of the form $\mathcal{E};\Gamma \vdash t : A$ ('in the equational theory $\mathcal{E}$ and the context Γ, the λ-term t is a proof-term of A') that is defined from the rules of Fig. 2.

In what follows, we shall say that a typing rule—or a deduction step—is *computationally transparent* if it does not affect the current proof-term. Most of the rules of Fig. 2 are computationally transparent: the conversion rule, the introduction and elemination rules of universal quantification, as well as the introduction and elimination rules of equational implication.

E. Expressiveness

1) Encoding arithmetic reasoning: From the rules of Fig. 2 one can derive the usual introduction and elimination rules of falsity, negation, conjunction, disjunction, existential quantification and Leibniz equality using the encodings given in the end of Section II-B. The typing rule of œ implements Peirce's law, from which we can derive the excluded middle as well as other classical reasoning principles.

Using the recursor rec^τ, we can easily encode the predecessor and nullity test functions from which one easily derives that the successor function is injective and non surjective using appropriate conversions. Finally, reasoning by induction is mimicked (as in [6], [11]) by relativizing all quantifications of kind ι using the predicate $\mathrm{nat} : \iota \to o$ defined by

$$\mathrm{nat} \ \equiv\ \lambda x^\iota . \forall Z\, (Z\, 0 \Rightarrow \forall y\, (Z\, y \Rightarrow Z\, (s\, y)) \Rightarrow Z\, x)\,.$$

2) Equivalence of $\langle M = M'\rangle A$ and $M =_\tau M' \Rightarrow A$: It is easy to check that for all terms M, M' (of kind τ) and A (of kind o), the propositions $\langle M = M'\rangle A$ and $M =_\tau M' \Rightarrow A$ are provably equivalent in PAω^+:

$$\begin{array}{lll} \vdash & \lambda xy\, .\, y\, x \quad : & (\langle M = M'\rangle A) \ \Rightarrow\ M =_\tau M' \Rightarrow A \\ \vdash & \lambda x\, .\, x\, (\lambda y\, .\, y) \quad : & (M =_\tau M' \Rightarrow A) \ \Rightarrow\ \langle M = M'\rangle A \end{array}$$

3) The proposition $\top$: We can define the proposition $\top$ as $\top \equiv \langle \mathsf{tt} = \mathsf{ff}\rangle\bot$ where $\mathsf{tt} = \lambda x^o y^o . x$ and $\mathsf{ff} = \lambda x^o y^o . y$. Using the fact that in the equational theory $\mathcal{E} = \{\mathsf{tt} = \mathsf{ff}\}$, all propositions are convertible (since $A \cong_{\mathcal{E}} \mathsf{tt}\, A\, A' \cong_{\mathcal{E}} \mathsf{ff}\, A\, A' \cong_{\mathcal{E}} A'$ for all propositions A and A'), we easily get:

Lemma 1: — For all equational theories $\mathcal{E}$, for all contexts Γ and for all proof-terms t such that $FV(t) \subseteq \mathrm{dom}(\Gamma)$, the judgment $\mathcal{E};\Gamma \vdash t : \top$ is derivable.

As a consequence, all raw proof-terms are typable, hence system PAω^+ enjoys no normalization property.

III. Classical realizability semantics

The operational behavior of proof-terms can be described via a classical realizability model (based on the λ_c-calculus) that constitutes the natural higher-order extension of the model presented in [13]. (A similar model was presented in [16].)

A. The λ_c-calculus

The λ_c-calculus is defined from three kinds of syntactic entities: terms, stacks and processes, that are inductively defined as follows:

$$\begin{array}{llcll}
\textbf{Terms} & t, u & ::= & x \mid \lambda x \,.\, t \mid tu & \\
 & & \mid & \kappa \mid \mathsf{k}_\pi & (\kappa \in \mathcal{K}) \\
\textbf{Stacks} & \pi & ::= & \alpha \mid t \cdot \pi & (\alpha \in \mathcal{B},\ t \text{ closed}) \\
\textbf{Processes} & p, q & ::= & t \star \pi & (t \text{ closed})
\end{array}$$

Formally, the syntax of the λ_c-calculus is thus parameterized by two sets: a set $\mathcal{K}$ of *instructions*, that contains (at least) the instruction $\mathfrak{c}\!\mathfrak{c}$ ('call/cc'); and a nonempty set $\mathcal{B}$ of *stack constants* (or *stack bottoms*).

Terms (notation: t, u, etc.) of the λ_c-calculus are ordinary λ-terms enriched with constants of two forms: *instructions* $\kappa \in \mathcal{K}$, including the instruction $\mathfrak{c}\!\mathfrak{c}$ ('call/cc'), and *continuation constants* k_π, one for every stack π. In particular, the set of (open) λ_c-terms is a strict superset of the set of raw proof-terms introduced in Section II-D. *Stacks* (notation: π, π', etc.) are lists of closed terms terminated by a stack constant; in particular they are closed objects—so that continuation constants k_π are actually constant. Finally, *processes* are pairs $t \star \pi$ formed by a closed term t and a stack π.

In what follows, we respectively denote by Λ_c and Π the sets of all closed λ_c-terms and of all (closed) stacks. The set of all processes is written $\Lambda_c \star \Pi$.

1) Evaluation: We assume that the set of processes $\Lambda_c \star \Pi$ comes with a binary relation of *evaluation*, written $p \succ p'$, that fulfils the following axioms:

$$\begin{array}{lrcl}
\textsc{Grab} & \lambda x \,.\, t \star u \cdot \pi & \succ & t\{x := u\} \star \pi \\
\textsc{Push} & tu \star \pi & \succ & t \star u \cdot \pi \\
\textsc{Save} & \mathfrak{c}\!\mathfrak{c} \star t \cdot \pi & \succ & t \star \mathsf{k}_\pi \cdot \pi \\
\textsc{Restore} & \mathsf{k}_\pi \star t \cdot \pi' & \succ & t \star \pi
\end{array}$$

Note that these rules do not constitute a *definition* of evaluation, but a *specification* of what evaluation should do—at least. The relation of evaluation is a parameter of the calculus (as for the sets $\mathcal{K}$ and $\mathcal{B}$); and it may follow other rules describing the computational behavior of extra instructions $\kappa \in \mathcal{K}$.

B. The realizability interpretation

1) Poles: The definition of the classical realizability model for system $\mathrm{PA}\omega^+$ is actually parameterized by a *pole*, that is, by a set of processes $\perp\!\!\!\perp \subseteq \Lambda_c \star \Pi$ that is closed under anti-evaluation, in the sense that $p \succ p'$ and $p' \in \perp\!\!\!\perp$ imply $p \in \perp\!\!\!\perp$ for all $p, p' \in \Lambda_c \star \Pi$.

