3- Krivine's classical realizability

This chapter aims at being a survey on Krivine's classical realizability. Our intention is twofold. On the one hand, we recall in broad lines the key definitions of Krivine's classical realizability, and we take advantage of this to introduce some techniques that we use in the sequel of this thesis. On the other hand, we present standard applications of Krivine realizability to the study of the computational content of classical proofs and to models theory. These applications are again loosely introduced, with references pointing to articles where they are presented more in details. Nonetheless, we hope that this overview justifies our interest in the topic and in particular the third part of this manuscript, which is dedicated to the study of algebraic structures for Krivine realizability.

3.1 Realizability in a nutshell

3.1.1 Intuitionistic realizability

The very first ideas of realizability are to be found in the Brouwer-Heyting-Kolmogoroff (BHK) interpretation, which was in fact anterior to its actual formulation, done independently by Heyting for propositional logic [72] and Kolmogoroff for predicate logic [88]. The BHK-interpretation gives the meaning of a statement A by explaining what constitutes an evidence[1] while 'evidence of A' for logically compound A is explained by giving evidences of its constituents. For propositional logic:

1. a evidence of $A \wedge B$ is given by presenting a evidence of A and a evidence of B;
2. a evidence of $A \vee B$ is given by presenting either a evidence of A or a evidence of B (plus the stipulation that this evidence is presented as evidence for $A \vee B$);
3. a evidence of $A \to B$ is a construction which transforms any evidence of A into a evidence of B;
4. absurdity $\bot$ (contradiction) has no evidence; a evidence of $A \to \bot$ is a construction which transforms any evidence of A into a evidence of $\bot$.

In this definition, notions such as *"construction"*, *"transformation"* or the very notion of *"evidence"* can be understood in different ways, and indeed they have been. Intuitionistic realizability can precisely be viewed as the replacement of the notion of *"evidence"* by the formal notion of *"realizer"*, which, again, can be defined in different ways. The original presentation of realizability, due to Kleene [87], define realizers as computable functions. Each function φ is in fact identified to its Gödel's number[2] n, and *"transformation"* is defined by means of function application. Kleene's definition can be reformulated[3] as follows:

[1] We voluntarily use the terminology of "evidence" instead of "proof", to which we already gave a syntactic meaning. Besides, if we regard the BHK-interpretation of propositions with the λ-calculus in mind, we observe that evidences of A correspond to "values" of type A rather than "proofs".

[2] In practice, any other enumeration of computable functions do the job just as well, that is to say that encoding is irrelevant to the principle of Kleene's realizability.

[3] In the original presentation, a pair (n, m) is encoded by its Gödel's number $2^n 3^m$, $\mathtt{left}(n)$ is the pair $(0, n)$ and $\mathtt{right}(m)$ is the pair $(1, m)$.

1. 0 realizes $\top$;
2. if n realizes A and m realizes B, then the pair (n,m) realizes $A \wedge B$;
3. if n realizes A, then $\texttt{left}(n)$ realizes $A \vee B$, and similarly, $\texttt{right}(m)$ realizes $A \vee B$ if m realizes B;
4. the function φ_n realizes $A \to B$ if for any m realizing A, $\varphi_n(m)$ realizes B;
5. a realizer of $\neg A$ is a function realizing $A \to \bot$.

This definition can be revisited using the $\lambda^{\times+}$-calculus extended with natural numbers as the language for computable functions. We do not describe formally this calculus here[4], but only assume that the calculus contains a term $\bar{n}$ for each natural number n. We give the interpretation for first-order arithmetic formulas (see Example 1.3).

1. $t \Vdash \top$ if $t \xrightarrow{*} 0$;
2. $t \Vdash e_1 = e_2$ if $e_1^{\mathbb{N}} = e_2^{\mathbb{N}}$ and $t \xrightarrow{*} 0$;
3. $t \Vdash A \wedge B$ if $t \xrightarrow{*} (t_1, t_2)$ such that $t_1 \Vdash A$ and $t_2 \Vdash B$;
4. $t \Vdash A \vee B$ if $t \xrightarrow{*} \iota_1(u)$ and $u \Vdash A$, or if $t \xrightarrow{*} \iota_2(u)$ and $u \Vdash B$;
5. $t \Vdash A \to B$ if for any $u \Vdash A$, $tu \Vdash B$;
6. $t \Vdash \neg A$ if $t \Vdash A \to \bot$;
7. $t \Vdash \forall x.A$ if for any n, $t\,\bar{n} \Vdash A(n)$;
8. $t \Vdash \exists x.A$ if $t \xrightarrow{*} (\bar{n}, u)$ and $u \Vdash A(n)$.

where $e^{\mathbb{N}}$ is the valuation of the first-order expression e in the standard model $\mathbb{N}$ (see Section 1.2.4).

The main observation is that this definition is purely computational, as opposed to the syntactic definition of typing. In fact, it is a strict generalization of typing in the sense that it can be shown that a term of type A is a realizer of A: this is the property of *adequacy*. One of the consequence of the computational definition is that the relation $t \Vdash A$ is undecidable: given a term t and a formula A, there is no algorithm deciding whether t is a realizer of A. This is again to be opposed with the typing relation.

If this interpretation has shown to be fruitful over the years[5], it is intrinsically bound to intuitionistic logic and incompatible with an extension to classical logic. Indeed, Kleene's realizability takes position against the excluded-middle, as shown by the following proposition:

Proposition 3.1. *There exists a formula H such that the negation of $\forall x(H(x) \vee \neg H(x))$ is realized.*

Proof. Consider the primitive recursive function $h : \mathbb{N}^2 \to N$ defined by:

$$h(n,k) = \begin{cases} 1 & \text{if the } n^{\text{th}} \text{ Turing machine stops after } k \text{ steps} \\ 0 & \text{otherwise} \end{cases}$$

and define the formula $H(x) \triangleq \exists y.(h(x,y) = 1)$, also called *halting predicate*. Assume now that there is a term t realizing the formula $\forall x.(H(x) \vee \neg H(x))$ and define $u \triangleq \lambda n.\texttt{match}\ t\ n\ \texttt{with}\ [x \mapsto 1 \mid y \mapsto 0]$. Then, for any $n \in \mathbb{N}$:

[4] You can think of the syntax and reduction rules of the (untyped) $\lambda^{\times+}$-calculus (Section 2.4.1) extended with terms $0, S, \texttt{rec}$ standing for zero, the successor and a recursion operator. The `rec` operator can be defined in various way, the point being that it allows to perform recursion over natural numbers. For instance, it could be given the following reduction rules :

$$\begin{array}{rcl} \texttt{rec}\ 0\ t_0\ t_S & \to & t_0 \\ \texttt{rec}\ (S\,u)\ t_0\ t_S & \to & t_S\ u\ (\texttt{rec}\ u\ t_0\ t_S) \end{array}$$

Formally, this can also be seen as a fragment of PCF [137].

[5] See for instance Van Oosten's historical essay [159] on this topic.

1. either $t\bar{n} \xrightarrow{*} \iota_1(t')$ in which case $u\,\bar{n} \xrightarrow{*} 1$ and $H(n)$ is realized (by t'), *i.e.* the n^{th} Turing machine halts,
2. either $t\bar{n} \xrightarrow{*} \iota_2(t')$ in which case $u\,\bar{n} \xrightarrow{*} 0$ and $\neg H(n)$ is realized (by t'), *i.e.* the n^{th} Turing machine does not halt.

