Math. Struct. in Comp. Science (2016), *vol.* 26, *pp.* 1269–1303. © Cambridge University Press 2014
doi:10.1017/S0960129514000450 First published online 17 November 2014

Specifying Peirce's law in classical realizability

MAURICIO GUILLERMO[†] and ALEXANDRE MIQUEL[‡]

[†]*Centro de Matemática, Facultad de Ciencias, Universidad de la República, Iguá 4225 esq. Mataojo, Montevideo, Uruguay*
Email: mguille@fing.edu.uy
[‡]*Laboratoire d'Informatique du parallélisme, École Normale Supérieure de Lyon 46, allée d'Italie, 69364 Lyon Cedex 07, France*

Received 8 April 2011; revised 27 April 2014

This paper deals with the specification problem in classical realizability (such as introduced by Krivine (2009 *Panoramas et synthèses* **27**)), which is to characterize the universal realizers of a given formula by their computational behaviour. After recalling the framework of classical realizability, we present the problem in the general case and illustrate it with some examples. In the rest of the paper, we focus on Peirce's law, and present two game-theoretic characterizations of its universal realizers. First, we consider the particular case where the language of realizers contains no extra instruction such as 'quote' (Krivine 2003 *Theoretical Computer Science* **308** 259–276). We present a first game $\mathbb{G}_0$ and show that the universal realizers of Peirce's law can be characterized as the uniform winning strategies for $\mathbb{G}_0$, using the technique of interaction constants. Then we show that in the presence of extra instructions such as 'quote', winning strategies for the game $\mathbb{G}_0$ are still adequate but no more complete. For that, we exhibit an example of a wild realizer of Peirce's law, that introduces a purely game-theoretic form of backtrack that is not captured by $\mathbb{G}_0$. We finally propose a more sophisticated game $\mathbb{G}_1$, and show that winning strategies for the game $\mathbb{G}_1$ are both adequate and complete in the general case, without any further assumption about the instruction set used by the language of classical realizers.

1. Introduction

The correspondence between proofs and programs – also known as the *Curry–Howard correspondence* (Curry and Feys 1958; Girard 1989; Howard 1969) – has brought a deep renewal in proof theory, by establishing strong connections between the concepts of proof theory and the concepts of functional programming. For a long time, the computational interpretation of proofs induced by this correspondence was strictly limited to intuitionistic logic and to constructive mathematics, and the computational contents of classical proofs could only be studied indirectly, via clever translations – the so called *negative translations* – from classical logic to intuitionistic logic (Friedman 1978) or to linear logic (Girard 2006).

In Griffin(1990), he discovered that in the programming language scheme (Sperber *et al.* 2009), the control operator `call/cc` (for 'call with current continuation') could be given the type $((A \to B) \to A) \to A$ corresponding to *Peirce's law* through the formulas-as-types interpretation (Howard 1969). Since, Peirce's law constructively implies all the other forms of classical reasoning (excluded middle, double negation elimination, *reductio ad absurdum*, de Morgan laws, etc.), this discovery opened the way to a direct computational interpretation of all classical proofs, using control operators and their ability to implement *backtrack* to interpret classical reasoning principles. Many classical λ-calculi have been introduced from these ideas, such as Parigot's $\lambda\mu$-calculus (Parigot 1997), Barbanera and Berardi's symmetric λ-calculus (Barbanera and Berardi 1996), Krivine's λ_c-calculus (Krivine 2009) or Curien and Herbelin's $\bar{\lambda}\mu$-calculus (Curien and Herbelin 2000).

However, the analysis of the computational behaviour of programs extracted from classical proofs quickly proved to be difficult. One reason for this was the presence of control operators, which naturally break the linearity of the execution flow of programs. But the main reason was the lack of a theory relating the point of view of typing (which corresponds to deduction in logic) with the point of view of computation. Such a theory already existed for intuitionistic logic: the theory of *realizability*, that was initially introduced by Kleene (1945) to interpret the computational contents of the proofs of Heyting arithmetic, and later extended to more expressive frameworks, including intuitionistic set theories (Friedman 1973; McCarty 1984; Myhill 1973). Alas, the theory of realizability such as designed by Kleene and his successors was not only limited to intuitionistic logic, but it was also fundamentally incompatible with classical logic[†].

1.1. *The theory of classical realizability*

To address this problem, Krivine introduced in the middle of the nineties the theory of *classical realizability* (Krivine 2009), which is a complete reformulation of the very principles of realizability to make them compatible with classical reasoning. (As noticed in Miquel (2010) and Oliva and Streicher (2008), classical realizability can be seen as a reformulation of Kleene's realizability through Friedman's A-translation (Friedman 1978).) Although it was initially introduced to interpret the proofs of classical second-order arithmetic, the theory of classical realizability can be scaled to more expressive theories such as Zermelo–Fraenkel set theory (Krivine 2001) or the calculus of constructions with universes (Miquel 2007).

As in intuitionistic realizability, every formula A is interpreted in classical realizability as a set $|A|$ of programs called the *realizers* of A, that share a common computational behaviour dictated by the structure of the formula A. This point of view is related to the point of view of deduction (and of typing) via the property of *adequacy*, that expresses that any program extracted from a proof of A – that is: any program of type A – realizes the formula A, and thus has the computational behaviour expected from the formula A.

But the difference between intuitionistic and classical realizability is that in the latter, the set of realizers of A is defined indirectly, that is: from a set $\|A\|$ of execution contexts (represented as argument stacks) that are intended to challenge the truth of A. Intuitively,

[†] For instance, the formula $\forall x\,(H(x) \vee \neg H(x))$ – where $H(x)$ denotes the halting predicate – is classically provable, but its *negation* is intuitionistically realizable (Kleene 1945). The same holds for the formula $\forall X\,(X \vee \neg X)$ expressing the law of excluded middle in second-order logic, whose negation is intuitionistically realizable.

the set $\|A\|$ – which we shall call the *falsity value of A* – can be understood as the set of all possible counter-arguments to the formula A. In this framework, a program realizes the formula A – i.e. belongs to the *truth value* $|A|$ – if and only if it is able to defeat all the attempts to refute A using a stack in $\|A\|$. (The definition of the classical notion of a realizer is also parameterized by a *pole* representing a particular challenge, that we shall define and discuss in Section 4.1.1.)

By giving an equal importance to programs – or terms – that 'defend' the formula A, and to execution contexts – or stacks – that 'attack' the formula A, the theory of classical realizability is thus able to describe the interpretation of classical reasoning in terms of manipulation of whole stacks (as first class citizens) using control operators.

1.2. *The λ_c-calculus*

The language of realizers that is traditionally used in classical realizability is Krivine's λ_c-calculus (Krivine 2009), an extension of Church's λ-calculus (Church 1941) with an instruction cc (representing the control operator `call/cc`) together with the machinery for manipulating *continuations constants* embedding stacks. Unlike the traditional λ-calculi, the λ_c-calculus relies on a particular reduction strategy – the *call by name* strategy – which is implemented using Krivine's abstract machine (KAM). As a consequence, the property of *confluence* – which plays a central role in traditional λ-calculi – does not make sense anymore in this architecture. In the KAM, the property of confluence is replaced by the property of *determinism*, which is not only simpler, but which is also closer to the point of view of real programming languages.

An important feature of the λ_c-calculus is that it can be freely enriched with extra instructions that can be used to optimize extracted programs (for instance: instructions manipulating primitive numerals (Miquel 2010)) or even to realize additional reasoning principles. The emblematic example is given by the instruction '`quote`', that computes the Gödel code of a stack (according to a fixed enumeration of stacks), and that is used in Krivine (2003) to realize the axiom of dependent choices[†]. In this paper, we shall also consider two other extra instructions: the instruction '`eq`', that tests the syntactic equality between two (closed) λ_c-terms, and the non-deterministic choice operator $\pitchfork$ ('fork').

1.3. *The specification problem*

A fundamental problem of classical realizability is the *specification problem*, which is to characterize the (universal) realizers of a given formula A from their computational behaviour. This problem has received little attention in intuitionistic realizability, mainly because the specification attached to a formula A can be directly inferred from the definition of the set of realizers of A. For instance, intuitionistic realizers of the formula $\exists^{\mathsf{N}} x\, A(x)$ are exactly the programs reducing to a pair whose first component is a witness $n \in \mathbb{N}$

[†] This axiom is crucial to prove the Baire category theorem – to which it is actually equivalent (Goldblatt 1985).

and whose second component is a realizer of the formula $A(n)$ (Krivine 1993). In intuitionistic realizability, formulas already constitute specifications.

The situation is much more subtle in classical realizability. This subtlety does not only come from the particular architecture of classical realizability (that involves notions such as a *falsity value* or a *pole* that are alien to Kleene's realizability), but it primarily comes from the fact that the underlying programming language λ_c contains the control operator `call/cc`, so that realizers can embed continuation constants that may issue a backtrack at any time. Of course, these features are essential to interpret classical reasoning principles such as excluded middle, but on the other hand we cannot hope anymore that the first projection of a classical realizer of $\exists^{\mathsf{N}} x\, A(x)$ will give us the desired witness for free. (For an account of witness extraction techniques in classical realizability, see Miquel (2010) .)

As we shall see in Sections 6.4 and 6.5, the problem becomes even more difficult when considering extensions of the λ_c-calculus with instructions such as '`quote`' or '`eq`', that are able to discriminate programs from their syntactic differences, and not only from their computational behaviour.

1.4. *Specifying Peirce's law*

The opposition between λ_c-terms (seen as defenders) and stacks (seen as attackers) constitutes the heart of classical realizability, and it naturally suggests that the specification problem has to be studied in game-theoretic terms.

In this paper, we shall study the specification problem for the (fundamentally classical) law of Peirce, whose second-order formulation is $\forall X\, \forall Y\, (((X \Rightarrow Y) \Rightarrow X) \Rightarrow X)$ – namely: the type of `call/cc`. This problem was given a first and partial solution by the first author Guillermo (2008), who proposed a game-theoretic characterization $\mathbb{G}_0$ of the universal realizers of Peirce's law in the particular case, where the underlying calculus of realizers is deterministic and contains infinitely many *interaction constants*, a notion we shall define in Section 5.3. (We shall present here a simplified proof of this first solution that does not rely on the assumption of determinism.)

However, the presence of interaction constants – which is crucial in the proof of completeness presented in Guillermo (2008) – is known to be incompatible with the presence of instructions such as '`quote`' or '`eq`', that are able to detect syntactic differences in λ_c-terms that would be otherwise considered as computationally equivalent. This first result thus left open the specification problem for Peirce's law in the general case, where the calculus of realizers may rely on an arbitrary set of instructions – including the instruction '`quote`' used to realize the axiom of dependent choices (Krivine 2003).

In this paper, we shall see that the first specification $\mathbb{G}_0$ remains adequate in the general case (in the sense that any term which gives a winning strategy in the game $\mathbb{G}_0$ is a realizer of Peirce's law), but that it is not complete anymore. Indeed, we shall exhibit a closed λ_c-term (that can be implemented either from '`eq`' or from '`quote`') that constitutes a *wild realizer of Peirce's law*, in the sense that it is not captured by the game $\mathbb{G}_0$. We shall see that this counter-example introduces a new – and purely game theoretic – form of backtrack that does not come from control operators, but from the fact that realizers can now test (using syntactic equality) whether a position already appeared before in the play.

From the point of view of metatheory, we shall also see that this new form of backtrack is treated in the corresponding proof of adequacy by using the meta-theoretic law of Peirce, thus making the proof classical. (This contrasts to the traditional proof of adequacy of the domestic realizer `call/cc`, which is purely intuitionistic.)

To capture our (counter-)example of a wild realizer, we shall present a second game $\mathbb{G}_1$ that takes into account the second form of backtrack. Then, we shall prove that this second game $\mathbb{G}_1$ is both adequate and complete in the general case (i.e. without any assumption on the instruction set), thus constituting the definitive specification of Peirce's law.

2. The language λ_c

2.1. *Terms and stacks*

The λ_c-calculus distinguishes two kinds of syntactic expressions: *terms*, that represent programs, and *stacks*, that represent evaluation contexts. The terms of the λ_c-calculus are pure λ-terms (Barendregt 1984; Church 1941) enriched with two kinds of constants:

— *Continuation constants* k_π, one for every stack π;
— *Instructions*, such as the control operator `call/cc` (written here cc), that are taken in a fixed set $\mathcal{C}$ of constants.

The stacks of the λ_c-calculus are finite lists of closed terms terminated by a stack constant taken in a fixed set $\mathcal{B}$ of *stack constants*, also known as *stack bottoms*.

Formally, terms and stacks of the λ_c-calculus are thus defined from three auxiliary sets of symbols, that are pairwise disjoint:

— A denumerable set $\mathcal{V}_\lambda$ of λ-variables (notation: x, y, z, etc.).
— A countable set $\mathcal{C}$ of instructions, that contains at least an instruction $\text{cc} \in \mathcal{C}$ ('call/cc', for: *call with current continuation*).
— A non-empty countable set $\mathcal{B}$ of stack constants, also called stack bottoms (notation: α, β, γ, etc.).

Definition 1 (terms and stacks). Terms and stacks of the λ_c-calculus are defined by mutual induction from the following formation rules:

1. If $x \in \mathcal{V}_\lambda$ is a λ-variable, then x is a term, and $FV(x) = \{x\}$.
2. If $c \in \mathcal{C}$ is an instruction, then c is a term, and $FV(c) = \varnothing$.
3. If π is a stack, then k_π is a term, and $FV(\mathsf{k}_\pi) = \varnothing$.
4. If t and u are terms, then tu is a term, and $FV(tu) = FV(t) \cup FV(u)$.
5. If $x \in \mathcal{V}_\lambda$ is a λ-variable and if t is a term, then $\lambda x\,.\,t$ is a term, and $FV(\lambda x\,.\,t) = FV(t) \setminus \{x\}$.
6. If $\alpha \in \mathcal{B}$ is a stack constant, then α is a stack.
7. If t is a closed term (i.e. $FV(t) = \varnothing$) and if π is a stack, then $t \cdot \pi$ is a stack.

In this definition, we define every λ_c-term t together with its set of free variables $FV(t)$, so that we can restrict the application of rule (7) to closed terms t. Thanks to this restriction, stacks are always closed objects (i.e. they do not contain free variables) and continuation constants k_π are really constant.

In what follows, we adopt the same writing conventions as in the pure λ-calculus, by considering that application is left-associative and has higher precedence than abstraction. We also allow several abstractions to be regrouped under a single λ, so that the closed term $\lambda x\,.\,\lambda y\,.\,\lambda z\,.\,((zx)y)$ can be more simply written $\lambda xyz\,.\,zxy$.

As usual, terms and stacks are considered up to α-conversion (Barendregt 1984), and we denote by $t\{x := u\}$ the term obtained by replacing every free occurrence of the variable x by the term u in the term t, possibly renaming the bound variables of t to prevent name clashes. The sets of all closed terms and of all (closed) stacks are respectively denoted by Λ and Π.

Definition 2 (proof-like terms). We say that a λ_c-term t is *proof like* if t contains no continuation constant k_π.

The above terminology comes from the fact that every realizer coming from the proof of a theorem of PA2 is of this form (as we shall see in Theorem 17).

Finally, every natural number $n \in \mathbb{N}$ is represented in the λ_c-calculus as the closed proof-like term $\overline{n}$ defined by

$$\overline{n} \;\equiv\; \overline{s}^n\overline{0} \;\equiv\; \underbrace{\overline{s}(\cdots(\overline{s}}_{n}\,\overline{0})\cdots),$$

where $\overline{0} \equiv \lambda xf\,.\,x$ and $\overline{s} \equiv \lambda nxf\,.\,f(nxf)$ are Church's encodings of zero and the successor function in the pure λ-calculus. Note that this encoding slightly differs from the traditional encoding of numerals in the λ-calculus, although the term $\overline{n} \equiv \overline{s}^n\overline{0}$ is clearly β-convertible to Church's encoding $\lambda xf\,.\,f^nx$ – and thus computationally equivalent. The reason for preferring this modified encoding is that it is better suited to the call-by-name discipline of KAM we shall now present.

2.2. *Krivine's abstract machine*

In the λ_c-calculus, computation occurs through the interaction between a closed term and a stack within KAM. Formally, we call a *process* any pair $t \star \pi$ formed by a closed term t and a stack π. The set of all processes is written $\Lambda \star \Pi$ (which is just another notation for the Cartesian product $\Lambda \times \Pi$).

