

EXISTENTIAL WITNESS EXTRACTION IN CLASSICAL REALIZABILITY AND VIA A NEGATIVE TRANSLATION

ALEXANDRE MIQUEL

LIP (UMR 5668 – CNRS – ENS de Lyon – UCBL – INRIA) ENS de Lyon, Université de Lyon,
France
e-mail address: alexandre.miquel@ens-lyon.fr

ABSTRACT. We show how to extract existential witnesses from classical proofs using Krivine’s classical realizability—where classical proofs are interpreted as λ -terms with the call/cc control operator. We first recall the basic framework of classical realizability (in classical second-order arithmetic) and show how to extend it with primitive numerals for faster computations. Then we show how to perform witness extraction in this framework, by discussing several techniques depending on the shape of the existential formula. In particular, we show that in the Σ_1^0 -case, Krivine’s witness extraction method reduces to Friedman’s through a well-suited negative translation to intuitionistic second-order arithmetic. Finally we discuss the advantages of using call/cc rather than a negative translation, especially from the point of view of an implementation.

1. INTRODUCTION

Extracting an existential witness (i.e. an object t such that $A(t)$) from a proof of the formula $\exists x A(x)$ is now a well-understood technique in intuitionistic logic. The simplest way to do it is to normalize the proof and retrieve the witness from the premise of its normal form. Through the Brouwer-Heyting-Kolmogorov interpretation, one can also read the proof as a functional program that reduces to a pair whose first component is the desired witness. Such techniques are implemented in proof-assistants based on intuitionistic systems [29, 20, 27].

Extracting a witness from a classical proof of an existential formula is much more difficult, since classical logic is known not to enjoy the witness property. Such an extraction is actually not always feasible: for instance, we cannot expect to extract a witness from the obvious classical proof of the formula

$$\exists x ((x = 1 \wedge C) \vee (x = 0 \wedge \neg C))$$

in general—think of C being undecidable or, say, Riemann’s conjecture.

1998 ACM Subject Classification: F.4.1.

Key words and phrases: Proof theory, Classical lambda-calculus, Classical realizability, Program extraction.

This paper is an expanded version of [24].

However, several techniques [12, 13, 8, 6, 11] have been proposed in order to extract a witness from a classical proof of an existential formula in some particular cases—typically: when the formula is Σ_1^0 (i.e. of the form $\exists x f(x) = 0$).

1.1. Friedman’s method. One of the most popular methods to extract witnesses from classical proofs of Σ_1^0 -formulae has been introduced by Friedman [6]. The idea of Friedman is to generalize Gödel and Kolmogorov’s double negation translation by replacing the intuitionistic negation $\neg A \equiv A \Rightarrow \perp$ by a relative negation $\neg_R A \equiv A \Rightarrow R$ parameterized by an arbitrary formula R . (The only condition on R is that its free variables should not be captured in the formula or the proof we want to translate.) In first-order Peano arithmetic (PA) for instance, this negative R -translation $A \mapsto A^{\neg\neg}$ can be defined as follows

$$\begin{aligned} (e_1 = e_2)^{\neg\neg} &\equiv \neg_R \neg_R (e_1 = e_2) & (\neg A)^{\neg\neg} &\equiv \neg_R A^{\neg\neg} \\ (A \wedge B)^{\neg\neg} &\equiv A^{\neg\neg} \wedge B^{\neg\neg} & (\forall x A)^{\neg\neg} &\equiv \forall x A^{\neg\neg} \\ (A \vee B)^{\neg\neg} &\equiv \neg_R (\neg_R A^{\neg\neg} \wedge \neg_R B^{\neg\neg}) & (\exists x A)^{\neg\neg} &\equiv \neg_R \forall x \neg_R A^{\neg\neg} \end{aligned}$$

and it is easy to check that if a formula A is provable in Peano arithmetic, then the formula $A^{\neg\neg}$ is provable in Heyting Arithmetic (HA), independently from the choice of the formula R .

If we apply this translation to a classical proof p of the formula $\exists x f(x) = 0$ (i.e. a Σ_1^0 -formula), then we get an intuitionistic proof p^* of the formula

$$\neg_R \forall x \neg_R \neg_R \neg_R f(x) = 0.$$

By simplifying the triple (relative) negation and by unfolding the relative negation $\neg_R A \equiv A \Rightarrow R$, we thus get an intuitionistic proof $p^{*'}$ of the formula

$$\forall x (f(x) = 0 \Rightarrow R) \Rightarrow R.$$

(The proof $p^{*'}$ we get is parametric w.r.t. the formula R .)

Now, let us introduce Friedman’s trick, which is to instantiate the parameter R with the formula we want to prove, letting $R \equiv \exists y f(y) = 0$. Thus $p^{*'}$ is an intuitionistic proof of the implication

$$\forall x (f(x) = 0 \Rightarrow \exists y f(y) = 0) \Rightarrow \exists y f(y) = 0.$$

whose left member is the introduction rule of existential quantification. Combining the modus ponens with the introduction rule of existential quantification, we finally get an intuitionistic proof $\tilde{p} \equiv (p^{*'})$ (\exists -intro) of the formula

$$\exists y f(y) = 0$$

from which we can perform the standard extraction techniques.

The transformation above actually shows that classical arithmetic is conservative over intuitionistic arithmetic on the class of Σ_1^0 -formulae. Since the transformation even works when the inner formula depends on free variables, it is easy to generalize the latter result to a result of conservativity on the class of Π_2^0 -formulae:

$$\frac{\frac{\frac{\vdash_{\text{PA}} \forall x \exists y f(x, y) = 0}{\vdash_{\text{PA}} \exists y f(x_0, y) = 0} \text{ (\forall-elim, } x_0 \text{ fresh)}}{\vdash_{\text{HA}} \exists y f(x_0, y) = 0} \text{ (Friedman's transformation)}}{\vdash_{\text{HA}} \forall x \exists y f(x, y) = 0} \text{ (\forall-intro)}$$

This conservativity result has been extended by Friedman (using the same technique) to much stronger pairs of classical and intuitionistic theories, such as PA2/HA2, ..., PA ω /HA ω , Z/IZ, ZF/IZF_C [6].

1.2. Krivine’s classical realizability. Up to the 90’s, the computational contents of classical proofs was only studied indirectly, via clever translations to intuitionistic logic [8, 12, 6] or to linear logic. The situation quickly changed with the discovery of a strong connection between classical reasoning principles (such as Peirce’s law) and control operators (such as call/cc) [9]. This led to the rise of many extensions of the λ -calculus with control primitives, such as Krivine’s λ_c -calculus [19], Parigot’s $\lambda\mu$ -calculus [26], Barbanera and Berardi’s (non deterministic) symmetric λ -calculus [1] or Curien and Herbelin’s $\lambda\bar{\lambda}\mu\tilde{\mu}$ -calculus [5]. (This list is far from being exhaustive.)

Among these different proposals to extend the proofs-as-programs paradigm to classical logic, Krivine’s theory of classical realizability [16, 19] enjoys a particular position. First, it is based on realizability rather than on typing, which makes it naturally more flexible and more powerful than systems that are simply based on typing. Second, the simplicity on the underlying calculus of realizers (the λ -calculus extended with the call/cc control primitive) and of its evaluation policy (weak head normalization) hides its main feature, which is its ability to incorporate new instructions in order to realize new formulæ, such as (for instance) several forms of the axiom of choice [17]. Although classical realizability is traditionally presented in second-order classical arithmetic, it can be extended to much more expressive logical frameworks such as Zermelo-Fraenkel set theory [16] or the calculus of constructions with universes [21].

Less known is the fact that Krivine’s framework allows to perform classical witness extraction directly (especially from realizers of Σ_1^0 -formulæ), without going through a negative translation such as Friedman’s. The purpose of this paper is twofold. First, it aims at presenting some methods that naturally come with classical realizability in order to extract witnesses from classical proofs of existential formulæ—especially Σ_1^0 -formulæ. Second, it aims to relate the extraction method for Σ_1^0 -formulæ with Friedman’s, by showing that through a well-chosen negative translation (inspired from [25]), both methods are basically the same (up to the details of the translation).