2) Truth and falsity values: We call a *falsity value* any set of stacks $S \subseteq \Pi$. Every falsity value $S \subseteq \Pi$ induces a truth value $S^{\perp\!\!\!\perp} \subseteq \Lambda_c$ that is defined by

$$S^{\perp\!\!\!\perp} \;=\; \{t \in \Lambda_c \;:\; \forall \pi \in S\ (t \star \pi) \in \perp\!\!\!\perp\}\,.$$

The larger the falsity value S, the smaller the truth value $S^{\perp\!\!\!\perp}$.

3) Interpreting kinds: In the classical realizability model (parameterized by the pole $\perp\!\!\!\perp$), individuals are interpreted as natural numbers, propositions as falsity values and higher-order functions as set-theoretic functions, that is:

$$[\![\iota]\!] = \mathbb{N}, \qquad [\![o]\!] = \mathfrak{P}(\Pi), \qquad [\![\tau \to \sigma]\!] \;=\; [\![\sigma]\!]^{[\![\tau]\!]}$$

A *valuation* is a function ρ that maps every variable x^τ of kind τ to an element $\rho(x^\tau) \in [\![\tau]\!]$. As usual, we denote by $\rho, x^\tau \leftarrow v$ the valuation obtained from the valuation ρ by rebinding the variable x^τ to the denotation $v \in [\![\tau]\!]$ (thus erasing the value previously bound to x^τ in ρ).

4) Interpreting higher-order terms: Every higher-order term M of kind τ is interpreted as an element $[\![M]\!]_\rho \in [\![\tau]\!]$ depending on a valuation ρ. Abstraction, application and arithmetic constructions are interpreted the obvious way using their set-theoretic equivalents:

$$\begin{array}{rclrcl}
[\![x]\!]_\rho & = & \rho(x) & [\![0]\!] & = & 0 \\
[\![\lambda x^\tau \,.\, M]\!]_\rho & = & (v \in [\![\tau]\!] \mapsto [\![M]\!]_{\rho, x \leftarrow v}) & [\![s]\!] & = & n \mapsto n+1 \\
[\![MN]\!]_\rho & = & [\![M]\!]_\rho([\![N]\!]_\rho) & [\![\mathrm{rec}_\tau]\!] & = & \mathsf{rec}_{[\![\tau]\!]}
\end{array}$$

(where $\mathsf{rec}_{[\![\tau]\!]}$ denotes the expected set-theoretic recursor over the set $[\![\tau]\!]$). Implication and universal quantification are given their standard negative interpretation following [13], and equational implication is interpreted following the informal explanation given in II-B3:

$$\begin{array}{rcl}
[\![A \Rightarrow B]\!]_\rho & = & [\![A]\!]_\rho^{\perp\!\!\!\perp} \cdot [\![B]\!]_\rho \\
 & = & \big\{t \cdot \pi \;:\; t \in [\![A]\!]_\rho^{\perp\!\!\!\perp},\ \pi \in [\![B]\!]_\rho\big\} \\
[\![\forall x^\tau A]\!]_\rho & = & \displaystyle\bigcup_{v \in [\![\tau]\!]} [\![A]\!]_{\rho, x \leftarrow v} \\
[\![\langle M = M' \rangle A]\!] & = & \begin{cases} [\![A]\!]_\rho & \text{if } [\![M]\!]_\rho = [\![M']\!]_\rho \\ \varnothing & \text{otherwise} \end{cases}
\end{array}$$

(Here the empty falsity value denotes the true proposition $\top$, i.e. the proposition that has non opponent.)

Since the denotation $[\![M]\!]_\rho$ (implicitly) depends on the pole $\perp\!\!\!\perp$, we shall write it sometimes $[\![M]\!]_{\perp\!\!\!\perp,\rho}$ to recall the dependency. Given a proposition A, a valuation ρ and a closed term t, we say that *t realizes A in the valuation ρ* and write $t \Vdash A[\rho]$ when $t \in ([\![A]\!]_{\perp\!\!\!\perp,\rho})^{\perp\!\!\!\perp}$. Again, this notion is relative to a particular pole $\perp\!\!\!\perp$. Finally, we say that *t universally realizes the proposition A in the valuation ρ* and write $t \Vvdash A[\rho]$ when $t \in ([\![A]\!]_{\perp\!\!\!\perp,\rho})^{\perp\!\!\!\perp}$ for all poles $\perp\!\!\!\perp$.

Again, this construction is nothing but the straightforward extension of Krivine's model to higher-order arithmetic, and formulas of second-order arithmetic $\mathrm{PA2} \subset \mathrm{PA}\omega^+$ are interpreted in our model the very same way as in [13]. In particular, the classical extraction techniques described in [15] are still valid in this framework.

C. Soundness properties

Lemma 2 (Substitution): — For all higher-order terms M^τ and N^σ and for all valuations ρ we have

$$[\![M\{x^\sigma := N\}]\!] \;=\; [\![M]\!]_{\rho, x^\sigma \leftarrow [\![N]\!]_\rho}\,.$$

We write $\rho \models \mathcal{E}$ when $[\![M]\!]_\rho = [\![M']\!]_\rho$ for all equations $(M = M') \in \mathcal{E}$. We easily check that:

Lemma 3 (Conversion): — If $M \cong_{\mathcal{E}} M'$, then for all valuations ρ such that $\rho \models \mathcal{E}$ we have $[\![M]\!]_\rho = [\![M']\!]_\rho$.

Proposition 2 (Adequacy): — If the judgement

$$\mathcal{E};\ x_1 : B_1, \ldots, x_n : B_n \vdash t : A$$

is derivable in system PAω^+, then for all valuations ρ such that $\rho \models \mathcal{E}$ and for all closed λ_c-terms $u_1, \ldots, u_n$ such that $u_1 \Vdash B_1[\rho], \ldots, u_n \Vdash B_n[\rho]$ we have

$$t\{x_1 := u_1; \ldots; x_n := u_n\} \Vdash A[\rho]\,.$$

In the particular case of an empty equational theory and of an empty context, the property of adequacy simply expresses that if $\vdash t : A$ is derivable in system PAω^+, then t is a universal realizer of A in any valuation ρ.

IV. Representing forcing conditions

We follow Krivine [12], [14] by representing forcing conditions as the elements of an upwards closed subset C of a meet semi-lattice $(\kappa, \cdot, 1)$ over a fixed sort κ. This slightly non standard presentation of forcing conditions will be justified by the computational analysis of the underlying program transformation we will present in section V-F, and we shall give an example of such a structure in section IV-C.

A. Forcing parameters

Formally, we fix a kind κ of *conditions* (notation: p, q, r, etc.) with a relativization predicate C of kind $\kappa \to o$ delimiting the class of *well-formed conditions*. Given a condition p, we write $C[p]$ ('p is a well-formed condition') for $C\,p$.

We assume that the kind κ comes with

- A term 1 (of kind κ) representing the largest condition.
- A binary operation (of kind $\kappa \to \kappa \to \kappa$) that associates to every pair of conditions p and q the *product of* p *and* q, written pq.

