

SPECIFYING PEIRCE'S LAW IN CLASSICAL REALIZABILITY

MAURICIO GUILLERMO AND ALEXANDRE MIQUEL

1. INTRODUCTION

The correspondence between proofs and programs—also known as the *Curry-Howard correspondence* [5, 12, 9]—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 [7] or to linear logic [8].

In 1990, Griffin discovered [10] that in the programming language Scheme [25], the control operator `call/cc` (for ‘call with current continuation’) could be given the type $((A \rightarrow B) \rightarrow A) \rightarrow A$ corresponding to *Peirce’s law* through the formulas-as-types interpretation [12]. 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 [24], Barbanera and Berardi’s symmetric λ -calculus [1], Krivine’s λ_c -calculus [17] or Curien and Herbelin’s $\tilde{\lambda}\mu$ -calculus [4].

However, the analysis of the computational behavior 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 [13] to interpret the computational contents of the proofs of Heyting arithmetic, and later extended to more expressive frameworks, including intuitionistic set theories [22, 6, 19]. 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 logic¹.

1.1. The theory of classical realizability. To address this problem, Krivine introduced in the middle of the nineties the theory of *classical realizability* [17], which is a complete reformulation of the very principles of realizability to make them compatible with classical reasoning. (As noticed in [23, 21], classical realizability can be seen as a reformulation of Kleene’s realizability through Friedman’s *A*-translation [7].) 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 [15] or the calculus of constructions with universes [20].

¹For 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 [13]. 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.

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 behavior 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 behavior 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, 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 [17], an extension of Church’s λ -calculus [3] with an instruction \mathfrak{c} (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 [21]) 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 [16] to realize the axiom of dependent choices². 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 \mathfrak{h} (‘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 behavior. 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 $\exists^{\mathbb{N}} x A(x)$ are exactly the programs reducing to a pair whose first component is a witness $n \in \mathbb{N}$ and whose second component is a realizer of the formula $A(n)$ [14]. 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

²In analysis, this axiom is crucial to prove Baire’s lemma—which is actually equivalent.

any time. Of course, these features are essential to interpret classical reasoning principles such as the excluded middle, but on the other hand, we cannot hope anymore that the first projection of a classical realizer of $\exists^N x A(x)$ will give us the desired witness for free. (For an account of witness extraction techniques in classical realizability, see [21].)

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 behavior.

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 [11], 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 [11]—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 [16].

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 captured by 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 counterexample 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 meta theory, 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 [3, 2] 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 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}_λ of λ -variables (notation: x, y, z , etc.)
- A countable set \mathcal{C} of instructions, that contains at least an instruction $\mathfrak{c} \in \mathcal{C}$ ('call/cc', for: *call with current continuation*).
- A nonempty 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 six 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 $FV(c) = \emptyset$.
- (3) If t and u are terms, then tu is a term, and $FV(tu) = FV(t) \cup FV(u)$.
- (4) If $x \in \mathcal{V}_\lambda$ is a λ -variable and if t is a term, then $\lambda x . t$ is a term, and $FV(\lambda x . t) = FV(t) \setminus \{x\}$.
- (5) If $\alpha \in \mathcal{B}$ is a stack constant, then α is a stack.
- (6) If t is a closed term (i.e. $FV(t) = \emptyset$) 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 (6) to closed terms t . Thanks to this restriction, stacks are always closed objects and continuation constants κ_π are actually 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 [2], 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 κ_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 1 p. 14).

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

$$\bar{n} \equiv \bar{s}^n \bar{0} \equiv \underbrace{\bar{s}(\cdots(\bar{s}\bar{0})\cdots)}_n,$$

where $\bar{0} \equiv \lambda x f . x$ and $\bar{s} \equiv \lambda n x f . f(nxf)$ 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 $\bar{n} \equiv \bar{s}^n \bar{0}$ is clearly β -convertible to Church's encoding $\lambda x f . f^n x$ —and thus computationally equivalent. The reason for preferring this modified encoding is that it is better suited to the call-by-name discipline of Krivine's Abstract Machine (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 Krivine's Abstract Machine (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 of Λ by Π).

Definition 3 (Relation of evaluation). — We call a relation of one step evaluation any binary relation $>_1$ over the set $\Lambda \star \Pi$ of processes that fulfils the following four axioms:

$$\begin{array}{lll}
\text{(PUSH)} & tu \star \pi & >_1 & t \star u \cdot \pi \\
\text{(GRAB)} & (\lambda x. t) \star u \cdot \pi & >_1 & t\{x := u\} \star \pi \\
\text{(SAVE)} & \mathbb{C} \star t \cdot \pi & >_1 & t \star k_\pi \cdot \pi \\
\text{(RESTORE)} & k_\pi \star t \cdot \pi' & >_1 & t \star \pi
\end{array}$$

The reflexive-transitive closure of $>_1$ is written $>$.

One of the specificities of the λ_c -calculus is that it comes with a binary relation of (one step) evaluation $>_1$ that is not *defined*, but *axiomatized* via the rules (PUSH), (GRAB), (SAVE) and (RESTORE). In practice, the binary relation $>_1$ 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 $>_1$. (The set \mathcal{V}_λ 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 $>_1$ (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 $>_1$. On the other hand, the definitions of classical realizability [17] as well as its main properties do not depend on the particular choice of \mathcal{B} , \mathcal{C} and $>_1$, 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

$$\text{(QUOTE)} \quad \text{quote} \star t \cdot \pi >_1 t \star \bar{n}_\pi \cdot \pi,$$

where $\pi \mapsto n_\pi$ is a recursive 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 $n \mapsto \bar{n}$ described in Section 2.1) to the term t . This instruction was introduced in [16] to realize the axiom of dependent choices.

- The instruction `eq`, that comes with the evaluation rule

$$\text{(EQ)} \quad \text{eq} \star t_1 \cdot t_2 \cdot u \cdot v \cdot \pi >_1 \begin{cases} u \star \pi & \text{if } t_1 \equiv t_2 \\ v \star \pi & \text{if } t_1 \not\equiv t_2 \end{cases}$$

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 behavior as `eq`, by letting

$$\text{eq}' \equiv \lambda x_1 x_2. \text{quote} (\lambda n_1 y_1. \text{quote} (\lambda n_2 y_2. \text{eq_nat } n_1 n_2) x_2) x_1,$$

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

- The instruction `fork` (‘fork’), that comes with the two evaluation rules

$$\text{(FORK)} \quad \text{fork} \star t_0 \cdot t_1 \cdot \pi >_1 t_0 \star \pi \quad \text{and} \quad \text{fork} \star t_0 \cdot t_1 \cdot \pi >_1 t_1 \star \pi.$$

Intuitively, the instruction `fork` 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 $>_1$ is deterministic when the two conditions $p >_1 p'$ and $p >_1 p''$ implies $p' \equiv p''$ (syntactic identity) for all processes p, p' and p'' . Otherwise, $>_1$ 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 $\text{eq} (\neq \mathbb{C})$ together with the aforementioned evaluation rules, or with an instruction $\text{quote} (\neq \mathbb{C})$ such as presented in [16, 17].

On the other hand, the presence of an instruction th 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 $\text{th}(p)$ the set of all process p' such that $p > p'$:

$$\text{th}(p) = \{p' \in \Lambda \star \Pi : p > p'\}.$$

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

$$\mathbf{I} \star \delta \delta \cdot \alpha >_1 \delta \delta \star \alpha >_1 \delta \star \delta \cdot \alpha >_1 \delta \delta \star \alpha >_1 \dots$$

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

$$\mathbf{II} \star \alpha >_1 \mathbf{I} \star \mathbf{I} \cdot \alpha >_1 \mathbf{I} \star \alpha.$$

- *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 $\delta' \equiv \lambda x . x x \mathbf{I}$:

$$\delta' \delta' \star \alpha >_3 \delta' \delta' \star \mathbf{I} \cdot \alpha >_3 \delta' \delta' \star \mathbf{I} \cdot \mathbf{I} \cdot \alpha >_3 \delta' \delta' \star \mathbf{I} \cdot \mathbf{I} \cdot \mathbf{I} \cdot \alpha >_3 \dots$$

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:

$$\begin{aligned} x\{c := u\} &\equiv x & c\{c := u\} &\equiv u \\ (\lambda x . t)\{c := u\} &\equiv \lambda x . t\{c := u\} & c'\{c := u\} &\equiv c' \quad (\text{if } c' \neq c) \\ (t_1 t_2)\{c := u\} &\equiv t_1\{c := u\} t_2\{c := u\} & \alpha\{c := u\} &\equiv \alpha \\ k_\pi\{c := u\} &\equiv k_{\pi\{c := u\}} & (t \cdot \pi)\{c := u\} &\equiv t\{c := u\} \cdot \pi\{c := u\} \end{aligned}$$

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:

$$\begin{aligned} x\{\alpha := \pi_0\} &\equiv x & c\{\alpha := \pi_0\} &\equiv c \\ (\lambda x . t)\{\alpha := \pi_0\} &\equiv \lambda x . t\{\alpha := \pi_0\} & \alpha\{\alpha := \pi_0\} &\equiv \pi_0 \\ (t_1 t_2)\{\alpha := \pi_0\} &\equiv t_1\{\alpha := \pi_0\} t_2\{\alpha := \pi_0\} & \alpha'\{\alpha := \pi_0\} &\equiv \alpha' \quad (\text{if } \alpha' \neq \alpha) \\ k_\pi\{\alpha := \pi_0\} &\equiv k_{\pi\{\alpha := \pi_0\}} & (t \cdot \pi)\{\alpha := \pi_0\} &\equiv t\{\alpha := \pi_0\} \cdot \pi\{\alpha := \pi_0\} \end{aligned}$$

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 to 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*³, 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 $k \geq 0$ and if e_1, \dots, e_k are first-order expressions, then $f(e_1, \dots, e_k)$ 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 $e\{x := e'\}$.