One of the difficulties of tracking *arithmetic* reasoning through a negative translation is that some parts of the proof carry over logical invariants whereas other parts are only devoted to arithmetic computations. To solve this problem, we shall introduce *primitive numerals* in the language of realizers, while showing that they (essentially) realize the same formulæ as Church numerals. As a side effect, replacing Church numerals with primitive numerals also makes the corresponding extraction technique much more realistic—and we believe, much more efficient—in the perspective of a practical implementation.

1.3. Outline of the paper. In section 2, we present a type system for classical second-order arithmetic (PA2) based on the λ -calculus extended with the primitive call/cc. This type system is given its semantics in section 3, by defining a family of classical realizability models (following [19]). In section 4, we extend the calculus of realizers and the type system for PA2 with primitive numerals to make arithmetic computations more efficient (in proof-terms) and more easily tractable through the negative translation. The classical witness extraction methods are presented in section 5 and we illustrate them with an example

based on the minimum principle in section 6. In section 7, we define a more traditional type system for intuitionistic second-order arithmetic (HA2), which we relate to the type system for PA2 by defining in section 8 a negative translation in the spirit of [25].

2. CLASSICAL SECOND-ORDER ARITHMETIC (PA2)

2.1. The language of second-order arithmetic. The language of PA2 (Fig. 1 p. 5) is made of two kinds of syntactic expressions: *arithmetic expressions* (a.k.a. first-order terms¹) that represent individuals, and *formulae* that represent mathematical propositions.

Arithmetic expressions (notation: e, e', e_1 , etc.) are built from an infinite set of first-order variables (notation: x, y, z , etc.) using function symbols (notation: f, g, h , etc.) defined in a given first-order signature. Here, we assume that the signature contains a constant symbol ‘0’ for zero, a unary function symbol ‘ s ’ for the successor function, and more generally, a function symbol f of arity k for every primitive recursive definition of a function with k arguments. In the sequel, we shall use binary function symbols ‘+’ (addition) and ‘ \times ’ (multiplication) as well as unary function symbols ‘pred’ (predecessor) and ‘neg’ (boolean negation) with the following definitions:

$$\begin{array}{ll} 0 + y & = y & 0 \times y & = 0 \\ s(x) + y & = s(x + y) & s(x) \times y & = (x \times y) + y \\ \text{pred}(0) & = 0 & \text{neg}(0) & = 1 \\ \text{pred}(s(x)) & = x & \text{neg}(s(x)) & = 0 \end{array}$$

(writing $1 = s(0)$, $2 = s(1)$, $3 = s(2)$, etc.) The set of all free variables of an arithmetic expression e is written $FV(e)$. The notion of (first-order) substitution in an arithmetic expression is defined as usual and written $e\{x := e'\}$.

Formulae of the language of second-order arithmetic (notation: A, B, C , etc.) are formed from second-order variables (notation: X, Y, Z , etc.) of all arities using implication and first- and second-order universal quantification (Fig. 1). We slightly deviate from the traditional presentation of the syntax of the language [7, 14] by explicitly introducing a unary predicate symbol ‘null’ expressing that its argument yields zero. The main reason for introducing this symbol is that it facilitates the construction of a simple proof-term for Peano’s 4th axiom within the type system presented in section 2.3.

The set of all free (first- and second-order) variables of a formula A is written $FV(A)$. The notions of first- and second-order substitution in a formula are defined as usual, and written $A\{x := e\}$ and $A\{X(x_1, \dots, x_k) := B\}$ respectively. (See [7, 14] for a more detailed presentation of the two forms of substitutions.)

2.1.1. Second-order encodings. Propositional units (\top and \perp), negation, conjunction, disjunction, first- and second-order existential quantification as well as Leibniz equality are represented using the second-order encodings given in Fig. 1. Here, we define the propositional constant \top as a shorthand for the formula $\text{null}(0)$, which is consistent with the type system of section 2.3 and the realizability interpretation of section 3. Intuitively, the formula \top is the type of *all* proof-terms, and it is important not to confuse it with the (true) formula $\mathbf{1} \equiv \forall Z (Z \Rightarrow Z)$ that has much less proof-terms.

¹We shall prefer the terminology of ‘arithmetic expression’ to the more standard terminology of ‘first-order term’ to prevent a confusion with the proof-terms we shall introduce in section 2.3.

The language of PA2

Arithmetic expr.	$e ::= x \mid f(e_1, \dots, e_k)$
Formulæ	$A, B ::= \text{null}(e) \mid X(e_1, \dots, e_k)$ $\mid A \Rightarrow B \mid \forall x A \mid \forall X A$
Proof-terms	$t, u ::= x \mid \lambda x. t \mid tu \mid \mathbf{c}$
Contexts	$\Gamma ::= \emptyset \mid \Gamma, x : A$

The congruences $e \cong e'$ and $A \cong A'$

$$\begin{aligned}
0 + y &\cong y & \text{pred}(0) &\cong 0 & \text{neg}(0) &\cong 1 & (\text{etc.}) \\
s(x) + y &\cong s(x + y) & \text{pred}(s(x)) &\cong x & \text{neg}(s(x)) &\cong 0 \\
\text{null}(s(x)) &\cong \perp
\end{aligned}$$

Abbreviations

$$\begin{aligned}
\top &\equiv \text{null}(0) \\
\perp &\equiv \forall Z Z \\
\neg A &\equiv A \Rightarrow \perp \\
A \wedge B &\equiv \forall Z ((A \Rightarrow B \Rightarrow Z) \Rightarrow Z) \\
A \vee B &\equiv \forall Z ((A \Rightarrow Z) \Rightarrow (B \Rightarrow Z) \Rightarrow Z) \\
\exists x A(x) &\equiv \forall Z (\forall x (A(x) \Rightarrow Z) \Rightarrow Z) \\
\exists X A(X) &\equiv \forall Z (\forall X (A(X) \Rightarrow Z) \Rightarrow Z) \\
e = e' &\equiv \forall Z (Z(e) \Rightarrow Z(e')) \\
\text{nat}(e) &\equiv \forall Z (Z(0) \Rightarrow \forall y (Z(y) \Rightarrow Z(s(y))) \Rightarrow Z(x))
\end{aligned}$$

Typing rules of PA2

$$\begin{array}{c}
\frac{}{\Gamma \vdash_{\text{NK}} x : A} \quad (x:A) \in \Gamma \qquad \frac{}{\Gamma \vdash_{\text{NK}} t : \top} \quad FV(t) \subseteq \text{dom}(\Gamma) \\
\\
\frac{}{\Gamma \vdash_{\text{NK}} \mathbf{c} : ((A \Rightarrow B) \Rightarrow A) \Rightarrow A} \qquad \frac{\Gamma \vdash_{\text{NK}} t : A}{\Gamma \vdash_{\text{NK}} t : A'} \quad A \cong A' \\
\\
\frac{\Gamma, x : A \vdash_{\text{NK}} t : B}{\Gamma \vdash_{\text{NK}} \lambda x. t : A \Rightarrow B} \qquad \frac{\Gamma \vdash_{\text{NK}} t : A \Rightarrow B \quad \Gamma \vdash_{\text{NK}} u : A}{\Gamma \vdash_{\text{NK}} tu : B} \\
\\
\frac{\Gamma \vdash_{\text{NK}} t : A}{\Gamma \vdash_{\text{NK}} t : \forall x A} \quad x \notin FV(\Gamma) \qquad \frac{\Gamma \vdash_{\text{NK}} t : \forall x A}{\Gamma \vdash_{\text{NK}} t : A\{x := e\}} \\
\\
\frac{\Gamma \vdash_{\text{NK}} t : A}{\Gamma \vdash_{\text{NK}} t : \forall X A} \quad X \notin FV(\Gamma) \qquad \frac{\Gamma \vdash_{\text{NK}} t : \forall X A}{\Gamma \vdash_{\text{NK}} t : A\{X(x_1, \dots, x_k) := B\}}
\end{array}$$

Figure 1: Classical second-order arithmetic (PA2)

2.2. The congruences $e \cong e'$ and $A \cong A'$. We introduce two congruences $e \cong e'$ and $A \cong A'$ over arithmetic expressions and formulæ that will be used to incorporate the definitional equalities of the function symbols of the signature in the conversion rule of the type system we shall introduce in section 2.3. The same mechanism will be used to build proof-terms for Peano’s 3rd and 4th axioms.