Definition 3 (relation of evaluation). We call a relation of *one step evaluation* any binary relation $\succ_1$ over the set $\Lambda \star \Pi$ of processes that fulfils the following four axioms:

(Push)	$tu \star \pi$	$\succ_1$	$t \star u \cdot \pi$
(Grab)	$(\lambda x\,.\,t) \star u \cdot \pi$	$\succ_1$	$t\{x := u\} \star \pi$
(Save)	$\mathbf{cc} \star t \cdot \pi$	$\succ_1$	$t \star \mathsf{k}_\pi \cdot \pi$
(Restore)	$\mathsf{k}_\pi \star t \cdot \pi'$	$\succ_1$	$t \star \pi$

The reflexive-transitive closure of $\succ_1$ is written $\succ$.

One of the specificities of the λ_c-calculus is that it comes with a binary relation of (one step) evaluation $\succ_1$ that is not *defined*, but *axiomatized* via the rules (Push), (Grab), (Save) and (Restore). In practice, the binary relation $\succ_1$ is simply another parameter of the definition of the calculus, just like the sets $\mathcal{C}$ and $\mathcal{B}$. Strictly speaking, the λ_c-calculus is not a particular extension of the λ-calculus, but a family of extensions of the λ-calculus parameterized by the sets $\mathcal{B}$, $\mathcal{C}$ and the relation of one step evaluation $\succ_1$. (The set $\mathcal{V}_\lambda$ of λ-variables – that is interchangeable with any other denumerable set of symbols – does not really constitute a parameter of the calculus.)

2.3. *Adding new instructions*

The main interest of keeping open the definition of the sets $\mathcal{B}$, $\mathcal{C}$ and of the relation evaluation $\succ_1$ (by axiomatizing rather than defining them) is that it makes possible to enrich the calculus with extra instructions and evaluation rules, simply by putting additional axioms about $\mathcal{C}$, $\mathcal{B}$ and $\succ_1$. On the other hand, the definitions of classical realizability (Krivine 2009) as well as its main properties do not depend on the particular choice of $\mathcal{B}$, $\mathcal{C}$ and $\succ_1$, although the fine structure of the corresponding realizability models is of course affected by the presence of additional instructions and evaluation rules.

For the needs of the discussion in Section 5, we shall sometimes consider the following extra instructions in the set $\mathcal{C}$:

— The instruction `quote`, that comes with the evaluation rule

$$\text{(QUOTE)}\qquad \mathtt{quote} \star t \cdot \pi \ \succ_1\ t \star \overline{n}_\pi \cdot \pi\,,$$

where $\pi \mapsto n_\pi$ is an injection from Π to $\mathbb{N}$. Intuitively, the instruction `quote` computes the 'code' n_π of the stack π, and passes it (using the encoding $n \mapsto \overline{n}$ described in Section 2.1) to the term t. This instruction was introduced in Krivine (2003) to realize the axiom of dependent choices.

— The instruction eq, that comes with the evaluation rule

$$\text{(EQ)}\qquad \mathsf{eq} \star t_1 \cdot t_2 \cdot u \cdot v \cdot \pi \ \succ_1\ \begin{cases} u \star \pi & \text{if } t_1 \equiv t_2 \\ v \star \pi & \text{if } t_1 \not\equiv t_2. \end{cases}$$

Intuitively, the instruction eq tests the syntactic equality of its first two arguments t_1 and t_2 (up to α-conversion), giving the control to the next argument u if the test succeeds, and to the second next argument v otherwise. In presence of the `quote` instruction, it is possible to implement a closed λ_c-term eq' that has the very same computational behaviour as eq, by letting

$$\mathsf{eq}' \ \equiv\ \lambda x_1 x_2\,.\,\mathtt{quote}\,(\lambda n_1\,.\,\mathtt{quote}\,(\lambda n_2\,.\,\mathsf{eq\_nat}\, n_1\, n_2)\, x_2)\, x_1\,,$$

where eq_nat is any closed λ-term that tests the equality between two numerals (using the encoding $n \mapsto \overline{n}$).

— The instruction ⋔ ('fork'), that comes with the two evaluation rules

$$\text{(FORK)}\qquad \pitchfork \star t_0 \cdot t_1 \cdot \pi \succ_1 t_0 \star \pi \qquad \text{and} \qquad \pitchfork \star t_0 \cdot t_1 \cdot \pi \succ_1 t_1 \star \pi\,.$$

Intuitively, the instruction ⋔ behaves as a non-deterministic choice operator, that indifferently selects its first or its second argument. The main interest of this instruction is that it makes evaluation non-deterministic, in the following sense:

Definition 4 (deterministic evaluation). We say that the relation of evaluation $\succ_1$ is *deterministic* when the two conditions $p \succ_1 p'$ and $p \succ_1 p''$ imply $p' \equiv p''$ (syntactic identity) for all processes p, p' and p''. Otherwise, $\succ_1$ is said to be *non-deterministic*.

The smallest relation of evaluation, that is defined as the union of the four rules (Push), (Grab), (Save) and (Restore), is clearly deterministic. The property of determinism still holds if we enrich the calculus with an instruction $\texttt{eq}\,(\not\equiv \text{œ})$ together with the aforementioned evaluation rules, or with the instruction $\texttt{quote}\,(\not\equiv \text{œ})$.

On the other hand, the presence of an instruction $\pitchfork$ with the corresponding evaluation rules definitely makes the relation of evaluation non-deterministic.

2.4. *The thread of a process and its anatomy*

Given a process p, we call the *thread* of p and write $\mathbf{th}(p)$ the set of all processes p' such that $p \succ p'$:

$$\mathbf{th}(p) \;=\; \{p' \in \Lambda \star \Pi \;:\; p \succ p'\}\,.$$

This set has the structure of a finite or infinite (di)graph whose edges are given by the relation $\succ_1$ of one step evaluation. In the case where the relation of evaluation is deterministic, the graph $\mathbf{th}(p)$ can be either:

— *Finite and cyclic from a certain point*, because the evaluation of p loops at some point. A typical example is the process $\mathbf{I} \star \delta\delta \cdot \alpha$ (where $\mathbf{I} \equiv \lambda x\,.\,x$ and $\delta \equiv \lambda x\,.\,xx$), that enters into a 2-cycle after one evaluation step:

$$\mathbf{I} \star \delta\delta \cdot \alpha \;\succ_1\; \delta\delta \star \alpha \;\succ_1\; \delta \star \delta \cdot \alpha \;\succ_1\; \delta\delta \star \alpha \;\succ_1\; \cdots$$

— *Finite and linear*, because the evaluation of p reaches a state where no more rule applies. For example:

$$\mathbf{II} \star \alpha \;\succ_1\; \mathbf{I} \star \mathbf{I} \cdot \alpha \;\succ_1\; \mathbf{I} \star \alpha\,.$$

— *Infinite and linear*, because p has an infinite execution that never reaches twice the same state. A typical example is given by the process $\delta'\delta' \star \alpha$, where $\delta' \equiv \lambda x\,.\,x\,x\,\mathbf{I}$:

$$\delta'\delta' \star \alpha \;\succ_3\; \delta'\delta' \star \mathbf{I} \cdot \alpha \;\succ_3\; \delta'\delta' \star \mathbf{I} \cdot \mathbf{I} \cdot \alpha \;\succ_3\; \delta'\delta' \star \mathbf{I} \cdot \mathbf{I} \cdot \mathbf{I} \cdot \alpha \;\succ_3\; \cdots$$

2.5. *Substituting term and stack constants*

In some situations, it is desirable to substitute a closed term u to a particular constant $c \in \mathcal{C}$ throughout the structure of a term t or of a stack π. Unlike the traditional form of substitution $t\{x := u\}$ (which is only defined for terms), the substitutions $t\{c := u\}$ and $\pi\{c := u\}$ propagate through the continuation constants k_π as well. Formally, these substitutions are defined as follows:

$$\begin{array}{rcl@{\qquad}rcll}
x\{c := u\} & \equiv & x & c\{c := u\} & \equiv & u & \\
(\lambda x\,.\,t)\{c := u\} & \equiv & \lambda x\,.\,t\{c := u\} & c'\{c := u\} & \equiv & c' & (\text{if } c' \not\equiv c) \\
(t_1 t_2)\{c := u\} & \equiv & t_1\{c := u\} t_2\{c := u\} & \alpha\{c := u\} & \equiv & \alpha & \\
\mathsf{k}_\pi\{c := u\} & \equiv & \mathsf{k}_{\pi\{c:=u\}} & (t \cdot \pi)\{c := u\} & \equiv & t\{c := u\} \cdot \pi\{c := u\} &
\end{array}$$

Similarly, we also define two operations of substitutions $t\{\alpha := \pi_0\}$ and $\pi\{\alpha := \pi_0\}$ where a stack constant $\alpha \in \mathcal{B}$ is replaced by a given stack π_0 throughout the structure of a term t or of a stack π, by letting:

$$\begin{array}{rclrcll} x\{\alpha := \pi_0\} & \equiv & x & c\{\alpha := \pi_0\} & \equiv & c & \\ (\lambda x\,.\,t)\{\alpha := \pi_0\} & \equiv & \lambda x\,.\,t\{\alpha := \pi_0\} & \alpha\{\alpha := \pi_0\} & \equiv & \pi_0 & \\ (t_1 t_2)\{\alpha := \pi_0\} & \equiv & t_1\{\alpha := \pi_0\} t_2\{\alpha := \pi_0\} & \alpha'\{\alpha := \pi_0\} & \equiv & \alpha' & (\text{if } \alpha' \not\equiv \alpha) \\ \mathsf{k}_\pi\{\alpha := \pi_0\} & \equiv & \mathsf{k}_{\pi\{\alpha := \pi_0\}} & (t \cdot \pi)\{\alpha := \pi_0\} & \equiv & t\{\alpha := \pi_0\} \cdot \pi\{\alpha := \pi_0\} & \end{array}$$

This operation is generalized to parallel substitutions $\{\alpha_1 := \pi_1, \ldots, \alpha_n := \pi_n\}$ in the obvious way.

3. Classical second-order arithmetic

In Section 2, we have presented the *computing facet* of the theory of classical realizability. In this section, we shall now present its *logical facet*, by introducing the language of classical second-order logic with the corresponding type system. In Section 3.3, we shall focus on the particular case of *second-order arithmetic*, and present its axioms.

3.1. *The language of second-order logic*

The language of second-order logic distinguishes two kinds of expressions: *first-order expressions*[†], that represent individuals, and *formulas*, that represent propositions about individuals and sets of individuals (represented using second-order variables as we shall see below).

3.1.1. *First-order expressions.* First-order expressions are formally defined from the following sets of symbols:

— A *first-order signature* Σ defining *function symbols* with their arities, and considering *constant symbols* as function symbols of arity 0.

— A denumerable set $\mathcal{V}_1$ of *first-order variables*. For convenience, we shall still use the lowercase letters x, y, z, etc. to denote first-order variables, but these variables should not be confused with the λ-variables introduced in Section 2.

Definition 5 (first-order expressions). First-order expressions are inductively defined from the following two rules:

1. If $x \in \mathcal{V}_1$ is a first-order variable, then x is a first-order expression.
2. If $f \in \Sigma$ is a function symbol of arity $k \geqslant 0$ and if $e_1, \ldots, e_k$ are first-order expressions, then $f(e_1, \ldots, e_k)$ is a first-order expression.

The set $FV(e)$ of all (free) variables of a first-order expression e is defined as expected, as well as the corresponding operation of substitution, that we still write $e\{x := e'\}$.

[†] Here, we prefer the terminology of a *first-order expression* to the more standard terminology of a *first-order term* to avoid a possible confusion with λ_c-terms.

3.1.2. *Formulas.* Formulas of second-order logic are defined from an additional set of symbols $\mathcal{V}_2$ of *second-order variables* (or *predicate variables*), using the uppercase letters X, Y, Z, etc. to represent such variables. We assume that each second-order variable X comes with an arity $k \geqslant 0$ (that we shall often leave implicit, since it can be easily inferred from the context), and that for each arity $k \geqslant 0$, the subset of $\mathcal{V}_2$ formed by all second-order variables of arity k is denumerable.

Intuitively, second-order variables of arity 0 represent (unknown) propositions, unary predicate variables represent predicates over individuals (or *sets* of individuals) whereas binary predicate variables represent binary relations (or sets of pairs), etc.

Definition 6 (formulas). Formulas of second-order logic are inductively defined from the following four rules:

1. If $X \in \mathcal{V}_2$ is a predicate variable of arity $k \geqslant 0$ and if $e_1, \ldots, e_k$ are first-order expressions, then $X(e_1, \ldots, e_k)$ is a formula.
2. If A and B are formulas, then $A \Rightarrow B$ is a formula.
3. If $x \in \mathcal{V}_1$ is a first-order variable and if A is a formula, then $\forall x\, A$ is a formula.
4. If $X \in \mathcal{V}_2$ is a second-order variable and if A is a formula, then $\forall X\, A$ is a formula.

The set of free variables of a formula A is written $FV(A)$. (This set may contain both first-order and second-order variables.) As usual, formulas are identified up to α-conversion, neglecting differences in bound variable names. Given a formula A, a first-order variable x and a first-order expression e, we denote by $A\{x := e\}$ the formula obtained by replacing every free occurrence of x by the first-order expression e in the formula A, possibly renaming some bound variables of A to avoid name clashes.

3.1.3. *Predicates and second-order substitution.* We call a *predicate of arity k* any expression of the form $P \equiv \lambda x_1 \cdots x_k . C$ where $x_1, \ldots, x_k$ are k pairwise distinct first-order variables and where C is an arbitrary formula. Here, we (ab)use the λ-notation to indicate which variables $x_1, \ldots, x_k$ are abstracted in the formula C, but this notation should not be confused with the abstraction of the λ_c-calculus.

The set of free variables of a k-ary predicate $P \equiv \lambda x_1 \cdots x_k . C$ is defined by $FV(P) \equiv FV(C) \setminus \{x_1, \ldots, x_k\}$, and the application of the predicate $P \equiv \lambda x_1 \cdots x_k . C$ to a k-tuple of first-order expressions $e_1, \ldots, e_k$ is defined by letting

$$P(e_1, \ldots, e_k) \equiv (\lambda x_1 \cdots x_k . C)(e_1, \ldots, e_k) \equiv C\{x_1 := e_1, \ldots, x_k := e_k\}$$

(by analogy with β-reduction). From this definition, it is clear that every predicate variable X of arity k can be seen as a k-ary predicate as well, namely, as the k-ary predicate $\lambda x_1 \cdots x_k . X(x_1, \ldots, x_k)$, whose only free variable is X. The reader can easily check that the meaning of the notation $X(e_1, \ldots, e_k)$ does not depend on whether we read it as an atomic formula (considering X as a predicate variable) or as the application of the predicate $X \equiv \lambda x_1 \cdots x_k . X(x_1, \ldots, x_k)$ to the k-tuple of first-order expressions $e_1, \ldots, e_k$.

Given a formula A, a k-ary predicate variable X and an actual k-ary predicate P, we finally define the operation of *second-order substitution* $A\{X := P\}$ as follows:

$$\begin{array}{rcll} X(e_1,\dots,e_k)\{X := P\} & \equiv & P(e_1,\dots,e_k) & \\ Y(e_1,\dots,e_m)\{X := P\} & \equiv & Y(e_1,\dots,e_m) & (Y \not\equiv X) \\ (A \Rightarrow B)\{X := P\} & \equiv & A\{X := P\} \Rightarrow B\{X := P\} & \\ (\forall x\, A)\{X := P\} & \equiv & \forall x\, A\{X := P\} & (x \notin FV(P)) \\ (\forall X\, A)\{X := P\} & \equiv & \forall X\, A & \\ (\forall Y\, A)\{X := P\} & \equiv & \forall Y\, A\{X := P\} & (Y \not\equiv X,\ Y \notin FV(P)) \end{array}$$

3.1.4. *Second-order encodings.* Although the formulas of the language of second-order logic are constructed from atomic formulas only using implication and first- and second-order universal quantifications, we can define other logical constructions (negation, conjunction disjunction, first- and second-order existential quantification as well as Leibniz equality) using the so called second-order encodings:

$$\begin{array}{rclrcl} \bot & \equiv & \forall Z\, Z & A \Leftrightarrow B & \equiv & (A \Rightarrow B) \wedge (B \Rightarrow A) \\ \neg A & \equiv & A \Rightarrow \bot & \exists x\, A(x) & \equiv & \forall Z\, (\forall x\, (A(x) \Rightarrow Z) \Rightarrow Z) \\ A \wedge B & \equiv & \forall Z\, ((A \Rightarrow B \Rightarrow Z) \Rightarrow Z) & \exists X\, A(X) & \equiv & \forall Z\, (\forall X\, (A(X) \Rightarrow Z) \Rightarrow Z) \\ A \vee B & \equiv & \forall Z\, ((A \Rightarrow Z) \Rightarrow (B \Rightarrow Z) \Rightarrow Z) & e_1 = e_2 & \equiv & \forall Z\, (Z(e_1) \Rightarrow Z(e_2)) \end{array}$$

(where Z is a fresh second-order variable).