Finally, we assume that the predicate C comes with the following *combinators*, that is, proof-terms

$$\begin{array}{lcl}
\alpha_* & : & C[1] \\
\alpha_1 & : & \forall p^\kappa\ \forall q^\kappa\ (C[pq] \Rightarrow C[p]) \\
\alpha_2 & : & \forall p^\kappa\ \forall q^\kappa\ (C[pq] \Rightarrow C[q]) \\
\alpha_3 & : & \forall p^\kappa\ \forall q^\kappa\ (C[pq] \Rightarrow C[qp]) \\
\alpha_4 & : & \forall p^\kappa\ (C[p] \Rightarrow C[pp]) \\
\alpha_5 & : & \forall p^\kappa\ \forall q^\kappa\ \forall r^\kappa\ (C[(pq)r] \Rightarrow C[p(qr)]) \\
\alpha_6 & : & \forall p^\kappa\ \forall q^\kappa\ \forall r^\kappa\ (C[p(qr)] \Rightarrow C[(pq)r]) \\
\alpha_7 & : & \forall p^\kappa\ (C[p] \Rightarrow C[p1]) \\
\alpha_8 & : & \forall p^\kappa\ (C[p] \Rightarrow C[1p])
\end{array}$$

(This set of combinators is not minimal: for instance, α_2, α_6 and α_8 can be defined from the other combinators.)

Intuitively, these axioms express that the set C is upwards closed w.r.t. the ordering induced by the binary meet operation $(p, q) \mapsto pq$. (The preorder on conditions will be formally defined from these parameters in section IV-B.)

We say that two conditions p and q are *compatible* when $C[pq]$ holds. The proposition $C[pq]$ ('p and q are compatible') obviously implies that both conditions p and q are well-formed (from axioms α_1 and α_2), but the converse is not true in general. (This point is crucial in the definition of forcing.)

In what follows, we shall also need the following derived combinators:

$$\begin{array}{lcl}
\alpha_9 & \equiv & \alpha_3 \circ \alpha_1 \circ \alpha_6 \circ \alpha_3 \\
 & : & \forall p\ \forall q\ \forall r\ (C[(pq)r] \Rightarrow C[pr]) \\
\alpha_{10} & \equiv & \alpha_2 \circ \alpha_5 \\
 & : & \forall p\ \forall q\ \forall r\ (C[(pq)r] \Rightarrow C[qr]) \\
\alpha_{11} & \equiv & \alpha_9 \circ \alpha_4 \\
 & : & \forall p\ \forall q\ (C[pq] \Rightarrow C[p(pq)]) \\
\alpha_{12} & \equiv & \alpha_5 \circ \alpha_3 \\
 & : & \forall p\ \forall q\ \forall r\ (C[p(qr)] \Rightarrow C[q(rp)]) \\
\alpha_{13} & \equiv & \alpha_3 \circ \alpha_{12} \\
 & : & \forall p\ \forall q\ \forall r\ (C[p(qr)] \Rightarrow C[(rp)q]) \\
\alpha_{14} & \equiv & \alpha_5 \circ \alpha_3 \circ \alpha_{10} \circ \alpha_4 \circ \alpha_2 \\
 & : & \forall p\ \forall q\ \forall r\ (C[p(qr)] \Rightarrow C[q(rr)]) \\
\alpha_{15} & \equiv & \alpha_9 \circ \alpha_3 \\
 & : & \forall p\ \forall q\ \forall r\ (C[p(qr)] \Rightarrow C[qp])
\end{array}$$

B. Preorder on conditions

We define the relation $p \le q$ between conditions by letting

$$p \le q \;\equiv\; \forall r^\kappa\ (C[pr] \Rightarrow C[qr])\,.$$

It is easy to check that $\le$ is a preorder with greatest element 1

$$\begin{array}{lcl}
\lambda c\,.\,c & : & \forall p\ (p \le p) \\
\lambda xyc\,.\,y(xc) & : & \forall p\ \forall q\ \forall r\ (p \le q \Rightarrow q \le r \Rightarrow p \le r) \\
\alpha_8 \circ \alpha_2 & : & \forall p\ (p \le 1)
\end{array}$$

and that non well-formed conditions are smallest elements:

$$\lambda xc\,.\,x\,(\alpha_1\,c) \quad : \quad \forall p\ \forall q\ (\neg C[p] \Rightarrow p \le q)$$

(so that all non well-formed conditions are actually collapsed w.r.t. the equivalence induced by the preorder). Moreover, the product of two conditions is their least upper bound:

$$\begin{array}{lcl}
\alpha_9 & : & \forall p\ \forall q\ (pq \le p) \\
\alpha_{10} & : & \forall p\ \forall q\ (pq \le q) \\
\multicolumn{3}{l}{\lambda xy\,.\,\alpha_{13} \circ y \circ \alpha_{12} \circ x \circ \alpha_{11}} \\
 & : & \forall p\ \forall q\ \forall r\ (r \le p \Rightarrow r \le q \Rightarrow r \le pq)
\end{array}$$

C. An example of a set of conditions

The typical—and historical—example of a set of forcing conditions is the set of finite functions from a given set τ to the pair $\{0; 1\}$. (When τ is taken large enough, such a set of conditions can be used to force the negation of the continuum hypothesis [3], [4], [2].)

In our framework, this set is modelled as follows.

From a fixed sort τ, we define the sort $\kappa \equiv \tau \to \iota \to o$ of binary relations between objects of sort τ and ι. The product of two relations p and q is their union

$$pq \;\equiv\; \lambda xy\,.\,p\,x\,y \vee q\,x\,y$$

and the unit in the empty relation: $1 \equiv \lambda xy\,.\,\bot$. Well formed conditions are then defined as the binary relations $p : \kappa$ that are finite functions from τ to $\{0;1\}$, that is:

$$\begin{array}{rcll} C[p] & \equiv & \forall x^\tau\,\forall y^\iota\,(p\,x\,y \Rightarrow y = 0 \vee y = 1) & \wedge \\ & & \forall x^\tau\,\forall y_1^\iota\,\forall y_2^\iota\,(p\,x\,y_1 \Rightarrow p\,x\,y_2 \Rightarrow y_1 = y_2) & \wedge \\ & & p \text{ finite} & \end{array}$$

It is a straightforward exercise to check that for any two well-formed conditions p and q, the ordering $p \leq q$ defined in section IV-B coincides with the reverse inclusion $p \supseteq q$.