The congruence $e \cong e'$ over arithmetic expressions is simply defined as the congruence generated by the defining equations of the primitive recursive function symbols of the signature. (We already gave the equations associated with the function symbols ‘+’, ‘×’, ‘pred’ and ‘neg’ in section 2.1.) Of course, these equations can be oriented in such a way that they form a confluent and terminating system of rewrite rules, so that the congruence $e \cong e'$ is decidable. But we shall not need such a level of detail in the sequel.

The congruence $A \cong A'$ over formulæ is defined by adding the equation $\text{null}(s(x)) \cong \perp$ to the system of equations defining the congruence $e \cong e'$. Again, this new equation can be oriented from left to right so that the resulting system of rewrite rules (including the rewrite rules for function symbols) is confluent and terminating, and the congruence $A \cong A'$ is thus decidable.

2.3. A type system for classical second-order arithmetic. The type/proof system of PA2 closely follows the spirit of Second-order functional arithmetic (FA2) [14]. As in FA2, first- and second-order universal quantifications are treated uniformly, by using Curry-style proof-terms that do not keep track of introduction and elimination of universal quantifiers.² As usual in such a framework, numeric quantifications require a special treatment we shall recall in Section 2.4.

Formally, the type system of PA2 is based on a typing judgment of the form $\Gamma \vdash_{\text{NK}} t : A$, where Γ is a *typing context*, t a (Curry-style) *proof-term*, and where A is a formula of the language of PA2 (section 2.1).

Proof-terms of PA2 (notation: t, u , etc.) are just pure λ -terms³ enriched with a special constant \mathfrak{c} (‘call/cc’) to prove Peirce’s law. The operational semantics of proof-terms (that slightly differs from the traditional operational semantics of pure λ -calculus) will be given in section 3.

A typing context (notation: $\Gamma, \Gamma', \Gamma_1$, etc.) is a finite unordered list of declarations of the form $\Gamma \equiv x_1 : A_1, \dots, x_n : A_n$ where x_1, \dots, x_n are pairwise distinct proof-variables and where A_1, \dots, A_n are arbitrary formulæ. Given a typing context $\Gamma \equiv x_1 : A_1, \dots, x_n : A_n$, we write $\text{dom}(\Gamma) = \{x_1; \dots; x_n\}$ and $FV(\Gamma) = FV(A_1) \cup \dots \cup FV(A_n)$.

The inference rules for the judgment $\Gamma \vdash_{\text{NK}} t : A$ are given in Fig. 1. These rules contain the standard typing rules of AF2 [14] (that correspond to the deduction rules of intuitionistic natural deduction in second-order predicate logic), plus a typing rule for the constant \mathfrak{c} (Peirce’s axiom) to recover classical logic. These rules also contain a conversion rule as well an introduction rule for the propositional constant \top . (These rules are specifically needed to build proof-terms for the axioms of arithmetic.) In particular:

²For this reason, a (Curry-style) proof-term should not be confused with the proof (i.e. the derivation) it comes from, since the latter contains much more information that cannot be reconstructed from the proof-term. In such a setting, the proof-term is merely a computational digest of the formal proof, where some computationally irrelevant parts of the proof have been already removed.

³Proof variables (i.e. variables of the λ -calculus) are written x, y, z , etc. in the sequel, but it is important not to confuse them with first-order variables (written using the same letters) that occur in arithmetic expressions and formulæ.

expressing that x belongs to the smallest set of individuals containing zero and stable under the successor function. With this notation, the relativized form of the induction principle

$$\forall Z (Z(0) \Rightarrow \forall y (\text{nat}(y) \Rightarrow Z(y) \Rightarrow Z(s(y))) \Rightarrow \forall x (\text{nat}(x) \Rightarrow Z(x)))$$

can be given a closed proof-term in our setting. (See [14] for instance.)

More generally, we associate to every formula A a formula A^{nat} that is obtained by relativizing all the first-order quantifications with the predicate nat . Formally, the formula A^{nat} is defined by induction of A with the equations:

$$\begin{aligned} (\text{null}(e))^{\text{nat}} &\equiv \text{null}(e) \\ (X(e_1, \dots, e_k))^{\text{nat}} &\equiv X(e_1, \dots, e_k) \\ (A \Rightarrow B)^{\text{nat}} &\equiv A^{\text{nat}} \Rightarrow B^{\text{nat}} \\ (\forall x A)^{\text{nat}} &\equiv \forall x (\text{nat}(x) \Rightarrow A^{\text{nat}}) \\ (\forall X A)^{\text{nat}} &\equiv \forall X (A^{\text{nat}}) \end{aligned}$$

We then easily check that

Proposition 2.1. *If a closed formula A is provable in classical second-order arithmetic (with the unrelativized induction principle), then the formula A^{nat} has a closed proof-term in the type system defined in Fig. 1.*

3. CLASSICAL REALIZABILITY

We shall now present the classical realizability interpretation of the type system presented in section 2.3, following the method introduced by Krivine [19].

First, we shall introduce a calculus of realizers (Krivine's language λ_c) containing the proof-terms of Fig. 1, and give its evaluation rules, that constitute the small-step operational semantics of the language. From this, we shall see how to interpret every formula A of PA2 as a set of realizers $|A|$, reading the formula A as a specification of the computational behavior of the realizers of A . The connection between the classical realizability interpretation and big-step operational semantics in λ_c should become clear in sections 4 and 5.

3.1. A calculus of realizers. Krivine's language λ_c [19] is a strict extension of the calculus of proof-terms of PA2 (section 2.3). The language λ_c actually distinguishes three kinds of syntactic entities: *terms*, *stacks* and *processes*.

Terms	$t, u ::= x \mid \lambda x. t \mid tu \mid \kappa \mid k_\pi$	$(\kappa \in \mathcal{K})$
Stacks	$\pi ::= \diamond \mid t \cdot \pi$	$(t \text{ closed})$
Processes	$p, q ::= t \star \pi$	$(t \text{ closed})$

Terms of λ_c are pure λ -terms enriched with two kinds of constants:

- *instructions* $\kappa \in \mathcal{K}$, where \mathcal{K} is a fixed set of constants that contains (at least) an instruction written \mathfrak{c} (call/cc);
- *continuation constants* k_π , one for every stack π .

Stacks are finite lists of *closed* terms terminated by the stack constant \diamond .⁴ Note that unlike terms (that may be open or closed), stacks only contain closed terms and are thus closed objects—so that the continuation constant k_π associated to every stack π is actually a

⁴Krivine allows the formation of stacks using many stack constants (representing as many empty stacks), but we will not need more than one stack constant here.

constant. (The details of the mutual definition of terms and stacks are given in [19].) Finally, a *process* is simply a pair formed by a closed term t and a stack π . The set of closed terms (resp. the set of stacks) is written Λ_c (resp. Π), and the set of processes is written $\Lambda_c \star \Pi$.

In section 4 we shall extend the calculus with extra instructions to perform fast arithmetic computations. (See also Remark 3.1.)