3.2. *A type system for classical second-order logic*

Through the formulas-as-types correspondence (Girard 1989; Howard 1969), we can see any formula A of second-order logic as a type, namely, as the type of its proofs. We shall thus present the deduction system of classical second-order logic as a type system based on a typing judgment of the form $\Gamma \vdash t : A$, where

— Γ is a typing context of the form $\Gamma \equiv x_1 : B_1, \dots, x_n : B_n$, where $x_1, \dots, x_n$ are pairwise distinct λ-variables and where $B_1, \dots, B_n$ are arbitrary propositions;
— t is a proof-like term, i.e. a λ_c-term containing no continuation constant k_π;
— A is a formula of second-order logic.

Given a typing context $\Gamma \equiv x_1 : B_1, \dots, x_n : B_n$, we write $\mathrm{dom}(\Gamma) = \{x_1, \dots, x_n\}$ (this is a finite set of λ-variables) and $FV(\Gamma) = FV(B_1) \cup \cdots \cup FV(B_n)$ (this is a finite set of first- and second-order variables).

The type system of classical second-order logic is then defined from the typing rules of Figure 1. These typing rules are the usual typing rules of AF2 (Krivine 1993), plus a specific typing rule for the instruction $\mathbf{c\!c}$ that permits to recover the full strength of classical logic.

Using the encodings of Section 3.1.4, we can derive from the typing rules of Figure 1 the usual introduction and elimination rules of absurdity, conjunction, disjunction, (first- and second-order) existential quantification and Leibniz equality (Krivine 1993). The typing rule for call/cc (law of Peirce) allows us to construct proof-terms for classical reasoning principles such as the excluded middle, *reductio ad absurdum*, de Morgan laws, etc.

$$\frac{}{\Gamma \vdash x : A}\ {\scriptstyle (x:A)\in\Gamma}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x . t : A \Rightarrow B} \qquad \frac{\Gamma \vdash t : A \Rightarrow B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : B}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t : \forall x\, A}\ {\scriptstyle x \notin FV(\Gamma)} \qquad \frac{\Gamma \vdash t : \forall x\, A}{\Gamma \vdash t : A\{x := e\}}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t : \forall X\, A}\ {\scriptstyle X \notin FV(\Gamma)} \qquad \frac{\Gamma \vdash t : \forall X\, A}{\Gamma \vdash t : A\{X := P\}}$$

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

Fig. 1. Typing rules of second-order logic.

3.3. *Classical second-order arithmetic (PA2)*

From now on, we consider the particular case of *second-order arithmetic* (PA2), where first-order expressions are intended to represent natural numbers. For that, we assume that every k-ary function symbol $f \in \Sigma$ comes with an interpretation in the standard model of arithmetic as a function $[\![f]\!] : \mathbb{N}^k \to \mathbb{N}$, so that we can give a denotation $[\![e]\!] \in \mathbb{N}$ to every closed first-order expression e.

For convenience, we assume that the signature Σ contains a constant symbol 0 ('zero'), a unary function symbol s ('successor') as well as a function symbol f for every primitive recursive function (including symbols $+$, $\times$, etc.), each of them being given its standard interpretation in $\mathbb{N}$. In this way, every numeral $n \in \mathbb{N}$ is represented in the world of first-order expressions as the closed expression $s^n(0)$ that we still write n, since $[\![s^n(0)]\!] = n$.

3.3.1. *Induction.* Following Dedekind's construction of natural numbers, we consider the predicate $\mathsf{Nat}(x)$ (Girard 1989; Krivine 1993) defined by

$$\mathsf{Nat}(x) \;\equiv\; \forall Z\, (Z(0) \Rightarrow \forall y\, (Z(y) \Rightarrow Z(s(y))) \Rightarrow Z(x)),$$

that defines the smallest class of individuals containing zero and closed under the successor function. One of the main properties of the logical system presented above is that the axiom of induction, that we can write $\forall x\, \mathsf{Nat}(x)$, is not derivable from the rules of Figure 1. As proved in Krivine (2009, Theorem 12), this axiom is even not (universally) realizable in general. To recover the strength of arithmetic reasoning, we need to relativize all first-order quantifications to the class $\mathsf{Nat}(x)$ of Dedekind numerals using the shorthands for *numeric quantifications*[†]

$$\begin{aligned} \forall^{\mathsf{N}} x\, A(x) &\equiv \forall x\, (\mathsf{Nat}(x) \Rightarrow A(x)) \\ \exists^{\mathsf{N}} x\, A(x) &\equiv \forall Z\, (\forall x (\mathsf{Nat}(x) \Rightarrow A(x) \Rightarrow Z) \Rightarrow Z) \end{aligned}$$

[†] From a computational point of view, the numeric quantifications $\forall^{\mathsf{N}} x\, A(x)$ and $\exists^{\mathsf{N}} x\, A(x)$ play the same role as the dependent product $\Pi x : \mathsf{Nat}. A(x)$ and the dependent sum $\Sigma x : \mathsf{Nat}. A(x)$ in type theory (Martin-Löf 1998), putting aside the subtleties coming from the fact that we work here in a system that is both classical and impredicative.

so that the *relativized induction axiom* becomes provable in second-order logic (Krivine 1993):

$$\forall Z\,(Z(0) \Rightarrow \forall^{\mathsf{N}} x\,(Z(x) \Rightarrow Z(s(x))) \Rightarrow \forall^{\mathsf{N}} Z(x))\,.$$

3.3.2. *The axioms of PA2.* Formally, a formula A is a *theorem* of second-order arithmetic (PA2) if, considered as a type, it has a term according to Figure 1 under a typing context of the form $x_1 : A_1, \ldots, x_n : A_n$, where $A_1, \ldots, A_n$ are *axioms* of PA2.

Here, the axioms of PA2 are the two axioms

— $\forall x\,\forall y\,(s(x) = s(y) \Rightarrow x = y)$ (Peano 3rd axiom)
— $\forall x\,\neg(s(x) = 0)$ (Peano 4th axiom)

expressing that the successor function is injective and not surjective, as well as the definitional equalities attached to the (primitive recursive) function symbols of the signature:

— $\forall x\,(x + 0 = x)$, $\forall x\,\forall y\,(x + s(y) = s(x + y))$
— $\forall x\,(x \times 0 = 0)$, $\forall x\,\forall y\,(x \times s(y) = (x \times y) + x)$
— etc.

Unlike the non-relativized induction axiom – that requires a special treatment in PA2 – we shall see in Section 4.6 that all these axioms are realized by simple proof-like terms.

4. Classical realizability semantics

4.1. *Generalities*

Given a particular instance of the λ_c-calculus (defined from particular sets $\mathcal{B}$, $\mathcal{C}$ and from a particular relation of evaluation $\succ_1$ as described in Section 2), we shall now build a classical realizability model in which every closed formula A of the language of PA2 will be interpreted as a set of closed terms $|A| \subseteq \Lambda$, called the *truth value* of A, and whose elements will be called the *realizers* of A.

4.1.1. *Poles, truth values and falsity values.* Formally, the construction of the realizability model is parameterized by a *pole*[†] $\perp\!\!\!\perp$ in the sense of the following definition:

Definition 7 (poles). A *pole* is any set of processes $\perp\!\!\!\perp \subseteq \Lambda \star \Pi$ which is closed under anti-evaluation, in the sense that both conditions $p \succ p'$ and $p' \in \perp\!\!\!\perp$ together imply that $p \in \perp\!\!\!\perp$ for all processes $p, p' \in \Lambda \star \Pi$.

[†] In Guillermo (2008) and Krivine (2003;2009), *poles* are also called *models*. The reason is that each pole $\perp\!\!\!\perp$ defines a theory $\mathcal{T}_{\perp\!\!\!\perp}$, which is formed by all the closed formulas realized by a proof-like term. The theory $\mathcal{T}_{\perp\!\!\!\perp}$ – which is an extension of PA2 by Theorem 17 p. 1287 – is consistent if and only if the formula $\bot$ is realized by no proof-like term, in which case we say that the pole $\perp\!\!\!\perp$ is *consistent*. In this case, the theory $\mathcal{T}_{\perp\!\!\!\perp}$ induced by $\perp\!\!\!\perp$ has (by completeness) at least a model in the sense of Tarski, which is also a particular model of PA2.

There are mainly two methods to define a pole $\perp\!\!\!\perp$ from an arbitrary set of processes P:

— The first method is to define the pole $\perp\!\!\!\perp$ as the set of all processes that reach an element of P after zero, one or several evaluation steps, that is:

$$\perp\!\!\!\perp \;\equiv\; \{p \in \Lambda \star \Pi \;:\; \exists p' \in P\ (p \succ p')\}\,.$$

By definition, the set $\perp\!\!\!\perp$ is the smallest pole that contains the set of processes P as a subset. In what follows, we shall say that this definition is *goal-oriented.*

— The second method is to define the pole $\perp\!\!\!\perp$ as the complement set of the union of all threads starting from an element of P, that is:

$$\perp\!\!\!\perp \;\equiv\; \Big(\bigcup_{p \in P} \mathbf{th}(p)\Big)^c \;\equiv\; \bigcap_{p \in P} \big(\mathbf{th}(p)\big)^c\,.$$

Here, the set $\perp\!\!\!\perp$ is now the largest pole that does not intersect P. In what follows, we shall say that this definition is *thread oriented.*

Let us now consider a fixed pole $\perp\!\!\!\perp$. We call a *falsity value* any set of stacks $S \subseteq \Pi$. Every falsity value $S \subseteq \Pi$ induces a *truth value* $S^{\perp\!\!\!\perp} \subseteq \Lambda$ that is defined by

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

Intuitively, every falsity value $S \subseteq \Pi$ represents a particular set of *tests*, while the corresponding truth value $S^{\perp\!\!\!\perp}$ represents the set of all *programs* that passes all tests in S (w.r.t. the pole $\perp\!\!\!\perp$, that can be seen as the *challenge*). From the definition of $S^{\perp\!\!\!\perp}$, it is clear that the larger the falsity value S, the smaller the corresponding truth value $S^{\perp\!\!\!\perp}$, and vice versa.

In classical realizability, the semantics of a closed formula A is primarily given by a falsity value $\|A\| \subseteq \Pi$ that defines the set of all tests that should be passed by all the realizers of A. The corresponding truth value $|A| \subseteq \Lambda$ (i.e. the set of all realizers of A) is then defined indirectly from the equation $|A| = \|A\|^{\perp\!\!\!\perp}$.

4.1.2. *Formulas with parameters.* In order to interpret second-order variables that occur in a given formula A, it is convenient to enrich the language of PA2 with a new predicate symbol $\dot{F}$ of arity k for every *falsity value function* F of arity k, that is, for every function $F : \mathbb{N}^k \to \mathfrak{P}(\Pi)$ that associates a falsity value $F(n_1,\ldots,n_k) \subseteq \Pi$ to every k-tuple $(n_1,\ldots,n_k) \in \mathbb{N}^k$. A formula of the language enriched with the predicate symbols $\dot{F}$ is then called a *formula with parameters*. Formally:

Definition 8 (formulas with parameters). The set of all formulas with parameters is inductively defined from the rules (1)–(4) of Definition 6 (replacing the expression 'formula' by 'formula with parameters') plus the following rule:

5. If $F : \mathbb{N}^k \to \mathfrak{P}(\Pi)$ is a falsity value function of arity $k \geqslant 0$ and if $e_1,\ldots,e_k$ are first-order expressions, then $\dot{F}(e_1,\ldots,e_k)$ is a formula with parameters.

The notions of a *predicate with parameters* and of a *typing context with parameters* are defined similarly. The notations $FV(A)$, $FV(P)$, $FV(\Gamma)$, $\mathrm{dom}(\Gamma)$, $A\{x := e\}$, $A\{X := P\}$, etc. are extended to all formulas A with parameters, to all predicates P with parameters and to all typing contexts Γ with parameters in the obvious way.

Let us insist on the fact that this extension only affects the language of formulae, predicates and typing contexts, whose cardinality jumps from the denumerable to the power of continuum (i.e. the cardinality of the sets $\mathfrak{P}(\Pi)$ and $\mathbb{N}^k \to \mathfrak{P}(\Pi)$). On the other hand, proof terms and realizers (stacks, and processes) remain unchanged.

In what follows, we shall write $\top \equiv \dot{\varnothing}$ the formula associated with the empty falsity value. (Since $\varnothing^{\perp\!\!\!\perp} = \Lambda$, the formula $\top$ will represent the type of all λ_c-terms.)

4.2. *Definition of the interpretation function*

The interpretation of the closed formulas with parameters is defined as follows:

Definition 9 (interpretation of closed formulas with parameters). The falsity value $\|A\| \subseteq \Pi$ of a closed formula A with parameters is defined by induction on the number of connectives/quantifiers in A from the equations

$$\begin{aligned}
\|\dot{F}(e_1,\ldots,e_k)\| &= F([\![e_1]\!],\ldots,[\![e_k]\!]) \\
\|A \Rightarrow B\| &= |A| \cdot \|B\| \quad = \quad \{t \cdot \pi \ : \ t \in |A|,\ \pi \in \|B\|\} \\
\|\forall x\, A\| &= \bigcup_{n \in \mathbb{N}} \|A\{x := n\}\| \\
\|\forall X\, A\| &= \bigcup_{F : \mathbb{N}^k \to \mathfrak{P}(\Pi)} \|A\{X := \dot{F}\}\| \qquad (\text{if } X \text{ has arity } k)
\end{aligned}$$

whereas its truth value $|A| \subseteq \Lambda$ is defined by $|A| = \|A\|^{\perp\!\!\!\perp}$.

Since the falsity value $\|A\|$ (resp. the truth value $|A|$) of A actually depends on the pole $\perp\!\!\!\perp$, we shall write it sometimes $\|A\|_{\perp\!\!\!\perp}$ (resp. $|A|_{\perp\!\!\!\perp}$) to recall the dependence. Given a closed formula A with parameters and a closed term $t \in \Lambda$, we say that

— t *realizes* A and write $t \Vdash A$ when $t \in |A|_{\perp\!\!\!\perp}$.
(This notion is relative to a particular pole $\perp\!\!\!\perp$.)

— t *universally realizes* A and write $t \Vvdash A$ when $t \in |A|_{\perp\!\!\!\perp}$ for all poles $\perp\!\!\!\perp$.

From these definitions, we clearly have

$$|\forall x\, A| \ = \ \bigcap_{n \in \mathbb{N}} |A\{x := n\}| \qquad \text{and} \qquad |\forall X\, A| \ = \ \bigcap_{F : \mathbb{N}^k \to \mathfrak{P}(\Pi)} |A\{X := \dot{F}\}|\,.$$

On the other hand, the truth value $|A \Rightarrow B|$ of an implication $A \Rightarrow B$ slightly differs from its traditional interpretation in Kleene's realizability (Kleene 1945). Writing

$$|A| \to |B| \ = \ \{t \in \Lambda \ : \ \text{for all } u \in \Lambda,\ u \in |A| \text{ implies } tu \in |B|\}\,,$$

we easily check that:

Lemma 10. For all closed formulas A and B with parameters:

1. $|A \Rightarrow B| \subseteq |A| \to |B|$ (adequacy of modus ponens).

2. The converse inclusion does not hold in general, unless the pole $\perp\!\!\!\perp$ is insensitive to the rule (PUSH), that is: $tu \star \pi \in \perp\!\!\!\perp$ iff $t \star u \cdot \pi \in \perp\!\!\!\perp$ (for all $t, u \in \Lambda$, $\pi \in \Pi$).
3. In all cases, $t \in (|A| \to |B|)$ implies $\lambda x\,.\,tx \in |A \Rightarrow B|$ (for all $t \in \Lambda$).

Proof.