V. The forcing translation

From the parameters κ, C, $(p,q) \mapsto pq$, 1 and the combinators $\alpha_*, \alpha_1, \ldots, \alpha_8$ introduced in section IV, we now want to define the forcing translation $A \mapsto (p \Vdash A)$ on propositions together with the corresponding program transformation $t \mapsto t^*$ on proof-terms. Some care has to be taken in the definition of the proposition $p \Vdash A$ in PAω^+, since we do not only want to define the corresponding proof transformation at the level of *derivations*, but at the level of *proof-terms*, that contain much less information. In practice, this means that the proposition $p \Vdash A$ has to be defined in such a way that all the computationally transparent deduction steps in the proof of A remain computationally transparent in the proof of $p \Vdash A$.

For that we shall proceed in two steps. First, we shall define an auxiliary translation $M \mapsto M^*$ over all the higher-order terms, through which a proposition A will be translated into a set A^* of forcing conditions (i.e. of kind $\kappa \to o$). Then we shall define the forcing relation $p \Vdash A$ from the set of conditions A^* in such a way that the set of all $p \Vdash A$ is an element of the complete Boolean algebra generated by the set of forcing conditions [2]. (See section V-C.)

A. Translating kinds

Every kind τ is translated into a kind τ^* defined by induction on τ as follows:

$$\iota^* \equiv \iota, \qquad o^* \equiv \kappa \to o, \qquad (\tau \to \sigma)^* \equiv \tau^* \to \sigma^*\,.$$

From this definition, we can already see that the forcing translation will essentially affect propositions (i.e. terms of kind o), that will be interpreted as sets of conditions (kind $\kappa \to o$), but that individuals (kind ι) will not be affected.

B. The auxiliary translation $M \mapsto M^*$

We now define an auxiliary translation mapping each term M of sort τ to a term M^* of sort τ^*. This translation simply propagates through abstractions and applications, and is trivial on arithmetic constructions:

$$\begin{array}{rclrcl} (x^\tau)^* & \equiv & x^{\tau^*} & 0^* & \equiv & 0 \\ (\lambda x^\tau\,.\,M)^* & \equiv & \lambda x^{\tau^*}\,.\,M^* & s^* & \equiv & s \\ (MN)^* & \equiv & M^* N^* & (\mathrm{rec}_\tau)^* & \equiv & \mathrm{rec}_{\tau^*} \end{array}$$

Universal quantification and equational implication—whose deduction are computationally transparent—are translated in a quite obvious way:

$$\begin{array}{rcl} (\forall x^\tau A)^* & \equiv & \lambda r^\kappa\,.\,\forall x^{\tau^*} A^* r \\ (\langle M_1 = M_2\rangle A)^* & \equiv & \lambda r^\kappa\,.\,\langle M_1^* = M_2^*\rangle A^*\,r \end{array}$$

All the complexity of the translation actually lies in implication, that is, in the only connective whose deduction rules have a real computational contents:

$$\begin{array}{l} (A \Rightarrow B)^* \quad \equiv \\ \qquad \lambda r^\kappa\,.\,\forall q\,\forall r'\,\langle r = qr'\rangle(\forall s\,(C[qs] \Rightarrow A^* s) \Rightarrow B^* r')\,. \end{array}$$

(This definition will be justified in section V-D4.)

The translation $M \mapsto M^*$ immediately extends to equational theories $\mathcal{E} = \{M_1 = M_1'; \ldots; M_k = M_k'\}$ by letting

$$\mathcal{E}^* \;=\; \{M_1^* = M_1'^*; \ldots; M_k^* = M_k'^*\}\,.$$

C. The forcing translation $A \mapsto (p \Vdash A)$

Given a condition p and a proposition A, we define the relation of forcing $p \Vdash A$ by letting

$$p \Vdash A \;\equiv\; \forall r^\kappa\;(C[pr] \Rightarrow A^* r)$$

(where r is a fresh variable). To understand the meaning of this definition, let us introduce some terminology and notations.

We say that two conditions p and q are *incompatible* and write $p \perp q$ when $\neg C[pq]$. Given a set of conditions S (of kind $\kappa \to o$) we write

$$S^\perp \;=\; \lambda q^\kappa\,.\,\forall p^\kappa\,(S\,p \Rightarrow p \perp q)$$

the set of all conditions that are incompatible with all the elements of S. In the theory of Boolean valued models [2], it is well known that the set of sets

$$\mathcal{B} \;=\; \lambda S^{\kappa \to o}\,.\,S =_{\mathrm{ext}} S^{\perp\perp}$$

(where $=_{\mathrm{ext}}$ denotes extensional equality between sets) formed by all the sets of conditions that are (extensionally) equal to their bi-orthogonal forms a complete Boolean algebra (ordered by inclusion) generated by the set of conditions $(C, \leq)$[1].

Coming back to the definition of the forcing relation, we have the (classical) equivalence:

$$p \Vdash A \quad\Leftrightarrow\quad \forall r^\kappa(\neg A^* r \Rightarrow p \perp r)$$

which means that the set of all conditions forcing A is defined as $\{p : p \Vdash A\} = (\complement A^*)^\perp$ (using suggestive notations). Therefore the set $\{p : p \Vdash A\}$ is equal to its bi-orthogonal, and thus belongs to the Boolean algebra $\mathcal{B}$. (On the other hand, there is no relationship of inclusion between the sets A^* and $\{p : p \Vdash A\}$ in general.)

The notation $p \Vdash A$ is extended to arbitrary typing contexts $\Gamma \equiv x_1 : A_1, \ldots, x_n : A_n$ by letting

$$p \Vdash \Gamma \;\equiv\; x_1 : (p \Vdash A_1), \ldots, x_n : (p \Vdash A_n)\,.$$

[1]The elements of this algebra are also known to be the *regular open subsets* of the set of conditions (for the topology whose open sets are the downwards closed sets of conditions).

D. *Properties of forcing*

1) Substitutivity and congruence: The auxiliary translation is substitutive and compatible with the congruence $\cong_{\mathcal{E}}$:

Lemma 4:

1) $(M\{x^\tau := N\})^* \;\equiv\; M^*\{x^{\tau^*} := N^*\}$.
2) If $M \cong_{\mathcal{E}} M'$, then $M^* \cong_{\mathcal{E}^*} M'^*$

The same holds for the forcing relation:

Lemma 5:

1) $p \Vdash (A\{x^\tau := N\}) \;\equiv\; (p \Vdash A)\{x^{\tau^*} := N^*\}$ (provided $x^\tau \notin FV(p)$)
2) If $p \cong_{\mathcal{E}} p'$ and $A \cong_{\mathcal{E}} A'$, then $(p \Vdash A) \cong_{\mathcal{E}^*} (p' \Vdash A')$.