3.1.1. Evaluation. The set of processes is equipped with a binary relation of one step *evaluation* written $p \succ p'$, whose reflexive-transitive closure is written $p \succ^* p'$ as usual. We assume that this relation satisfies (at least) the following axioms:

(GRAB)	$\lambda x . t \star u \cdot \pi$	\succ	$t\{x := u\} \star \pi$
(PUSH)	$tu \star \pi$	\succ	$t \star u \cdot \pi$
(CALL/CC)	$\mathfrak{c} \star t \cdot \pi$	\succ	$t \star \kappa_\pi \cdot \pi$
(RESUME)	$\kappa_\pi \star t \cdot \pi'$	\succ	$t \star \pi$

for all $t, u \in \Lambda_c$ and $\pi, \pi' \in \Pi$. Note that only processes are subject to evaluation: there is no notion of reduction for either terms or stacks in Λ_c .

This list of axioms—that basically implements weak head β -reduction in presence of the control operator *call/cc*—can be extended with extra axioms to describe the computational behavior of the other instructions $\kappa \in \mathcal{K}$.

Remark 3.1. Formally, the definition of the language λ_c thus depends on two parameters: the set \mathcal{K} of instructions (containing at least the instruction \mathfrak{c}), and the relation of evaluation \succ that fulfils the four axioms given above. In particular, the rules (GRAB), (PUSH), (CALL/CC) and (RESUME) are only *conditions* on the relation \succ , but they do not constitute a *defi* (by cases) of this relation. (The reader is invited to check that these conditions are actually the minimal conditions for proving Prop. 3.10.) Putting conditions on the set \mathcal{K} and on the relation of evaluation—rather than defining them completely—naturally makes the calculus modular, since this design allows us to enrich the calculus with extra instructions (by putting extra conditions on \mathcal{K}) and extra evaluation rules (by putting extra conditions on \succ), while keeping all the results that have been proved using a smaller set of conditions on \mathcal{K} and \succ ⁵. Technically, this open design has only one drawback, which is that it forbids any form of reasoning by ‘case analysis’ on an instruction or on an evaluation step—since the contents of \mathcal{K} and the definition of \succ are not (completely) known. Again, the reader is invited to check that this form of reasoning is never used in the results presented in Sections 3, 4 and 5—with the sole exception of Lemma 5.5 in section 5.3. The set of available instructions and evaluation rules will only be closed in section 8, in order to define the negative translation and to study its properties.

⁵This is the point of view that is taken in [16, 17, 19, 18].

3.2. The realizability interpretation.

3.2.1. *The notion of a pole.* The construction of the classical realizability model is parameterized by a set of processes $\perp \subseteq \Lambda_c \star \Pi$, which we called the *pole* of the model. We assume that this set is closed under anti-evaluation (or saturated according to the terminology of [19]). Formally:

Definition 3.2. A *pole* is any set of processes $\perp \subseteq \Lambda_c \star \Pi$ such that the conditions $p \succ p'$ and $p' \in \perp$ together imply $p \in \perp$ for all $p, p' \in \Lambda_c \star \Pi$.

Remark 3.3. Since the definition of a pole explicitly depends on the relation of evaluation \succ , all the conditions we put on the relation of evaluation (see Remark 3.1) are mechanically reflected in the definition of the notion of a pole. For instance, the rule (PUSH) is reflected in all poles \perp by the fact that $t \star u \cdot \pi \in \perp$ implies $tu \star \pi \in \perp$ (for all terms t, u and for all stacks π). The same holds for the rules (GRAB), (CALL/CC) and (RESUME), as well as for the new rules we shall introduce in Section 4. Putting more conditions on the relation of evaluation thus reduces the number of available poles.

Note that there are two generic ways to define a pole \perp from an arbitrary set of processes $P_0 \subseteq \Lambda_c \times \Pi$:

- The first method is to consider P_0 as a set of final (or ‘accepting’) states, and to take \perp as the closure of P_0 by anti-evaluation, that is: $\perp = (\succ P_0)$, which is defined by $(\succ P_0) \equiv \{p : \exists p_0 \in P_0 p \succ^* p_0\}$.
- The second method is to consider P_0 as a set of initial (‘forbidden’) states, and to take \perp as the *complement* of the closure of P_0 by evaluation, that is: $\perp = (\Lambda_c \star \Pi) \setminus (P_0 \succ)$, where $(P_0 \succ) \equiv \{p : \exists p_0 \in P_0 p_0 \succ^* p\}$.

In this paper, we shall build particular poles (in Section 5) only using the first method, but interesting uses of the second method can be found in [17].

3.2.2. *Truth and falsity values.* From now on, \perp denotes a fixed pole. We call a *falsity value* any set of stacks $S \subseteq \Pi$. By orthogonality, every falsity value $S \subseteq \Pi$ induces a *truth value* $S^\perp \subseteq \Lambda_c$ defined as:

$$S^\perp = \{t \in \Lambda_c : \forall \pi \in S t \star \pi \in \perp\}.$$

3.2.3. *Valuations and parametric formulæ.* A *valuation* is a function ρ whose domain is a finite set of (first- and second-order) variables, such that:

- $\rho(x) \in \mathbb{N}$ for every first-order variable $x \in \text{dom}(\rho)$;
- $\rho(X)$ is a (total) function from \mathbb{N}^k to $\mathfrak{P}(\Pi)$ (i.e. a *falsity value function*) for every k -ary second-order variable $X \in \text{dom}(\rho)$.

A *parametric expression* (resp. a *parametric formula*) is simply an arithmetic expression e (resp. a formula A) equipped with a valuation ρ , that we write $e[\rho]$ (resp. $A[\rho]$). Parametric contexts are defined similarly. A parametric expression (formula, context) is said to be closed when every free variable of the underlying expression (formula, context) belongs to the domain of the attached valuation.

For every closed parametric expression $e[\rho]$ we write $\mathbf{Val}(e[\rho]) \in \mathbb{N}$ the *value* of $e[\rho]$, interpreting variables by their images in ρ while giving to the primitive recursive function symbols in e their standard interpretation.

We easily check that:

Lemma 3.4. *If e and e' are two arithmetic expressions such that $e \cong e'$, then for all valuations ρ closing e and e' we have $\mathbf{Val}(e[\rho]) = \mathbf{Val}(e'[\rho])$.*

Proof. By induction on the derivation of $e \cong e'$. \square

3.2.4. The interpretation function. Every closed parametric formula $A[\rho]$ is interpreted as two sets, namely: a *falsity value* $\|A[\rho]\| \subseteq \Pi$ and a *truth value* $|A[\rho]| \subseteq \Lambda_c$. Both sets are defined by induction on the formula A as follows:

$$\begin{aligned} \|X(e_1, \dots, e_k)[\rho]\| &= \rho(X)(\mathbf{Val}(e_1[\rho]), \dots, \mathbf{Val}(e_k[\rho])) \\ \|\text{null}(e)[\rho]\| &= \begin{cases} \emptyset & \text{if } \mathbf{Val}(e[\rho]) = 0 \\ \Pi & \text{if } \mathbf{Val}(e[\rho]) \neq 0 \end{cases} \\ \|(A \Rightarrow B)[\rho]\| &= |A[\rho]| \cdot \|B[\rho]\| = \{t \cdot \pi : t \in |A[\rho]|, \pi \in \|B[\rho]\|\} \\ \|(\forall x A)[\rho]\| &= \bigcup_{n \in \mathbb{N}} \|A[\rho; x \leftarrow n]\| \\ \|(\forall X A)[\rho]\| &= \bigcup_{F: \mathbb{N}^k \rightarrow \mathfrak{P}(\Pi)} \|A[\rho; x \leftarrow F]\| \\ |A[\rho]| &= \|A[\rho]\|^\perp = \{t \in \Lambda_c : \forall \pi \in \|A[\rho]\| t \star \pi \in \perp\} \end{aligned}$$

The reader is invited to check that the sets $\|A[\rho]\|$ and $|A[\rho]|$ only depend on the values given by ρ to the free variables of A , so that we can drop the valuation ρ when A is closed and simply write $\|A\|$ and $|A|$ for $\|A[\rho]\|$ and $|A[\rho]|$.