1. Let $t \in |A \Rightarrow B|$ and $u \in |A|$. To prove that $tu \in |B|$, we consider an arbitrary stack $\pi \in \|B\|$. By applying the rule (PUSH) we get $tu \star \pi \succ_1 t \star u \cdot \pi \in \perp\!\!\!\perp$, since $t \in |A \Rightarrow B|$ and $u \cdot \pi \in \|A \Rightarrow B\|$. Hence, $tu \star \pi \in \perp\!\!\!\perp$ by anti-evaluation.
2. Let $t \in |A| \to |B|$. To prove that $t \in |A \Rightarrow B|$, we consider an arbitrary element of the falsity value $\|A \Rightarrow B\|$, that is, a stack $u \cdot \pi$ where $u \in |A|$ and $\pi \in \|B\|$. We clearly have $tu \star \pi \in \perp\!\!\!\perp$, since $tu \in |B|$ from our assumption on t. But since $\perp\!\!\!\perp$ is insensitive to the rule (PUSH), we also have $t \star u \cdot \pi \in \perp\!\!\!\perp$.
3. Let $t \in |A| \to |B|$. To prove that $\lambda x\,.\,tx \in |A \Rightarrow B|$, we consider an arbitrary element of the falsity value $\|A \Rightarrow B\|$, that is, a stack $u \cdot \pi$ where $u \in |A|$ and $\pi \in \|B\|$. We have $\lambda x\,.\,tx \star u \cdot \pi \succ_1 tu \star \pi \in \perp\!\!\!\perp$ (since $tu \in |B|$), hence $\lambda x\,.\,tx \star u \cdot \pi \in \perp\!\!\!\perp$ by anti-evaluation.

□

Lemma 11 (law of Peirce). Let A and B be two closed formulas with parameters:

1. If $\pi \in \|A\|$, then $\mathsf{k}_\pi \Vdash A \Rightarrow B$.
2. $\text{cc} \Vvdash ((A \Rightarrow B) \Rightarrow A) \Rightarrow A$.

Proof.

1. Let $\pi \in \|A\|$. To prove that $\mathsf{k}_\pi \in |A \Rightarrow B|$, we need to check that $\mathsf{k}_\pi \star t \cdot \pi' \in \perp\!\!\!\perp$ for all $t \in |A|$ and $\pi' \in \|B\|$. By applying the rule (RESTORE) we get $\mathsf{k}_\pi \star t \cdot \pi' \succ_1 t \star \pi \in \perp\!\!\!\perp$ (since $t \in |A|$ and $\pi \in \|A\|$), hence $\mathsf{k}_\pi \star t \cdot \pi' \in \perp\!\!\!\perp$ by anti-evaluation.
2. To prove that $\text{cc} \Vdash ((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ (for an arbitrary pole $\perp\!\!\!\perp$), we need to check that $\text{cc} \star t \cdot \pi \in \perp\!\!\!\perp$ for all $t \in |(A \Rightarrow B) \Rightarrow A|$ and $\pi \in \|A\|$. By applying the rule (SAVE) we get $\text{cc} \star t \cdot \pi \succ_1 t \star \mathsf{k}_\pi \cdot \pi$. But since $\mathsf{k}_\pi \in |A \Rightarrow B|$ (from (1)) and $\pi \in \|A\|$, we have $\mathsf{k}_\pi \cdot \pi \in \|(A \Rightarrow B) \Rightarrow A\|$, so that $t \star \mathsf{k}_\pi \cdot \pi \in \perp\!\!\!\perp$. Hence $\text{cc} \star t \cdot \pi \in \perp\!\!\!\perp$ by anti-evaluation.

□

4.3. *Valuations and substitutions*

In order to express the soundness invariants relating the type system of Section 3 with the classical realizability semantics defined above, we need to introduce some more terminology.

Definition 12 (valuations). A *valuation* is a function ρ that associates a natural number $\rho(x) \in \mathbb{N}$ to every first-order variable x and a falsity value function $\rho(X) : \mathbb{N}^k \to \mathfrak{P}(\Pi)$ to every second-order variable X of arity k.

— Given a valuation ρ, a first-order variable x and a natural number $n \in \mathbb{N}$, we denote by $(\rho, x \leftarrow n)$ the valuation defined by:

$$(\rho, x \leftarrow n) \;=\; \rho_{|\,\mathrm{dom}(\rho)\setminus\{x\}} \cup \{x \leftarrow n\}\,.$$

— Given a valuation ρ, a second-order variable X of arity k and a falsity value function $F : \mathbb{N}^k \to \mathfrak{P}(\Pi)$, we denote by $(\rho, x \leftarrow F)$ the valuation defined by:

$$(\rho, x \leftarrow F) \;=\; \rho_{|\operatorname{dom}(\rho)\setminus\{X\}} \cup \{X \leftarrow F\}\,.$$

To every pair (A, ρ) formed by a (possibly open) formula A of PA2 and a valuation ρ, we associate a *closed* formula with parameters $A[\rho]$ that is defined by

$$A[\rho] \;\equiv\; A\{x_1 := \rho(x_1), \dots, x_n := \rho(x_n), X_1 := \dot{\rho}(X_1), \dots, X_m := \dot{\rho}(X_m)\}$$

where $x_1, \dots, x_n, X_1, \dots, X_m$ are the free variables of A, and writing $\dot{\rho}(X_i)$ the predicate symbol associated to the falsity value function $\rho(X_i)$. This operation naturally extends to typing contexts by letting $(x_1 : A_1, \dots, x_n : A_n)[\rho] \equiv x_1 : A_1[\rho], \dots, x_n : A_n[\rho]$.

Definition 13 (substitutions). A *substitution* is a finite function σ from λ-variables to closed λ_c-terms. Given a substitution σ, a λ-variable x and a closed λ_c-term u, we denote by $\sigma, x := u$ the substitution defined by $(\sigma, x := u) \equiv \sigma_{|\operatorname{dom}(\sigma)\setminus\{x\}} \cup \{x := u\}$.

Given an open λ_c-term t and a substitution σ, we denote by $t[\sigma]$ the term defined by

$$t[\sigma] \;\equiv\; t\{x_1 := \sigma(x_1), \dots, x_n := \sigma(x_n)\}$$

where $\operatorname{dom}(\sigma) = \{x_1, \dots, x_n\}$. Notice that $t[\sigma]$ is closed as soon as $FV(t) \subseteq \operatorname{dom}(\sigma)$. We say that a substitution σ *realizes* a closed context Γ with parameters and write $\sigma \Vdash \Gamma$ if:

— $\operatorname{dom}(\sigma) = \operatorname{dom}(\Gamma)$;
— $\sigma(x) \Vdash A$ for every declaration $(x : A) \in \Gamma$.

4.4. *Adequacy*

Given a fixed pole $\perp\!\!\!\perp$, we say that

— A typing judgment $\Gamma \vdash t : A$ is *adequate* (w.r.t. the pole $\perp\!\!\!\perp$) if for all valuations ρ and for all substitutions $\sigma \Vdash \Gamma[\rho]$ we have $t[\sigma] \Vdash A[\rho]$.
— More generally, we say that an inference rule

$$\frac{J_1 \quad \cdots \quad J_n}{J_0}$$

is adequate (w.r.t. the pole $\perp\!\!\!\perp$) if the adequacy of all typing judgments $J_1, \dots, J_n$ implies the adequacy of the typing judgment J_0.

From the latter definition, it is clear that a typing judgment that is derivable from a set of adequate inference rules is adequate too. In Section 4.5, we shall extend the notion of adequacy to new judgments of subtyping and of subtyping equivalence.

Proposition 14 (adequacy). The typing rules of Figure 1 are adequate w.r.t. any pole $\perp\!\!\!\perp$, as well as all the judgments $\Gamma \vdash t : A$ that are derivable from these rules.

(The proof of this result can be found in Krivine (2009).)

Since the typing rules of Figure 1 involve no continuation constant, every realizer that comes from a proof of second order logic by Proposition 14 is thus a proof-like term.

$$\frac{}{A \leqslant A} \qquad \frac{A \leqslant B \quad B \leqslant C}{A \leqslant C} \qquad \frac{\Gamma \vdash t : A \quad A \leqslant B}{\Gamma \vdash t : B}$$

$$\frac{A \leqslant B \quad B \leqslant A}{A \simeq B} \qquad \frac{A \simeq B}{A \leqslant B} \qquad \frac{A \simeq B}{B \leqslant A}$$

$$\frac{}{\bot \leqslant A} \qquad \frac{}{A \leqslant \top} \qquad \frac{}{\top \leqslant A \Rightarrow \top}$$

$$\frac{}{\forall x\, A \leqslant A\{x := e\}} \qquad \frac{}{\forall X\, A \leqslant A\{X := P\}}$$

$$\frac{A \leqslant B}{A \leqslant \forall x\, B}\ {\scriptstyle x \notin FV(A)} \qquad \frac{A \leqslant B}{A \leqslant \forall X\, B}\ {\scriptstyle X \notin FV(A)} \qquad \frac{A' \leqslant A \quad B \leqslant B'}{A \Rightarrow B \leqslant A' \Rightarrow B'}$$

$$\frac{}{\forall x\,(A \Rightarrow B) \leqslant A \Rightarrow \forall x\, B}\ {\scriptstyle x \notin FV(A)} \qquad \frac{}{\forall X\,(A \Rightarrow B) \leqslant A \Rightarrow \forall X\, B}\ {\scriptstyle X \notin FV(A)}$$

Fig. 2. Adequate rules of subtyping and of subtyping equivalence.

4.5. *Subtyping and subtyping equivalence*

In many situations, it is convenient to consider a subtyping judgment $A \leqslant B$ as well as a judgment of subtyping equivalence $A \simeq B$, where A and B are two formulas with parameters. Given a pole $\perp\!\!\!\perp$, we say that

— The subtyping judgment $A \leqslant B$ is *adequate* (w.r.t. the pole $\perp\!\!\!\perp$) if for all valuations ρ we have $\|B[\rho]\| \subseteq \|A[\rho]\|$ (so that $|A[\rho]| \subseteq |B[\rho]|$).

— The subtyping equivalence judgment $A \simeq B$ is *adequate* (w.r.t. the pole $\perp\!\!\!\perp$) if for all valuations ρ, we have $\|A[\rho]\| = \|B[\rho]\|$ (so that $|A[\rho]| = |B[\rho]|$).

The notion of an adequate inference rule (cf Section 4.4) is extended to all the inference rules involving the new forms of judgments $A \leqslant B$ and $A \simeq B$. As before, it is clear that a judgment (of typing, subtyping or of subtyping equivalence) that is derivable from a set of adequate inference rules is adequate too.

Proposition 15. The inference rules of Figure 2 are adequate w.r.t. all poles $\perp\!\!\!\perp$.

Proof. Immediately follows from the definitions. □

From the inference rules of Figure 2, one can derive well-known equivalences of (intuitionistic or classical) realizability, such as:

— $\forall x\, A \simeq A$ and $\forall x\,(A \Rightarrow B) \simeq A \Rightarrow \forall x\, B$ (if $x \notin FV(A)$);

— $\forall X\, A \simeq A$ and $\forall X\,(A \Rightarrow B) \simeq A \Rightarrow \forall X\, B$ (if $X \notin FV(A)$);

— $\forall x\, \forall y\, A \simeq \forall y\, \forall x\, A$, $\forall X\, \forall Y\, A \simeq \forall Y\, \forall X\, A$, and $\forall x\, \forall Y\, A \simeq \forall Y\, \forall x\, A$;

— $A \Rightarrow \top \simeq \top$, etc.

4.6. *Realizing the axioms of PA2*

Let us recall that in second-order arithmetic, Leibniz equality $e_1 = e_2$ is defined by $e_1 = e_2 \equiv \forall Z\,(Z(e_1) \Rightarrow Z(e_2))$.

Proposition 16 (realizing Peano axioms). :

1. $\lambda z\,.\,z \ \Vdash\ \forall x\,\forall y\,(s(x) = s(y) \Rightarrow x = y)$,
2. $\lambda z\,.\,zu \ \Vdash\ \forall x\,(s(x) = 0 \Rightarrow \bot)$ (where u is any term such that $FV(u) \subseteq \{z\}$).
3. $\lambda z\,.\,z \ \Vdash\ \forall x_1 \cdots \forall x_k\,(e_1(x_1,\ldots,x_n) = e_2(x_1,\ldots,x_k))$,
   for all arithmetic expressions $e_1(x_1,\ldots,x_n)$ and $e_2(x_1,\ldots,x_k)$ such that
   $\mathbb{N} \models \forall x_1 \cdots \forall x_k\,(e_1(x_1,\ldots,x_n) = e_2(x_1,\ldots,x_k))$.

(The proof of this proposition can be found in Krivine (2009).)

From this we deduce the main theorem:

Theorem 17 (realizing the theorems of PA2). If A is a theorem of PA2 (in the sense defined in Section 3.3.2), then there is a closed proof-like term t such that $t \Vdash A$.

Proof. Immediately follows from Propositions 14 and 16. □

4.7. *The full standard model of PA2 as a degenerate case*

It is easy to see that when the pole $\perp\!\!\!\perp$ is empty, the classical realizability model defined above collapses to the *full standard model of PA2*, that is, to the model (in the sense of Tarski) where individuals are interpreted by the elements of $\mathbb{N}$ and where second-order variables of arity k are interpreted by all the subsets of $\mathbb{N}^k$. For that, we first notice that when $\perp\!\!\!\perp = \varnothing$, the truth value $S^{\perp\!\!\!\perp}$ associated to an arbitrary falsity value $S \subseteq \Pi$ can only take two different values: $S^{\perp\!\!\!\perp} = \Lambda_c$ when $S = \varnothing$, and $S^{\perp\!\!\!\perp} = \varnothing$ when $S \neq \varnothing$. Moreover, we easily check that the realizability interpretation of implication and universal quantification mimics the standard truth value interpretation of the corresponding logical construction in the case where $\perp\!\!\!\perp = \varnothing$. Writing $\mathscr{M}$ for the full standard model of PA2, we thus easily show that:

Proposition 18. If $\perp\!\!\!\perp = \varnothing$, then for every closed formula A of PA2 we have

$$|A| = \begin{cases} \Lambda & \text{if } \mathscr{M} \models A \\ \varnothing & \text{if } \mathscr{M} \not\models A \end{cases}$$

Proof. We more generally show that for all formulas A and for all valuations ρ closing A (in the sense defined in Section 4.3) we have

$$|A[\rho]| = \begin{cases} \Lambda & \text{if } \mathscr{M} \models A[\tilde{\rho}] \\ \varnothing & \text{if } \mathscr{M} \not\models A[\tilde{\rho}] \end{cases}$$

where $\tilde{\rho}$ is the valuation in $\mathscr{M}$ (in the usual sense) defined by

— $\tilde{\rho}(x) = \rho(x)$ for all first-order variables x;
— $\tilde{\rho}(X) = \{(n_1,\ldots,n_k) \in \mathbb{N}^k : \rho(X)(n_1,\ldots,n_k) = \varnothing\}$ for all second-order variables X of arity k.

(This characterization is proved by a straightforward induction on A.) □

An interesting consequence of the above lemma is the following:

Corollary 19. If a closed formula A has a universal realizer $t \Vdash\!\!\vdash A$, then A is true in the full standard model $\mathscr{M}$ of PA2.

Proof. If $t \Vdash\!\!\vdash A$, then $t \in |A|_\varnothing$. Therefore $|A|_\varnothing = \Lambda$ and $\mathscr{M} \models A$. □

However, the converse implication is false in general, since the formula $\forall x\, \mathsf{Nat}(x)$ (cf Section 3.3.1) that expresses the induction principle over individuals is obviously true in $\mathscr{M}$, but it has no universal realizer when evaluation is deterministic (Krivine 2009, Theorem 12).

5. The specification problem

From now on, we are interested in the *specification problem*, which is to give a purely computational characterization of the universal realizers of a given formula A.

As mentioned in the introduction, this problem is much more subtle in classical realizability than in intuitionistic realizability, which is mainly due to fact that realizers may perform a backtrack at any time. Another source of difficulty comes from the fact that the definition of the λ_c-calculus is open to the introduction of extra instructions such as the ones we have presented in Section 2.3. We cannot reason anymore as in the closed world of the pure λ-calculus, where closed programs start either with an abstraction or with an application. Here, extra instructions can do anything: they can compute the code of a term or a stack (`quote`), they can decode a stack or a term from its code (by introducing a dual instruction `unquote`), they can introduce non-determinism in computations ($\pitchfork$), and they can even introduce non-recursive computations (which is the case if we introduce an instruction solving the halting problem for any Turing machine).

In this section, we shall study the case of very simple formulas for which the specification problem has a simple solution, that does not even depend on any particular set of instructions. In the next section, we shall consider the more ambitious case of Peirce's law, where control structures play a crucial role.

5.1. *The identity type*

In the language of second-order logic, the *identity type* is described by the formula $\forall X\,(X \Rightarrow X)$. We say that a closed term $t \in \Lambda$ is *identity like* if $t \star u \cdot \pi \succ u \star \pi$ for all $u \in \Lambda$ and $\pi \in \Pi$. Examples of identity-like terms are of course the identity function $\mathbf{I} \equiv \lambda x\,.\,x$, but also terms such as $\mathbf{I}\,\mathbf{I}$, $\delta\,\mathbf{I}$ (where $\delta \equiv \lambda x\,.\,xx$), etc.