2) Universal quantification and equational implication: From the definition of the forcing relation $p \Vdash A$ and of the congruence $M \cong_{\varnothing} M'$ we immediately get:

Lemma 6:

1) $p \Vdash \forall x^\tau A \;\cong_{\varnothing}\; \forall x^{\tau^*}(p \Vdash A)$ (if $x^\tau \notin FV(p)$)
2) $p \Vdash \langle M = M'\rangle A \;\cong_{\varnothing}\; \langle M^* = M'^*\rangle(p \Vdash A)$

Since the two propositions $p \Vdash (\forall x^\tau A)$ and $\forall x^{\tau^*}(p \Vdash A)$ are convertible (in any equational theory), they are logically equivalent in PAω^+, which is mandatory in any definition of forcing. (The same remark holds for equational implication.) However, this stronger property of convertibility will give us more: it will allow us to translate every deduction step that does not affect the current proof-term (i.e. a conversion step, an introduction or an elimination of a universal quantification or of an equational implication) into a combination of deduction steps that do not affect the current proof-term. This property will be crucial to lift the forcing translation at the level of proof terms—and not only at the level of typing derivations.

3) General closure properties: Before considering the subtle case of implication, let us establish some provable closure properties of the forcing relation:

Proposition 3: — In PAω^+ we have:

$$\begin{array}{lcll}
\beta_1 & \equiv & \lambda xyc\,.\,y\,(x\,c) & \\
 & : & \forall p\;\forall q\;(q \le p \Rightarrow (p \Vdash A) \Rightarrow (q \Vdash A)) & \\
\beta_2 & \equiv & \lambda xc\,.\,x\,(\alpha_1\,c) & :\;\forall p\;(\neg C[p] \Rightarrow p \Vdash A) \\
\beta_3 & \equiv & \lambda xc\,.\,x\,(\alpha_9\,c) & :\;\forall p\;\forall q\;((p \Vdash A) \Rightarrow (pq \Vdash A)) \\
\beta_4 & \equiv & \lambda xc\,.\,x\,(\alpha_{10}\,c) & :\;\forall p\;\forall q\;((q \Vdash A) \Rightarrow (pq \Vdash A))
\end{array}$$

In other words, the forcing relation is anti-monotonic: if a proposition A is forced by some condition p, then A is forced by all the conditions $q \le p$ (β_1). In particular, non well-formed conditions force all propositions (β_2). The last two items (β_3 and β_4) are particular cases of the property of anti-monotonicity that will be quite useful in the following.

4) The case of implication: From our definition of the forcing relation, we have:

$$\begin{array}{ll}
p \Vdash A \Rightarrow B & \equiv\; \forall r\,(C[pr] \Rightarrow (A \Rightarrow B)^* r) \\
\cong & \forall r\,(C[pr] \Rightarrow \forall q \forall r'\,\langle r = qr'\rangle((q \Vdash A) \Rightarrow B^* r'))
\end{array}$$

On the other hand, the theory of forcing requires that:

$$p \Vdash A \Rightarrow B \quad\Leftrightarrow\quad \forall q\,((q \Vdash A) \Rightarrow pq \Vdash B)\,.$$

This equivalence comes from the following proposition:

Proposition 4: — In PAω^+ we have:

$$\begin{array}{lcl}
\gamma_1 & \equiv & \lambda xcy\,.\,x\,y\,(\alpha_6\,c) \\
 & : & (\forall q\,((q \Vdash A) \Rightarrow (pq \Vdash B))\;\Rightarrow\; p \Vdash A \Rightarrow B) \\
\gamma_2 & \equiv & \lambda xyc\,.\,x\,(\alpha_5\,c)\,y \\
 & : & (p \Vdash A \Rightarrow B)\;\Rightarrow\;\forall q\;((q \Vdash A) \Rightarrow (pq \Vdash B)) \\
\gamma_3 & \equiv & \lambda xyc\,.\,x\,(\alpha_{11}\,c)\,y \\
 & : & (p \Vdash A \Rightarrow B)\;\Rightarrow\;(p \Vdash A)\;\Rightarrow\;(p \Vdash B) \\
\gamma_4 & \equiv & \lambda xcy\,.\,x\,(y\,(\alpha_{15}\,c)) \\
 & : & \neg A^* p\;\Rightarrow\; p \Vdash A \Rightarrow B
\end{array}$$

Intuitively, the proof-term γ_1 'folds' a proof of the proposition $\forall q\,((q \Vdash A) \Rightarrow (pq \Vdash B))$ into a proof of $p \Vdash A \Rightarrow B$, while γ_2 performs the corresponding unfolding operation. The proof-term γ_3 is a specialized form of γ_2 corresponding to the modus ponens (or application) through the forcing transformation. We also introduce a proof-term γ_4 that will be a key ingredient of the translation of continuations.

5) Forcing Peirce's law: We can already check that the law of Peirce is forced by any condition:

Proposition 5: — In PAω^+ we have:

$$\begin{array}{lcl}
\text{cc}^* & \equiv & \lambda cx\,.\,\text{cc}\,(\lambda k\,.\,x\,(\alpha_{14}\,c)\,(\gamma_4\,k)) \\
 & : & p \Vdash ((A \Rightarrow B) \Rightarrow A) \Rightarrow A\,.
\end{array}$$

E. *Translating proof-terms*

We can now define the translation $t \mapsto t^*$ on *raw* proof-terms as follows:

$$\begin{array}{rcl}
x^* & \equiv & x \\
(t\,u)^* & \equiv & \gamma_3\,t^*\,u^* \\
(\lambda x\,.\,t)^* & \equiv & \gamma_1\,(\lambda x\,.\,t^*\{x_i := \beta_3 x_i\}_{i=1}^n\{x := \beta_4 x\}) \\
\text{cc}^* & \equiv & \lambda cx\,.\,\text{cc}\,(\lambda k\,.\,x\,(\alpha_{14}\,c)\,(\gamma_4\,k))
\end{array}$$

(In the definition of $(\lambda x\,.\,t)^*$, we assume that $x, x_1, \ldots, x_n$ are all the free variables of the proof-term t.)

Basically, the translation $t \mapsto t^*$ inserts the combinator γ_1 in front of every abstraction, and the combinator γ_3 in front of every application, while translating the call/cc constant into the proof-term cc^* of Prop. 5.

The main subtlety of the translation lies in the treatment of variables: at each abstraction, the translation inserts the combinator β_3 in front of every free occurrence of a variable x_i that is not bound by the abstraction, while inserting the combinator β_4 in front of every free occurrence of the variable x (that is bound by the abstraction) in the term t. It is easy to see that, when applied to a closed term t, the translation reveals the de Bruijn structure of t, since every occurrence of a variable x in the term t is translated into the term $\beta_3^n(\beta_4 x)$, where n is the de Bruijn index of that occurrence (starting indices from 0).

We now have all the necessary material to prove that the program transformation $t \mapsto t^*$ is sound w.r.t. typing:

Proposition 6 (Soundness): — If the judgment $\mathcal{E}; \Gamma \vdash t : A$ is derivable in PAω^+, then for all conditions p, the sequent $\mathcal{E}^*; (p \Vdash \Gamma) \vdash t^* : (p \Vdash A)$ is derivable in PAω^+ too.