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 $k \geq 0$ (that we shall often leave implicit, since it can be easily inferred from the context), and that for each arity $k \geq 0$, 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 $k \geq 0$ and if e_1, \dots, e_k are first-order expressions, then $X(e_1, \dots, e_k)$ 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 closed 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, \dots, 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, \dots, x_k are abstracted in the formula C , but this notation should not be confused with the abstraction of the λ_c -calculus.

³Here 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.

The set of free variables of a k -ary predicate $P \equiv \lambda x_1 \cdots x_k . C$ is defined by $FV(P) \equiv FV(C) \setminus \{x_1; \dots; x_k\}$, 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, \dots, e_k is defined by letting

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

(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 $\lambda x_1 \cdots x_k . X(x_1, \dots, x_k)$, whose only free variable is X . The reader can easily check that the meaning of the notation $X(e_1, \dots, e_k)$ 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 $X \equiv \lambda x_1 \cdots x_k . X(x_1, \dots, x_k)$ to the k -tuple of first-order expressions e_1, \dots, 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:

$$\begin{aligned} X(e_1, \dots, e_k)\{X := P\} &\equiv P(e_1, \dots, e_k) \\ Y(e_1, \dots, e_m)\{X := P\} &\equiv Y(e_1, \dots, e_m) && (Y \neq X) \\ (A \Rightarrow B)\{X := P\} &\equiv A\{X := P\} \Rightarrow B\{X := P\} \\ (\forall x A)\{X := P\} &\equiv \forall x A\{X := P\} && (x \notin FV(P)) \\ (\forall X A)\{X := P\} &\equiv \forall X A \\ (\forall Y A)\{X := P\} &\equiv \forall Y A\{X := P\} && (Y \neq X, Y \notin FV(P)) \end{aligned}$$

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:

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

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

3.2. A type system for classical second-order logic. Through the formulas-as-types correspondence [12, 9], 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.

The type system of classical second-order logic is then defined from the typing rules of Fig. 1. These typing rules are the usual typing rules of AF2 [14], plus a specific typing rule for the instruction \mathfrak{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 Fig. 1 the usual introduction and elimination rules of absurdity, conjunction, disjunction, (first- and second-order) existential quantification and Leibniz equality [14]. 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.

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 $\llbracket f \rrbracket : \mathbb{N}^k \rightarrow \mathbb{N}$, so that we can give a denotation $\llbracket e \rrbracket \in \mathbb{N}$ to every closed first-order expression e .

$$\begin{array}{c}
\hline\hline
\frac{}{\Gamma \vdash x : A} \quad (x:A) \in \Gamma \\
\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x. t : A \Rightarrow B} \quad \frac{\Gamma \vdash t : A \Rightarrow B \quad \Gamma \vdash t : A}{\Gamma \vdash tu : B} \\
\frac{\Gamma \vdash t : A}{\Gamma \vdash t : \forall x A} \quad x \notin FV(\Gamma) \quad \frac{\Gamma \vdash t : \forall x A}{\Gamma \vdash t : A\{x := e\}} \\
\frac{\Gamma \vdash t : A}{\Gamma \vdash t : \forall X A} \quad X \notin FV(\Gamma) \quad \frac{\Gamma \vdash t : \forall X A}{\Gamma \vdash t : A\{X := P\}} \\
\hline\hline
\Gamma \vdash \mathfrak{c} : ((A \Rightarrow B) \Rightarrow A) \Rightarrow A
\end{array}$$

FIGURE 1. Typing rules of second-order logic

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 $\llbracket s^n(0) \rrbracket = n$.

3.3.1. *Induction.* Following Dedekind's construction of natural numbers, we consider the predicate $\text{Nat}(x)$ [9, 14] defined by

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

that defines the smallest class of individuals containing zero and closed under the successor function. We easily check that the judgments

$$\vdash \bar{0} \equiv \lambda x f. x : \text{Nat}(0) \quad \text{and} \quad \vdash \bar{s} \equiv \lambda n x f. f(n x f) : \forall x (\text{Nat}(x) \Rightarrow \text{Nat}(s(x)))$$

(cf Section 2.1) are derivable in second-order logic, so that $\bar{n} : \text{Nat}(n)$ for all $n \in \mathbb{N}$.

One of the main properties of the logical system presented above is that the axiom of induction, that we can write $\forall x \text{Nat}(x)$, is not derivable from the rules of Fig. 1. As proved in [17, 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 $\text{Nat}(x)$ of Dedekind numerals using the shorthands for *numeric quantifications*⁴

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

so that the *relativized induction axiom* becomes provable in second-order logic:

$$\forall Z (Z(0) \Rightarrow \forall^{\text{N}}_x (Z(x) \Rightarrow Z(s(x))) \Rightarrow \forall^{\text{N}}_x Z(x)).$$

(Proof-term: $\lambda z_0 z_1 n. n (\lambda u. u \bar{0} z_0) (\lambda z. z (\lambda x y u. u (\bar{s} x) (z_1 x y))) (\lambda x y. y)$.)

3.3.2. *The axioms of PA2.* Formally, a formula A is a *theorem* of second-order arithmetic (PA2) if it can be derived (using the rules of Fig. 1) from 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, and from the definitional equalities attached to the (primitive recursive) function symbols of the signature:

- $\forall x (x + 0 = x)$
- $\forall x \forall y (x + s(y) = s(x + y))$
- $\forall x (x \times 0 = 0)$

⁴From a computational point of view, the numeric quantifications $\forall^{\text{N}}_x A(x)$ and $\exists^{\text{N}}_x A(x)$ play the same role as the dependent product $\Pi x : \text{Nat}. A(x)$ and the dependent sum $\Sigma x : \text{Nat}. A(x)$ in type theory [18], putting aside the subtleties coming from the fact that we work here in a system that is both classical and impredicative.

- $\forall x \forall y (x \times s(y) = (x \times y) + x)$
- 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 $>_1$ 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*⁵ $\perp\!\!\!\perp$ 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 $p > p'$ and $p' \in \perp\!\!\!\perp$ together imply that $p \in \perp\!\!\!\perp$ for all processes $p, p' \in \Lambda \star \Pi$.

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:

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

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:

$$\perp\!\!\!\perp \equiv \left(\bigcup_{p \in P} \text{th}(p) \right)^c \equiv \bigcap_{p \in P} (\text{th}(p))^c.$$

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* $S^\perp \subseteq \Lambda$ that is defined by

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

Intuitively, every falsity value $S \subseteq \Pi$ represents a particular set of *tests*, while the corresponding truth value S^\perp represent 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 , it is clear that the larger the falsity value S , the smaller the corresponding truth value S^\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 $|A| = \|\|A\|\|^\perp$.

⁵In [17, 16, 11], *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 1 p. 14—is consistent if and only if the formula \perp 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.

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 $F : \mathbb{N}^k \rightarrow \mathfrak{F}(\Pi)$ that associates a falsity value $F(n_1, \dots, n_k) \subseteq \Pi$ to every k -tuple $(n_1, \dots, n_k) \in \mathbb{N}^k$. 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 Def. 6 (replacing the expression ‘formula’ by ‘formula with parameters’) plus the following rule:*

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

The notions of a *predicate with parameters* and of a *typing context with parameters* are defined similarly. The notations $FV(A)$, $FV(P)$, $FV(\Gamma)$, $\text{dom}(\Gamma)$, $A\{x := e\}$, $A\{X := P\}$, 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.