We easily check that:

Lemma 3.5. *If A and A' are two formulæ of PA2 such that $A \cong A'$, then for all valuations ρ closing A and A' we have $\|A[\rho]\| = \|A'[\rho]\|$.*

Proof. By induction on the derivation of $A \cong A'$ using Lemma 3.4 together with the fact that

$$\|\perp\| = \bigcup_{S \subseteq \Pi} S = \Pi = \|\text{null}(s(e))[\rho]\|$$

for all closed parametric expressions $e[\rho]$ (to interpret $\perp \cong \text{null}(s(e))$). \square

Since the truth value $|A[\rho]|$ and the falsity value $\|A[\rho]\|$ of the formula A actually depend on the pole \perp , we shall sometimes use the notations $|A[\rho]|_\perp$ and $\|A[\rho]\|_\perp$ to indicate this dependency explicitly.

Definition 3.6 (Realizability). Given a pole \perp , a closed parametric formula $A[\rho]$ and a closed term t , we say that t *realizes* $A[\rho]$ and write $t \Vdash_{\text{NK}} A[\rho]$ when $t \in |A[\rho]|_\perp$, keeping in mind that this notation depends on the choice of the particular pole \perp . When $t \in |A[\rho]|_\perp$ for all poles \perp , we say that t *universally realizes* $A[\rho]$ and write $t \Vdash_{\text{NK}} A[\rho]$.

3.2.5. *Writing parametric formulæ.* In what follows, we shall often use the convenient shorthand

$$\dot{F}(e_1, \dots, e_k) \equiv (X(e_1, \dots, e_k))[X \leftarrow F]$$

to denote a parametric formula built from a *k*-ary predicate variable X that is bound to a particular falsity value function $F : \mathbb{N}^k \rightarrow \mathfrak{P}(\Pi)$ in the attached valuation. (The dot above the symbol \dot{F} is here to recall that F is an object that belongs to the semantics, not to the syntax.) By systematically using this notation, we can write parametric formulæ without explicitly mentioning valuations. In the sequel, we shall consider (for instance) that the notation

$$\forall z (\dot{F}(z) \Rightarrow \dot{S})$$

refers to the parametric formula

$$(\forall z (X(z) \Rightarrow Y))[X \leftarrow F, Y \leftarrow S]$$

where X and Y are arbitrarily chosen fresh variables. Note that the parametric formula defined by such a notation is defined up to the names of the variables that are bound in the valuation—but it is easy to see that these names have no impact in the interpretation of the corresponding parametric formula.

3.3. The full standard model of PA2 as a degenerate case. In the case where $\perp = \emptyset$, the classical realizability model defined above collapses to the full standard model of PA2 (i.e. the model 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). To understand this point, 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, 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:

Lemma 3.7. *If $\perp = \emptyset$, then for every closed formula A of PA2 we have*

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

Proof. We more generally show that for all formulæ A and for all valuations ρ closing A (in the sense defined in section 3.2) we have

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

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

- $\tilde{\rho}(x) = \rho(x)$ if x is a first-order variable such that $x \in \text{dom}(\rho)$;
- $\tilde{\rho}(X) = \{(n_1, \dots, n_k) \in \mathbb{N}^k : \rho(X)(n_1, \dots, n_k) = \emptyset\}$ if X is a second-order variable of arity k such that $X \in \text{dom}(\rho)$.

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

An interesting consequence of the above lemma is the following:

Lemma 3.8. *If a closed formula A has a universal realizer $t \Vdash_{\text{NK}} A$, then A is true in the full standard model of PA2.*

Proof. If $t \Vdash_{\text{NK}} A$, then $t \in |A|_{\emptyset}$. Therefore $|A|_{\emptyset} = \Lambda_c$ and $\mathcal{M} \models A$. \square

However, the converse implication is wrong in general, since the formula $\forall x \text{nat}(x)$ (cf Fig. 1) that expresses the induction principle over individuals is obviously true in \mathcal{M} , but has no universal realizer [19, Theorem 12]⁶. Nevertheless, the converse implication becomes true when we restrict it to *arithmetic formulæ*, that is, to the formulæ of the following language:

Arithmetic formulæ $P, Q ::= e_1 = e_2 \mid P \Rightarrow Q \mid \forall x (\text{nat}(x) \Rightarrow P)$

(This is a consequence of a slightly more general result in [19, Theorem 21].)

Remark 3.9. In the case where $\perp \neq \emptyset$, every truth value S^{\perp} is inhabited, for instance by any term of the form $k_{\pi_0} t_0$ where $t_0 \star \pi_0 \in \perp$. An important consequence of this remark is that a classical realizer of a formula A (w.r.t. to a nonempty pole) can never be taken as a ‘certificate’ that the formula A is true, even when A is an equality. (This remark is crucial to understand the specific difficulty of witness extraction in classical realizability.)

3.4. Adequacy. We call a *substitution* any finite function from proof-variables to the set Λ_c of closed λ_c -terms, and we denote by $t[\sigma]$ the term obtained by applying a substitution σ to a term t . Given a substitution σ and a closed parametric context $\Gamma[\rho]$, we write $\sigma \Vdash_{\text{NK}} \Gamma[\rho]$ when the following conditions are fulfilled:

- (1) $\text{dom}(\Gamma) \subseteq \text{dom}(\sigma)$;
- (2) $\sigma(x) \Vdash_{\text{NK}} A[\rho]$ for every declaration $(x : A) \in \Gamma$.

We say that:

- A judgment $\Gamma \vdash_{\text{NK}} t : A$ is *sound* (w.r.t. the pole \perp) when for all valuations ρ and for all substitutions σ such that $\sigma \Vdash_{\text{NK}} \Gamma[\rho]$, we have $t[\sigma] \Vdash_{\text{NK}} A[\rho]$.
- An inference rule $\frac{P_1 \dots P_n}{C}$ (where P_1, \dots, P_n and C are typing judgments) is *sound* (w.r.t. the pole \perp) when the soundness of its premises P_1, \dots, P_n (in the above sense) implies the soundness of its conclusion C .

From these definitions, it is clear that the conclusion of any typing derivation formed with only sound inference rules is sound w.r.t. all poles \perp .

Proposition 3.10 (Adequacy). *The typing rules of PA2 (Fig. 1) are sound w.r.t. all poles $\perp \subseteq \Lambda_c \times \Pi$.*

Proof. The soundness of the introduction rule of \top is obvious (since $|\top| = \Lambda_c$) and the soundness of the conversion rule follows from Lemma 3.5. The soundness of the remaining typing rules is proved in [19]. \square

⁶This explains the special treatment of the induction principle in section 2.4.

A consequence of this proposition is that closed proof-terms that are built using the type system of PA2 are actually universal realizers of the corresponding formulæ. (But not all realizers can be detected via typing [19].)

4. PRIMITIVE NATURAL NUMBERS

Through the formulæ-as-types paradigm, the relativized form of first-order universal quantification $\forall x (\text{nat}(x) \Rightarrow A(x))$ corresponds to the (dependent) type of all functions mapping realizers of the formula $\text{nat}(n)$ to realizers of the formula $A(n)$ for every $n \in \mathbb{N}$. To get a realizer of the formula $A(n)$ (for a particular value of $n \in \mathbb{N}$) from a realizer t of the formula $\forall x (\text{nat}(x) \Rightarrow A(x))$, it suffices to apply the term t to the Church numeral $\lambda x f . f^n x$ using the following fact

Fact 4.1. For every $n \in \mathbb{N}$ one has: $\vdash_{\text{NK}} \lambda x f . f^n x : \text{nat}(n)$.

combined with the property of adequacy (Prop. 3.10).

On the other hand, Church numeral $\lambda x f . f^n x$ is far from being the only realizer of the formula $\text{nat}(n)$ —the situation being much more complex than in intuitionistic realizability due to the presence of continuations in realizers. However, it is always possible to effectively retrieve (in some sense) the natural number n from an arbitrary realizer of the formula $\text{nat}(n)$, and the traditional way to achieve this in classical realizability is to use a storage operator [15, 19]. We propose here another method by changing the representation of numerals.