Proof: We actually prove the result in the case where p is a fresh condition variable, by induction on the derivation of $\mathcal{E}; \Gamma \vdash t : A$, using the types of the combinators γ_1, γ_3, β_3, β_4, cc^* and the properties of substitutivity for system PAω^+. The general case follows by substitutivity. ■

F. Analysis of the computational behavior of t^*

Before analyzing the computational behavior of the translated term t^*, we need to recall that the term t^*, that has a type of the form $p \Vdash A \equiv \forall r\,(C[pr] \Rightarrow A^*r)$, is intended to be evaluated in front of a stack whose first argument c has type $C[pr]$, where p and r are some forcing conditions. As we shall see, the condition p represents logical invariants attached to the current term t^* that is currently evaluated, whereas the condition r represents logical invariants attached to the stack facing t^* during evaluation. In what follows, we shall call a *computational condition*—as opposed to a logical condition—any closed term c of type $C[pr]$ (or any realizer of $C[pr]$) for some conditions p and r.

Using the definition of the combinators γ_1, γ_3, œ^* and γ_4, we easily discover the following evaluation scheme for the translated program t^*:

Proposition 7: — For all terms t_x such that $FV(t_x) \subseteq \{x\}$, for all closed terms t, u, c and for all stacks π we have:

$$\begin{array}{rcr}
(\lambda x\,.\,t_x)^* \star c \cdot u \cdot \pi & \succ^* & t_x^*\{x := \beta_4 u\} \star \alpha_6\ c \cdot \pi \\
(tu)^* \star c \cdot \pi & \succ^* & t^* \star \alpha_{11} c \cdot u^* \cdot \pi \\
\text{œ}^* \star c \cdot u \cdot \pi & \succ^* & u \star \alpha_{14} c \cdot \mathsf{k}_\pi^* \cdot \pi \\
\mathsf{k}_\pi^* \star c \cdot u \cdot \pi' & \succ^* & u \star \alpha_{15} c \cdot \pi
\end{array}$$

writing k_π^* as a shorthand for $\gamma_4\,\mathsf{k}_\pi$.

From this picture, we can see that the translated program t^* essentially behaves the same way as the initial program t, with the difference that the first slot of the stack is now reserved to the computational condition that evolves during evaluation. All the stack operations are thus performed one slot further in the stack (slots are thus intuitively re-indexed), while each operation updates the current computational condition (in the first slot of the stack) by inserting the appropriate combinator.

Most notably, the translated call/cc operator œ^* does not save the current computational condition, as well as the translated continuation constant k_π^* (that is dynamically generated by œ^*) does not restore any formerly saved computational condition—it just updates the current computational condition using the appropriate combinator. As noticed by Krivine [12], [14], the first slot of the stack[2] is thus subtracted from the normal save/restore mechanism induced by call/cc, and now behaves as a mutable reference (or as a 'global memory' according to Krivine's terminology).

Let us now study the types (cf section IV-A) of the combinators α_6, α_{11}, α_{14} and α_{15} that are inserted in the first slot of the stack at each step of the evaluation of t^*:

GRAB	α_6	:	$C[p(qr)]$	$\Rightarrow$	$C[(pq)r]$
PUSH	α_{11}	:	$C[pr]$	$\Rightarrow$	$C[p(pr)]$
SAVE	α_{14}	:	$C[p(qr)]$	$\Rightarrow$	$C[q(rr)]$
RESTORE	α_{15}	:	$C[p(qr)]$	$\Rightarrow$	$C[qp]$

[2]In Krivine's work [12], [14], the current computational condition c is actually stored in the very last slot of the stack, using two specific instructions χ and χ' swapping the first and last elements of the current stack. Our work shows that we can do the same by reserving the first slot of the stack instead of the last one, thus removing the need of the two extra instructions χ and χ'.

These figures suggest the following scenario, which is that logical conditions are actually attached to pieces of data—and even to particular *closures*[3]—within the currently executed process, and that the evolution of the current logical condition pr (given by the type $C[pr]$ of the current computational condition) simply reflects the move of these pieces of data during evaluation.

For instance, the type of α_6 reflects the fact that, during a GRAB step, the first element of the stack (the condition q is attached to) is removed from the stack (the condition r is attached to) and then incorporated in the environment of the term t (the condition p is attached to). Similarly, the type of combinator α_{11} reflects the fact that, during a PUSH step, the environment of the currently executed term (the condition p is attached to) is duplicated and that a closure containing a copy of it is then put on the top of the stack (the condition r is attached to). A similar analysis can be done for SAVE and RESTORE steps, noticing that a continuation constant generated by call/cc is always labelled with the same condition as the corresponding stack.

In the author's opinion, all these features are reminiscent from well-known techniques in computer architecture, such as *virtualization* or *protection rings*, that allow the system to execute a program by only giving it access to a part of the resources (here: the tail of the stack), thus allowing the system to maintain and update critical information (here: the computational condition) in the back of the executed program. The difference is that these features are implemented here via a program transformation while in most computer architectures, these are hardwired in processors that usually provide several execution modes, or protection rings.

In the next section, we shall see that we can also put the forcing transformation 'into the hardware' by introducing a variant of Krivine's Abstract Machine (KAM) with two execution modes (regular mode versus forcing mode), thus avoiding the cost of the program transformation.

VI. AN ABSTRACT MACHINE FOR FORCING

A. Krivine's Forcing Abstract Machine (KFAM)

We now present an abstract machine—the KFAM—that extends the standard Krivine Abstract Machine (with explicit environments). The main novelty of this abstract machine is that it distinguishes two kinds of closures: *regular closures*, that are intended to be executed the usual way, and *forcing closures* (bearing a star as a superscript), that are intended to be executed as through the program transformation defined in section V-E. The current execution mode of the machine will thus be given by the mode of the currently executed closure[4].

1) Syntactic entities: The KFAM manipulates five kinds of syntactic entities—terms, environments, closures, stacks and

[3]This notion will be given a precise meaning in section VI.

[4]In particular, we do not need a special instruction to switch from one mode to the other. This design (where closures bear their execution mode) is mostly dictated by the realizability results we shall present in section VI-C.

processes—that are defined by the following BNF:

$$\begin{array}{llcl}
\textbf{Terms} & t,u & ::= & x \mid \lambda x\,.\,t \mid tu \mid \mathtt{cc} \\
\textbf{Environments} & e & ::= & \emptyset \mid e, x{\leftarrow}c \\
\textbf{Closures} & c & ::= & (t|e) \mid \mathsf{k}_\pi \quad (FV(t){\subseteq}\mathrm{dom}(e)) \\
 & & \mid & (t|e)^* \mid \mathsf{k}^*_\pi \quad (FV(t){\subseteq}\mathrm{dom}(e)) \\
\textbf{Stacks} & \pi & ::= & \diamond \mid c\cdot\pi \\
\textbf{Processes} & P & ::= & c\star\pi
\end{array}$$

Here, terms are classical proof-terms (with call/cc), environments are finite ordered association lists assigning closures to variables, whereas stacks are finite lists of closures.