Thus u decides the halting problem, which is absurd. As a consequence, there is no such t, and in particular, any term realizes the formula $\neg(\forall x(H(x) \vee \neg H(x)))$. □

3.1.2 Classical realizability

To address the incompatibility of Kleene's realizability with classical reasoning, Krivine introduced in the middle of the 90s the theory of *classical realizability* [97], which is a complete reformulation[6] of the very principles of realizability to make them compatible with classical reasoning. 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 [93] or the calculus of constructions with universes [117].

This theory has shown in the past twenty years to be a very powerful framework, both as a tool to analyze programs and as a way to build new models of set theory. We shall now present briefly these aspects before introducing formally Krivine classical realizability.

3.1.2.1 A powerful tool to reason on programs

Krivine realizability, in what concerns the analysis of programs, can be understood as a relaxation of the Curry-Howard isomorphism. As a proof-as-program correspondence, it is indeed more flexible in that it includes programs that are correct with respect to the execution, but that are not typable. In other words, given a formula A and a problem t, when the Curry-Howard isomorphism somewhat said that t is a *proof* of A if its *syntax* matches the structure of A; Krivine realizability rather has for slogan:

if t *computes* correctly, then it is a *realizer*.

For instance, the following dummy program:

```
program dummy(n):
  if n=n+1 then { return 'Hello'} else { return 27 }
```

can not be simply typed with $\mathsf{Nat} \to \mathsf{Nat}$ while this program has the computational behavior that is expected from this type: when applied to a natural number, it always returns the natural number 27.

If this example is easy to understand, it is quite arbitrary and does not bring any interesting perspective. Yet they are more interesting cases, for instance the term of Maurey $M_{a,b}$. This term, defined by:

$$M_{a,b} \triangleq \lambda nm.n\,F\,(\lambda x.a)\,(m\,F\,(\lambda x.b))$$

where $F \triangleq \lambda fg.gf$ and a, b are free variables, decides which of two natural numbers is the smaller. Indeed, when applied to the Church numerals $\overline{n}$ and $\overline{m}$, $M_{a,b}\,\overline{n}\,\overline{m}$ reduces[7] to a if $n \leq m$ and to b if $\overline{m} < \overline{n}$. In particular, if $\mathtt{tt}$ and $\mathtt{ff}$ are the Boolean term for *true* and *false*, $M_{\mathtt{tt},\mathtt{ff}}$ reduces to $\mathtt{tt}$ if $n \leq m$ and to $\mathtt{ff}$ otherwise. Following our realizability motto, since the term $M_{\mathtt{tt},\mathtt{ff}}$ computes the formula *"n is lower than m"*, *a fortiori* it should realize it [8]. However, as shown by Krivine [91], it can not be typed

[6] As observed in several articles [129, 118], classical realizability can in fact be seen as a reformulation of Kleene's realizability through Friedman's A-translation [53].

[7] We recall that the Church numeral $\overline{n}$ is defined by $\lambda fx.f^n x$: $\overline{0} = \lambda fx.x$, $\overline{1} = \lambda fx.fx$, $\overline{2} = \lambda fx.f(fx)$, etc... The verification of the statement is a pleasant exercise of λ-calculus.

[8] This claim can be formalized with a clever definition of the realized formula, and is a nice (but tricky) exercise of realizability.

in Peano second-order arithmetic (or System F), which is the language of Krivine realizability. This illustrates perfectly the fact that realizability includes strictly more programs (and not only dummy ones) than just typed programs.

As we will see in the next sections, the definition of Krivine realizability interpretation of formulas is again purely computational, and thus the relation of $t \Vdash A$ is also undecidable. Worse, the computational analysis of programs is harder than in the intuitionistic case because of the `call/cc` operator which enables programs to backtrack. Even though, Krivine realizability has shown to be a powerful tool to prove properties on the computational behavior of programs. In particular, the adequacy of the interpretation (with respect to typing rules) gives for free the normalization of typed terms. Besides, the computational content of a realizer can be specified by means of a game-theoretic interpretation, but we will come back to this in Section 3.5.2.

3.1.2.2 Terms as semantics

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 determined 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 of type A realizes the formula A, and thus has the computational behavior expected from the formula A.

However 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 by a stack in $\|A\|$. Another difference with the intuitionistic setting resides in the classical notion of a realizer whose definition is parameterized by a *pole*, which represents a particular sets of challenges and that we shall define and discuss in Section 3.4.1.1.

We shall discuss the underlying game-theoretic intuition more in depth at the end of this chapter (Section 3.5.2.2), and say a word about some surprisingly new model-theoretic perspectives brought by this semantics (Section 3.5.3).

3.1.2.3 Modular implementation of logic

As we advocated in the previous chapter (Section 2.4.3), the proofs-as-programs interpretation of logic suggests that any logical extension should be made through an extension of the programming language. Krivine classical realizability precisely follows this slogan, since classical logic is obtained through the λ_c-calculus which is an extension of the λ-calculus with the `call/cc` operator. Much more than that, as we shall explain in Section 3.2.3, the λ_c-calculus is modular in essence and really turns the motto:

"With side-effects come new reasoning principles."

into a general recipe: to extend the logic with an axiom A, one should add an extra instruction with the adequate reduction rules, and give it the type A. If the computational behavior is indeed correct with respect to A, then the typing rules will automatically be adequate with respect to the realizability interpretation. This is for instance the methodology followed by Krivine to obtain a realizer of the axiom of dependent choice with the `quote` instruction, [94].

3.2 The λ_c-calculus

3.2.1 Terms and stacks

The λ_c-calculus distinguishes two kinds of syntactic expressions: *terms*, which represent programs, and *stacks*, which represent evaluation contexts. Formally, terms and stacks of the λ_c-calculus are defined from three auxiliary sets of symbols, that are pairwise disjoint:

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

The syntax of terms, stacks and processes is given by the following grammar:

Terms	t, u	::=	$x \mid \lambda x.t \mid tu \mid \mathbf{k}_\pi \mid \kappa$	$x, \in \mathcal{V}_\lambda, \kappa \in C$
Stacks	π	::=	$\alpha \mid t \cdot \pi$	($\alpha \in \mathcal{B}$, t closed)
Processes	p, q	::=	$t \star \pi$	(t closed)

As usual, terms and stacks are considered up to α-conversion and we denote by $t[u/x]$ 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 3.2 (Proof-like terms)**.** – We say that a λ_c-term t is *proof-like* if t contains no continuation constant $\mathbf{k}_\pi$. We denote by PL the set of all proof-like terms. ⌟

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

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

where $\overline{0} \equiv \lambda xf\,.\,x$ and $\overline{s} \equiv \lambda nxf\,.\,f(nxf)$ are Church's encodings of zero and the successor function in the pure λ-calculus. Note that this encoding slightly differs from the traditional encoding of numerals in the λ-calculus, although the term $\overline{n} \equiv \overline{s}^n\overline{0}$ is clearly β-convertible to Church's encoding $\lambda xf\,.\,f^n x$—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 will now present.