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*

$$\begin{aligned} \|\dot{F}(e_1, \dots, e_k)\| &= F(\llbracket e_1 \rrbracket, \dots, \llbracket e_k \rrbracket) \\ \|A \Rightarrow B\| &= |A| \cdot \|B\| = \{t \cdot \pi : t \in |A|, \pi \in \|B\|\} \\ \|\forall x A\| &= \bigcup_{n \in \mathbb{N}} \|A\{x := n\}\| \\ \|\forall X A\| &= \bigcup_{F: \mathbb{N}^k \rightarrow \mathfrak{F}(\Pi)} \|A\{X := \dot{F}\}\| \quad (\text{if } X \text{ has arity } k) \end{aligned}$$

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

Since the falsity value $\|A\|$ (resp. the truth value $|A|$) of A actually depends on the pole \perp , we shall write it sometimes $\|A\|_\perp$ (resp. $|A|_\perp$) to recall the dependency. 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 $t \in |A|_\perp$.
(This notion is relative to a particular pole \perp .)
- t universally realizes A and write $t \Vdash\!\!\! \Vdash A$ when $t \in |A|_\perp$ for all poles \perp .

From these definitions, we clearly have

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

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 [13]. Writing

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

we easily check that:

Lemma 1. — *For all closed formulas A and B with parameters:*

- (1) $|A \Rightarrow B| \subseteq |A| \rightarrow |B|$ (adequacy of modus ponens).
- (2) The converse inclusion does not hold in general, unless the pole \perp is insensitive to the rule (PUSH), that is: $tu \star \pi \in \perp$ iff $t \star u \cdot \pi \in \perp$ (for all $t, u \in \Lambda, \pi \in \Pi$).
- (3) In all cases, $t \in (|A| \rightarrow |B|)$ 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 $\pi \in \llbracket B \rrbracket$. By applying the rule (PUSH) we get $tu \star \pi >_1 t \star u \cdot \pi \in \perp$, since $t \in |A \Rightarrow B|$ and $u \cdot \pi \in \llbracket A \Rightarrow B \rrbracket$. Hence $tu \star \pi \in \perp$ by anti-evaluation.

(2) Let $t \in |A| \rightarrow |B|$. To prove that $t \in |A \Rightarrow B|$, we consider an arbitrary element of the falsity value $\llbracket A \Rightarrow B \rrbracket$, that is, a stack $u \cdot \pi$ where $u \in |A|$ and $\pi \in \llbracket B \rrbracket$. We clearly have $tu \star \pi \in \perp$, since $tu \in |B|$ from our assumption on t . But since \perp is insensitive to the rule (PUSH), we also have $t \star u \cdot \pi \in \perp$.

(3) Let $t \in |A| \rightarrow |B|$. To prove that $\lambda x . tx \in |A \Rightarrow B|$, we consider an arbitrary element of the falsity value $\llbracket A \Rightarrow B \rrbracket$, that is, a stack $u \cdot \pi$ where $u \in |A|$ and $\pi \in \llbracket B \rrbracket$. We have $\lambda x . tx \star u \cdot \pi >_1 tu \star \pi \in \perp$ (since $tu \in |B|$), hence $\lambda x . tx \star u \cdot \pi \in \perp$ by anti-evaluation. \square

Lemma 2 (Law of Peirce). — *Let A and B be two closed formulas with parameters:*

- (1) *If $\pi \in \llbracket A \rrbracket$, then $k_\pi \Vdash A \Rightarrow B$.*
- (2) *$\mathbb{C} \Vdash ((A \Rightarrow B) \Rightarrow A) \Rightarrow A$.*

Proof. (1) Let $\pi \in \llbracket A \rrbracket$. To prove that $k_\pi \in |A \Rightarrow B|$, we need to check that $k_\pi \star t \cdot \pi' \in \perp$ for all $t \in |A|$ and $\pi' \in \llbracket B \rrbracket$. By applying the rule (RESTORE) we get $k_\pi \star t \cdot \pi' >_1 t \star \pi \in \perp$ (since $t \in |A|$ and $\pi \in \llbracket A \rrbracket$), hence $k_\pi \star t \cdot \pi' \in \perp$ by anti-evaluation.

(2) To prove that $\mathbb{C} \Vdash ((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ (for an arbitrary pole \perp), we need to check that $\mathbb{C} \star t \cdot \pi \in \perp$ for all $t \in \llbracket (A \Rightarrow B) \Rightarrow A \rrbracket$ and $\pi \in \llbracket A \rrbracket$. By applying the rule (SAVE) we get $\mathbb{C} \star t \cdot \pi >_1 t \star k_\pi \cdot \pi$. But since $k_\pi \in |A \Rightarrow B|$ (from (1)) and $\pi \in \llbracket A \rrbracket$, we have $k_\pi \cdot \pi \in \llbracket (A \Rightarrow B) \Rightarrow A \rrbracket$, so that $t \star k_\pi \cdot \pi \in \perp$. Hence $\mathbb{C} \star t \cdot \pi \in \perp$ by anti-evaluation. \square

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 10 (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 $\rho(X) : \mathbb{N}^k \rightarrow \mathfrak{F}(\Pi)$ 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:*

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

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

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

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 11 (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 $(\sigma, x := u) \equiv \sigma_{|\text{dom}(\sigma) \setminus \{x\}} \cup \{x := u\}$.*

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

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

4.4. **Adequacy.** Given a fixed pole \perp , we say that:

- A typing judgment $\Gamma \vdash t : A$ is *adequate* (w.r.t. the pole \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) 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 1 (Adequacy). — *The typing rules of Fig. 1 are adequate w.r.t. any pole \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 [17].)

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

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

- The subtyping judgment $A \leq B$ is *adequate* (w.r.t. the pole \perp) if for all valuations ρ we have $\|B\| \subseteq \|A\|$ (so that $|A| \subseteq |B|$).
- The subtyping equivalence judgment $A \simeq B$ is *adequate* (w.r.t. the pole \perp) if for all valuations ρ we have $\|A\| = \|B\|$ (so that $|A| = |B|$).

The notion of an adequate inference rule (cf Section 4.4) is extended to all the inference rules involving the new forms of judgments $A \leq B$ 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 2. — *The inference rules of Fig. 2 are adequate w.r.t. all poles \perp .*

Proof. Immediately follows from the definitions. □

$$\begin{array}{c}
 \hline \hline
 \frac{}{A \leq A} \quad \frac{A \leq B \quad B \leq C}{A \leq C} \quad \frac{\Gamma \vdash t : A \quad A \leq B}{\Gamma \vdash t : B} \\
 \frac{A \leq B \quad B \leq A}{A \simeq B} \quad \frac{A \simeq B \quad A \leq B}{B \leq A} \\
 \frac{}{\perp \leq A} \quad \frac{}{A \leq \top} \quad \frac{}{\top \leq A \Rightarrow \top} \\
 \frac{}{\forall x A \leq A\{x := e\}} \quad \frac{}{\forall X A \leq A\{X := P\}} \\
 \frac{A \leq B}{A \leq \forall x B} \quad x \notin FV(A) \quad \frac{A \leq B}{A \leq \forall X B} \quad X \notin FV(A) \quad \frac{A' \leq A \quad B \leq B'}{A \Rightarrow B \leq A' \Rightarrow B'} \\
 \hline \hline
 \frac{}{\forall x (A \Rightarrow B) \leq A \Rightarrow \forall x B} \quad x \notin FV(A) \quad \frac{}{\forall X (A \Rightarrow B) \leq A \Rightarrow \forall X B} \quad X \notin FV(A)
 \end{array}$$

FIGURE 2. Adequate rules of subtyping and of subtyping equivalence

From the inference rules of Fig. 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 PA2, Leibniz equality $e_1 = e_2$ is defined by $e_1 = e_2 \equiv \forall Z (Z(e_1) \Rightarrow Z(e_2))$. The falsity value of an equality $e_1 = e_2$ between two closed arithmetic expressions e_1 and e_2 is characterized as follows:

Lemma 3. — *For all closed arithmetic expressions e_1 and e_2 , we have*

$$\|e_1 = e_2\| = \begin{cases} \|\forall X (X \Rightarrow X)\| = \{t \cdot \pi : t \star \pi \in \perp\} & \text{if } \llbracket e_1 \rrbracket = \llbracket e_2 \rrbracket \\ \|\top \Rightarrow \perp\| = \Lambda \cdot \Pi & \text{otherwise} \end{cases}$$

Proof. We distinguish two cases:

- $\llbracket e_1 \rrbracket = \llbracket e_2 \rrbracket = n \in \mathbb{N}$. In this case we have:

$$\|e_1 = e_2\| = \bigcup_{F: \mathbb{N} \rightarrow \mathfrak{P}(\Pi)} F(n)^\perp \cdot F(n) = \bigcup_{S \in \mathfrak{P}(\Pi)} S^\perp \cdot S = \|\forall X (X \Rightarrow X)\|.$$

Moreover, it is clear that $S^\perp \cdot S \subseteq \{t \cdot \pi : t \star \pi \in \perp\}$ for all $S \subseteq \Pi$, from which we get the inclusion $\|e_1 = e_2\| \subseteq \{t \cdot \pi : t \star \pi \in \perp\}$. Conversely, if we take t and π such that $t \star \pi \in \perp$, we have $(t \cdot \pi) \in (S^\perp \cdot S) \subseteq \|e_1 = e_2\|$, by taking $S = \{\pi\}$.