The KFAM uses two different kinds of regular closures: closures of the form $(t|e)$, where t is a term whose free variables are all assigned in the environment e, and *continuation closures* of the form k_π (where π is an arbitrary stack) that are generated by the call/cc instruction during evaluation. There are also starred versions $(t|e)^*$ and k^*_π of these closures, that are intended to be executed in forcing mode. (Here, the star is just a mark put on the closure.)

B. Evaluation

1) Evaluation in regular mode: The evaluation rules for regular closures are the standard evaluation rules of the KAM with continuations and explicit environments:

$$\begin{array}{rcll}
(x|e,y{\leftarrow}c)\star\pi & \succ & (x|e)\star\pi & (y{\not\equiv}x) \\
(x|e,x{\leftarrow}c)\star\pi & \succ & c\star\pi & \\
(\lambda x\,.\,t|e)\star c\cdot\pi & \succ & (t|e,x{\leftarrow}c)\star\pi & \\
(tu|e)\star\pi & \succ & (t|e)\star(u|e)\cdot\pi & \\
(\mathtt{cc}|e)\star c\cdot\pi & \succ & c\star\mathsf{k}_\pi\cdot\pi & \\
\mathsf{k}_\pi\star c\cdot\pi' & \succ & c\star\pi &
\end{array}$$

Here we can find the usual rules GRAB, PUSH, SAVE and RESTORE (adapted to the new setting), plus two specific rules to lookup a variable in the current environment.

2) Evaluation in forcing mode: The evaluation rules for forcing closures are similar to the rules for regular closures, except that in this case, the first slot is now reserved for the computational condition—a closure—that is updated at each evaluation step by the insertion of one of the 6 combinators α_6, α_9, α_{10}, α_{11}, α_{14} and α_{15}. These combinators are now fixed closures that are parameterizing the machine. (At this stage, these parameters can be taken arbitrarily.)

$$\begin{array}{rcll}
(x|e,y{\leftarrow}c)^*\star c_0\cdot\pi & \succ & (x|e)^*\star\alpha_9\ c_0\cdot\pi & (y{\not\equiv}x) \\
(x|e,x{\leftarrow}c)^*\star c_0\cdot\pi & \succ & c\star\alpha_{10}c_0\cdot\pi & \\
(\lambda x\,.\,t|e)^*\star c_0\cdot c\cdot\pi & \succ & (t|e,x{\leftarrow}c)^*\star\alpha_6\ c_0\cdot\pi & \\
(tu|e)^*\star c_0\cdot\pi & \succ & (t|e)^*\star\alpha_{11}c_0\cdot(u|e)^*\cdot\pi & \\
(\mathtt{cc}|e)^*\star c_0\cdot c\cdot\pi & \succ & c\star\alpha_{14}c_0\cdot\mathsf{k}^*_\pi\cdot\pi & \\
\mathsf{k}^*_\pi\star c_0\cdot c\cdot\pi' & \succ & c\star\alpha_{15}c_0\cdot\pi &
\end{array}$$

(In the above rules, the application of a combinator α_i to a closure c_0 is defined by $\alpha_i c_0 \equiv (xy|x{\leftarrow}\alpha_i, y{\leftarrow}c_0)$).

The last four rules clearly mimic the computational behavior of transformed programs such as described by Prop. 7. Also note that lookup operations also insert their own combinators (α_9 and α_{10}) in the first slot of the stack. These two combinators—that were actually hidden in the proof-terms β_3 and β_4 (cf sections V-D3 and V-F)—simply reflect the destruction of the current environment during lookup.

C. Realizability models induced by the KFAM

It is now time to relate the abstract machine introduced above with logic and forcing.

For that, it suffices to remark that the KFAM describes a small step operational semantics that naturally induces new classical realizability models. Provided we give to closures the role formerly played by closed λ_c-terms, and provided we use our new definitions for stacks, processes and for the relation of evaluation[5], the construction of these realizability models is exactly the same as the one presented in section III-B. We shall not rewrite the defining equations of these models since they are the same as in section III-B; only the sets in which these equations have to be understood are different.

As usual, there are as many realizability models as poles (in the sense of the KFAM), and given a fixed pole $\perp\!\!\!\perp$, the realizability relation is written $c \Vdash A[\rho]$, where c is a closure, A a proposition and ρ a valuation in the corresponding model. Up to the end of this section, we assumed fixed a pole $\perp\!\!\!\perp$.

The presence of two kinds of closures (regular or forcing) now induces two properties of adequacy.

1) Adequacy in regular mode: The first property of adequacy is the exact analogous of Prop. 2:

Proposition 8 (Adequacy in regular mode): — If the typing judgment $\mathcal{E}; x_1 : A_1, \ldots, x_n : A_n \vdash t : B$ is derivable in PAω^+, then for all valuations ρ such that $\rho \models \mathcal{E}$ and for all closures $c_1 \Vdash A_1[\rho], \ldots, c_n \Vdash A_n[\rho]$, we have

$$(t|x_1{\leftarrow}c_1, \ldots, x_n{\leftarrow}c_n) \Vdash B[\rho]\,.$$

Note that here, we close the term t with an environment rather than with a substitution. Up to such minor changes, the proof of Prop. 8 is the same as for Prop. 2. Also note that this proposition does not make any assumption on the combinators α_6, α_9, α_{10}, α_{11}, α_{14} and α_{15}, that can be taken arbitrarily. The reason is that none of the evaluation rules of the forcing mode is involved in the proof of Prop. 8.

2) Adequacy in forcing mode: In order to state the adequacy property corresponding to the forcing mode—that is: to forcing closures—we now need to make suitable assumptions on the combinators α_6, α_9, α_{10}, α_{11}, α_{14} and α_{15}. Formally, we say that these combinators are *adequate* to the notion of forcing induced by $(\kappa, C, \cdot, 1)$ and to the pole $\perp\!\!\!\perp$ if:

$$\begin{array}{rcl}
\alpha_6 & \Vdash & \forall p\,\forall q\,\forall r\ (C[p(qr)] \Rightarrow C[(pq)r]) \\
\alpha_9 & \Vdash & \forall p\,\forall q\,\forall r\ (C[(pq)r] \Rightarrow C[pr]) \\
\alpha_{10} & \Vdash & \forall p\,\forall q\,\forall r\ (C[(pq)r] \Rightarrow C[qr]) \\
\alpha_{11} & \Vdash & \forall p\,\forall q\,\forall r\ (C[pq] \Rightarrow C[p(pq)]) \\
\alpha_{14} & \Vdash & \forall p\,\forall q\,\forall r\ (C[p(qr)] \Rightarrow C[q(rr)]) \\
\alpha_{15} & \Vdash & \forall p\,\forall q\,\forall r\ (C[p(qr)] \Rightarrow C[qp])
\end{array}$$

[5]Formally, the set of closures together with the set of stacks, the set of processes and the relation of evaluation (according to the new definitions) constitute a *realizability algebra* in the sense of [14], which is thus suitable to the definition of classical realizability models.