Proposition 20. For all terms $t \in \Lambda$, the following assertions are equivalent:

1. $t \Vdash\!\!\vdash \forall X\,(X \Rightarrow X)$;
2. t is identity like.

Proof. (2) $\Rightarrow$ (1). Let t be a closed λ_c-term that is identity like. To prove that $t \Vdash\!\!\vdash \forall X\,(X \Rightarrow X)$, let us consider an arbitrary pole $\perp\!\!\!\perp$ and an arbitrary falsity value $S \subseteq \Pi$

to instantiate the variable X. To show that $t \Vdash \dot{S} \Rightarrow \dot{S}$, let us consider an arbitrary element of $\|\dot{S} \Rightarrow \dot{S}\|$, that is, a stack of the form $u \cdot \pi$ where $u \in |\dot{S}| = S^{\perp\!\!\!\perp}$ and $\pi \in \|\dot{S}\| = S$. We have $t \star u \cdot \pi \succ u \star \pi \in \perp\!\!\!\perp$, hence $t \star u \cdot \pi \in \perp\!\!\!\perp$ by anti-evaluation.

(1) $\Rightarrow$ (2). Given a universal realizer $t \Vvdash \forall X\,(X \Rightarrow X)$, a term $u \in \Lambda$ and a stack $\pi \in \Pi$, let us consider the pole $\perp\!\!\!\perp = \{p \ : \ p \succ u\star\pi\}$ and the falsity value $S = \{\pi\}$. Since $u\star\pi \in \perp\!\!\!\perp$ (from the definition of $\perp\!\!\!\perp$), we get $u \Vdash \dot{S}$ (from the definition of S). Hence $u \cdot \pi \in \|\dot{S} \Rightarrow \dot{S}\|$ and thus $t \star u \cdot \pi \in \perp\!\!\!\perp$ (since $t \Vdash \dot{S} \Rightarrow \dot{S}$), which precisely means that $t \star u \cdot \pi \succ u \star \pi$. □

We have thus proved that the universal realizers of the formula $\forall X\,(X \Rightarrow X)$ are precisely the identity-like λ_c-terms, and this, independently of any particular set of instructions $\mathcal{C}$. However, we should not forget that there are many ways to implement identity-like terms using call/cc or other instructions, for instance:

— $\lambda x\,.\,\text{cc}\,(\lambda k\,.\,x)$, $\lambda x\,.\,\text{cc}\,(\lambda k\,.\,k\,x)$, $\lambda x\,.\,\text{cc}\,(\lambda k\,.\,k\,x\,(\delta\,\delta))$,

— $\text{cc}\,(\lambda k\,.\,\mathbf{I})$, $\text{cc}\,(\lambda k\,.\,k\,\mathbf{I})$, $\text{cc}\,(\lambda k\,.\,k\,\mathbf{I}\,\delta\,k)$, $\pitchfork\,\mathbf{I}\,(\delta\,\delta)$, etc.

5.1.1. *Goal-oriented poles versus thread-oriented poles.* It is interesting to notice that in order to prove that universal realizers of $\forall X\,(X \Rightarrow X)$ are identity-like terms, we have introduced for each pair $(u,\pi) \in \Lambda \times \Pi$ the pole $\perp\!\!\!\perp_{u,\pi} = \{p : p \succ u \star \pi\}$ that is generated from the expected final state $u \star \pi$, thus using a goal-oriented definition. However, we could also prove the same implication by using a *thread-oriented* definition as follows:

Alternative proof of (1) $\Rightarrow$ (2). Let us assume that $t \Vvdash \forall X\,(X \Rightarrow X)$, and take two elements $u \in \Lambda$ and $\pi \in \Pi$. We now consider the pole

$$\perp\!\!\!\perp \ \equiv \ \big(\mathbf{th}(t \star u \cdot \pi)\big)^c \ \equiv \ \{p \in \Lambda \star \pi \ : \ (t \star u \cdot \pi \not\succ p)\}$$

as well as the falsity value $S = \{\pi\}$. From the definition of $\perp\!\!\!\perp$, we have $t \star u \cdot \pi \notin \perp\!\!\!\perp$. But since $t \Vdash \dot{S} \Rightarrow \dot{S}$ and $\pi \in \|\dot{S}\|$, we immediately get $u \nVdash \dot{S}$. Which precisely means that $u \star \pi \notin \perp\!\!\!\perp$, so that $u \star \pi \in \mathbf{th}(t \star u \cdot \pi)$. □

Let us briefly compare the ingredients used in both proofs of (1) $\Rightarrow$ (2).

— In the first proof of (1) $\Rightarrow$ (2), we use a goal-oriented definition of the pole $\perp\!\!\!\perp$, by generating $\perp\!\!\!\perp$ from the final state $u \star \pi$ we want to reach. Moreover, this proof of (1) $\Rightarrow$ (2) is purely intuitionistic (from the point of view of meta-theory).

— In the second proof of (1) $\Rightarrow$ (2), we use a thread-oriented definition of the pole $\perp\!\!\!\perp$, by *excluding* from $\perp\!\!\!\perp$ the process $p_0 \equiv t \star u \cdot \pi$ we start from. As a consequence, this second proof is classical, since it relies on the equality $\big(\big(\mathbf{th}(p_0)\big)^c\big)^c = \mathbf{th}(p_0)$.

For the simple formulas we are studying in this section, it is possible to use both constructions indifferently in order to prove that a universal realizer of the considered formula meets the expected specification. However, this is not always the case, and the thread-oriented construction (which is slightly less natural than the goal-oriented construction) proves to be much more powerful in many situations, as illustrated in Krivine (2003) and Guillermo (2008).

5.2. *The Booleans*

Let us now consider the unary predicate $\mathsf{Bool}(x)$ defined by

$$\mathsf{Bool}(x) \;\equiv\; \forall X\,(X(0) \Rightarrow X(1) \Rightarrow X(x))\,.$$

We also denote by Bool the formula $\forall X\,(X \Rightarrow X \Rightarrow X)$ of second-order propositional logic that we get by erasing first-order dependencies in the predicate $\mathsf{Bool}(x)$. From the rules of Figure 2, we can derive that the formula $\mathsf{Bool}(x)$ is a subtype of the formula Bool.

We say that a closed term $t \in \Lambda$ is:

— *True like* if $t \star u_0 \cdot u_1 \cdot \pi \succ u_0 \star \pi$ for all $u_0, u_1 \in \Lambda$ and $\pi \in \Pi$.
— *False like* if $t \star u_0 \cdot u_1 \cdot \pi \succ u_1 \star \pi$ for all $u_0, u_1 \in \Lambda$ and $\pi \in \Pi$.
— *Boolean like* if for all $u_0, u_1 \in \Lambda$ and $\pi \in \Pi$, we have either
$t \star u_0 \cdot u_1 \cdot \pi \succ u_0 \star \pi$ or $t \star u_0 \cdot u_1 \cdot \pi \succ u_1 \star \pi$.

From these definitions, True-like and False-like terms are particular cases of Boolean-like terms. We easily check that:

Proposition 21. For all closed terms $t \in \Lambda$:

1. $t \Vdash \mathsf{Bool}(0)$ iff t is True like;
2. $t \Vdash \mathsf{Bool}(1)$ iff t is False like;
3. The formula $\mathsf{Bool}(n)$ has no universal realizer as soon as $n \geqslant 2$.
4. $t \Vdash \mathsf{Bool}$ iff t is Boolean like.

Proof. The proofs of (1), (2) and (4) proceed similarly to the proof of Proposition 20. (In all cases, we can choose either a goal-oriented or a thread-oriented definition of the pole to show that the universal realizer t meets the expected specification.) For (3), it suffices to notice that the formula $\mathsf{Bool}(n)$ (where $n \geqslant 2$) is false in the standard model of PA2, and thus has no universal realizer. □

The above proposition shows that in classical realizability, universal realizers of the formulas $\mathsf{Bool}(0)$, $\mathsf{Bool}(1)$ and Bool have the same computational behaviour as the intuitionistic realizers of these formulas in AF2 (Krivine 1993) – *mutatis mutandis*[†]. But in AF2, we can actually prove that the set of (intuitionistic) realizers of Bool is the disjoint union of the sets of realizers of the formulas $\mathsf{Bool}(0)$ and $\mathsf{Bool}(1)$. In classical realizability, the situation is more complex due to the presence of extra instructions, as we shall now see.

5.2.1. *Non-deterministic choice operators.*

We say that a closed λ_c-term $t \in \Lambda$ is a *non-deterministic choice operator* if t is both True like and False like. An example of such an operator is the instruction $\pitchfork$ introduced in Section 2.3.

It follows from Proposition 21 that non-deterministic choice operators are exactly the closed λ_c-terms t such that $t \Vdash \mathsf{Bool}(0)$ and $t \Vdash \mathsf{Bool}(1)$ (simultaneously). In classical realizability, the sets of universal realizers of the formulas $\mathsf{Bool}(0)$ and $\mathsf{Bool}(1)$ may thus

[†] In AF2, formulas are not primarily interpreted as sets of stacks (as in classical realizability), but as sets of (possibly open) λ-terms closed under β-equivalence, in the spirit of Kleene realizability. Moreover, the notion of computation of AF2 is ordinary β-reduction rather than weak head-reduction. Up to these differences, the specification of the formulas $\mathsf{Bool}(0)$, $\mathsf{Bool}(1)$ and Bool is the same in AF2 as indicated by Proposition 21.

have a non-empty intersection, depending on the presence of a non-deterministic operator in the calculus. If the calculus provides the instruction $\pitchfork$ described in Section 2.3, then the intersection is nonempty. But if the relation of one step evaluation is deterministic, then the sets of universal realizers of the formulas $\mathsf{Bool}(0)$ and $\mathsf{Bool}(1)$ do not intersect.

5.2.2. *Versatile Booleans.* We call a *versatile Boolean* any Boolean-like term that is neither True like nor False like. Intuitively, a versatile Boolean is a Boolean-like term that sometimes returns its first argument or its second argument, depending on the two arguments it is applied to, or depending on the rest of the stack.

Versatile Booleans cannot exist in the pure λ-calculus, for obvious reasons. But in the λ_c-calculus, it is easy to implement such objects using the instruction `quote` or the instruction `eq` (see Section 2.3), for instance:

— The term $D \equiv \lambda xy\,.\,\mathtt{quote}\,(\lambda n\,.\,\mathsf{even}\,n\,x\,y)$, where even is a pure λ-term that tests whether the numeral it is applied to is even. Notice that the answer ('first' or 'second') given by D does not actually depend on the two arguments it is applied to, since it only depends on the (code of the) rest of the stack!

— The term $E \equiv \lambda xy\,.\,\mathtt{eq}\,x\,\mathbf{I}\,x\,y$. Here the term E returns its first argument if it is equal to $\mathbf{I}$, and its second argument otherwise.

From Proposition 21, both terms D and E are universal realizers of the formula Bool, but none of them universally realizes $\mathsf{Bool}(0)$ or $\mathsf{Bool}(1)$.

5.3. *Interaction constants*

The two examples of versatile Booleans presented above crucially depend on two extra instructions `quote` and `eq` that have no equivalent in the $\lambda(\mu)$-calculus. Indeed, these instructions are able to distinguish syntactically different terms that are computationally equivalent, such as the terms $\mathbf{I}$ and $\mathbf{I}\,\mathbf{I}$ for instance[†].

To understand better the impact of such extra instructions in the λ_c-calculus, we need to introduce the important notion of an interaction constant.

Definition 22 (interaction constants). A constant $\kappa \in \mathcal{C}$ is said to be

— *inert* if for all $\pi \in \Pi$, there is no process p such that $\kappa \star \pi \succ_1 p$;

— *substitutive* if for all $u \in \Lambda$ and for all processes $p, p' \in \Lambda \star \Pi$,
$p \succ_1 p'$ implies $p\{\kappa := u\} \succ_1 p'\{\kappa := u\}$;

— *non-generative* if for all processes $p, p' \in \Lambda \star \Pi$ such that $p \succ_1 p'$, the constant κ cannot occur in p' unless it already occurs in p.

A constant $\kappa \in \mathcal{C}$ that is inert, substitutive and non-generative is then called an *interaction constant*. Similarly, we say that a stack constant $\alpha \in \mathcal{B}$ is

— *substitutive* if for all $\pi \in \Pi$ and for all processes $p, p' \in \Lambda \star \Pi$,
$p \succ_1 p'$ implies $p\{\alpha := \pi\} \succ_1 p'\{\alpha := \pi\}$;

[†] In particular, it is clear that naively extending the pure λ-calculus with any of these two instructions would immediately break the property of confluence. In the λ_c-calculus, the property of determinism is preserved only because computation proceeds according to a fixed evaluation strategy.

— *non-generative* if for all $p, p' \in \Lambda \star \Pi$ such that $p \succ_1 p'$, the stack constant α cannot occur in p' unless it already occurs in p.

We can first notice that substitutive constants are incompatible with the two instructions quote and eq:

Proposition 23. If the calculus of realizers contains one of both instructions quote or eq, then none of the constants $\kappa \in \mathcal{C}$ is substitutive.

Proof. Let us assume that the calculus contains an instruction eq with the evaluation rule (Eq) described in Section 2.3. Given an arbitrary constant $\kappa \in \mathcal{C}$, let us consider the process $p \equiv \mathtt{eq} \star \kappa \cdot \mathbf{I} \cdot \delta \cdot \mathbf{I} \cdot \alpha$, where $\alpha \in \mathcal{B}$ is a fixed stack constant. We notice that $p \succ_1 \mathbf{I} \star \alpha$ (since $\kappa \not\equiv \mathbf{I}$) whereas $p\{\kappa := \mathbf{I}\} \equiv \mathtt{eq} \star \mathbf{I} \cdot \mathbf{I} \cdot \delta \cdot \mathbf{I} \cdot \alpha \succ_1 \delta \star \alpha \not\equiv (\mathbf{I} \star \alpha)\{\kappa := \mathbf{I}\}$, hence the constant κ is not substitutive. The same argument applies to the instruction quote, since the instruction eq can be implemented from it. □

On the other hand, it is clear that if the relation of evaluation $\succ_1$ is defined from the only rules (Grab), (Push), (Save) and (Restore) – and possibly: the rule (Fork) – then all the remaining constants κ in $\mathcal{C}$ (i.e. $\kappa \neq \text{œ}, ⫛$) are interaction constants (and thus substitutive), whereas all the stack constants in $\mathcal{B}$ are substitutive and non-generative.

Substitutive (term and stack) constants are useful to analyse the computational behaviour of realizers in a uniform way. For instance, if we know that a closed term $t \in \Lambda$ is such that

$$t \star \kappa_1 \cdots \kappa_n \cdot \alpha \ \succ\ p$$

where $\kappa_1, \ldots, \kappa_n$ are substitutive constants that do not occur in t, and where α is a substitutive stack constant that does not occur in t too, then we more generally know that

$$t \star u_1 \cdots u_n \cdot \pi \ \succ\ p\{\kappa_1 := u_1, \ldots, \kappa_n := u_n, \alpha := \pi\}$$

for all terms $u_1, \ldots, u_n \in \Lambda$ and for all stacks $\pi \in \Pi$. Intuitively, substitutive constants play in the λ_c-calculus the same role as free variables in the pure λ-calculus.

Using the uniformity of computations that is brought by the presence of substitutive constants, we easily check that:

Proposition 24. If the calculus of realizers has infinitely many substitutive constants and infinitely many substitutive stack constants, then every Boolean-like term is either True like or False like. (Which means that there are no versatile Booleans.)

Proof. Let t be a Boolean-like term, and consider two distinct substitutive constants κ_0 and κ_1 that do not occur in the closed term t as well as a substitutive stack constant α that does not occur in t. (We can always find such constants outside t, since t only contains a finite number of them.) We distinguish two cases:

— Either $t \star \kappa_0 \cdot \kappa_1 \cdot \alpha \succ \kappa_0 \star \alpha$. By substitutivity, we have

$$\begin{aligned} t \star u_0 \cdot u_1 \cdot \pi \ &\equiv\ (t \star \kappa_0 \cdot \kappa_1 \cdot \alpha)\{\kappa_0 := u_0, \kappa_1 := u_1, \alpha := \pi\} \\ &\succ\ (\kappa_0 \star \alpha)\{\kappa_0 := u_0, \kappa_1 := u_1, \alpha := \pi\} \ \equiv\ u_0 \star \pi \end{aligned}$$

for all $u_0, u_1 \in \Lambda$ and $\pi \in \Pi$, which means that t is True like.