- $\llbracket e_1 \rrbracket = n_1$ and $\llbracket e_2 \rrbracket = n_2$, where $n_1 \neq n_2$. In this case we have:

$$\|e_1 = e_2\| = \bigcup_{F: \mathbb{N} \rightarrow \mathfrak{P}(\Pi)} F(n_1)^\perp \cdot F(n_2) = \bigcup_{S_1, S_2 \in \mathfrak{P}(\Pi)} S_1^\perp \cdot S_2 = \|\forall X \forall Y (X \Rightarrow Y)\|.$$

We notice that the judgment $\forall X \forall Y (X \Rightarrow Y) \simeq (\top \Rightarrow \perp)$ is derivable from the rules of Fig. 2, hence $\|e_1 = e_2\| = \|\forall X \forall Y (X \Rightarrow Y)\| = \|\top \Rightarrow \perp\| = \Lambda \cdot \Pi$. \square

In the classical realizability model, true equalities are thus interpreted the same way as the true formula $\forall X (X \Rightarrow X)$ (identity type) whereas false equalities are interpreted the same way as the false formula $\top \Rightarrow \perp$.

Proposition 3 (Realizing true equations). — *Let $e_1(x_1, \dots, x_k)$ and $e_2(x_1, \dots, x_k)$ be two arithmetic expressions depending on first-order variables x_1, \dots, x_k . If $\llbracket e_1(n_1, \dots, n_k) \rrbracket = \llbracket e_2(n_1, \dots, n_k) \rrbracket$ for all $(n_1, \dots, n_k) \in \mathbb{N}^k$, then*

$$\lambda z . z \Vdash \forall x_1 \cdots \forall x_k (e_1(x_1, \dots, x_k) = e_2(x_1, \dots, x_k)).$$

Proof. Obvious from Lemma 3. \square

As a consequence, all the defining equations coming with the primitive recursive symbols $f \in \Sigma$ are universally realized by the identity term $\lambda z . z$. Moreover:

Proposition 4 (Realizing Peano's 3rd and 4th axioms). :

- (1) $\lambda z . z \Vdash \forall x \forall y (s(x) = s(y) \Rightarrow x = y)$
- (2) $\lambda z . zu \Vdash \forall x (s(x) = 0 \Rightarrow \perp)$ (where u is any term such that $FV(u) \subseteq \{z\}$).

Proof. (1) Obvious since $\|s(n) = s(m)\| = \|n = m\|$ (Lemma 3) for all $n, m \in \mathbb{N}$.

(2) Since $\|s(n) = 0 \Rightarrow \perp\| = \|(\top \Rightarrow \perp) \Rightarrow \perp\|$ for all $n \in \mathbb{N}$ (Lemma 3), it suffices to check that $\lambda z . zu \Vdash (\top \Rightarrow \perp) \Rightarrow \perp$, which is obvious. \square

From this we deduce the main theorem:

Theorem 1 (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 Prop. 1, 3 and 4. \square

4.7. The full standard model of PA2 as a degenerate case. It is easy to see that when the pole \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 $\perp = \emptyset$, the truth value S^\perp associated to an arbitrary falsity value $S \subseteq \Pi$ can only take two different values: $S^\perp = \Lambda_c$ when $S = \emptyset$, and $S^\perp = \emptyset$ when $S \neq \emptyset$. 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 $\perp = \emptyset$. Writing \mathcal{M} for the full standard model of PA2, we thus easily show that:

Proposition 5. — *If $\perp = \emptyset$, then for every closed formula A of PA2 we have*

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

Proof. We more generally show that for all formulas A and for all valuations ρ closing A (in the sense defined in section 4.2) we have

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

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

- $\tilde{\rho}(x) = \rho(x)$ for all first-order variables x ;
- $\tilde{\rho}(X) = \{(n_1, \dots, n_k) \in \mathbb{N}^k : \rho(X)(n_1, \dots, n_k) = \emptyset\}$ 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 1. — *If a closed formula A has a universal realizer $t \Vdash A$, then A is true in the full standard model \mathcal{M} of PA2.*

Proof. If $t \Vdash A$, then $t \in |A|_\emptyset$. Therefore $|A|_\emptyset = \Lambda$ and $\mathcal{M} \models A$. □

However, the converse implication is false in general, since the formula $\forall x \text{Nat}(x)$ (cf Section 3.3.1) that expresses the induction principle over individuals is obviously true in \mathcal{M} , but it has no universal realizer when evaluation is deterministic [17, 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 issue 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 (`h`), 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 even not 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 $t \star u \cdot \pi > u \star \pi$ 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{II} , $\delta \mathbf{I}$ (where $\delta \equiv \lambda x . xx$), etc.

Proposition 6. — *For all terms $t \in \Lambda$, the following assertions are equivalent:*

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

Proof. (2) \Rightarrow (1). Let t be a closed λ_c -term that is identity-like. To prove that $t \Vdash \forall X (X \Rightarrow X)$, let us consider an arbitrary pole \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 $u \in |\dot{S}| = S^\perp$ and $\pi \in \|\dot{S}\| = S$. We have $t \star u \cdot \pi > u \star \pi \in \perp$, hence $t \star u \cdot \pi \in \perp$ by anti-evaluation.

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

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 from any particular set of instructions C . However, we should not forget that there are many ways to implement identity-like terms using call/cc or other instructions, for instance:

- $\lambda x . \mathbf{cc}(\lambda k . x)$, $\lambda x . \mathbf{cc}(\lambda k . kx)$, $\lambda x . \mathbf{cc}(\lambda k . kx(\delta\delta))$
- $\mathbf{cc}(\lambda k . \mathbf{I})$, $\mathbf{cc}(\lambda k . k\mathbf{I})$, $\mathbf{cc}(\lambda k . k\mathbf{I}\delta k)$, $\mathfrak{m}\mathbf{I}(\delta\delta)$, etc.

Goal-oriented poles vs 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 $\perp_{u, \pi} = \{p : p > u \star \pi\}$ 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 $t \Vdash X (X \Rightarrow X)$, and take two elements $u \in \Lambda$ and $\pi \in \Pi$. We now consider the pole

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

as well as the falsity value $S = \{\pi\}$. From the definition of \perp , we have $t \star u \cdot \pi \notin \perp$. But since $t \Vdash \dot{S} \Rightarrow \dot{S}$ and $\pi \in \|\dot{S}\|$, we immediately get $u \not\Vdash \dot{S}$. Which precisely means that $u \star \pi \notin \perp$, so that $u \star \pi \in \mathbf{th}(t \star u \cdot \pi)$. \square

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 , by generating \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 , by *excluding* from \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 $(\mathbf{th}(p_0))^{cc} = \mathbf{th}(p_0)$.

In the simple cases we are studying in this section, it is always 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 [16] and [11].

5.2. **The Booleans.** Let us now consider the unary predicate $\text{Bool}(x)$ defined by

$$\text{Bool}(x) \equiv \forall X (X(0) \Rightarrow X(1) \Rightarrow X(x)).$$

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 $\text{Bool}(x)$. It is clear from the rules of Fig. 2 that the formula $\text{Bool}(x)$ is a subtype of the formula Bool .

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

- *True-like* if $t \star u_0 \cdot u_1 \cdot \pi > u_0 \star \pi$ for all $u_0, u_1 \in \Lambda$ and $\pi \in \Pi$.
- *False-like* if $t \star u_0 \cdot u_1 \cdot \pi > u_1 \star \pi$ 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 $t \star u_0 \cdot u_1 \cdot \pi > u_0 \star \pi$ or $t \star u_0 \cdot u_1 \cdot \pi > u_1 \star \pi$.

It is clear from these definitions that both True-like and False-like terms are particular cases of Boolean-like terms. We easily check that:

Proposition 7. — *For all closed terms $t \in \Lambda$:*

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

Proof. The proofs of (1), (2) and (4) proceed similarly to the proof of Prop. 6. (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 $\text{Bool}(n)$ (where $n \geq 2$) is false in the standard model of PA2, and thus has no universal realizer. \square

The above proposition shows that in classical realizability, universal realizers of the formulas $\text{Bool}(0)$, $\text{Bool}(1)$ and Bool have exactly the same computational behavior as the intuitionistic realizers of these formulas in AF2 [14]. But in AF2, it is also clear that the set of (intuitionistic) realizers of Bool is exactly the disjoint union of the sets of realizers of the formulas $\text{Bool}(0)$ and $\text{Bool}(1)$. In classical realizability, the situation is more complex due to the presence of extra instructions, as we shall now see.

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 \uparrow introduced in Section 2.3.