3.2.2 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). Before turning into a central piece of classical realizability, this abstract machine was a very standard tool to implement (call-by-name) λ-calculus [96]. 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 3.3 (Relation of evaluation)**.** We call a relation of *one step evaluation* any binary relation $\succ_1$ over the set $\Lambda \star \Pi$ of processes that fulfills the following four axioms:

(Push)	$tu \star \pi$	$\succ_1$	$t \star u \cdot \pi$
(Grab)	$(\lambda x\,.\,t) \star u \cdot \pi$	$\succ_1$	$t[u/x] \star \pi$
(Save)	$\mathrm{cc} \star t \cdot \pi$	$\succ_1$	$t \star \mathbf{k}_\pi \cdot \pi$
(Restore)	$\mathbf{k}_\pi \star t \cdot \pi'$	$\succ_1$	$t \star \pi$

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

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

3.2.3 Adding new instructions

The main interest of keeping open the definition of the sets $\mathcal{B}$, $\mathcal{C}$ and of the relation evaluation $\succ_1$ (by axiomatizing rather than defining them) is that it makes possible to enrich the calculus with extra instructions and evaluation rules, simply by putting additional axioms about $\mathcal{C}$, $\mathcal{B}$ and $\succ_1$. On the other hand, the definitions of classical realizability [97] as well as its main properties do not depend on the particular choice of $\mathcal{B}$, $\mathcal{C}$ and $\succ_1$, although the fine structure of the corresponding realizability models is of course affected by the presence of additional instructions and evaluation rules. Standard examples of extra instructions in the set $\mathcal{C}$ are:

1. The instruction quote, which comes with the evaluation rule

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

   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 $n \mapsto \overline{n}$ described in Section 3.2.1) to the term t. This instruction was originally introduced to realize the axiom of dependent choices [94].

2. The instruction eq, which comes with the evaluation rule

   (Eq) $$\mathtt{eq} \star t_1 \cdot t_2 \cdot u \cdot v \cdot \pi \ \succ_1 \ \begin{cases} u \star \pi & \text{if } t_1 \equiv t_2 \\ v \star \pi & \text{if } t_1 \not\equiv t_2 \end{cases}$$

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

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

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

3. The instruction stop, which comes with no evaluation rule. The only purpose of this instruction is to stop evaluation; the contents of the facing stack is implicitly the result of the computation. This instruction turns out to be very useful for witness extraction procedures [118].

4. The instruction $\pitchfork$ ('fork'), which comes with the two evaluation rules

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

   Intuitively, the instruction $\pitchfork$ 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 3.4 (Deterministic evaluation)**.** We say that the relation of evaluation $\succ_1$ is *deterministic* when the two conditions $p \succ_1 p'$ and $p \succ_1 p''$ imply $p' \equiv p''$ (syntactic identity) for all processes p, p' and p''. Otherwise, $\succ_1$ is said to be *non deterministic*. ⌟

The smallest relation of evaluation, that is defined as the union of the four rules (Push), (Grab), (Save) and (Restore), is clearly deterministic. The property of determinism still holds if we enrich the calculus with an instruction `eq` together with the aforementioned evaluation rules or with the instruction `quote`.

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

3.2.4 The thread of a process and its anatomy

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

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

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

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

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

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

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

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

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

3.3 Classical second-order arithmetic

In Section 3.2 we focused on the *computing facet* of the theory of classical realizability. In this section, we will now present its *logical facet* by introducing the language of classical second-order logic with the corresponding type system. In Section 3.3.3, we will deal with the particular case of *second-order arithmetic* and present its axioms.

3.3.1 The language of second-order logic

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

3.3.1.1 First-order expressions and formulas

First-order expressions are formally defined as in first-order arithmetic (see Example 1.3) from

1. a *first-order signature* Σ which we assume to contain 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 $\mathbb{N}$ (see Section 3.3.3).
2. 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 3.2.

This results in the following formal definition:

$$\textbf{First-order terms} \qquad e_1, e_2 \ ::= \ x \mid f(e_1, \dots, e_k) \qquad (x \in \mathcal{V}_1, f \in \Sigma)$$

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 (see Definitions 1.5 and 1.6).

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:

$$\textbf{Formulas} \qquad A, B \ ::= \ X(e_1, \dots, e_k) \mid A \to B \mid \forall x.A \mid \forall X.A \qquad (X \in \mathcal{V}_2)$$

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.

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 $A[e/x]$ 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.

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

$$\begin{array}{rcl@{\qquad}rcl}
\bot & \triangleq & \forall Z.Z & A \Leftrightarrow B & \triangleq & (A \to B) \wedge (B \to A) \\
\neg A & \triangleq & A \to \bot & \exists x.A(x) & \triangleq & \forall Z.(\forall x.(A(x) \to Z) \to Z) \\
A \wedge B & \triangleq & \forall Z.((A \to B \to Z) \to Z) & \exists X.A(X) & \triangleq & \forall Z.(\forall X.(A(X) \to Z) \to Z) \\
A \vee B & \triangleq & \forall Z.((A \to Z) \to (B \to Z) \to Z) & e_1 = e_2 & \triangleq & \forall W.(W(e_1) \to W(e_2))
\end{array}$$

3.3.1.2 Predicates and second-order substitution

We call a *predicate of arity k* any expression which associates to the variable $x_1, \dots, x_k$ a formula C depending on these variables. More formally, we could (ab)use the λ-notation to define them as expressions of the form $P \equiv \lambda x_1 \cdots x_k \,.\, C$ where C is then an arbitrary formula. 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

$$P(e_1, \dots, e_k) \ \equiv \ (\lambda x_1 \cdots x_k \,.\, C)(e_1, \dots, e_k) \ \equiv \ C[e_1/x_1, \dots, e_k/x_k]$$

$$\frac{(x:A)\in\Gamma}{\Gamma\vdash x:A}\ (\text{Ax}) \qquad \frac{\Gamma,x:A\vdash t:B}{\Gamma\vdash \lambda x\,.\,t:A\to B}\ (\to_I) \qquad \frac{\Gamma\vdash t:A\to B \qquad \Gamma\vdash t:A}{\Gamma\vdash tu:B}\ (\to_E)$$

$$\frac{\Gamma\vdash t:A \quad x\notin FV(\Gamma)}{\Gamma\vdash t:\forall x.A}\ (\forall^1_I) \qquad \frac{\Gamma\vdash t:\forall x.A}{\Gamma\vdash t:A\{x:=e\}}\ (\forall^1_E) \qquad \frac{\Gamma\vdash t:A \quad X\notin FV(\Gamma)}{\Gamma\vdash t:\forall X.A}\ (\forall^2_I)$$

$$\frac{\Gamma\vdash t:\forall X.A}{\Gamma\vdash t:A\{X:=P\}}\ (\forall^2_E) \qquad \frac{}{\Gamma\vdash \mathrm{cc}:((A\to B)\to A)\to A}\ (\mathrm{cc})$$

Figure 3.1: Typing rules of second-order logic

(by analogy with β-reduction). 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* $A[P/X]$ as follows:

$$\begin{array}{rcll} X(e_1,\ldots,e_k)[P/X] & \triangleq & P(e_1,\ldots,e_k) & \\ Y(e_1,\ldots,e_m)[P/X] & \triangleq & Y(e_1,\ldots,e_m) & (Y\neq X)\\ (A\to B)[P/X] & \triangleq & A[P/X]\to B[P/X] & \\ (\forall x.A)[P/X] & \triangleq & \forall x.A[P/X] & (x\notin FV(P))\\ (\forall X.A)[P/X] & \triangleq & \forall X.A & \\ (\forall Y.A)[P/X] & \triangleq & \forall Y.A[P/X] & (Y\neq X,\ Y\notin FV(P)) \end{array}$$

3.3.2 A type system for classical second-order logic

We shall now present the deduction system of classical second-order logic as a type system based on typing judgments of the form $\Gamma\vdash t:A$, where t is a proof-like term, i.e. a λ_c-term containing no continuation constant $\mathbf{k}_\pi$; and A is a formula of second-order logic.