Thanks to Prop. 8, we can take for instance $\alpha_i = (t_i|\emptyset)$, where t_i is a proof-term of the appropriate proposition.

Proposition 9 (Adequacy in forcing mode): — If the combinators α_i are adequate to the notion of forcing induced by $(\kappa, C, \cdot, 1)$ and to the pole $\perp\!\!\!\perp$, and if the typing judgment

$$\mathcal{E}; x_1 : A_1, \dots, x_n : A_n \vdash t : B$$

is derivable in PAω^+, then for all valuations ρ such that $\rho \models \mathcal{E}^*$, for all pairwise distinct condition variables $p_0, p_1, \dots, p_n$ (of kind κ) and for all closures $c_1 \Vdash (p_1 \Vdash A_1)[\rho], \dots, c_n \Vdash (p_n \Vdash A_n)[\rho]$, we have

$$(t|x_1{\leftarrow}c_1, \dots, x_n{\leftarrow}c_n)^* \Vdash \big(((p_0 p_1) \cdots p_n) \Vdash B[\rho]\big)\,.$$

Note that here, we need to attach a condition variable p_i to each closure c_i so that the closure $(t|x_1{\leftarrow}c_1, \dots, x_n{\leftarrow}c_n)^*$ (in forcing mode) realizes the formula $((p_0 p_1) \cdots p_n) \Vdash B[\rho]$ (where p_0 is another condition variable).

The proof of Prop. 9 is quite tedious, but it essentially consists to rephrase the ingredients of the proof of the syntactic result of Prop. 6 into semantic terms.

D. The underlying methodology

The typical example of use of the KFAM is the following: Let us assume that we have a proof-term t of some theorem T that depends on a free variable x_0 representing some axiom A. If we are able to force this axiom using a suitable set of conditions $(\kappa, C, \cdot, 1)$, that is, if we can build a closed proof-term t_0 of $1 \Vdash A$, then by Prop. 8 we know that the regular closure $c_0 = (t_0|\emptyset)$ is a realizer of $1 \Vdash A$. Hence by Prop. 9 the forcing closure $c = (t|x_0{\leftarrow}c_0)^*$ is a realizer of $1 \Vdash T$.

Intuitively, we have embedded the *user program* t into a forcing closure in order to force its evaluation in forcing mode. Each time the variable x_0 representing the axiom A comes into head position (at each *system call*), the control is given to the regular closure $c = (t_0|\emptyset)$ that has a full access to the stack, and that can thus perform all the necessary operations using the current computational condition.

VII. Conclusion

In this paper we have presented and studied the program transformation underlying the forcing translation through the Curry-Howard correspondence in classical logic. We also introduced an abstract machine (the KFAM) devoted to the evaluation of proofs built by forcing techniques.

However, much work remains to be done in order to understand the difference between the realizers of $p \Vdash A$ and the realizers of A and to adapt extraction techniques such as described in [15] to the framework described here. (Part of this work has been already done in [14].)

Moreover, it could be interesting to see how forcing works in concrete cases, such as the ones described in [12], [14]. But before considering difficult problems such as the computational meaning of the continuum hypothesis, we could first investigate simple cases of forcing by studying the computational behavior of proofs of $p \Vdash A$ (for very simple formulæ A) that do not come from an already existing proof of A, and to understand how such 'extra proofs' behave.

Discovering new logical transformations

This paper is more generally an illustration of a fruitful methodology in logic: start from a logical translation $A \mapsto A^*$ (here: $p \Vdash A$), deduce from it a program transformation $t \mapsto t^*$ (on Curry-style proof-terms) and then internalize the program transformation into the abstract machine (or into the language) in such a way that the transformation becomes identity.

Historically, this methodology was already used for classical logic: starting from negative translations $A \mapsto A^{\neg\neg}$, we get the corresponding CPS-transform $t \mapsto t^{\neg\neg}$, before removing the need of CPS-transforms by adding control operators in the language. (Of course, real history was much less linear than this simplistic picture.)

But the reminiscence we observed in section V-F suggests that it could be interesting to try using this methodology the other way around: starting from a particular feature of computer architecture, we could first mimic this feature using a suitable program transform (in some λ-calculus) before finding the corresponding logical translation. In the author's opinion, this idea opens interesting perspectives in the discovery of new logical transformations similar to forcing.

References

[1] J. Avigad. Forcing in proof theory. *Bulletin of Symbolic Logic*, 10(3):305–333, 2004.

[2] J. L. Bell. *Boolean-Valued Models and Independence Proofs in Set Theory*. Oxford, 1985.

[3] P. J. Cohen. The independence of the continuum hypothesis. *Proceedings of the National Academy of Sciences of the United States of America*, 50(6):1143–1148, December 1963.

[4] P. J. Cohen. The independence of the continuum hypothesis II. *Proceedings of the National Academy of Sciences of the United States of America*, 51(1):105–110, January 1964.

[5] T. Coquand. Forcing and type theory. In E. Grädel and R. Kahle, editors, *CSL*, volume 5771 of *Lecture Notes in Computer Science*, page 2. Springer, 2009.

[6] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[7] K. Gödel. Consistency of the axiom of choice and of the generalized continuum-hypothesis with the axioms of set theory. *Proceedings of the National Academy of Sciences of the United States of America*, 24(12), 1938.

[8] N. Goodman. Relativized realizability in intuitionistic arithmetic of all finite types. *Journal of Symbolic Logic*, 43(1):23–44, 1978.

[9] T. Griffin. A formulae-as-types notion of control. In *Principles Of Programming Languages (POPL'90)*, pages 47–58, 1990.

[10] S. C. Kleene. On the interpretation of intuitionistic number theory. *Journal of Symbolic Logic*, 10:109–124, 1945.

[11] J. L. Krivine. *Lambda-calculus, types and models*. Masson, 1993.

[12] J.-L. Krivine. Structures de réalisabilité, RAM et ultrafiltre sur N. Manuscript, available on the author's web page, 2008.

[13] J.-L. Krivine. Realizability in classical logic. In *Interactive models of computation and program behaviour*, volume 27 of *Panoramas et synthèses*, pages 197–229. Société Mathématique de France, 2009.

[14] J.-L. Krivine. Realizability algebras : a program to well order R. Manuscript, available on the author's web page, 2010.

[15] A. Miquel. Existential witness extraction in classical realizability and via a negative translation. *Logical Methods for Computer Science*, 2010.

[16] C. Raffalli and F. Ruyer. Realizability of the axiom of choice in HOL (An analysis of Krivine's work). *Fundamenta Informaticae*, 84(2):241–258, 2008.