— Either $t \star \kappa_0 \cdot \kappa_1 \cdot \alpha \succ \kappa_1 \star \alpha$. Symmetrically, we deduce that t is False like. □

6. Specification of Peirce's law

In this section, we now consider the specification problem for an intrinsically classical reasoning principle: the law of Peirce. In the literature, Peirce's law is indifferently stated as $\forall X\,\forall Y\,(((X \Rightarrow Y) \Rightarrow X) \Rightarrow X)$ or as $\forall X\,((\neg X \Rightarrow X) \Rightarrow X)$. Since the judgment

$$\forall X\,\forall Y\,(((X \Rightarrow Y) \Rightarrow X) \Rightarrow X) \;\simeq\; \forall X\,((\neg X \Rightarrow X) \Rightarrow X)$$

is derivable from the rules of Figure 2, both formulations of this law have the same semantics, and thus the same set of (universal) realizers. In what follows, we shall prefer the simpler formulation $\forall X\,((\neg X \Rightarrow X) \Rightarrow X)$ that involves a single parameter X.

The aim of this section is to specify *all* the universal realizers of Peirce's law. It is clear from Lemma 11 that the instruction œ is one of them, but it is now time to see that the universal realizers of Peirce's law may have a much richer computational behaviour than the one of the instruction œ.

6.1. *The family of terms* $(\text{œ}_{n,p})_{n\geqslant p\geqslant 1}$

To illustrate the possible computational behaviours of the universal realizers of Peirce's law, it is useful to introduce a sequence of closed proof-like terms $\text{œ}_{n,p}$ indexed by all pairs of integers (n, p) such that $n \geqslant p \geqslant 1$.

Given a fixed pair (n, p) such that $n \geqslant p \geqslant 1$, we define for all $i \in [1..n]$ an open proof-like term $K^i_{n,p}[x_0, k, x_1, \ldots, x_{i-1}]$ that only depends on the variables $x_0, k, x_1, \ldots, x_{i-1}$. This finite sequence of open terms is defined from $i = n$ (down) to $i = 1$ by the equations:

$$\begin{aligned} K^n_{n,p}[x_0, k, x_1, \ldots, x_{n-1}] &\equiv \lambda x_n\,.\,k\,x_p \\ K^i_{n,p}[x_0, k, x_1, \ldots, x_{i-1}] &\equiv \lambda x_i\,.\,k\,(x_0\,K^{i+1}_{n,p}[x_0, k, x_1, \ldots, x_i]) \qquad (1 \leqslant i < n) \end{aligned}$$

(Notice that we actually have $FV(K^i_{n,p}[x_0, k, x_1, \ldots, x_{i-1}]) = \{x_0, k\} \cup \{x_p \text{ if } p < i\}$.)

The closed proof-like term $\text{œ}_{n,p}$ is then defined by

$$\text{œ}_{n,p} \;\equiv\; \lambda x_0\,.\,\text{œ}\,(\lambda k\,.\,x_0\,K^1_{n,p}[x_0, k])\,.$$

We easily check that:

Fact 25. For all $u_0, \ldots, u_n \in \Lambda$ and $\pi_0, \ldots, \pi_n \in \Pi$, we have:

$$\begin{aligned} \text{œ}_{n,k} \star u_0 \cdot \pi_0 &\succ u_0 \star K^1_{n,p}[u_0, \mathsf{k}_{\pi_0}] \cdot \pi_0 \\ K^i_{n,p}[u_0, \mathsf{k}_{\pi_0}, u_1, \ldots, u_{i-1}] \star u_i \cdot \pi_i &\succ u_0 \star K^{i+1}_{n,p}[u_0, \mathsf{k}_{\pi_0}, u_1, \ldots, u_i] \cdot \pi_0 \qquad (1 \leqslant i < n) \\ K^n_{n,p}[u_0, \mathsf{k}_{\pi_0}, u_1, \ldots, u_{n-1}] \star u_n \cdot \pi_n &\succ u_p \star \pi_0 \end{aligned}$$

In the particular case where $n = p = 1$, we thus have

$$\begin{aligned} \text{œ}_{1,1} \star u_0 \cdot \pi_0 &\succ u_0 \star (\lambda x_1\,.\,\mathsf{k}_{\pi_0}\,x_1) \cdot \pi_0 \qquad (\text{since } K^1_{1,1}[u_0, \mathsf{k}_{\pi_0}] \equiv \lambda x_1\,.\,\mathsf{k}_{\pi_0}\,x_1) \\ (\lambda x_1\,.\,\mathsf{k}_{\pi_0}\,x_1) \star u_1 \cdot \pi_1 &\succ u_1 \star \pi_0\,, \end{aligned}$$

which makes clear that $\text{œ}_{1,1}$ has the same computational behaviour as œ – which is not surprising since $\text{œ}_{1,1} \equiv \lambda x_0\,.\,\text{œ}\,(\lambda k\,.\,x_0\,(\lambda x_1\,.\,k\,x_1))$ is nothing but the η-long form of œ.

It is easy to check that:

Proposition 26. For all $n \geqslant p \geqslant 1$ the judgment $\vdash \text{œ}_{n,p} : \forall X\,((\neg X \Rightarrow X) \Rightarrow X)$ is derivable from the rules of Figure 1, so that $\text{œ}_{n,p}$ is a universal realizer of Peirce's law.

Proof. It suffices to check for $i = n$ (down) to $i = 1$ that the judgment

$$x_0 : \neg X \Rightarrow X,\ k : \neg X,\ x_1 : X,\ \ldots,\ x_{i-1} : X \vdash K^{i}_{n,p}[x_0, k, x_1, \ldots, x_{i-1}] : \neg X$$

is derivable from the rules of Figure 1 (where X is a fixed second-order variable), hence we have $\vdash \text{œ}_{n,p} \equiv \lambda x_0 \,.\, \text{œ}\,(\lambda k \,.\, x_0\, K^{1}_{n,p}[x_0, k]) : \forall X\,((\neg X \Rightarrow X) \Rightarrow X)$. □

In Section 6.3, we shall prove that in the presence of infinitely many interaction constants and of infinitely many substitutive and non-generative stack constants, every universal realizer of Peirce's law behaves as one of the terms $\text{œ}_{n,p}$.

6.2. *A first game* $\mathbb{G}_0$

To understand the computational behaviour of the universal realizers of Peirce's law, it is convenient to present computations in the form of a game between two players $\exists$ (player, or defender) and $\forall$ (opponent, or attacker), that we denote by $\mathbb{G}_0$.

During the game, the two players $\exists$ and $\forall$ exchange arguments for (using closed terms) and against (using stacks) particular formulas. The aim of player $\exists$ is then to exhibit a contradiction from the only arguments raised by the opponent, thus defeating the opponent's attack. The game $\mathbb{G}_0$ starts with an *initialization phase* of one round ($i = 0$), before entering the *main phase*, that contains all the subsequent rounds ($i = 1, 2, \ldots$).

The initialization phase (round $i = 0$) proceeds as follows:

— Player $\exists$ first plays a closed term t_0 to defend the formula $(\neg X \Rightarrow X) \Rightarrow X$, where the parameter X is unknown to $\exists$. (In what follows, it is convenient to think that this purely logical parameter is only known to the opponent $\forall$.)

— To attack the same formula, opponent $\forall$ plays a pair $(u_0, \pi_0) \in \Lambda \times \Pi$, claiming that u_0 proves $\neg X \Rightarrow X$ whereas π_0 refutes X, thus forming the process $p_0 \equiv t_0 \star u_0 \cdot \pi_0$, that constitutes the next $\exists$-position.

The rest of the game is then parameterized by the initial opponent's move (u_0, π_0), which we call the *handle*, since it is used by player $\exists$ to communicate its moves to opponent $\forall$. Each round $i \geqslant 1$ of the main phase of the game then proceeds as follows:

— The possible moves of player $\exists$ are determined by the thread of the process $p_{i-1} \equiv t_{i-1} \star u_{i-1} \cdot \pi_{i-1}$ formed in the previous round, which is the current $\exists$-position. Depending on the evaluation of this process, the possible moves of player $\exists$ are the following:

Winning move. If $p_{i-1} \succ u_j \star \pi_0$, where u_j is an argument for X played by opponent $\forall$ at some round $j \in [1..i-1]$, then player $\exists$ has a winning move. Intuitively, this move allows player $\exists$ to win the play by exhibiting a contradiction between the opponents, claims that X is true (u_j) and false (π_0). Note that such a move is only possible at a round $i \geqslant 2$, since at round $i = 1$, the opponent has not yet given any argument in defense of the formula X.

Continuation move. If $p_{i-1} \succ u_0 \star t \cdot \pi_0$ for some closed term t, then player $\exists$ has the possibility to play the term $t_i \equiv t$ in defense of the formula $\neg X$. This move is justified by the fact that if the term t is a correct argument for the formula $\neg X$, then the stack $t \cdot \pi_0$ is a correct refutation of the opponent's claim that the formula $\neg X \Rightarrow X$ is true (using u_0).

If none of these two kinds of moves is possible, then opponent $\forall$ wins.

— Once player $\exists$ has played a term t_i in defense of the formula $\neg X$, opponent $\forall$ attacks the formula $\neg X$ with a pair $(u_i, \pi_i) \in \Lambda \times \Pi$, claiming that u_i proves X whereas π_i refutes $\bot$, and forms the next $\exists$-position $p_i \equiv t_i \star u_i \cdot \pi_i$. (Of course, by doing this, opponent $\forall$ unwillingly provides an argument u_i in defense of X that player $\exists$ can use to win in a future turn.)

When a play is infinite, we consider that $\forall$ wins ('benefit of the doubter').

In this game-theoretic setting, it is easy to see that each $\propto_{n,p}$ $(n \geqslant p \geqslant 1)$ constitutes a winning strategy for $\exists$ (as an initial move) in exactly $2n+3$ moves. Indeed, if player $\exists$ plays $t_0 \equiv \propto_{n,p}$ as her first move, the game proceeds as follows:

— $\exists$ plays $t_0 \equiv \propto_{n,p}$.
— $\forall$ plays any move (u_0, π_0) (the 'handle').
— $\exists$ plays $t_1 \equiv K^1_{n,p}[u_0, \mathsf{k}_{\pi_0}]$ (since $\propto_{n,p} \star u_0 \cdot \pi_0 \succ u_0 \star t_1 \cdot \pi_0$).
— $\forall$ plays any move (u_1, π_1)
— $\exists$ plays $t_2 \equiv K^2_{n,p}[u_0, \mathsf{k}_{\pi_0}, u_1]$ (since $t_1 \star u_1 \cdot \pi_1 \succ u_0 \star t_2 \cdot \pi_0$).

$\vdots$

— $\forall$ plays any move (u_{n-1}, π_{n-1})
— $\exists$ plays $t_n \equiv K^n_{n,p}[u_0, \mathsf{k}_{\pi_0}, u_1, \ldots, u_{n-1}]$ (since $t_{n-1} \star u_{n-1} \cdot \pi_{n-1} \succ u_0 \star t_n \cdot \pi_0$).
— $\forall$ plays any move (u_n, π_n)
— $\exists$ wins (since $t_n \star u_n \cdot \pi_n \succ u_p \star \pi_0$, where $1 \leqslant p \leqslant n$).

Let us now formalize the notion of a winning strategy for the game $\mathbb{G}_0$ (from the point of view of player $\exists$). We call a $\mathbb{G}_0$-*state* (or simply: a *state*) any pair $\langle p, \ell \rangle$ where p is a process and where ℓ is a finite set of closed terms that intuitively represents the former moves of $\forall$. Given a fixed handle $(u_0, \pi_0) \in \Lambda \times \Pi$, we define by (generalized) induction the set $W_{(u_0,\pi_0)}$ of *winning states* from the following two inference rules:

$$\frac{}{\langle p, \ell \rangle \in W_{(u_0,\pi_0)}} \qquad (\text{if } p \succ u \star \pi_0 \text{ for some } u \in \ell)$$

$$\frac{\langle t \star u \cdot \pi, \ell \cup \{u\} \rangle \in W_{(u_0,\pi_0)} \quad \text{for all } (u,\pi) \in \Lambda \times \Pi}{\langle p, \ell \rangle \in W_{(u_0,\pi_0)}} \qquad (\text{if } p \succ u_0 \star t \cdot \pi_0).$$

Notice that the second inference rule is an ω-rule that has infinitely many premises, which correspond to all the possible $\forall$-moves $(u, \pi) \in \Lambda \times \Pi$. A derivation of $\langle p, \ell \rangle \in W_{(u_0,\pi_0)}$ is thus, from the generalized inductive definition, an infinitely branching well-founded tree. Finally, we say that a closed term t_0 is a *winning strategy for the game* $\mathbb{G}_0$ if $\langle t_0 \star u_0 \cdot \pi_0, \varnothing \rangle \in W_{(u_0,\pi_0)}$ for all handles $(u_0, \pi_0) \in \Lambda \times \Pi$.

We easily check that:

Fact 27. For all $n \geqslant p \geqslant 1$, $\propto_{n,p}$ is a winning strategy for the game $\mathbb{G}_0$.

Moreover, the terms $\text{\ae}_{n,p}$ are *uniform* strategies, in the sense that all the corresponding plays have the very same structure; they all have the same length ($2n+3$ moves) and they all use the pth opponent's move (u_p, π_p) to end the game (with $u_p \star \pi_0$). In the presence of the `quote` instruction, it is easy to implement winning strategies for the game $\mathbb{G}_0$ that are not uniform – for instance by dynamically extracting n and/or p from the code of the handle (u_0, π_0) or of one of the first opponent's moves (u_i, π_i). But in all cases:

Proposition 28 (adequacy of winning strategies for the game $\mathbb{G}_0$). If $t_0 \in \Lambda$ is a winning strategy for the game $\mathbb{G}_0$, then $t_0 \Vdash \forall X\,((\neg X \Rightarrow X) \Rightarrow X)$.

Proof. Let us assume that $t_0 \in \Lambda$ is a winning strategy for the game $\mathbb{G}_0$, and take a pole $\perp\!\!\!\perp$, a falsity value $S \subseteq \Pi$, a realizer $u_0 \in |\neg\dot{S} \Rightarrow \dot{S}|$ as well as a stack $\pi_0 \in S$. We want to show that $t_0 \star u_0 \cdot \pi_0 \in \perp\!\!\!\perp$. For that, we more generally prove that

For all states $\langle p, \ell\rangle \in W_{(u_0,\pi_0)}$, if $\ell \subseteq S^{\perp\!\!\!\perp}$, then $p \in \perp\!\!\!\perp$.

We proceed by induction on the derivation of $\langle p, \ell\rangle \in W_{(u_0,\pi_0)}$, distinguishing two cases:

1. *$\langle p, \ell\rangle \in W_{(u_0,\pi_0)}$ since $p \succ u \star \pi_0$ for some $u \in \ell$.*
   If we assume that $\ell \subseteq S^{\perp\!\!\!\perp}$, we thus get $u \star \pi_0 \in \perp\!\!\!\perp$ (since $u \in S^{\perp\!\!\!\perp}$ and $\pi_0 \in S$). We then conclude by anti-evaluation.
2. *$\langle p, \ell\rangle \in W_{(u_0,\pi_0)}$ since $p \succ u_0 \star t \cdot \pi_0$ for some term $t \in \Lambda$ such that $\langle t \star u \cdot \pi,\ \ell \cup \{u\}\rangle \in W_{(u_0,\pi_0)}$ for all $(u, \pi) \in \Lambda \times \Pi$.*
   Let us assume that $\ell \subseteq S^{\perp\!\!\!\perp}$. We first want to prove that $t \Vdash \neg\dot{S}$. For that, we take a term $u \in S^{\perp\!\!\!\perp}$ and a stack $\pi \in \Pi$. Since $\ell \cup \{u\} \subseteq S^{\perp\!\!\!\perp}$, we get $t \star u \cdot \pi \in \perp\!\!\!\perp$ from the induction hypothesis. Hence $t \Vdash \neg\dot{S}$ and thus $t \cdot \pi_0 \in \|\neg\dot{S} \Rightarrow \dot{S}\|$, so that $u_0 \star t \cdot \pi_0 \in \perp\!\!\!\perp$. We conclude by anti-evaluation.

By induction, the property holds for all states $\langle p, \ell\rangle \in W_{(u_0,\pi_0)}$. In particular, we have $t_0 \star u_0 \cdot \pi_0 \in \perp\!\!\!\perp$, since $\langle t_0 \star u_0 \cdot \pi_0, \varnothing\rangle \in W_{(u_0,\pi_0)}$ from our assumption on t_0. □

6.3. *Completeness of $\mathbb{G}_0$ in the presence of interaction constants*

In this section, we prove that in the presence of infinitely many interaction constants and of infinitely many substitutive stack constants, the converse of Proposition 28 holds, in the sense that every universal realizer of the law of Peirce is a winning strategy for the game $\mathbb{G}_0$.