The type system of classical second-order logic is defined from the typing rules of Figure 3.1. These typing rules are the usual typing rules of AF2 [92], plus a specific typing rule for the instruction cc which permits to recover the full strength of classical logic.

Using the encodings of second-order logic, we can derive from the typing rules of Figure 3.1 the usual introduction and elimination rules of absurdity, conjunction, disjunction, (first- and second-order) existential quantification and Leibniz equality [92]. As explained in Section 1.1.2.1, 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.

3.3.3 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 first-order arithmetic (Section 1.2.4) as a function $[\![f]\!]:\mathbb{N}^k\to\mathbb{N}$, so that we can give a denotation $[\![e]\!]\in\mathbb{N}$ to every closed first-order expression e. Moreover, we assume that each function symbol associated to a primitive recursive definition (cf Section 3.3.1.1) is given its standard interpretation in $\mathbb{N}$. In this way, every numeral $n\in\mathbb{N}$ is represented in the world of first-order expressions as the closed expression $s^n(0)$ that we still write n, since $[\![s^n(0)]\!]=n$.

3.3.3.1 Induction

Following Dedekind's construction of natural numbers, we consider the predicate $\mathrm{Nat}(x)$ [60, 92] defined by

$$\mathrm{Nat}(x)\ \triangleq\ \forall Z.(Z(0)\to\forall y.(Z(y)\to Z(s(y)))\to Z(x))\,,$$

that defines the smallest class of individuals containing zero and closed under the successor function. One of the main properties of the logical system presented above is that the axiom of induction, that we can write $\forall x.\mathrm{Nat}(x)$, is not derivable from the rules of Figure 3.1. As proved by Krivine [97, Theorem 12], this axiom is not even (universally) realizable in general. To recover the strength of arithmetic reasoning, we need to relativize all first-order quantifications to the class $\mathrm{Nat}(x)$ of Dedekind numerals using the shorthands for *numeric quantifications*

$$\begin{array}{rcl} \forall^{\mathrm{nat}} x.A(x) & \triangleq & \forall x.(\mathrm{Nat}(x) \to A(x)) \\ \exists^{\mathrm{nat}} x.A(x) & \triangleq & \forall Z.(\forall x.(\mathrm{Nat}(x) \to A(x) \to Z) \to Z) \end{array}$$

so that the *relativized induction axiom* becomes provable in second-order logic [92]:

$$\forall Z.(Z(0) \to \forall^{\mathrm{nat}} x.(Z(x) \to Z(s(x))) \to \forall^{\mathrm{nat}} x.Z(x))\,.$$

3.3.3.2 The axioms of PA2

Formally, a formula A is a *theorem* of second-order arithmetic (PA2) if it can be derived from Peano axioms (see Example 1.12), expressing that the successor function is injective and not surjective:

$$\text{(PA5)}\quad \forall x.\forall y.(s(x) = s(y) \to x = y) \qquad\qquad \text{(PA6)}\quad \forall x.(s(x) \neq 0)$$

and from the definitional equalities attached to the (primitive recursive) function symbols of the signature:

$$\begin{array}{llll} \text{(PA1)} & \forall x.(0 + x = x) & \text{(PA2)} & \forall x.\forall y.(s(x) + y = s(x + y)) \\ \text{(PA3)} & \forall x.(0 \times x = 0) & \text{(PA4)} & \forall x.\forall y.(s(x) \times y = (x \times y) + y) \end{array}$$

etc... Unlike the non relativized induction axiom—that requires a special treatment in PA2—we shall see in Section 3.4.6 that all these axioms are realized by simple proof-like terms.

Observe that we consider here an unusual definition of (PA2), since the usual one includes the induction rule as an axiom. Nonetheless, the two theories are related through the relativization of first-order quantifications. Namely, if A is a theorem of (PA2) with induction, then the relativized formula A^{Nat} is a theorem of $(PA2)$ without induction.

3.4 Classical realizability semantics

3.4.1 Generalities

Given a particular instance of the λ_c-calculus (defined from particular sets $\mathcal{B}, C$ and from a particular relation of evaluation $\succ_1$ as described in Section 3.2), 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.

3.4.1.1 Poles, truth values and falsity values

Formally, the construction of the realizability model is parameterized by a *pole* $\perp\!\!\!\perp$ in the sense of the following definition:

Definition 3.5 (Poles)**.** A *pole* is any set of processes $\perp\!\!\!\perp \subseteq \Lambda \star \Pi$ which is closed under anti-evaluation, in the sense that both conditions $p \succ p'$ and $p' \in \perp\!\!\!\perp$ together imply that $p \in \perp\!\!\!\perp$ for all processes $p, p' \in \Lambda \star \Pi$. ⌟

Given a fixed set of processes, the following two examples are standard methods to define a pole. The first one is straightforward in that it simply consists in taking the closure by anti-evaluation. The second one might be more disconcerting, and consists in taking the set of processes which are unreachable by reduction.

Example 3.6 (Goal-oriented pole)**.** Given a set of processes P, the set of all processes that reach an element of P after zero, one or several evaluation steps, that is:

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

is a valid pole. Indeed, if p,p' are processes such that $p \succ p'$ and $p' \in \perp\!\!\!\perp$, by definition there is a process $p_0 \in P$ such that $p' \succ p_0$. Thus $p \succ p' \succ p_0$ and $p \in \perp\!\!\!\perp$, which concludes the proof that $\perp\!\!\!\perp$ is closed by anti-reduction. By definition, the set $\perp\!\!\!\perp$ is the smallest pole that contains the set of processes P as a subset. ⌟

Example 3.7 (Thread-oriented pole)**.** Given a set of processes P, the complement set of the union of all threads starting from an element of P, that is:

$$\perp\!\!\!\perp \;\triangleq\; \left(\bigcup_{p\in P} \mathbf{th}(p)\right)^c \;\equiv\; \bigcap_{p\in P} \big(\mathbf{th}(p)\big)^c$$

is a valid pole. It is indeed quite easy to check that $\perp\!\!\!\perp$ is closed by anti-reduction. Consider two processes p,p' such that $p \succ p'$ and $p' \in P$, and assume that there is a process $p_0 \in P$ such that $p_0 \succ p$. Then $p_0 \succ p'$ which contradicts the fact that $p' \in \perp\!\!\!\perp$. Thus there is no such process p_0 and $p \in \perp\!\!\!\perp$. This pole is also the largest one that does not intersect P. ⌟

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

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

Intuitively, every falsity value $S \subseteq \Pi$ represents a particular set of *tests*, while the corresponding truth value $S^{\perp\!\!\!\perp}$ 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* or the *referee*). From the definition of $S^{\perp\!\!\!\perp}$, it is clear that the larger the falsity value S, the smaller the corresponding truth value $S^{\perp\!\!\!\perp}$, and vice-versa.