From Prop. 7, it is clear that non-deterministic choice operators are exactly the closed λ_c -terms t such that $t \Vdash \text{Bool}(0)$ and $t \Vdash \text{Bool}(1)$ (simultaneously). In classical realizability, the sets of universal realizers of the formulas $\text{Bool}(0)$ and $\text{Bool}(1)$ may thus have a nonempty intersection, depending on the presence of a non deterministic operator in the calculus. If the calculus provides the instruction \uparrow 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 $\text{Bool}(0)$ and $\text{Bool}(1)$ do not intersect.

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.

It is clear that versatile Booleans cannot exist in the pure λ -calculus. 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 $D \equiv \lambda xy. \text{quote}(\lambda n. \text{even } n \ xy)$, 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 $E \equiv \lambda xy. \text{eq } x \mathbf{I} x y$. Here the term E returns its first argument if it is equal to \mathbf{I} , and its second argument otherwise.

From Prop. 7, both terms D and E are universal realizers of the formula Bool , but none of them universally realizes $\text{Bool}(0)$ or $\text{Bool}(1)$.

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{II} for instance⁶.

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 12 (Interaction constants). — *A constant $\kappa \in C$ is said to be*

- inert if for all $\pi \in \Pi$, there is no process p such that $\kappa \star \pi >_1 p$;
- substitutive if for all $u \in \Lambda$ and for all processes $p, p' \in \Lambda \star \Pi$, $p >_1 p'$ implies $p\{\kappa := u\} >_1 p'\{\kappa := u\}$;
- non generative if for all processes $p, p' \in \Lambda \star \Pi$ such that $p >_1 p'$, the constant κ cannot occur in p' unless it already occurs in p .

A constant $\kappa \in C$ 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 $p, p' \in \Lambda \star \Pi$, $p >_1 p'$ implies $p\{\alpha := \pi\} >_1 p'\{\alpha := \pi\}$;
- non generative if for all $p, p' \in \Lambda \star \Pi$ such that $p >_1 p'$, 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 8. — *If the calculus of realizers contains one of both instructions `quote` or `eq`, then none of the constants $\kappa \in C$ 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 $\kappa \in C$, let us consider the process $p \equiv \text{eq} \star \kappa \cdot \mathbf{I} \cdot \delta \cdot \mathbf{I} \cdot \alpha$, where $\alpha \in \mathcal{B}$ is a fixed stack constant. We notice that $p >_1 \mathbf{I} \star \alpha$ (since $\kappa \neq \mathbf{I}$) whereas $p\{\kappa := \mathbf{I}\} \equiv \text{eq} \star \mathbf{I} \cdot \mathbf{I} \cdot \delta \cdot \mathbf{I} \cdot \alpha >_1 \delta \star \alpha \neq (\mathbf{I} \star \alpha)\{\kappa := \mathbf{I}\}$, hence the constant κ is not substitutive. The same argument applies to the instruction `quote`, since the instruction `eq` can be implemented from it. \square

On the other hand, it is clear that if the relation of evaluation $>_1$ is defined from the only rules (GRAB), (PUSH), (SAVE) and (RESTORE)—and possibly: the rule (FORK)—then all the remaining constants κ in C (i.e. $\kappa \neq \mathbf{cc}, \mathfrak{h}$) 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 analyze the computational behavior of realizers in a uniform way. For instance, if we know that a closed term $t \in \Lambda$ is such that

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

where $\kappa_1, \dots, \kappa_n$ 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

$$t \star u_1 \cdots u_n \cdot \pi > p\{\kappa_1 := u_1; \dots; \kappa_n := u_n; \alpha := \pi\}$$

for all terms $u_1, \dots, u_n \in \Lambda$ 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.

⁶In 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.

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

Proposition 9. — *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 $t \star \kappa_0 \cdot \kappa_1 \cdot \alpha > \kappa_0 \star \alpha$. By substitutivity, we have

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

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

- Either $t \star \kappa_0 \cdot \kappa_1 \cdot \alpha > \kappa_1 \star \alpha$. Symmetrically, we deduce that t is False-like. \square

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 Fig. 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 2 that the instruction \mathfrak{c} 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 behavior than the one of the instruction \mathfrak{c} .

6.1. The family of terms $(\mathfrak{c}_{n,p})_{n \geq p \geq 1}$. To illustrate the possible computational behaviors of the universal realizers of Peirce's law, it is useful to introduce a sequence of closed proof-like terms $\mathfrak{c}_{n,p}$ indexed by all pairs of integers (n, p) such that $n \geq p \geq 1$.

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

$$\begin{aligned} K_{n,p}^n(x_0, k, x_1, \dots, x_{n-1}) &\equiv \lambda x_n . k x_p \\ K_{n,p}^i(x_0, k, x_1, \dots, x_{i-1}) &\equiv \lambda x_i . k (x_0 K_{n,p}^{i+1}(x_0, k, x_1, \dots, x_i)) \quad (1 \leq i < n) \end{aligned}$$

(Notice that we actually have $FV(K_{n,p}^i(x_0, k, x_1, \dots, x_{i-1})) = \{x_0; k\} \cup \{x_p \text{ if } p < i\}$.)

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

$$\mathfrak{c}_{n,p} \equiv \lambda x_0 . \mathfrak{c} (\lambda k . x_0 K_{n,p}^1(x_0, k)).$$

We easily check that:

Fact 1. — *For all $u_0, \dots, u_n \in \Lambda$ and $\pi_0, \dots, \pi_n \in \Pi$, we have:*

$$\begin{aligned} \mathfrak{c}_{n,k} \star u_0 \cdot \pi_0 &> u_0 \star K_{n,p}^1(u_0, k_{\pi_0}) \cdot \pi_0 \\ K_{n,p}^i(u_0, k_{\pi_0}, u_1, \dots, u_{i-1}) \star u_i \cdot \pi_i &> u_0 \star K_{n,p}^{i+1}(u_0, k_{\pi_0}, u_1, \dots, u_i) \cdot \pi_0 \quad (1 \leq i < n) \\ K_{n,p}^n(u_0, k_{\pi_0}, u_1, \dots, u_{n-1}) \star u_n \cdot \pi_n &> u_p \star \pi_0 \end{aligned}$$

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

$$\begin{aligned} \mathfrak{C}_{1,1} \star u_0 \cdot \pi_0 &> u_0 \star (\lambda x_1 . k_{\pi_0} x_1) \cdot \pi_0 && (\text{since } K_{1,1}^1(u_0, k_{\pi_0}) \equiv \lambda x_1 . k_{\pi_0} x_1) \\ (\lambda x_1 . k_{\pi_0} x_1) \star u_1 \cdot \pi_1 &> u_1 \star \pi_0, \end{aligned}$$

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

We can now prove that every $\mathfrak{C}_{n,p}$ ($n \geq p \geq 1$) is a universal realizer of Peirce's law:

Proposition 10. — *For all $n \geq p \geq 1$, we have: $\mathfrak{C}_{n,p} \Vdash \forall X ((\neg X \Rightarrow X) \Rightarrow X)$.*

Proof. Let us consider a fixed pole \perp and a fixed falsity value $S \subseteq \Pi$. Given a realizer $u_0 \in |\neg \dot{S} \Rightarrow \dot{S}|$ and a stack $\pi_0 \in S$, we want to prove that $\mathfrak{C}_{n,p} \star u_0 \cdot \pi_0 \in \perp$.

For that, we first need to prove that for all $i \in [1..n]$ and for all $u_1, \dots, u_{i-1} \in |\dot{S}| = S^\perp$, we have $K(u_0, k_{\pi_0}, u_1, \dots, u_{i-1}) \Vdash \neg \dot{S}$. This property is proved from $i = n$ down to $i = 1$:

- $i = n$. Let $u_1, \dots, u_{n-1} \in S^\perp$. For all $u_n \in |\dot{S}| = S^\perp$ and $\pi_n \in |\perp| = \Pi$, we have

$$K_{n,p}^n(u_0, k_{\pi_0}, u_1, \dots, u_{n-1}) \star u_n \cdot \pi_n > u_p \star \pi_0 \in \perp$$

since $u_p \in S^\perp$ and $\pi_0 \in S$, hence $K_{n,p}^n(u_0, k_{\pi_0}, u_1, \dots, u_{n-1}) \Vdash \neg \dot{S}$.

- $1 \leq i < n$. Assume that the property holds for $i + 1$, and take $u_1, \dots, u_{i-1} \in S^\perp$. For all $u_i \in |\dot{S}| = S^\perp$ and $\pi_i \in |\perp| = \Pi$, we have

$$K_{n,p}^i(u_0, k_{\pi_0}, u_1, \dots, u_{i-1}) \star u_i \cdot \pi_i > u_0 \star K_{n,p}^{i+1}(u_0, k_{\pi_0}, u_1, \dots, u_i) \cdot \pi_0.$$

Since $K_{n,p}^{i+1}(u_0, k_{\pi_0}, u_1, \dots, u_i) \in |\neg \dot{S}|$ (by induction hypothesis) and $\pi_0 \in S$, we get $K_{n,p}^{i+1}(u_0, k_{\pi_0}, u_1, \dots, u_i) \cdot \pi_0 \in \|\neg \dot{S} \Rightarrow \dot{S}\|$, hence $u_0 \star K_{n,p}^{i+1}(u_0, k_{\pi_0}, u_1, \dots, u_i) \cdot \pi_0 \in \perp$.