Indeed, the main defect of Church numerals is not only their very poor efficiency in practical computations (especially for large values), but also the non atomicity of their encoding that makes them very hard to track through a negative translation towards intuitionistic logic. For this reason, we present here an alternative implementation of natural numbers in classical realizability, based on the introduction of specific constants to represent natural numbers with new instructions to compute with them.

4.1. Extending the language of realizers. We now enrich⁷ the instruction set \mathcal{K} with the following constants:

- For every $n \in \mathbb{N}$, a constant $\widehat{n} \in \mathcal{K}$ representing the natural number n as a pure datum. Here, the constant \widehat{n} hardly deserves the name of an instruction, since it comes with no evaluation rule. The intuition is that the constant \widehat{n} is only meaningful as a datum in the stack, not in head position.⁸
- Two constants \mathfrak{s} and rec with the evaluation rules

$$\begin{array}{lll}
 \text{(SUCC)} & \mathfrak{s} \star \widehat{n} \cdot u \cdot \pi & \succ \quad u \star \widehat{n+1} \cdot \pi \\
 \text{(REC-0)} & \text{rec} \star u_0 \cdot u_1 \cdot \widehat{0} \cdot \pi & \succ \quad u_0 \star \pi \\
 \text{(REC-S)} & \text{rec} \star u_0 \cdot u_1 \cdot \widehat{n+1} \cdot \pi & \succ \quad u_1 \star \widehat{n} \cdot (\text{rec } u_0 \ u_1 \ \widehat{n}) \cdot \pi
 \end{array}$$

for all $u, u_0, u_1 \in \Lambda_c$, $n \in \mathbb{N}$ and $\pi \in \Pi$.

⁷See remark 3.1 p. 9.

⁸This is similar to the situation in most programming languages, where numbers are represented using machine numbers (or blocks of machine numbers) that are meaningless as pointers, so that executing them usually raises a memory fault.

With these new instructions, it is more generally possible to implement every recursive function f of arity k as a term \check{f} with the reduction rule

$$\check{f} \star \hat{n}_1 \cdots \hat{n}_k \cdot u \cdot \pi \succ^* u \star \hat{m} \cdot \pi,$$

for all $(n_1, \dots, n_k) \in \text{dom}(f)$, writing $m = f(n_1, \dots, n_k)$.⁹ To improve efficiency, we can also introduce the \check{f} 's (or some of them) as new instructions.

Apart from the representation of numerals as pure data, every natural number $n \in \mathbb{N}$ can be also represented as a *program* \check{n} defined by $\check{n} \equiv \lambda x . x \hat{n}$. (We will momentarily see how to give a ‘type’ to this program in PA2.)

4.2. Extending the realizability interpretation. To understand the computational behavior of the instructions that come with our alternative representation of numerals, we extend the language of formulæ of PA2 with a new syntactic construct $\{e\} \Rightarrow B$ where e is an arithmetic expression and B a formula. (This extension is part of a larger system PA2⁺ that will be introduced in section 4.3.) Intuitively, this formula corresponds to the type of all functions taking the representation of the value of e as the constant \hat{n} (where $n = \mathbf{Val}(e)$) and return an object of type B .

Formally, the realizability interpretation of the formulæ of PA2 (section 3.2) is extended to the syntactic construct $\{e\} \Rightarrow B$ by letting:

$$\|(\{e\} \Rightarrow B)[\rho]\| = \{\hat{n} \cdot \pi : n = \mathbf{Val}(e[\rho]), \pi \in \|B[\rho]\|\}.$$

In this extended syntax, we can now give a type to the lazy numeral $\check{n} \equiv \lambda x . x \hat{n}$ by letting $\text{nat}'(e) \equiv \forall Z ((\{e\} \Rightarrow Z) \Rightarrow Z)$ and checking that:

Lemma 4.2. *For every $n \in \mathbb{N}$: $\check{n} \equiv \lambda x . x \hat{n} \Vdash_{\text{NK}} \text{nat}'(n)$*

Proof. Let \perp be a fixed pole, and consider an arbitrary element of falsity value $\|\text{nat}'(n)\| = \|\forall Z ((\{n\} \Rightarrow Z) \Rightarrow Z)\|$, that is: a stack of the form $u \cdot \pi$ where $u \in |\{n\} \Rightarrow \dot{S}|$ and $\pi \in S$ for some falsity value $S \in \mathfrak{P}(\Pi)$. We have $\lambda x . x \hat{n} \star u \cdot \pi \succ^* u \star \hat{n} \cdot \pi$. But since $\hat{n} \cdot \pi \in \|\{n\} \Rightarrow \dot{S}\|$ we get $u \star \hat{n} \cdot \pi \in \perp$, hence $\lambda x . x \hat{n} \star u \cdot \pi \in \perp$ by anti-evaluation. \square

Moreover:

Lemma 4.3. *Writing $\forall^{\mathbb{N}} x A(x) \equiv \forall x (\{x\} \Rightarrow A(x))$, we have:*

- (1) $\mathfrak{s} \Vdash_{\text{NK}} \forall^{\mathbb{N}} x \text{nat}'(s(x))$
- (2) $\text{rec} \Vdash_{\text{NK}} \forall Z (Z(0) \Rightarrow \forall^{\mathbb{N}} y (Z(y) \Rightarrow Z(s(y)))) \Rightarrow \forall^{\mathbb{N}} x Z(x)$

Proof. Let \perp be a fixed pole.

- (1) Let us consider an arbitrary element of $\|\forall^{\mathbb{N}} x \text{nat}'(s(x))\|$, that is: a stack of the form $\hat{n} \cdot u \cdot \pi$, where $n \in \mathbb{N}$, $u \in |\{s(n)\} \Rightarrow \dot{S}|$ and $\pi \in S$ for some falsity value $S \subseteq \Pi$. We want to show that $\mathfrak{s} \star \hat{n} \cdot u \cdot \pi \in \perp$. Using the evaluation rule of \mathfrak{s} , we get $\mathfrak{s} \star \hat{n} \cdot u \cdot \pi \succ u \star \widehat{n+1} \cdot \pi$. But since $\widehat{n+1} \cdot \pi \in \|\{s(n)\} \Rightarrow \dot{S}\|$, we have $u \star \widehat{n+1} \cdot \pi \in \perp$, hence $\mathfrak{s} \star \hat{n} \cdot u \cdot \pi \in \perp$ by anti-evaluation.
- (2) Let us take a falsity value function $F : \mathbb{N} \rightarrow \mathfrak{P}(\Pi)$ and consider two realizers $u_0 \in |\dot{F}(0)|$ and $u_1 \in |\forall^{\mathbb{N}} y (\dot{F}(y) \Rightarrow \dot{F}(s(y)))|$. We first show by induction on $n \in \mathbb{N}$ that for all stacks $\pi \in F(n)$ we have $\text{rec} \star u_0 \cdot u_1 \cdot \hat{n} \cdot \pi \in \perp$.

⁹In the case we want to go beyond primitive recursion, it is necessary to use a fixpoint combinator to implement minimization.