3.4.1.2 Formulas with parameters

In order to interpret second-order variables that occur in a given formula A, it is convenient to enrich the language of PA2 with a new predicate symbol $\dot{F}$ of arity k for every *falsity value function* F of arity k, that is, for every function $F : \mathbb{N}^k \to \mathcal{P}(\Pi)$ 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, this corresponds to the formulas defined by:

$$A,B \;::=\; X(e_1,\dots,e_k) \mid A \to B \mid \forall x.A \mid \forall X.A \mid \dot{F}(e_1,\dots,e_k) \qquad\qquad X \in \mathcal{V}_2, F \in \mathcal{P}(\Pi)^{\mathbb{N}^k}$$

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

3.4.2 Definition of the interpretation function

The interpretation of the closed formulas with parameters follows the intuition that the falsity value $\|A\|$ of a formula A contains tests that terms have to challenge to be in the corresponding truth value $|A|$. In particular, a test for $A \to B$ consists in a defender of A together with a test for B, while a test for a quantified formula $\forall x.A$ (resp. $\forall X.A$) is simply a test for one of the possible instantiations for the variable x (resp. X).

Definition 3.8 (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

$$\begin{array}{rcl} \|\dot{F}(e_1,\ldots,e_k)\| & \triangleq & F(\llbracket e_1 \rrbracket,\ldots,\llbracket e_k \rrbracket) \\ \|A \to B\| & \triangleq & |A| \cdot \|B\| \quad = \quad \big\{ t \cdot \pi \ : \ t \in |A|,\ \pi \in \|B\| \big\} \\ \|\forall x.A\| & \triangleq & \bigcup_{n \in \mathbb{N}} \|A[n/x]\| \\ \|\forall X.A\| & \triangleq & \bigcup_{F:\mathbb{N}^k \to \mathcal{P}(\Pi)} \|A[\dot{F}/X]\| \qquad \text{(if } X \text{ has arity } k\text{)} \end{array}$$

whereas its truth value $|A| \subseteq \Lambda$ is defined by $|A| = \|A\|^{\perp\!\!\!\perp}$. Finally, defining $\top \equiv \dot{\emptyset}$ (recall that we have $\bot \equiv \forall X.X$), one can check that we have :

$$\|\top\| = \emptyset \qquad\qquad |\top| = \Lambda \qquad\qquad \|\bot\| = \Pi$$

⌟

Since the falsity value $\|A\|$ (resp. the truth value $|A|$) of A actually depends on the pole $\perp\!\!\!\perp$, we shall write it sometimes $\|A\|_{\perp\!\!\!\perp}$ (resp. $|A|_{\perp\!\!\!\perp}$) to recall the dependency.

Definition 3.9 (Realizers)**.** Given a closed formula A with parameters and a closed term $t \in \Lambda$, we say that:

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

⌟

From these definitions, we clearly have

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

On the other hand, the truth value $|A \to B|$ of an implication $A \to B$ slightly differs from its traditional interpretation in Kleene's realizability (Section Section 3.1.1). Writing

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

we can check that:

Lemma 3.10. *For all closed formulas A and B with parameters:*

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

Proof. These simple statements are a nice pretext to a first manipulation of the definitions.

1. Let $t \in |A \to B|$ and $u \in |A|$. To prove that $tu \in |B|$, we consider an arbitrary stack $\pi \in \|B\|$. By applying the rule (PUSH) we get $tu \star \pi \succ_1 t \star u \cdot \pi$. Since $t \in |A \to B|$ and $u \cdot \pi \in \|A \to B\|$, the process $t \star u \cdot \pi$ belongs to $\perp\!\!\!\perp$. Hence $tu \star \pi \in \perp\!\!\!\perp$ by anti-evaluation.

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

□

Besides, it is easy to prove that cc is indeed a universal realizer of Peirce's law:

Lemma 3.11 (Law of Peirce)**.** *Let A and B be two closed formulas with parameters:*

1. *If $\pi \in \|A\|$, then $\mathbf{k}_\pi \Vdash A \to B$.*
2. *$\mathrm{cc} \Vdash ((A \to B) \to A) \to A$.*

Proof. 1. Let $\pi \in \|A\|$. To prove that $\mathbf{k}_\pi \in |A \to B|$, we need to check that $\mathbf{k}_\pi \star t \cdot \pi' \in \bot\!\!\!\bot$ for all $t \in |A|$ and $\pi' \in \|B\|$. By applying the rule (Restore) we get $\mathbf{k}_\pi \star t \cdot \pi' \succ_1 t \star \pi \in \bot\!\!\!\bot$ (since $t \in |A|$ and $\pi \in \|A\|$), hence $\mathbf{k}_\pi \star t \cdot \pi' \in \bot\!\!\!\bot$ by anti-evaluation.

2. To prove that $\mathrm{cc} \Vdash ((A \to B) \to A) \to A$ (for any pole $\bot\!\!\!\bot$), we need to check that $\mathrm{cc} \star t \cdot \pi \in \bot\!\!\!\bot$ for all $t \in |(A \to B) \to A|$ and $\pi \in \|A\|$. By applying the rule (Save) we get $\mathrm{cc} \star t \cdot \pi \succ_1 t \star \mathbf{k}_\pi \cdot \pi$. But since $\mathbf{k}_\pi \in |A \to B|$ (from (1)) and $\pi \in \|A\|$, we have $\mathbf{k}_\pi \cdot \pi \in \|(A \to B) \to A\|$, so that $t \star \mathbf{k}_\pi \cdot \pi \in \bot\!\!\!\bot$. Hence $\mathrm{cc} \star t \cdot \pi \in \bot\!\!\!\bot$ by anti-evaluation.

□

3.4.3 Valuations and substitutions

In order to express the soundness invariants relating the type system of Section 3.3.3 with the classical realizability semantics defined above, we need to introduce some more terminology.

Definition 3.12 (Valuations)**.** A *valuation* is a function ρ that associates a natural number $\rho(x) \in \mathbb{N}$ to every first-order variable x and a falsity value function $\rho(X) : \mathbb{N}^k \to \mathcal{P}(\Pi)$ to every second-order variable X of arity k.

1. 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:

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

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

$$(\rho, X \leftarrow F) \triangleq \rho_{|\,\mathrm{dom}(\rho)\setminus\{X\}} \cup \{X \leftarrow F\}\,.$$

⌟

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

$$A[\rho] \triangleq A[\rho(x_1)/x_1, \ldots, \rho(x_n)/x_n, \dot\rho(X_1)/X_1, \ldots, \dot\rho(X_m)/X_m]$$

where $x_1, \ldots, x_n, X_1, \ldots, X_m$ 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

$$(x_1 : A_1, \ldots, x_n : A_n)[\rho] \triangleq x_1 : A_1[\rho], \ldots, x_n : A_n[\rho]\,.$$

Definition 3.13 (Substitutions)**.** A *substitution* is a finite function σ from λ-variables to closed λ_c-terms. Given a substitution σ, a λ-variable x and a closed λ_c-term u, we denote by $\sigma, x := u$ the substitution defined by $(\sigma, x := u) \equiv \sigma_{|\,\mathrm{dom}(\sigma)\setminus\{x\}} \cup \{x := u\}$. ⌟

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

$$t[\sigma] \triangleq t[\sigma(x_1)/x_1, \ldots, \sigma(x_n)/x_n]$$

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

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

3.4.4 Adequacy

The adequacy of typing judgments and typing rules with respect to a pole is defined exactly like the adequacy with respect to a model (Definition 1.17). Given a fixed pole $\perp\!\!\!\perp$, we say that:

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

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

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

Proposition 3.14 (Adequacy)**.** *The typing rules of Figure 3.1 are adequate w.r.t. any pole $\perp\!\!\!\perp$, as well as all the judgments $\Gamma \vdash t : A$ that are derivable from these rules.*

Proof. The rule for cc directly stems from Lemma 3.11, while introduction and elimination rules for universal quantifiers results from the definition of the corresponding falsity values. We will only sketch the proof for the introduction and elimination rules of implication.