By anti-evaluation, we get $K_{n,p}^i(u_0, k_{\pi_0}, u_1, \dots, u_{i-1}) \star u_i \cdot \pi_i \in \perp$, and this for all $u_i \in |\dot{S}|$ and $\pi_i \in \Pi$, hence $K_{n,p}^i(u_0, k_{\pi_0}, u_1, \dots, u_{i-1}) \Vdash \neg \dot{S}$.

To conclude that $\mathfrak{C}_{n,p} \star u_0 \cdot \pi_0 \in \perp$, it suffices to check that

$$\mathfrak{C}_{n,p} \star u_0 \cdot \pi_0 > u_0 \star K_{n,p}^1(u_0, k_{\pi_0}) \cdot \pi_0 \in \perp,$$

which is obvious since $u_0 \Vdash \neg \dot{S} \Rightarrow \dot{S}$, $K_{n,p}^1(u_0, k_{\pi_0}) \Vdash \neg \dot{S}$ and $\pi_0 \in S$. \square

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 in the calculus of realizers, every universal realizer of Peirce's law behaves as one of the terms $\mathfrak{C}_{n,p}$.

6.2. A first game \mathfrak{G}_0 . To understand the computational behavior 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) and \forall (opponent), that we denote by \mathfrak{G}_0 .

The game \mathfrak{G}_0 starts with an initialization phase, where \exists brings a closed term $t_0 \in \Lambda$ that is intended to realize the law of Peirce, while \forall brings a 'handle' $(u_0, \pi_0) \in \Lambda \times \Pi$ that is intended to be used by \exists to communicate its moves to \forall . The rest of the game is then parameterized by the handle (u_0, π_0) , and starts at the \exists -position $t_0 \star u_0 \cdot \pi_0$.

During the game, \exists -positions are indicated as processes of the form $t \star u \cdot \pi$, while \forall -positions are indicated as closed terms t (corresponding to the last \exists -move). The game alternates \exists -moves and \forall -moves that are defined as follows:

- \exists -moves. Starting from an \exists -position $t \star u \cdot \pi$, player \exists evaluates the process $t \star u \cdot \pi$ in order to reach one of the following states:
 - If $t \star u \cdot \pi > u \star \pi_0$, where u is the first component of a pair (u, π) that was formerly played by \forall (not counting the initial pair (u_0, π_0)), then \exists wins.
 - If $t \star u \cdot \pi > u_0 \star t' \cdot \pi_0$, then \exists may communicate its answer t' to \forall through the handle (u_0, π_0) , in which case the answer t' becomes the current \forall -position. Notice that such a term t' is not necessarily unique (even when the relation of evaluation is deterministic), so that \exists can freely choose any such term t' .

If none of the above two moves is possible, then \forall wins.

- \forall -moves. Starting from an \exists -position t , opponent \forall can play any pair $(u, \pi) \in \Lambda \times \Pi$. The process $t \star u \cdot \pi$ then becomes the current \exists -position.

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 $\mathfrak{cc}_{n,p}$ ($n \geq p \geq 1$) constitutes a winning strategy for \exists in exactly $2n + 1$ moves. Indeed, if we start from the initial position $\mathfrak{cc}_{n,p} \star u_0 \cdot \pi_0$ (using an arbitrary handle (u_0, π_0)), then the game proceeds as follows:

- \exists plays $t_1 \equiv K_{n,p}^1(u_0, k_{\pi_0})$ (since $\mathfrak{cc}_{n,p} \star u_0 \cdot \pi_0 > u_0 \star t_1 \cdot \pi_0$).
- \forall plays any move (u_1, π_1)
- \exists plays $t_2 \equiv K_{n,p}^2(u_0, k_{\pi_0}, u_1)$ (since $t_1 \star u_1 \cdot \pi_1 > u_0 \star t_2 \cdot \pi_0$).
- \vdots
- \forall plays any move (u_{n-1}, π_{n-1})
- \exists plays $t_n \equiv K_{n,p}^n(u_0, k_{\pi_0}, u_1, \dots, u_{n-1})$ (since $t_{n-1} \star u_{n-1} \cdot \pi_{n-1} > u_0 \star t_n \cdot \pi_0$).
- \forall plays any move (u_n, π_n)
- \exists wins (since $t_n \star u_n \cdot \pi_n > u_p \star \pi_0$, where $1 \leq p \leq n$).

Let us now formalize the notion of a winning strategy for the game \mathfrak{G}_0 (from the point of view of player \exists). We call a \mathfrak{G}_0 -state (or simply: a state) any quadruple $(t \star u \cdot \pi)[\ell]$ where $t \star u \cdot \pi$ is a process and where ℓ is a finite list of closed terms, that intuitively represents the former moves of \forall . Given a fixed handle $(u_0, \pi_0) \in \Lambda \times \Pi$, we inductively define the set $W_{(u_0, \pi_0)}$ of *winning states* from the following two inference rules:

$$\frac{}{(t \star u \cdot \pi)[\ell] \in W_{(u_0, \pi_0)}} \quad (\text{if } t \star u \cdot \pi > u' \star \pi_0 \text{ and } u' \in \ell)$$

$$\frac{(t' \star u' \cdot \pi')[\ell, u'] \in W_{(u_0, \pi_0)} \text{ for all } (u', \pi') \in \Lambda \times \Pi}{(t \star u \cdot \pi)[\ell] \in W_{(u_0, \pi_0)}} \quad (\text{if } t \star u \cdot \pi > u_0 \star t' \cdot \pi_0)$$

Notice that the second inference rule has infinitely many premises, that correspond to all the possible \forall -moves $(u', \pi') \in \Lambda \times \Pi$. A derivation of $(t \star u \cdot \pi)[\ell] \in W_{(u_0, \pi_0)}$ is thus an infinitely branching well-founded tree. Finally, we say that a closed term t is a *winning strategy for the game \mathfrak{G}_0* if $(t \star u_0 \cdot \pi_0)[\ell] \in W_{(u_0, \pi_0)}$ for all handles $(u_0, \pi_0) \in \Lambda \times \Pi$.

We easily check that:

Fact 2. — For all $n \geq p \geq 1$, $\mathfrak{cc}_{n,p}$ is a winning strategy for the game \mathfrak{G}_0 .

Moreover, the terms $\mathfrak{cc}_{n,p}$ are *uniform* strategies, in the sense that all the corresponding plays have the very same structure: they all have the same length ($2n + 1$ moves) and they all use the p th 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 \mathfrak{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 11 (Adequacy of winning strategies for the game \mathfrak{G}_0). — If $t_0 \in \Lambda$ is a winning strategy for the game \mathfrak{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 \mathfrak{G}_0 , and take a pole \perp , a falsity value $S \subseteq \Pi$, a realizer $u_0 \in |\neg \dot{S} \Rightarrow \dot{S}|$ as well as a stack $\pi \in \dot{S}$. We want to show that $t_0 \star u_0 \cdot \pi_0 \in \perp$. For that, we more generally prove that

$$\text{For all states } (t \star u \cdot \pi)[\ell] \in W_{(u_0, \pi_0)}, \text{ if } \ell \subseteq S^\perp, \text{ then } t \star u \cdot \pi \in \perp.$$

We proceed by induction on the derivation of $(t \star u \cdot \pi)[\ell] \in W_{(u_0, \pi_0)}$, distinguishing the following two cases:

- (1) $(t \star u \cdot \pi)[\ell] \in W_{(u_0, \pi_0)}$ since $t \star u \cdot \pi > u' \star \pi_0$, where $u' \in \ell$.
If we assume that $\ell \subseteq S^\perp$, we thus get $u' \star \pi_0 \in \perp$ (since $u' \in S^\perp$ and $\pi_0 \in S$). We then conclude by anti-evaluation.
- (2) $(t \star u \cdot \pi)[\ell] \in W_{(u_0, \pi_0)}$ since $t \star u \cdot \pi > u_0 \star t' \cdot \pi_0$ for some term $t' \in \Lambda$ such that $(t' \star u' \cdot \pi')[\ell, u'] \in W_{(u_0, \pi_0)}$ for all $(u', \pi') \in \Lambda \times \Pi$.

Let us assume that $\ell \subseteq S^\perp$. We first want to prove that $t' \Vdash \neg \dot{S}$. For that, we take a term $u' \in S^\perp$ and a stack $\pi' \in \Pi$. Since $\ell \cup \{u'\} \subseteq S^\perp$, we get $t' \star u' \cdot \pi' \in \perp$ 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 $u_0 \star t' \cdot \pi_0 \in \perp$. We conclude by anti-evaluation.