- Base case. Take $\pi \in F(0)$. We have $\text{rec} \star u_0 \cdot u_1 \cdot \widehat{0} \cdot \pi \succ u_0 \star \pi \in \perp$ (since $u_0 \in F(0)^\perp$), hence $\text{rec} \star u_0 \cdot u_1 \cdot \widehat{0} \cdot \pi \in \perp$ by anti-evaluation.
- Let us assume that the property holds for $n \in \mathbb{N}$, and consider a stack $\pi \in F(n+1)$. We have $\text{rec} \star u_0 \cdot u_1 \cdot \widehat{n+1} \cdot \pi \succ u_1 \star \widehat{n} \cdot (\text{rec } u_0 \ u_1 \ \widehat{n}) \cdot \pi$. We now want to show that $\text{rec } u_0 \ u_1 \ \widehat{n} \in |F(n)|$. For that, we take a stack $\pi' \in F(n)$ and get $\text{rec } u_0 \ u_1 \ \widehat{n} \star \pi' \succ^* \text{rec} \star u_0 \cdot u_1 \cdot \widehat{n} \cdot \pi' \in \perp$ by induction hypothesis, hence $\text{rec } u_0 \ u_1 \ \widehat{n} \star \pi' \in \perp$ by anti-evaluation. Thus we have $\text{rec } u_0 \ u_1 \ \widehat{n} \in |F(n)|$, hence we get

$$\begin{aligned} \widehat{n} \cdot (\text{rec } u_0 \ u_1 \ \widehat{n}) \cdot \pi &\in \|\{n\} \Rightarrow \dot{F}(n) \Rightarrow \dot{F}(s(n))\| \\ &\subseteq \|\forall^{\mathbb{N}} y (\dot{F}(y) \Rightarrow \dot{F}(s(y)))\|. \end{aligned}$$

Therefore $u_1 \star \widehat{n} \cdot (\text{rec } u_0 \ u_1 \ \widehat{n}) \cdot \pi \in \perp$, and $\text{rec} \star u_0 \cdot u_1 \cdot \widehat{n+1} \cdot \pi \in \perp$ by anti-evaluation. We have shown that $\text{rec} \star u_0 \cdot u_1 \cdot \widehat{n} \cdot \pi \in \perp$ for all $F : \mathbb{N} \rightarrow \mathfrak{P}(\Pi)$ and for all $u_0 \in |\dot{F}(0)|$, $u_1 \in |\forall^{\mathbb{N}} y (\dot{F}(y) \Rightarrow \dot{F}(s(y)))|$, $n \in \mathbb{N}$ and $\pi \in F(n)$. But this precisely means that rec realizes the desired formula. \square

4.3. Extending the type system. To facilitate the construction of universal realizers using the new instructions, we define an extension of PA2 (1), which we call PA2⁺. The specific formation rules and typing rules of this system are summarized in Fig. 3.

Syntactic constructs	
Formulæ	$A, B ::= \dots \mid \{e\} \Rightarrow B$
Proof-terms	$t, u ::= \dots \mid \widehat{n} \mid s \mid \text{rec}$
Contexts	$\Gamma ::= \dots \mid \Gamma, x : \{e\}$
Abbreviations	
$\text{nat}'(e)$	$\equiv \forall Z ((\{e\} \Rightarrow Z) \Rightarrow Z)$
$\forall^{\mathbb{N}} x A$	$\equiv \forall x (\{x\} \Rightarrow A)$
$\exists^{\mathbb{N}} x A$	$\equiv \forall Z (\forall x (\{x\} \Rightarrow A \Rightarrow Z) \Rightarrow Z)$
Typing rules	
$\Gamma \vdash_{\text{NK}} \text{rec} : \forall Z (Z(0) \Rightarrow \forall^{\mathbb{N}} y (Z(y) \Rightarrow Z(s(y)))) \Rightarrow \forall^{\mathbb{N}} x Z(x)$	
$\Gamma \vdash_{\text{NK}} s : \forall^{\mathbb{N}} x \text{nat}'(s(x))$	$\frac{\Gamma, x : \{e\} \vdash_{\text{NK}} t : B}{\Gamma \vdash_{\text{NK}} \lambda x. t : \{e\} \Rightarrow B}$
$\frac{\Gamma \vdash_{\text{NK}} t : \{e\} \Rightarrow B}{\Gamma \vdash_{\text{NK}} t x : B}$	$\frac{\Gamma \vdash_{\text{NK}} t : \{n\} \Rightarrow B}{\Gamma \vdash_{\text{NK}} t \widehat{n} : B} \quad (x:\{e\}) \in \Gamma$

Figure 3: Extending PA2 with primitive numerals

Compared to PA2, the grammar of the formulæ of PA2⁺ is enriched with the syntactic construct $\{e\} \Rightarrow B$ introduced in Section 4.2. (Arithmetic expressions remain unchanged.) To reflect the presence of a second form of implication, typing contexts of system PA2⁺

introduce a second form of declaration, written $x : \{e\}$, that expresses that the proof-variable x is bound to the constant \widehat{n} , where n is the value of e (i.e. $n = \mathbf{Val}(e)$).

Proof-terms of PA2^+ are the proof-terms of PA2 enriched with the constants \widehat{n} (for all $n \in \mathbb{N}$), \mathbf{s} and \mathbf{rec} . System PA2^+ provides typing rules for the constants \mathbf{s} and \mathbf{rec} , as well as an introduction rule and two elimination rules for the formula $\{e\} \Rightarrow B$. Note that in this system, we can only apply a proof-term of type $\{e\} \Rightarrow B$ to a variable (declared with $x : \{e\}$) or to a constant of the form \widehat{n} —in which case we must have $e \equiv s^n(0)$.

4.3.1. The realizability interpretation of PA2^+ . The realizability interpretation of PA2^+ is defined as for PA2 , using the interpretation of the formula $\{e\} \Rightarrow B$ described in Section 4.2. (Of course, we now work with a set \mathcal{K} of instructions and a relation of evaluation that fulfill the conditions given in Section 3 and 4.)

To express the soundness of the new typing rules, we first have to adapt the definition of $\sigma \Vdash_{\text{NK}} \Gamma[\rho]$ to the extended notion of context. For that, we say that a substitution σ realizes a closed parametric context $\Gamma[\rho]$ and write $\sigma \Vdash_{\text{NK}} \Gamma[\rho]$ when the following conditions are fulfilled:

- (1) $\text{dom}(\Gamma) \subseteq \text{dom}(\sigma)$;
- (2) $\sigma(x) \Vdash_{\text{NK}} A[\rho]$ for every declaration $(x : A) \in \Gamma$;
- (3) $\sigma(x) = \widehat{n}$ where $n = \mathbf{Val}(e[\rho])$ for every declaration $(x : \{e\}) \in \Gamma$.

(This definition obviously coincides with the former definition in the case where the context Γ only contains declarations of the form $(x : A)$.) The definition of sound judgments and of sound valid rules (w.r.t. a fixed pole) immediately extends to the new system, so that we can check the following:

Proposition 4.4 (Adequacy). *The typing rules of PA2^+ are sound w.r.t. all poles $\perp \subseteq \Lambda_c \times \Pi$.*

Proof. Let \perp be a pole. We only treat the specific rules of PA2^+ (Fig. 3).

- Typing rules for \mathbf{s} and \mathbf{rec} : immediately follows from Lemma 4.3.
- Introduction rule of $\{e\} \Rightarrow B$. Let us assume that $\Gamma, x : \{e\} \vdash_{\text{NK}} t : B$ is sound. To show that the judgment $\Gamma \vdash_{\text{NK}} \lambda x . t : \{e\} \Rightarrow B$ is sound too, consider a valuation ρ with a substitution σ such that $\sigma \Vdash_{\text{NK}} \Gamma[\rho]$. We want to prove that $(\lambda x . t)[\sigma] \in |(\{e\} \Rightarrow B)[\rho]|$. For that, let us consider an arbitrary element of $|(\{e\} \Rightarrow B)[\rho]|$, that is: a stack of the form $\widehat{n} \cdot \pi$ where $n = \mathbf{Val}(e[\rho])$ and $\pi \in |B[\rho]|$, and let us prove that $(\lambda x . t)[\sigma] \star n \cdot \pi \in \perp$. Let $\sigma' = (\sigma, x := \widehat{n})$. We have $\sigma' \Vdash (\Gamma, x : \{e\})[\rho]$, hence $t[\sigma'] \in |B[\rho]|$ from the soundness of judgment $\Gamma, x : \{e\} \vdash_{\text{NK}} t : B$. By evaluating the process $(\lambda x . t)[\sigma] \star n \cdot \pi$ we get $(\lambda x . t)[\sigma] \star n \cdot \pi \succ t[\sigma'] \star \pi \in \perp$ (since $t[\sigma'] \in |B[\rho]|$), hence $(\lambda x . t)[\sigma] \star n \cdot \pi \in \perp$ by anti-evaluation.
- Elimination rules of $\{e\} \Rightarrow B$: both cases are straightforward. □