• **Case** $(\to_I)$. Assume that $\Gamma \vdash t : A \to B$ and $\Gamma \vdash u : B$ are adequate w.r.t. $\perp\!\!\!\perp$, and pick a valuation ρ and a substitution σ such that $\sigma \Vdash \Gamma[\rho]$. We want to show that $(tu)[\sigma] \Vdash B[\rho]$. It suffices to show that if $\pi \in \|B[\rho]\|$, then $(tu)[\sigma] \star \pi \in \perp\!\!\!\perp$. Applying the (Push) rule, we get :

$$(tu)[\sigma] \star \pi \succ t[\sigma] \star u[\sigma] \cdot \pi$$

By hypothesis, we have $u[\sigma] \Vdash A[\rho]$ (and then $u[\sigma] \cdot \pi \in \|(A \to B)[\rho]\|$), and $t[\sigma] \Vdash (A \to B)[\rho]$, so that $t[\sigma] \star u[\sigma] \cdot \pi$ belongs to $\perp\!\!\!\perp$. We conclude by anti-reduction.

• **Case** $(\to_E)$. Assume that $\Gamma, x : A \vdash t : B$ is adequate w.r.t $\perp\!\!\!\perp$. This means that for any valuation ρ, any $u \Vdash A[\rho]$ and any $\sigma \Vdash \Gamma[\rho]$, denoting by σ' the substitution $\sigma, x := u$, we have $t[\sigma'] \Vdash B[\rho]$. Let us pick a valuation ρ and a substitution σ such that $\sigma \Vdash \Gamma[\rho]$. We want to show that $(\lambda x.t)[\sigma] \Vdash (A \to B)[\rho]$. Let $u \cdot \pi$ be a stack in $\|(A \to B)[\rho]\|$. Applying the (Grab) rule, we have :

$$(\lambda x.t)[\sigma] \star u \cdot \pi \succ t[\sigma, x := u] \star \pi$$

By hypothesis, we have $u \Vdash A[\rho]$, and so $t[\sigma, x := u] \Vdash B[\rho]$. Thus $t[\sigma, x := u] \star \pi$ belongs to $\perp\!\!\!\perp$. and we conclude by anti-reduction. □

Since the typing rules of Figure 3.1 involve no continuation constant, every realizer that comes from a proof of second order logic by Proposition 3.14 is thus a proof-like term.

3.4.5 The induced model

It is not innocent if the sets $|A|$ introduced in the previous sections were called *truth values*. Indeed, this construction defined a model for second-order logic where truth values are made of λ_c-terms. In a nutshell, starting from the standard model $\mathbb{N}$ for first-order expressions and an instance of the λ_c-calculus (that is with `call/cc` only or other extras instructions), the choice of a particular pole $\bot\!\!\!\bot$ defines a truth value for all formulas of the language. Naively, we could be tempted to define the valid formulas as the one whose truth value is not empty. Yet, this raises a problem of consistency:

Proposition 3.15. *If $\bot\!\!\!\bot \neq \emptyset$, then there is a term t such that for all formula A, $t \in |A|$.*

Proof. Assume that the $\bot\!\!\!\bot$ is not empty, and let $\langle t \| \pi \rangle$ be a process in $\bot\!\!\!\bot$. Then for any formula A, $\mathbf{k}_\pi t \Vdash A$. Indeed, for any stack ρ (and in particular any stack in $\|A\|$), we have:

$$\mathbf{k}_\pi t \star \rho \succ \mathbf{k}_\pi \star t \cdot \rho \succ t \star \pi$$

The last process being in the pole, they all are by anti-evaluation, and thus $\mathbf{k}_\pi t \star \rho \in \bot\!\!\!\bot$. □

If we examine $k_\pi\ t$, the guilty term in the previous proof, there is two observations to do. First, it is worth noting that independently of t and π, this term can not be typed since there is no typing rule for continuations k_π. Second, sticking with the intuition that a realizer is a term that can challenge successfully any tests in the falsity value, this term is morally a cheater: in front of a test ρ, it actually refuses to challenge it, drops it and goes directly to the test π for which it already knows a winning defender t. Therefore, the problem comes from the presence of a continuation constant, and we should restrict truth values to terms without continuation constants, *i.e.* to proof-like terms.

To ease the next definition[9], we restrict ourselves to the *full standard model* of PA2. In this model, first-order individuals are interpreted by the elements of $\mathbb{N}$, while second-order objects of arity k are interpreted in the sets of k-ary relations on the set $\mathbb{N}$. We denote this model by $\mathcal{M}$.

Definition 3.16 (Realizability model)**.** Given the full standard model $\mathcal{M}$ of PA2 and a pole $\bot\!\!\!\bot$, we call realizability model and denote by $\mathcal{M}_{\bot\!\!\!\bot}$ the model in which the validity of formulas is defined by:

$$\mathcal{M}_{\bot\!\!\!\bot} \Vdash A \qquad \text{if and only if} \qquad |A| \cap PL \neq \emptyset$$

⌟

The previous definition gives a simple criterion of consistency for realizability models:

Proposition 3.17 (Consistency)**.** *The model $\mathcal{M}_{\bot\!\!\!\bot}$ induce by the pole $\bot\!\!\!\bot$ is consistent if and only if for each proof-like term t, there exists one stack π such that $t \star \bot\!\!\!\bot \notin \bot\!\!\!\bot$.*

Proof. Recall that $\|\bot\| = \Pi$. Hence $\mathcal{M}_{\bot\!\!\!\bot} \Vdash \bot$ if and only if there exists a proof-like term t such that $t \Vdash \bot$, *i.e.* for any stack π, $t \star \pi \in \bot\!\!\!\bot$. Thus $\mathcal{M}_{\bot\!\!\!\bot} \nVdash \bot$ if and only if for each proof-like term t there is at least one stack π such that $t \star \pi \notin \bot\!\!\!\bot$. □

3.4.6 Realizing the axioms of PA2

Let us recall that in PA2, Leibniz equality $e_1 = e_2$ is defined by $e_1 = e_2 \equiv \forall Z\,(Z(e_1) \to Z(e_2))$.

Proposition 3.18 (Realizing Peano axioms)**.** *:*

1. $\lambda z\,.\,z \Vdash \forall x\,\forall y\,(s(x) = s(y) \to x = y)$

[9] The definition of realizability models could be reformulated to consider a ground model of PA2 as parameter, but this would require a formal definition of the models of PA2. This would have been unnecessarily complex for the sole purpose of perceiving the spirit of realizability models.