By induction, the property holds for all states $(t \star u \cdot \pi)[\ell] \in W_{(u_0, \pi_0)}$. In particular, we have shown that $t_0 \star u_0 \cdot \pi_0 \in \perp$, since $(t_0 \star u_0 \cdot \pi_0)[\] \in W_{(u_0, \pi_0)}$ from our assumption on t_0 . \square

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 Prop. 11 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 4. — *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 $n \geq p \geq 1$ as well as a finite sequence t_1, \dots, t_n of n closed terms such that:*

$$\begin{aligned} t_0 \star \kappa_0 \cdot \alpha_0 &> \kappa_0 \star t_1 \cdot \alpha_0 \\ t_i \star \kappa_i \cdot \alpha_i &> \kappa_0 \star t_{i+1} \cdot \alpha_0 \\ t_n \star \kappa_n \cdot \alpha_n &> \kappa_p \star \alpha_0 \end{aligned} \quad (\text{for all } 1 \leq i < n)$$

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

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

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 defined by $\perp = (Q_\infty)^c$ as well as the falsity value $S = \{\alpha_0\}$. From the definition of \perp , we have $t_0 \star \kappa_0 \cdot \alpha_0 \notin \perp$. 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 $\kappa_0 \star t \cdot \alpha_0 \notin \perp$. From the latter, we deduce that $\kappa_0 \star t \cdot \alpha_0 \in Q_{p-1}$ for some $p \geq 1$. Hence $t \star \kappa_p \cdot \alpha_p \in Q_p$ (from the definition of Q_p) and thus $t \star \kappa_p \cdot \alpha_p \notin \perp$. But since $t \Vdash \neg \dot{S}$, we have $\kappa_p \not\Vdash \dot{S}$, hence $\kappa_p \star \alpha_0 \notin \perp$ and thus $\kappa_p \star \alpha_0 \in Q_n$ for some index $n \geq 0$. Using the fact that κ_p can only occur in the processes belonging to the sets Q_i for $i \geq p$, we get $n \geq p$ (so that $n \geq 1$). From the definition of Q_n , we immediately deduce the existence of n terms t_1, \dots, t_n such that

$$\begin{aligned} t_0 \star \kappa_0 \cdot \alpha_0 \in Q_0 &> \kappa_0 \star t_1 \cdot \alpha_0 \in Q_0 \\ t_i \star \kappa_i \cdot \alpha_i \in Q_i &> \kappa_0 \star t_{i+1} \cdot \alpha_0 \in Q_i \\ t_n \star \kappa_n \cdot \alpha_n \in Q_n &> \kappa_p \star \alpha_0 \in Q_n \end{aligned} \quad (\text{for all } 1 \leq i < n) \quad \square$$

Theorem 2 (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 (Prop. 11) that the terms t_0 that are winning strategies for the game \mathbb{G}_0 universally realize Peirce's law. Conversely, let $t_0 \Vdash \forall X ((\neg X \Rightarrow X) \Rightarrow X)$, 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 4, there exist two indices n and p such that $n \geq p \geq 1$ as well as a finite sequence

t_1, \dots, t_n of n closed terms such that:

$$\begin{aligned} t_0 \star \kappa_0 \cdot \alpha_0 &> \kappa_0 \star t_1 \cdot \alpha_0 \\ t_i \star \kappa_i \cdot \alpha_i &> \kappa_0 \star t_{i+1} \cdot \alpha_0 \\ t_n \star \kappa_n \cdot \alpha_n &> \kappa_p \star \alpha_0 \end{aligned} \quad (\text{for all } 1 \leq i < n)$$

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

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

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

(by substitutivity), so that

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

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

- $0 \leq i < n$. Let us now assume that

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

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

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

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 $(t_0 \star u_0 \cdot \pi_0)[\] \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. \square

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 behavior as one of the proof-like terms $\mathbf{cc}_{n,p}$ ($n \geq p \geq 1$).

6.4. A wild realizer of Peirce's law. Theorem 2 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 \mathfrak{h} ; actually, the specification expressed in Theorem 2 makes no assumption about the determinism or the non determinism of the relation of evaluation $>_1$.

However, the assumptions of Theorem 2 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 2 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, this is not possible, since the presence of the instruction `eq` (that can be mimicked using `quote`) allows to define *wild* realizers of the law of Peirce, that is, universal realizers $t \Vdash \forall X ((\neg X \Rightarrow X) \Rightarrow X)$ that are *not* winning strategies for the game \mathbb{G}_0 .

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

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

From these definitions we get:

$$\mathbf{cc}' \star u \cdot \pi >_1 T_2(u) \star \pi > u \star K(T_1(u), k_\pi) \cdot \pi$$

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

$$K(T_1(u), k_\pi) \star u' \cdot \pi' > \begin{cases} \mathbf{I} \star \pi' & \text{if } u' = T_2(u) \\ u' \star \pi & \text{otherwise} \end{cases}$$

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

Lemma 5. — *Let \perp be a fixed pole and $S \subseteq \Pi$ an arbitrary falsity value. For all $u \in \Lambda$ and $\pi \in S$ such that $T_2(u) \star \pi \notin \perp$, we have $K(T_1(u), k_\pi) \Vdash \neg \dot{S}$.*

Proof. To show that $K(T_1(u_0), k_\pi) \Vdash \neg \dot{S}$, let us consider a stack of the form $u' \cdot \pi'$ where $u' \in S^\perp$ and $\pi' \in \Pi$. Since $\pi \in S$, we have $u' \star \pi \in \perp$, and thus $u' \neq T_2(u)$. Hence $K(T_1(u), k_\pi) \star u' \cdot \pi' > u' \star \pi \in \perp$, so that $K(T_1(u), k_\pi) \star u' \cdot \pi' \in \perp$ by anti-evaluation. \square

Proposition 12. — $\mathbf{cc}' \Vdash \forall X ((\neg X \Rightarrow X) \Rightarrow X)$

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

- $T_2(u) \star \pi \in \perp$. In this case we have $\mathbf{cc}' \star u \cdot \pi >_1 T_2(u) \star \pi \in \perp$, from which we get $\mathbf{cc}' \star u \cdot \pi \in \perp$ by anti-evaluation.
- $T_2(u) \star \pi \notin \perp$. In this case we have $\mathbf{cc}' \star u \cdot \pi > u \star K(T_1(u), k_\pi) \cdot \pi$. Since $K(T_1(u), k_\pi) \Vdash \neg \dot{S}$ (from Lemma 5), we have $u \star K(T_1(u), k_\pi) \cdot \pi \in \perp$, hence $\mathbf{cc}' \star u \cdot \pi \in \perp$ by anti-evaluation. \square

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 5 and Prop. 12, 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 behavior of the process $K(T_1(u), k_\pi) \star u' \cdot \pi'$ since we are able to conclude that $\mathbf{cc}' \star u \cdot \pi \in \perp$ using other means (first case of the proof of Prop. 12). Also notice that the case distinction performed in the proof of Prop. 12 makes the corresponding proof intrinsically classical, which contrasts with the proof of adequacy for \mathbf{cc} (Lemma 2 p. 12), which is fully intuitionistic. In some sense, we can think of \mathbf{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 (Prop. 12) is classical too.

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

Lemma 6. — *Let us assume that the relation of one step evaluation $>_1$ is only defined from the rules (GRAB), (PUSH), (SAVE), (RESTORE) and (EQ). Then the universal realizer \mathbf{cc}' of Peirce's law is not a winning strategy for the game \mathfrak{G}_0 .*

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

- Player \exists is forced to play $t_1 \equiv K(T_1(\mathbf{I}), k_{\alpha_0})$, since t_1 is the only term t such that $\mathbf{I} \star t \cdot \alpha_0 \in \mathbf{th}(\mathbf{cc}' \star \mathbf{I} \cdot \alpha_0)$.
- 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 \mathfrak{G}_1 . Although the wild realizer \mathbf{cc}' of Peirce's law does not constitute a winning strategy for the game \mathfrak{G}_0 , we can still understand the computational behavior of \mathbf{cc}' in game-theoretic terms as follows. Starting from an arbitrary handle (u_0, π_0) :

- Player \exists plays $K(u_0, k_{\pi_0})$ (since $\mathbf{cc}' \star u_0 \cdot \pi_0 > u_0 \star K(u_0, k_{\pi_0}) \cdot \pi_0$)
- Then opponent \forall plays an arbitrary move $(u_1, \pi_1) \in \Lambda \times \Pi$.