This result is a consequence of the following technical lemma:

Lemma 29. Let t_0 be a universal realizer of the law of Peirce. If $(\kappa_i)_{i\in\omega}$ is an infinite sequence of (pairwise distinct) non-generative constants that do not occur in t_0 and if $(\alpha_i)_{i\in\omega}$ is an infinite sequence of stack constants, then there exist two indices n and p such that $n \geqslant p \geqslant 1$ as well as a finite sequence $t_1, \ldots, t_n$ of n closed terms such that

$$\begin{array}{rcll} t_0 \star \kappa_0 \cdot \alpha_0 & \succ & \kappa_0 \star t_1 \cdot \alpha_0 & \\ t_i \star \kappa_i \cdot \alpha_i & \succ & \kappa_0 \star t_{i+1} \cdot \alpha_0 & \qquad \text{(for all } 1 \leqslant i < n) \\ t_n \star \kappa_n \cdot \alpha_n & \succ & \kappa_p \star \alpha_0 & \end{array}$$

Proof. We consider the sequence of sets of processes $(Q_i)_{i\in\omega}$ that is defined by

$$Q_0 \;=\; \mathbf{th}(t_0 \star \kappa_0 \cdot \alpha_0) \qquad \text{and} \qquad Q_{i+1} \;=\; \bigcup_{\substack{t\in\Lambda \text{ s.t.}\\ \kappa_0\star t\cdot\alpha_0\in Q_i}} \mathbf{th}(t \star \kappa_{i+1} \cdot \alpha_{i+1})\,.$$

From this definition, it is clear that a process $q \in Q_i$ contains none of the constants κ_j for $j > i$, using the fact that these constants do not occur in t_0 and that they are non-generative.

Let us now write $Q_\infty = \bigcup_{i\in\omega} Q_i$, and consider the (thread-oriented) pole $\perp\!\!\!\perp$ defined by $\perp\!\!\!\perp = (Q_\infty)^c$ as well as the falsity value $S = \{\alpha_0\}$. From the definition of $\perp\!\!\!\perp$, we have $t_0 \star \kappa_0 \cdot \alpha_0 \notin \perp\!\!\!\perp$. But since $t_0 \Vdash (\neg\dot{S} \Rightarrow \dot{S}) \Rightarrow \dot{S}$ and $\alpha_0 \in S$, we deduce $\kappa_0 \not\Vdash \neg\dot{S} \Rightarrow \dot{S}$. Which means that there is a realizer $t \Vdash \neg\dot{S}$ such that $\kappa_0 \star t \cdot \alpha_0 \notin \perp\!\!\!\perp$. From the latter, we deduce that $\kappa_0 \star t \cdot \alpha_0 \in Q_{p-1}$ for some $p \geqslant 1$. Hence $t \star \kappa_p \cdot \alpha_p \in Q_p$ (from the definition of Q_p) and thus $t \star \kappa_p \cdot \alpha_p \notin \perp\!\!\!\perp$. But since $t \Vdash \neg\dot{S}$, we have $\kappa_p \not\Vdash \dot{S}$, hence $\kappa_p \star \alpha_0 \notin \perp\!\!\!\perp$ and thus $\kappa_p \star \alpha_0 \in Q_n$ for some index $n \geqslant 0$. Using the fact that κ_p can only occur in the processes belonging to the sets Q_i for $i \geqslant p$, we get $n \geqslant p$ (so that $n \geqslant 1$). From the definition of Q_n, we immediately deduce the existence of n terms $t_1, \ldots, t_n$ such that

$$\begin{array}{llll} t_0 \star \kappa_0 \cdot \alpha_0 & \in Q_0 \succ \kappa_0 \star t_1 \cdot \alpha_0 & \in Q_0 & \\ t_i \star \kappa_i \cdot \alpha_i & \in Q_i \succ \kappa_0 \star t_{i+1} \cdot \alpha_0 & \in Q_i & \text{(for all } 1 \leqslant i < n) \\ t_n \star \kappa_n \cdot \alpha_n & \in Q_n \succ \kappa_p \star \alpha_0 & \in Q_n. & \end{array}$$

□

Theorem 30 (specification of Peirce's law in the presence of interaction constants). If the calculus of realizers contains infinitely many interaction constants as well as infinitely many substitutive and non-generative stack constants, then the universal realizers of Peirce's law are exactly the uniform winning strategies for the game $\mathbb{G}_0$.

Proof. We have already proved (Proposition 28) that the terms t_0 that are winning strategies for the game $\mathbb{G}_0$ universally realize Peirce's law. Conversely, let $t_0 \Vvdash \forall X\, ((\neg X \Rightarrow X) \Rightarrow X)$, and take an infinite sequence $(\kappa_i)_{i\in\omega}$ of (pairwise distinct) interaction constants that do not occur in t_0 as well as an infinite sequence $(\alpha_i)_{i\in\omega}$ of (pairwise distinct) non-generative and substitutive stack constants that do not occur in t_0 too. (It is always possible to find such term and stack constants outside t_0, since t_0 only contains a finite number of them.) From Lemma 29, there exist two indices n and p such that $n \geqslant p \geqslant 1$ as well as a finite sequence $t_1, \ldots, t_n$ of n closed terms such that

$$\begin{array}{lll} t_0 \star \kappa_0 \cdot \alpha_0 & \succ \kappa_0 \star t_1 \cdot \alpha_0 & \\ t_i \star \kappa_i \cdot \alpha_i & \succ \kappa_0 \star t_{i+1} \cdot \alpha_0 & \text{(for all } 1 \leqslant i < n) \\ t_n \star \kappa_n \cdot \alpha_n & \succ \kappa_p \star \alpha_0. & \end{array}$$

Also notice that from our assumptions, each term t_i $(0 \leqslant i \leqslant n)$ may contain the constants κ_j/α_j for any $j < i$, but it contains none of them when $j \geqslant i$. To prove the desired result, we consider the threads $\mathbf{th}(t_i \star \kappa_i \cdot \alpha_i)$ $(1 \leqslant i \leqslant n)$ in reverse order.

— $i = n$. For all $u_0, \ldots, u_n \in \Lambda$ and $\pi_0, \ldots, \pi_n \in \Pi$ we have

$$t_n\{\kappa_j := u_j\}_{j=0}^{n-1}\{\alpha_j := \pi_j\}_{j=0}^{n-1} \star u_n \cdot \pi_n \;\succ\; u_p \star \pi_0$$

(by substitutivity), so that

$$\langle t_n\{\kappa_j := u_j\}_{j=0}^{n-1}\{\alpha_j := \pi_j\}_{j=0}^{n-1} \star u_n \cdot \pi_n,\ \{u_1,\dots,u_n\}\rangle \ \in\ W_{(u_0,\pi_0)}$$

from the first rule of the inductive definition of $W_{(u_0,\pi_0)}$.

— $0 \leqslant i < n$. Let us now assume that

$$\langle t_{i+1}\{\kappa_j := u_j\}_{j=0}^{i}\{\alpha_j := \pi_j\}_{j=0}^{i} \star u_{i+1} \cdot \pi_{i+1},\ \{u_1,\dots,u_{i+1}\}\rangle \ \in\ W_{(u_0,\pi_0)}$$

for all $u_0,\dots,u_{i+1} \in \Lambda$ and $\pi_0,\dots,\pi_{i+1} \in \Pi$. From the second rule of the inductive definition of $W_{(u_0,\pi_0)}$, we get

$$\langle t_i\{\kappa_j := u_j\}_{j=0}^{i-1}\{\alpha_j := \pi_j\}_{j=0}^{i-1} \star u_i \cdot \pi_i,\ \{u_1,\dots,u_i\}\rangle \ \in\ W_{(u_0,\pi_0)}$$

for all $u_0,\dots,u_i \in \Lambda$ and $\pi_0,\dots,\pi_i \in \Pi$.

In the case where $i = 0$, we have thus proved that $\langle t_0 \star u_0 \cdot \pi_0,\ \varnothing\rangle \in W_{(u_0,\pi_0)}$ for all $u_0 \in \Lambda$ and $\pi_0 \in \Lambda$, which means that t_0 is a winning strategy for $\mathbb{G}_0$. The fact that this strategy is uniform is obvious from the construction. □

In the presence of infinitely many interaction constants and of infinitely many substitutive and non-generative stack constants, every universal realizer of Peirce's law has thus the same computational behaviour as one of the proof-like terms $\text{œ}_{n,p}$ ($n \geqslant p \geqslant 1$).

6.4. *A wild realizer of Peirce's law*

Theorem 30 gives a specification of Peirce's law in the particular case where the language of realizers provides infinitely many interaction constants and infinitely many substitutive and non-generative stack constants. It is interesting to notice that these assumptions are compatible with the presence of the non-deterministic instruction ⫛; actually, the specification expressed in Theorem 30 makes no assumption about the determinism or the non-determinism of the relation of evaluation $\succ_1$.

However, the assumptions of Theorem 30 are definitely incompatible with the presence of instructions such as `eq` or `quote`, that break the property of substitutivity (for all term/stack constants), and it is tempting to extend the result expressed in Theorem 30 to a framework that allows such instructions – provided we drop the requirement of uniformity, since we know that `quote` allows to implement non-uniform winning strategy for the game $\mathbb{G}_0$. Alas, such an extension is not possible, since the presence of the instruction `eq` (that can be mimicked using `quote`) allows us to define universal realizers of Peirce's law that are *not* winning strategies for the game $\mathbb{G}_0$. In what follows, such realizers will be called *wild realizers of Peirce's law.*

Here is an example of such a wild realizer. Let us consider the terms

$$\begin{aligned} K[y,k] &\equiv \lambda z\,.\,\mathtt{eq}\,z\,(yy)\,\mathbf{I}\,(k\,z)\\ T_1[x] &\equiv \lambda y\,.\,\text{œ}\,(\lambda k\,.\,x\,K[y,k])\\ T_2[x] &\equiv T_1[x]\,T_1[x]\\ \text{œ}' &\equiv \lambda x\,.\,T_2[x]. \end{aligned}$$

From these definitions we get:

$$\text{œ}' \star u \cdot \pi \ \succ_1\ T_2[u] \star \pi\ \succ\ u \star K[T_1[u], \mathsf{k}_\pi] \cdot \pi$$

for all $u \in \Lambda$ and $\pi \in \Pi$, whereas

$$K[T_1[u], \mathsf{k}_\pi] \star u' \cdot \pi' \succ \begin{cases} \mathbf{I} \star \pi' & \text{if } u' \equiv T_2[u] \\ u' \star \pi & \text{otherwise} \end{cases}$$

for all $u, u' \in \Lambda$ and $\pi, \pi' \in \Pi$.

Lemma 31. Let $⫫$ be a fixed pole and $S \subseteq \Pi$ an arbitrary falsity value. For all $u \in \Lambda$ and $\pi \in S$ such that $T_2[u] \star \pi \notin ⫫$, we have $K[T_1[u], \mathsf{k}_\pi] \Vdash \neg\dot{S}$.

Proof. To show that $K[T_1[u], \mathsf{k}_\pi] \Vdash \neg\dot{S}$, let us consider a stack of the form $u' \cdot \pi'$ where $u' \in S^{⫫}$ and $\pi' \in \Pi$. Since $\pi \in S$, we have $u' \star \pi \in ⫫$, and thus $u' \not\equiv T_2[u]$. Hence $K[T_1[u], \mathsf{k}_\pi] \star u' \cdot \pi' \succ u' \star \pi \in ⫫$, so that $K[T_1[u], \mathsf{k}_\pi] \star u' \cdot \pi' \in ⫫$ by anti-evaluation. □

Proposition 32. $\text{cc}' \Vdash\!\Vdash \forall X\,((\neg X \Rightarrow X) \Rightarrow X)$

Proof. Let us consider a fixed pole $⫫$ as well as a falsity value $S \subseteq \Pi$. To show that $\text{cc}' \in |(\neg\dot{S} \Rightarrow \dot{S}) \Rightarrow \dot{S}|$, let us take $u \in |\neg\dot{S} \Rightarrow \dot{S}|$ and $\pi \in S$. We distinguish two cases:

— Either $T_2[u] \star \pi \in ⫫$. In this case we have $\text{cc}' \star u \cdot \pi \succ_1 T_2[u] \star \pi \in ⫫$, from which we get $\text{cc}' \star u \cdot \pi \in ⫫$ by anti-evaluation.

— Either $T_2[u] \star \pi \notin ⫫$. In this case we have $\text{cc}' \star u \cdot \pi \succ u \star K[T_1[u], \mathsf{k}_\pi] \cdot \pi$. Since $K[T_1[u], \mathsf{k}_\pi] \Vdash \neg\dot{S}$ (from Lemma 31), we get $u \star K[T_1[u], \mathsf{k}_\pi] \cdot \pi \in ⫫$, hence $\text{cc}' \star u \cdot \pi \in ⫫$ by anti-evaluation. □

Notice that the subterm $\mathbf{I}$ that appears in the definition of the continuation $K[y, k]$ never appears in head position in the proofs of Lemma 31 and Proposition 32, so that we could actually replace it by any closed λ_c-term. Intuitively, this is due to the fact that when $u' \equiv T_2[u]$, we are not interested anymore in the behaviour of the process $K[T_1[u], \mathsf{k}_\pi] \star u' \cdot \pi'$ since we are able to conclude that $\text{cc}' \star u \cdot \pi \in ⫫$ using other means (first case of the proof of Proposition 32). Also notice that the case distinction performed in the proof of Proposition 32 makes the corresponding proof intrinsically classical, which contrasts with the proof of adequacy for cc (Lemma 11 p. 1284), which is fully intuitionistic. In some sense, we can think of cc' as a universal realizer of Peirce's law that is twice classical. It is classical from the computational point of view, since it heavily relies on the machinery of continuations. But it is also classical from the meta-theoretic point of view, due to the fact that the corresponding proof of adequacy (Proposition 32) is classical too.

Before giving a game-theoretic interpretation of these strange phenomena, let us first check that cc' is not a winning strategy for our first game $\mathbb{G}_0$:

Lemma 33. Let us assume that the relation of one step evaluation $\succ_1$ is only defined from the rules (Grab), (Push), (Save), (Restore) and (Eq). Then the universal realizer cc' of Peirce's law is not a winning strategy for the game $\mathbb{G}_0$.

Proof. We start with the initial handle $(\mathbf{I}, \alpha_0)$, where α_0 is a stack constant. We notice that

— Player $\exists$ is forced to play $t_1 \equiv K[T_1[\mathbf{I}], \mathsf{k}_{\alpha_0}]$, since t_1 is the only term t such that $\mathbf{I} \star t \cdot \alpha_0 \in \mathbf{th}(\text{œ}' \star \mathbf{I} \cdot \alpha_0)$.
— Opponent $\forall$ can play $u_1 \equiv T_2[\mathbf{I}]$ and $\pi_1 \equiv \alpha_0$.
— Then $\exists$ loses, since the thread $\mathbf{th}(t_1 \star u_1 \cdot \pi_1)$ contains no process of the form $\mathbf{I} \star t_2 \cdot \alpha_0$ (to continue to play) or of the form $T_2[\mathbf{I}] \star \alpha_0$ (to win the game). $\square$

6.5. *A second game* $\mathbb{G}_1$

Although the wild realizer $\text{œ}'$ of Peirce's law does not constitute a winning strategy for the game $\mathbb{G}_0$, we can still understand the computational behaviour of $\text{œ}'$ in game-theoretic terms as follows:

— Player $\exists$ plays $t_0 \equiv \text{œ}'$.
— Opponent $\forall$ plays a handle (u_0, π_0).
— Player $\exists$ plays $t_1 \equiv K[T_1[u_0], \mathsf{k}_{\pi_0}]$ (since $\text{œ}' \star u_0 \cdot \pi_0 \succ u_0 \star K[T_1[u_0], \mathsf{k}_{\pi_0}] \cdot \pi_0$).
— Then opponent $\forall$ plays an arbitrary move $(u_1, \pi_1) \in \Lambda \times \Pi$.
— Now comes the crucial point:

1. In the case where $u_1 \not\equiv T_2[u_0]$, player $\exists$ wins as expected, since:

$$t_1 \star u_1 \cdot \pi_1 \;\equiv\; K[T_1[u_0], \mathsf{k}_{\pi_0}] \star u_1 \cdot \pi_1 \;\succ\; u_1 \star \pi_0\,.$$

2. But in the case where $u_1 \equiv T_2[u_0]$, player $\exists$ realizes that she could have played a winning move at the previous round, since:

$$t_0 \star u_0 \cdot \pi_0 \;\equiv\; \text{œ}' \star u_0 \cdot \pi_0 \;\succ_1\; T_2[u_0] \star \pi_0 \;\equiv\; u_1 \star \pi_0\,.$$

(Of course, she could not claim her victory at that time, because she did not know that u_1 is an opponent's argument for X.) To achieve her 'retrospective victory', player $\exists$ simply backtracks to her former position $\text{œ}' \star u_0 \cdot \pi_0$, from which she can win using the above indicated move. (Note that this new form of backtrack is purely game theoretic, with no computational counterpart.) At this point of the play, the $\exists$-position $K[T_1[u_0], \mathsf{k}_{\pi_0}] \star u_1 \cdot \pi_1$ is abandoned, which explains why its computational behaviour is irrelevant.