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

Proof. The proof is an easy verification, and can be found in [97]. □

From this we deduce the main theorem, proving that any realizability model is a model of PA2:

Theorem 3.19 (Realizing the theorems of PA2)**.** *If A is a theorem of PA2 (in the sense defined in Section 3.3.3.2), then there is a closed proof-like term t such that $t \Vdash A$.*

Proof. Immediately follows from Prop. 3.14 and 3.18. □

3.4.7 The full standard model of PA2 as a degenerate case

It is easy to see that when the pole $\perp\!\!\!\perp$ is empty, the classical realizability model defined above collapses to the full standard model $\mathcal{M}$ of PA2. For that, we first notice that when $\perp\!\!\!\perp = \varnothing$, the truth value $S^{\perp\!\!\!\perp}$ associated to an arbitrary falsity value $S \subseteq \Pi$ can only take two different values: $S^{\perp\!\!\!\perp} = \Lambda_c$ when $S = \varnothing$, and $S^{\perp\!\!\!\perp} = \varnothing$ when $S \neq \varnothing$. Moreover, we easily check that the realizability interpretation of implication and universal quantification mimics the standard truth value interpretation of the corresponding logical construction in the case where $\perp\!\!\!\perp = \varnothing$. It is easy to check that:

Proposition 3.20. *If $\perp\!\!\!\perp = \varnothing$, then for every closed formula A of PA2 we have*

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

An interesting consequence of the above proposition is the following:

Corollary 3.21. *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 $t \in |A|_\varnothing$. Therefore $|A|_\varnothing = \Lambda$ and $\mathcal{M} \models A$. □

However, the converse implication is false in general, since the formula $\forall x\ \mathrm{Nat}(x)$ (cf Section 3.3.3.1) that expresses the induction principle over individuals is obviously true in $\mathcal{M}$, but it has no universal realizer when evaluation is deterministic [97, Theorem 12].

3.5 Applications

We present in this section some applications of Krivine realizability, both on its logical and computational facets. While we introduce theses applications in the framework of the λ_c-calculus, keep in mind that they are not peculiar to this calculus. As we will see in the next sections, other calculi are suitable for a realizability interpretation *à la* Krivine, and can thus benefit from the results expressed thereafter.

3.5.1 Soundness and normalization

Once the realizability interpretation is defined and the adequacy proved, the soundness of the language is a direct consequence of the adequacy. Indeed, if there was a proof t of $\bot$, then by adequacy t would be a uniform realizer of $\bot$. Thus the existence of one consistent model is enough to contradict this possibility, ensuring the correction of the type system. Similarly, the normalization of the language is also a direct consequence of the adequacy and the following observation:

Proposition 3.22 (Normalizing processes)**.** *The set* $\perp\!\!\!\perp_{\Downarrow} \triangleq \{p \in \Lambda \times \Pi : p \text{ normalizes}\}$ *defines a valid pole.*

Proof. We need to check that $\perp\!\!\!\perp_{\Downarrow}$ is closed by anti-reduction, so let p, p' be two processes such that $p \succ p'$ and $p' \in \perp\!\!\!\perp_{\Downarrow}$. The latter means by definition that p' normalizes. Since $p \succ p'$, necessarily p normalizes too and thus belongs to the pole $\perp\!\!\!\perp_{\Downarrow}$. □

Note that we only consider the normalization with respect to the evaluation strategy of the processes, which corresponds to the weak-head reduction in the sense of the λ-calculus. In particular, this is weaker than the strong and weak normalizations of the λ-calculus (see Section 2.1.5). We will use this observation in Chapters 4 and 6 to prove normalization properties of different calculi.

3.5.2 Specification problem

The specification problem for a formula A can be expressed through the following question:

$$\textit{Which are the terms } t \textit{ such that } t \Vdash A \textit{ ?}$$

In other words, it poses the question of exhibiting a (computational) characterization for the realizers of A. Thanks to the adequacy of the interpretation with respect to typing, such a characterization would also apply to terms of type A.

3.5.2.1 Toy example: $\forall X.X \to X$

In the language of second-order logic, the type of the identity function $I = \lambda x.x$ is described by the formula $\forall X(X \to X)$. A closed term $t \in \Lambda$ is said to be *identity-like* if $t \star u \cdot \pi \succ u \star \pi$ for all $u \in \Lambda$ and $\pi \in \Pi$. Examples of identity-like terms are of course the identity function I but also terms such as II, δI (where $\delta \equiv \lambda x.xx$), $\lambda x.\mathrm{cc}(\lambda k.x)$, $\mathrm{cc}(\lambda k.kI\delta k)$, etc. It is easy to verify that any identity-like term is a universal realizer of the formula $\forall X.X \to X$. But the converse also holds, and thus provides an answer to the specification problem for the formula $\forall X.(X \to X)$.

Proposition 3.23. *For all terms* $t \in \Lambda$*, we have:*

$$t \Vdash \forall X.(X \to X) \qquad \Leftrightarrow \qquad t \textit{ is identity-like}$$

Proof. The interesting direction of the proof is from left to right. We prove it with the so-called *methods of threads* [63]. Assume $t \Vdash \forall X(X \to X)$, and consider $u \in \Lambda, \pi \in \Pi$. We want to prove that $t \star u \cdot \pi \succ u \star \pi$. We define the pole

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

as well as the falsity value $S = \{\pi\}$. From the definition of $\perp\!\!\!\perp$, we know that $t \star u \cdot \pi \notin \perp\!\!\!\perp$. As $t \Vdash \dot{S} \to \dot{S}$ and $\pi \in \|\dot{S}\|$, necessarily $u \nVdash S$. This means that $u \star \pi \notin \perp\!\!\!\perp$, that is $t \star u \cdot \pi \succ u \star \pi$. □

3.5.2.2 Game-theoretic interpretation

In the previous section we gave a toy example of specification that was proved using the method of threads. If this method is very useful, it has the drawbacks of becoming very painful when the formula to specify get more complex. A more scalable way to obtain specifications (which uses the threads method as a technical tool) is to strengthen the intuition of an opposition between two players underlying Krivine realizability. In addition to being a useful specification method, this idea that realizers of a formula are its defenders, turns out to be a helpful intuition when defining the realizability interpretation of a language.

As we only want to give an oversight of the corresponding game-theoretic intuitions, we will illustrate this methodology with an example. Precise definitions, proofs etc... can be found in [63, 64, 65]. We choose as a running example the formula $\Phi_f \triangleq \exists x.\forall y.f(x) \leq f(y)$, where f is any computable function from $\mathbb{N}$ to $\mathbb{N}$, expressing the fact that f admits a minimum. We could have chosen any arithmetical formula (see [65]), or second-order formulas, as Peirce's law (see [63, 64]). We believe this example to be representative enough of the general situation and easier to understand that an example in a second-order setting.

Eloise and Abelard Still writing $\mathcal{M}$ for the full standard model of PA2, the formula Φ_f naturally induces a game between two players $\exists$ and $\forall$, that we name[10] Eloise and Abelard. Both players instantiate the corresponding quantifiers in turns, Eloise for defending the formula and Abelard for attacking it. The game, whose depth is bounded by the number of quantifications, proceeds as follows:

- Eloise has to give an integer $m \in \mathbb{N}$ to instantiate the existential quantifier, and the game goes on over the closed formula $\forall y.f(m) \leq f(y)$.
- Abelard has to give an integer $n \in \mathbb{N}$, and the game goes on the closed formula $f(m) \leq f(n)$.
- Eloise has then two choices: either she backtracks to the first step to give another instantiation m' for x, and the game goes on; or she chooses to interrupt the game. If so, Eloise wins if $\mathcal{M} \models f(m) \leq f(n)$, otherwise Abelard wins. If the game goes on forever, Abelard wins.

Observe that the fact Eloise wins the game on a position (m,n) does not mean that m is a minimum for the function f: it only means that Abelard failed in finding an integer n such that $f(n) < f(m)$. Nonetheless, if Eloise actually knows that some integer m is a minimum for f, she will obviously win the game regardless of what Abelard plays.

We say that a player has a *winning strategy* if (s)he has a way of playing that ensures him/her the victory independently of the opponent moves, which corresponds to the definition of Coquand's game [27]. It is obvious from Tarski's definition of truth (see Section 1.2) that the closed formula Φ_f is valid in the ground model if and only if Eloise has a winning strategy.