We now need to distinguish the following two cases:

- (1) Either $u_1 \neq T_2(u_0)$. In this case, the play ends as expected:

- \exists wins, since $K(u_0, k_{\pi_0}) \star u_1 \cdot \pi_1 > u_1 \star \pi_0$.
- (2) Either $u_1 \equiv T_2(u_0)$. In this case, we can understand the strange computational behavior of \mathfrak{C}' as follows: since no interesting move can be obtained from the current \exists -position $K(u_0, k_{\pi_0}) \star u_1 \cdot \pi_1$, player \exists *backtracks* to the former \exists -position $\mathfrak{C}' \star u_0 \cdot \pi_0$ (the game started from). Then the play continues from the \exists -position $\mathfrak{C}' \star u_0 \cdot \pi_0$, and \exists now wins since

$$\mathfrak{C}' \star u_0 \cdot \pi_0 >_1 T_2(u_0) \star \pi_0 \equiv u_1 \star \pi_0.$$

In other words, we can still think of the closed term \mathfrak{C}' 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 \mathfrak{G}_1 that is defined from the same \exists/\forall -positions as before, as well as from the same \forall -moves as before. 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 \mathfrak{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 \mathfrak{G}_1 -states is inductively defined as follows:

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

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

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

Fact 3. — *The closed term \mathfrak{C}' is a winning strategy for the game \mathfrak{G}_1 .*

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

Proposition 13. — *If a closed λ_c -term is a winning strategy for the game \mathfrak{G}_0 , then it is also a winning strategy for the game \mathfrak{G}_1 .*

Proof. It suffices to prove that $(t \star u \cdot \pi)[\ell] \in W_{(u_0, \pi_0)}$ implies $\langle \{t \star u \cdot \pi\}, \ell \rangle \in W'_{(u_0, \pi_0)}$ for all \mathfrak{G}_0 -states $(t \star u \cdot \pi)[\ell]$, by induction on the derivation of $(t \star u \cdot \pi)[\ell] \in W_{(u_0, \pi_0)}$. 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 \mathfrak{G}_1 is adequate w.r.t. Peirce's law:

Proposition 14 (Adequacy of winning strategies for the game \mathfrak{G}_1). — *If t_0 is a winning strategy for the game \mathfrak{G}_1 , then $t_0 \Vdash \forall X ((\neg X \Rightarrow X) \Rightarrow X)$.*

Proof. Let \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 $t_0 \star u_0 \cdot \pi_0 \in \perp$. For that, we more generally prove that for all $\langle P, \ell \rangle \in W'_{(u_0, \pi_0)}$, $\ell \subseteq S^\perp$ implies $P \cap \perp \neq \emptyset$. 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 $p > u \star \pi_0$. If we assume that $\ell \subseteq S^\perp$, we thus get $u \star \pi_0 \in \perp$, hence $p \in \perp$ 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 $p > u_0 \star t' \cdot \pi_0$, 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 $\ell \subseteq S^\perp$. To show that $P \cap \perp \neq \emptyset$, let us assume that $P \cap \perp = \emptyset$ (using the meta-theoretic law of Peirce). We first want to show that $t' \Vdash \neg \dot{S}$. For that, let

us consider $u' \in S^\perp$ and $\pi' \in \Pi$. Since $\langle P \cup \{t' \star u' \cdot \pi'\}, \ell \cup \{u'\} \rangle \in W'_{(u_0, \pi_0)}$ and $(\ell \cup \{u'\}) \subseteq S^\perp$, we get $\langle P \cup \{t' \star u' \cdot \pi'\} \rangle \cap \perp \neq \emptyset$ by induction hypothesis. But since $P \cap \perp = \emptyset$, we deduce that $t' \star u' \cdot \pi' \in \perp$, which finishes the proof that $t' \Vdash \neg \dot{S}$. Therefore, $u_0 \star t' \cdot \pi_0 \in \perp$, and thus $p \in \perp$ by anti-evaluation.

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

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

Proposition 15 (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 $\langle \{t_0 \star u_0 \cdot \pi_0\}, \emptyset \rangle \notin W'_{(u_0, \pi_0)}$. 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 $\phi : \mathbb{N} \rightarrow \Lambda$ 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 $\ell_0 = \emptyset$, 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 $p > u_0 \star t \cdot \pi_0$. 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 $p > u_0 \star t \cdot \pi_0$. 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 $\perp = Q^c$ 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 $t_0 \star u_0 \cdot \pi_0 \notin \perp$ (from the definition of \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 $u_0 \star t \cdot \pi_0 \notin \perp$. Hence $u_0 \star t \cdot \pi_0 \in Q$, so that there is an index $n \geq 0$ and a process $p \in P_n$ such that $p > u_0 \star t \cdot \pi_0$. Let us consider an index $n' \geq n$ such that $\phi(n') \equiv t$. Since $p \in P_{n'} (\supseteq P_n)$ and $p > u_0 \star t \cdot \pi_0$, 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 $t \star u \cdot \pi \notin \perp$, hence $u \not\Vdash \dot{S}$ (since $t \Vdash \neg \dot{S}$ and $\pi \in \|\perp\|$), so that $u \star \pi_0 \notin \perp$. Hence we get $u \star \pi_0 \in Q$, so that there is an index $m \geq 0$ and a process $p \in P_m$ such that $p > u \star \pi_0$. Without loss of generality, we can assume that $m \geq n' + 1$ since the sequence $(P_i)_{i \in \mathbb{N}}$ is increasing. We have thus found an index $m \geq n' + 1$, a process $p \in P_m$ and a term $u \in \ell_m$ (since $u \in \ell_{n'+1} \subseteq \ell_m$) such that $p > u \star \pi_0$, 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. \square

From Prop. 14 and 15, we thus get the definitive specification of Peirce's law:

Theorem 3 (Specification of Peirce's law). — *The universal realizers of Peirce's law are exactly the winning strategies for the game \mathbb{G}_1 .*

REFERENCES

- [1] F. Barbanera and S. Berardi. A symmetric lambda calculus for classical program extraction. *Inf. Comput.*, 125(2):103–117, 1996.
- [2] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and The Foundations of Mathematics*. North-Holland, 1984.
- [3] A. Church. *The calculi of lambda-conversion*, volume 6 of *Annals of Mathematical Studies*. Princeton, 1941.
- [4] P.-L. Curien and H. Herbelin. The duality of computation. In *ICFP*, pages 233–243, 2000.
- [5] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, 1958.

- [6] H. Friedman. Some applications of Kleene's methods for intuitionistic systems. In *Cambridge Summer School in Mathematical Logic*, volume 337 of *Springer Lecture Notes in Mathematics*, pages 113–170. Springer-Verlag, 1973.
- [7] H. Friedman. Classically and intuitionistically provably recursive functions. *Higher Set Theory*, 669:21–28, 1978.
- [8] J.-Y. Girard. *Le point aveugle – Cours de logique – Volume I – vers la perfection*. Hermann, 2006.
- [9] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.
- [10] T. Griffin. A formulae-as-types notion of control. In *Principles Of Programming Languages (POPL'90)*, pages 47–58, 1990.
- [11] M. Guillermo. *Jeux de réalisabilité en arithmétique classique*. PhD thesis, Université Paris 7, 2008.
- [12] W. A. Howard. The formulae-as-types notion of construction. Privately circulated notes, 1969.
- [13] S. C. Kleene. On the interpretation of intuitionistic number theory. *Journal of Symbolic Logic*, 10:109–124, 1945.
- [14] J. L. Krivine. *Lambda-calculus, types and models*. Masson, 1993.
- [15] J.-L. Krivine. Typed lambda-calculus in classical Zermelo-Fraenkel set theory. *Arch. Math. Log.*, 40(3):189–205, 2001.
- [16] J.-L. Krivine. Dependent choice, 'quote' and the clock. *Th. Comp. Sc.*, 308:259–276, 2003.
- [17] J.-L. Krivine. Realizability in classical logic. In *interactive models of computation and program behaviour. Panoramas et synthèses*, 27, 2009.
- [18] P. Martin-Löf. An intuitionistic theory of types. In *twenty-five years of constructive type theory. Oxford Logic Guides*, 36:127–172, 1998.
- [19] D. McCarty. *Realizability and Recursive Mathematics*. PhD thesis, Carnegie-Mellon University, 1984.
- [20] A. Miquel. Classical program extraction in the calculus of constructions. In *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, volume 4646 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 2007.
- [21] A. Miquel. Existential witness extraction in classical realizability and via a negative translation. *Logical Methods for Computer Science*, 2010.
- [22] J. Myhill. Some properties of intuitionistic Zermelo-Fraenkel set theory. *Lecture Notes in Mathematics*, 337:206–231, 1973.
- [23] P. Oliva and T. Streicher. On Krivine's realizability interpretation of classical second-order arithmetic. *Fundam. Inform.*, 84(2):207–220, 2008.
- [24] M. Parigot. Proofs of strong normalisation for second order classical natural deduction. *J. Symb. Log.*, 62(4):1461–1479, 1997.
- [25] M. Sperber, R. Kent Dybvig, M. Flatt, A. Van Straaten, R. Findler, and J. Matthews. Revised⁶ report on the algorithmic language Scheme. *Journal of Functional Programming*, 19, Supplement S1:1–301, 2009.