This discussion shows that we can still think of the closed term $\text{œ}'$ as a winning strategy provided we give to player $\exists$ the possibility to compute its move from any $\exists$-position that was previously encountered during the play – and not only from the current $\exists$-position.

We thus get a new game $\mathbb{G}_1$, that relies on the same logical intuitions as before (*cf* Section 6.2). The only difference is that now, player $\exists$ keeps track of the history of all the preceding $\exists$-positions encountered during the game, and is allowed to compute its next move from any position recorded in this history.

Formally, we thus define a $\mathbb{G}_1$-*state* as a pair $\langle P, \ell \rangle$ where P is a finite set of processes (intuitively, the history of all the previously encountered $\exists$-positions, including the current position) and where ℓ is a finite set of closed terms (intuitively, the history of the first components u_i of the previous moves (u_i, π_i) of the opponent $\forall$). Given a handle

$(u_0, \pi_0) \in \Lambda \times \Pi$, the set $W'_{(u_0,\pi_0)}$ of *winning* $\mathbb{G}_1$-*states* is inductively defined as follows:

$$\frac{}{\langle P, \ell\rangle \in W'_{(u_0,\pi_0)}} \qquad (\text{if } p \succ u \star \pi_0 \text{ for some } p \in P,\ u \in \ell).$$

$$\frac{\langle P \cup \{t \star u \cdot \pi\}, \ell \cup \{u\}\rangle \in W'_{(u_0,\pi_0)} \quad \text{for all } (u, \pi)}{\langle P, \ell\rangle \in W'_{(u_0,\pi_0)}} \qquad (\text{if } p \succ u_0 \star t \cdot \pi_0 \text{ for some } p \in P).$$

As before, we say that a term t_0 is *a winning strategy for the game* $\mathbb{G}_1$ if for all handles $(u_0, \pi_0) \in \Lambda \times \Pi$, we have $\langle\{t_0 \star u_0 \cdot \pi_0\}, \varnothing\rangle \in W'_{(u_0,\pi_0)}$. It is a simple exercise to check that:

Fact 34. The closed term cc' is a winning strategy for the game $\mathbb{G}_1$.

Moreover, we easily check that winning strategies for the game $\mathbb{G}_0$ are particular cases of winning strategies for the game $\mathbb{G}_1$:

Proposition 35. If a closed λ_c-term is a winning strategy for the game $\mathbb{G}_0$, then it is also a winning strategy for the game $\mathbb{G}_1$.

Proof. It suffices to prove that $\langle p, \ell\rangle \in W_{(u_0,\pi_0)}$ implies $\langle\{p\}, \ell\rangle \in W'_{(u_0,\pi_0)}$ for all $\mathbb{G}_0$-states $\langle p, \ell\rangle$, by induction on the derivation of $\langle p, \ell\rangle \in W_{(u_0,\pi_0)}$. The second case of the proof relies on the property of monotonicity expressing that $\langle P, \ell\rangle \in W'_{(u_0,\pi_0)}$ and $P \subseteq P'$ together imply $\langle P', \ell\rangle \in W'_{(u_0,\pi_0)}$ (which is also proved by induction). $\square$

Let us now prove that the game $\mathbb{G}_1$ is adequate w.r.t. Peirce's law:

Proposition 36 (adequacy of winning strategies for the game $\mathbb{G}_1$). If t_0 is a winning strategy for the game $\mathbb{G}_1$, then $t_0 \Vdash \forall X\,((\neg X \Rightarrow X) \Rightarrow X)$.

Proof. Let $\perp\!\!\!\perp$ be a fixed pole, and consider a falsity value $S \subseteq \Pi$, a realizer $u_0 \Vdash \neg\dot{S} \Rightarrow \dot{S}$ as well as a stack $\pi_0 \in S$. We want to show that $t_0 \star u_0 \cdot \pi_0 \in \perp\!\!\!\perp$. For that, we more generally prove that for all $\langle P, \ell\rangle \in W'_{(u_0,\pi_0)}$, $\ell \subseteq S^{\perp\!\!\!\perp}$ implies $P \cap \perp\!\!\!\perp \neq \varnothing$. The proof proceeds by induction on the derivation on $\langle P, \ell\rangle \in W'_{(u_0,\pi_0)}$, distinguishing the following cases:

1. $\langle P, \ell\rangle \in W'_{(u_0,\pi_0)}$ since there exist some $p \in P$ and $u \in \ell$ such that $p \succ u \star \pi_0$. If we assume that $\ell \subseteq S^{\perp\!\!\!\perp}$, we thus get $u \star \pi_0 \in \perp\!\!\!\perp$, hence $p \in \perp\!\!\!\perp$ by anti-evaluation.
2. $\langle P, \ell\rangle \in W'_{(u_0,\pi_0)}$ since there exist some $p \in P$ and $t \in \Lambda$ such that $p \succ u_0 \star t \cdot \pi_0$, and since $\langle P \cup \{t \star u \cdot \pi\}, \ell \cup \{u\}\rangle \in W'_{(u_0,\pi_0)}$ for all $u \in \Lambda$ and $\pi \in \Pi$. Let us assume that $\ell \subseteq S^{\perp\!\!\!\perp}$. To show that $P \cap \perp\!\!\!\perp \neq \varnothing$, let us assume that $P \cap \perp\!\!\!\perp = \varnothing$ (using the meta-theoretic law of Peirce). We first want to show that $t \Vdash \neg\dot{S}$. For that, let us consider $u \in S^{\perp\!\!\!\perp}$ and $\pi \in \Pi$. Since $\langle P \cup \{t \star u \cdot \pi\}, \ell \cup \{u\}\rangle \in W'_{(u_0,\pi_0)}$ and $(\ell \cup \{u\}) \subseteq S^{\perp\!\!\!\perp}$, we get $(P \cup \{t \star u \cdot \pi\}) \cap \perp\!\!\!\perp \neq \varnothing$ by induction hypothesis. But since $P \cap \perp\!\!\!\perp = \varnothing$, we deduce that $t \star u \cdot \pi \in \perp\!\!\!\perp$, which finishes the proof that $t \Vdash \neg\dot{S}$. Therefore, $u_0 \star t \cdot \pi_0 \in \perp\!\!\!\perp$, and thus $p \in \perp\!\!\!\perp$ by anti-evaluation.

In particular, we have proved that $t_0 \star u_0 \cdot \pi_0 \in \perp\!\!\!\perp$, since $\langle\{t_0 \star u_0 \cdot \pi_0\}, \varnothing\rangle \in W'_{(u_0,\pi_0)}$. $\square$

But the converse implication also holds, without any further assumption on the parameters $\mathcal{B}$, $\mathcal{C}$ and $\succ_1$ that define the underlying calculus of realizers:

Proposition 37 (completeness of winning strategies for the game $\mathbb{G}_1$). If t_0 universally realizes Peirce's law, then t_0 is a winning strategy for the game $\mathbb{G}_1$.

Proof. We reason by contradiction by assuming that there is a handle $(u_0, \pi_0) \in \Lambda \times \Pi$ such that $\langle\{t_0 \star u_0 \cdot \pi_0\}, \varnothing\rangle \notin W'_{(u_0,\pi_0)}$. To reach a contradiction, we shall build an increasing sequence of $\mathbb{G}_1$-states $(\langle P_i, \ell_i\rangle)_{i\in\mathbb{N}}$ such that $\langle P_i, \ell_i\rangle \notin W'_{(u_0,\pi_0)}$ for all $i \in \mathbb{N}$. For that, we consider a fixed enumeration $\phi : \mathbb{N} \to \Lambda$ such that every term $t \in \Lambda$ appears infinitely many times in the range of ϕ. The sequence $(\langle P_i, \ell_i\rangle)_{i\in\mathbb{N}}$ is defined as follows:

— $P_0 = \{t_0 \star u_0 \cdot \pi_0\}$ and $\ell_0 = \varnothing$, so that $\langle P_0, \ell_0\rangle \notin W'_{(u_0,\pi_0)}$.

— Let us assume that we have built a $\mathbb{G}_1$-state $\langle P_i, \ell_i\rangle$ such that $\langle P_i, \ell_i\rangle \notin W'_{(u_0,\pi_0)}$. Writing $t \equiv \phi(i)$, we distinguish the following two cases:

1. Either there exists a process $p \in P_i$ such that $p \succ u_0 \star t \cdot \pi_0$. In this case, we know (from the second rule of the inductive definition of $W'_{(u_0,\pi_0)}$) that there is a pair $(u, \pi) \in \Lambda \times \Pi$ such that $\langle P_i \cup \{t \star u \cdot \pi\}, \ell_i \cup \{u\}\rangle \notin W'_{(u_0,\pi_0)}$. We pick such a pair (u, π) and let $P_{i+1} = P_i \cup \{t \star u \cdot \pi\}$ and $\ell_{i+1} = \ell_i \cup \{u\}$, so that by construction we have $\langle P_{i+1}, \ell_{i+1}\rangle \notin W'_{(u_0,\pi_0)}$.
2. Either there is no process $p \in P_i$ such that $p \succ u_0 \star t \cdot \pi_0$. In this case, we keep the same $\mathbb{G}_1$-state by letting $P_{i+1} = P_i$ and $\ell_{i+1} = \ell_i$.

It is clear from the above construction that $P_i \subseteq P_{i+1}$ and $\ell_i \subseteq \ell_{i+1}$ for all $i \in \mathbb{N}$. We then put $P_\infty = \bigcup_{i\in\mathbb{N}} P_i$, $Q = \bigcup_{p\in P_\infty} \mathbf{th}(p)$, and we consider the pole $\perp\!\!\!\perp = Q^c$ as well as the falsity value $S = \{\pi_0\}$. Since $t_0 \Vdash (\neg\dot{S} \Rightarrow \dot{S}) \Rightarrow \dot{S}$, $\pi_0 \in S$ and $t_0 \star u_0 \cdot \pi_0 \notin \perp\!\!\!\perp$ (from the definition of $\perp\!\!\!\perp$), we get $u_0 \not\Vdash \neg\dot{S} \Rightarrow \dot{S}$. Thus, there is a realizer $t \Vdash \neg\dot{S}$ such that $u_0 \star t \cdot \pi_0 \notin \perp\!\!\!\perp$. Hence $u_0 \star t \cdot \pi_0 \in Q$, so that there is an index $n \geqslant 0$ and a process $p \in P_n$ such that $p \succ u_0 \star t \cdot \pi_0$. Let us consider an index $n' \geqslant n$ such that $\phi(n') \equiv t$. Since $p \in P_{n'}$ $(\supseteq P_n)$ and $p \succ u_0 \star t \cdot \pi_0$, there exists $(u, \pi) \in \Lambda \times \Pi$ such that $P_{n'+1} = P_{n'} \cup \{t \star u \cdot \pi\}$ and $\ell_{n'+1} = \ell_{n'} \cup \{u\}$ (by construction of $\langle P_{n'+1}, \ell_{n'+1}\rangle$). Therefore $t \star u \cdot \pi \notin \perp\!\!\!\perp$, hence $u \not\Vdash \dot{S}$ (since $t \Vdash \neg\dot{S}$ and $\pi \in \|\bot\|$), so that $u \star \pi_0 \notin \perp\!\!\!\perp$. Hence we get $u \star \pi_0 \in Q$, so that there is an index $m \geqslant 0$ and a process $p' \in P_m$ such that $p' \succ u \star \pi_0$. Without loss of generality, we can assume that $m \geqslant n' + 1$ since the sequence $(P_i)_{i\in\mathbb{N}}$ is increasing. We have thus found an index $m \geqslant n' + 1$, a process $p' \in P_m$ and a term $u \in \ell_m$ (since $u \in \ell_{n'+1} \subseteq \ell_m$) such that $p' \succ u \star \pi_0$, which means that $\langle P_m, \ell_m\rangle \in W'_{(u_0,\pi_0)}$ (from the first rule of the inductive definition of $W'_{(u_0,\pi_0)}$) and brings us the desired contradiction. □

From Propositions 36 and 37, we thus get the definitive specification of Peirce's law:

Theorem 38 (specification of Peirce's law). The universal realizers of Peirce's law are exactly the winning strategies for the game $\mathbb{G}_1$.

References

Barendregt, H. (1984) *The Lambda Calculus: Its Syntax and Semantics*, Studies in Logic and The Foundations of Mathematics volume 103, North-Holland.

Barbanera, F. and Berardi, S. (1996) A symmetric lambda calculus for classical program extraction. *Information and Computation* **125** (2) 103–117.

Church, A. (1941) *The Calculi of Lambda-Conversion*, Annals of Mathematical Studies volume 6, Princeton.

Curien, P.-L. and Herbelin, H. (2000) The duality of computation. In: *International Conference on Functional Programming* 233–243.

Curry, H. B. and Feys, R. (1958) *Combinatory Logic*, volume 1, North-Holland.

Friedman, H. (1973) Some applications of Kleene's methods for intuitionistic systems. In: Cambridge Summer School in Mathematical Logic. *Springer-Verlag Lecture Notes in Mathematics* **337** 113–170.

Friedman, H. (1978) Classically and intuitionistically provably recursive functions. *Higher Set Theory* **669** 21–28.

Girard, J.-Y. (2006) *Le point aveugle – Cours de logique – Volume I – vers la perfection*, Hermann.

Girard, J.-Y., Lafont, Y. and Taylor, P. (1989) *Proofs and Types*, Cambridge University Press.

Goldblatt, R. (1985) On the role of the Baire category theorem and dependent choice in the foundations of logic. *Journal of Symbolic Logic* **50** 412–422.

Griffin, T. (1990) A formulae-as-types notion of control. In: *Principles of Programming Languages* 47–58.

Guillermo, M. (2008) *Jeux de réalisabilité en arithmétique classique*, Ph.D. thesis, Université Paris 7.

Howard., W. A. (1969) The formulae-as-types notion of construction. Privately circulated notes.

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

Krivine., J. -L. (1993) *Lambda-Calculus, Types and Models*, Masson.

Krivine, J.-L. (2001) Typed lambda-calculus in classical Zermelo-Fraenkel set theory. *Archive for Mathematical Logic* **40** (3) 189–205.

Krivine, J.-L. (2003) Dependent choice, 'quote' and the clock. *Theoretical Computer Science* **308** 259–276.

Krivine, J.-L. (2009) Realizability in classical logic. In: interactive models of computation and program behaviour. *Panoramas et synthèses*, **27** 197–229.

Martin-Löf, P. (1998) An intuitionistic theory of types. In twenty-five years of constructive type theory. *Oxford Logic Guides* **36** 127–172.

McCarty, D. (1984) *Realizability and Recursive Mathematics*, Ph.D. thesis, Carnegie-Mellon University.

Miquel, A. (2007) Classical program extraction in the calculus of constructions. In: Computer Science Logic, 21st International Workshop, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, *Springer Lecture Notes in Computer Science* **4646** 313–327.

Miquel, A. (2010) Existential witness extraction in classical realizability and via a negative translation. *Logical Methods for Computer Science* **7** (2) 1–47.

Myhill, J. (1973) Some properties of intuitionistic Zermelo–Fraenkel set theory. *Lecture Notes in Mathematics* **337** 206–231.

Oliva, P. and Streicher, T. (2008) On Krivine's realizability interpretation of classical second-order arithmetic. *Fundamenta Informaticae* **84** (2) 207–220.

Parigot, M. (1997) Proofs of strong normalisation for second order classical natural deduction. *The Journal of Symbolic Logic* **62** (4) 1461–1479.

Sperber, M., Kent Dybvig, R., Flatt, M., Van Straaten, A., Findler, R. and Matthews, J. (2009) Revised^6 report on the algorithmic language Scheme. *Journal of Functional Programming* **19** Supplement S1 1–301.