Intuitively, Eloise is playing as a realizer should, and Abelard is an opponent choosing amongst falsity values. This intuition can be formalized by implementing the previous game within the λ_c-calculus. A realizer will then corresponds to a winning strategy for Eloise, and reciprocally.

Relativization to canonical integers The implementation of the previous game in the λ_c-calculus actually requires a preliminary step. Indeed, as such first-order quantifications are not given any computational content: integers are instantiated in formulas which are only evaluated in the end within the ground model. To make these integers appear in the computations, we need to relativize first-order quantifications to the class $\mathrm{Nat}(x)$ (just like in Section 3.3.3.1). However, if we have as expected $\bar{n} \Vdash \mathrm{Nat}(n)$ for any $n \in \mathbb{N}$, there are realizers of $\mathrm{Nat}(n)$ different from $\bar{n}$. Intuitively, a term $t \Vdash \mathrm{Nat}(n)$ represents the integer n, but n might be present only as a computation, and not directly as a computed value.

The usual technique to retrieve $\bar{n}$ from such a term consist in the use of a *storage operator* T, which simulates a call-by-value reduction (for integers) on the first argument on the stack. While such a term is easy to define, it make the the definition of the game harder, and we do not want to bother the reader with such technical details[11]. Rather than that, we define a new asymmetrical implication where the left member must be an integer value (somehow forcing call-by-value reduction on all integers), and

[10] The names Eloise and Abelard are due to Thierry Coquand, who also defined the game in question [27].

[11] For further details about the relativization and storage operator, please refer to Section 2.9 and 2.10.1 of Rieg's Ph.D. thesis [144].

the interpretation of this new implication.

Formulas	$A, B ::= \ldots \mid \{e\} \to A$
Falsity value	$\|\|\{e\} \to A\|\| \triangleq \{\bar{n} \cdot \pi : [\![e]\!] = n \wedge \pi \in \|\|A\|\|\}$

We finally define the corresponding shorthands for relativized quantifications:

$$\begin{array}{lcl} \forall^{\mathsf{IN}} x\, A(x) & \triangleq & \forall x\, (\{x\} \to A(x)) \\ \exists^{\mathsf{IN}} x\, A(x) & \triangleq & \forall Z\, (\forall x(\{x\} \to A(x) \to Z) \to Z) \end{array}$$

which is easy to check to be equivalent (in terms of realizability) to the one defined in Section 3.3.3.1 [65].

Realizability game In order to play using realizers, we will slightly change the setting of the previous game, adding processes. One should notice that we only add more information, so that this new game is somewhat a "decorated" version of the previous one.

To describe the match, we use processes which evolve throughout the match according to the following rules:

1. Eloise proposes a term $t_0 \in \mathrm{PL}$ supposed to defend Φ_f and Abelard proposes a stack $u_0 \cdot \pi_0$ supposed to attack the formula Φ. We say that at time 0, the process $p_0 := t_0 \star u_0 \cdot \pi_0$ is the current process.
2. Assume that p_i is the current process. Eloise evaluates p_i in order to reach one of the following situations:
   - $p_i \succ u_0 \star \overline{m} \cdot t \cdot \pi$. If so, Eloise *can* decide to play by communicating her answer (t, m) to Abelard and standing for his answer, and Abelard *must* answer a new integer n together with a new stack $u' \cdot \pi'$. The current process then becomes $p_{i+1} := t \star \overline{n} \cdot u' \cdot \pi'$.
   - $p_i \succ u \star \pi$ for some u, π that were previously played by Abelard in a position in which x, y were instantiated by (m, n). In this case, Eloise wins if $\mathcal{M} \models f(m) \leq f(n)$.

   If none of the above moves is possible, then Abelard wins.

Starting with a term t is a "good move" for Eloise if and only if, proposed as a defender of the formula, t defines an initial winning state (for Eloise), independently from the initial stack proposed by Abelard. In this case, adopting the point of view of Eloise, we just say that t is a *winning strategy* for the formula Φ_f.

This furnishes us an answer to the specification problem for the formula Φ_f: winning strategies of this game exactly characterized the realizer of the formula Φ_f.

Theorem 3.24. *If a closed λ_c-term t is a winning strategy for Eloise if and only if $t \Vdash \Phi_f$.*

Proof. This is a particular case of the more general case of arithmetical formulas proved in [65]. □

3.5.3 Model theory

Up to this point, we only presented applications of Krivine realizability on its computational side. Yet, we explained that realizability offered a way to build models for second-order logic, (this can actually be extended, for instance for set theory [93]). More interestingly, classical realizability appears to be a generalization of Cohen's technique of forcing, introduced to construct a model of set theory in which the continuum hypothesis[12] is not valid. As shown by Krivine [98] and Miquel [120], the forcing

[12] The continuum hypothesis expresses the fact that there is no set whose cardinality would be strictly more than the cardinal of $\mathbb{N}$ and strictly less than the cardinal of $\mathcal{R}$.

construction can be computationally analyzed as a program transformation in the framework of the λ_c-calculus. In particular, classical realizability can simulate any forcing construction[13].

Even more surprising is the fact that the realizability semantics lead to the construction of new models, studied by Krivine in a series of papers [98, 99, 100, 101]. Briefly, the fact that $\forall x.\mathrm{Nat}(x)$ is not realized witnesses that a model has more individuals than the natural numbers. In a well-chosen model[14] $\mathcal{M}_{\perp\!\!\!\perp}$, one can show that $\mathcal{M}_{\perp\!\!\!\perp} \vDash \mathrm{Nat}(n)$ for any $n \in \mathbb{N}$ while $\mathcal{M}_{\perp\!\!\!\perp} \vDash \exists x.\neg\mathrm{Nat}(x)$. Otherwise said, the model attests the presence of unnamed elements. It turns out that this allows to define "pathological" infinite sets[15] $\nabla_n \triangleq \{x : x < n\}$ such that the following statements are valid for any $n, m \in \mathbb{N}$:

1. ∇_2 is not well-ordered
2. there is an injection from ∇_n to ∇_{n+1}
3. there is no surjection from ∇_n to ∇_{n+1}
4. $\nabla_m \times \nabla_n \simeq \nabla_{mn}$

These sets being subsets of $\mathcal{P}(\mathbb{N})$, observe that the first property implies that the axiom of choice (AC) is not valid, while items 2 and 3 prove that the continuum hypothesis (CH) is not valid either [99].

As far as we know, usual techniques to construct model of set theory do not allow to define directly a model in which both (AC) and (CH) are not valid. Besides, a construction by means of forcing can not break the axiom of choice, hence classical realizability is a strict generalization of forcing in this sense. For these reasons amongst others, classical realizability tends to be a promising framework to build new models. In particular, it justifies our quest (Part III) for an algebraic structure as general as possible in which the λ_c-calculus and these constructions can be embedded.

[13] An example of this is the extraction of Herbrand tree by forcing in [143].

[14] In Krivine's papers, it is the model of threads, in which each proof-like term t_n is associated with a stack constant α_n and the pole is defined as $\perp\!\!\!\perp \triangleq \bigcap_{n\in\mathbb{N}}(\mathbf{th}(t_n \star \alpha_n))^c$. This set is indeed a valid pole (see Example 3.7) and is consistent according to Proposition 3.17.

[15] In the ground model or any standard model, ∇_n is just $\{0, 1, \ldots, n-1\}$ *i.e.* n from a set-theoretic point of view.