Thanks to this extension, it is easy to check (by means of typing) that the new relativization predicate $\text{nat}'(x)$ is logically equivalent (in PA2^+) to the traditional relativization predicate $\text{nat}(x)$ defined in section 2.4:

$$\begin{aligned} \lambda z . z \check{0} (\lambda y . y \check{s}) & : \forall x (\text{nat}(x) \Rightarrow \text{nat}'(x)) \\ \lambda z . z (\mathbf{rec} (\lambda x f . x) (\lambda n x f . f (n x f))) & : \forall x (\text{nat}'(x) \Rightarrow \text{nat}(x)) \end{aligned}$$

(Intuitively, the above terms convert a Church numeral into the corresponding lazy numeral and vice-versa.) Moreover, we can check that the formula $\forall^{\mathbb{N}}x A(x)$ defined by the shorthand

$$\forall^{\mathbb{N}}x A(x) \equiv \forall x (\{x\} \Rightarrow A(x)) \quad (\text{nat-as-data relativization})$$

is logically equivalent to the formula

$$\forall x (\text{nat}'(x) \Rightarrow A(x)) \quad (\text{nat-as-program relativization})$$

by means of the following proof-terms:

$$\begin{aligned} \lambda f x . f (\lambda y . y x) & : \forall x (\text{nat}'(x) \Rightarrow A(x)) \Rightarrow \forall^{\mathbb{N}}x A(x) \\ \lambda f x . x f & : \forall^{\mathbb{N}}x A(x) \Rightarrow \forall x (\text{nat}'(x) \Rightarrow A(x)) \end{aligned}$$

(Intuitively, functions of type $\forall^{\mathbb{N}}x A(x)$ expect a fully computed natural number represented as a datum on the top of the stack, whereas functions of type $\forall x (\text{nat}'(x) \Rightarrow A(x))$ expect a lazy representation of a natural number on the top of the stack, whose corresponding value can be computed later.)

The same remark holds for the two different ways to relativize first-order existential quantification using primitive numerals

$$\begin{aligned} \forall Z (\forall x (\text{nat}'(x) \Rightarrow A(x) \Rightarrow Z) \Rightarrow Z) \\ \exists^{\mathbb{N}}x A(x) \equiv \forall Z (\forall x (\{x\} \Rightarrow A(x) \Rightarrow Z) \Rightarrow Z) \end{aligned}$$

that are provably equivalent.

In what follows, we shall thus only consider the problem of witness extraction from universal realizers of existential formulæ of the form $\exists^{\mathbb{N}}x A(x)$, whose witnesses are the most directly accessible.

5. WITNESS EXTRACTION IN CLASSICAL REALIZABILITY

In this section, we are interested in the problem of extracting a witness of a closed existential formula $\exists^{\mathbb{N}}x A(x)$ from a fixed universal realizer t_0 of this formula:

$$t_0 \Vdash_{\text{NK}} \exists^{\mathbb{N}}x A(x) \equiv \forall Z (\forall x (\{x\} \Rightarrow A(x) \Rightarrow Z) \Rightarrow Z).$$

(As a particular case, t_0 may be a proof term of $\exists^{\mathbb{N}}x A(x)$ in PA2.)

Throughout this section, we assume that the instruction set \mathcal{K} contains (at least) the extra instructions \widehat{n} , s and rec presented in Section 4.1, with their accompanying rules. For convenience, we also assume the existence of an instruction stop with no evaluation rule, that is intended to abort computation once the desired witness has been found. However, the proofs of Prop. 5.1, 5.3 and 5.6 do not rely on any particular assumption on stop , so that these propositions still hold if we consider that stop denotes a fixed closed λ_c -term.

The witness extraction methods discussed in Sections 5.2 and 5.4 are directly inspired from the techniques presented in [19], while the method presented in Section 5.6 is due to the author.

5.1. The failure of the naive method. To extract a witness from the universal realizer $t_0 \Vdash_{\text{NK}} \exists^{\mathbb{N}} x A(x)$, a natural idea would be to apply t_0 to the term $\lambda xy. \text{stop } x$ that extracts the first component of the ‘pair’ t_0 and passes it to stop . Applying this idea, we get the following:

Proposition 5.1. *For all $\pi \in \Pi$, the process $t_0 \star (\lambda xy. \text{stop } x) \cdot \pi$ evaluates (in a finite number of steps) to a process of the form $\text{stop} \star \hat{n} \cdot \pi$ for some $n \in \mathbb{N}$.*

Proof. Let us take a stack $\pi \in \Pi$ and work in the pole defined by

$$\perp\!\!\!\perp = \{p : \exists n \in \mathbb{N} p \succ^* \text{stop} \star \hat{n} \cdot \pi\}.$$

Writing $S = \{\pi\}$, we easily check that $\text{stop} \Vdash_{\text{NK}} \forall x (\{x\} \Rightarrow \dot{S})$ (from the definition of $\perp\!\!\!\perp$), so that $\lambda xy. \text{stop } x \Vdash_{\text{NK}} \forall x (\{x\} \Rightarrow A(x) \Rightarrow \dot{S})$ (by Prop. 3.10). Therefore $(\lambda xy. \text{stop } x) \cdot \pi \in \|\exists^{\mathbb{N}} x A(x)\|$, and thus $t_0 \star (\lambda xy. \text{stop } x) \cdot \pi \in \perp\!\!\!\perp$. \square

Alas, this result gives us no warranty that the natural number n we get by this method is such that $A(n)$ is true (in the full standard model). The mistake here is that we have dropped the second component y of the pair t_0 (that cannot be taken as a certificate that $A(n)$ holds), and we shall momentarily see that this component is actually the crucial ingredient of the extraction process.

5.2. Extraction in the Σ_1^0 -case. Let us now consider the particular case where the predicate $A(x)$ is of the form $A(x) \equiv f(x) = 0$, where f is a unary function symbol of the signature corresponding to (and denoted by) a primitive recursive function still written f .

To understand how to extract a (correct) witness from t_0 in this case, let us first study the denotation of equalities in the realizability model:

Lemma 5.2. *Let e_1 and e_2 be closed arithmetic expressions. For all poles $\perp\!\!\!\perp$ we have*

$$\|e_1 = e_2\| = \begin{cases} \{t \cdot \pi : (t \star \pi) \in \perp\!\!\!\perp\} = \|\mathbf{1}\| & \text{if } \mathbf{Val}(e_1) = \mathbf{Val}(e_2) \\ \Lambda \cdot \Pi = \|\top \Rightarrow \perp\| & \text{if } \mathbf{Val}(e_1) \neq \mathbf{Val}(e_2) \end{cases}$$

(writing $\mathbf{1} \equiv \forall Z (Z \Rightarrow Z)$).

In other words, true equalities are interpreted the same way as the formula $\mathbf{1} \equiv \forall Z (Z \Rightarrow Z)$ whereas false equalities are interpreted the same way as the formula $\top \Rightarrow \perp$ in the classical realizability model. If u is a realizer of the formula $f(n) = 0$ (w.r.t. a particular pole $\perp\!\!\!\perp$), then we can distinguish two cases:

- The equality $f(n) = 0$ is true. In this case, we can think of u ($\Vdash_{\text{NK}} \mathbf{1}$) as a term that essentially behaves as the identity term $\lambda z. z$: when coming in head position, it simply vanishes and gives the control to its argument.
- The equality $f(n) = 0$ is false. In this case, we can think of u ($\Vdash_{\text{NK}} \top \Rightarrow \perp$) as a term that consumes its argument (whatever it is) and then backtracks to an earlier point in the computation.

Of course, this informal description is only an loose approximation of the actual behavior of the realizer $u \Vdash_{\text{NK}} f(n) = 0$ (which may considerably vary depending on the choice of $\perp\!\!\!\perp$), but it gives us the clue to fix the naive extraction method.

The idea is to apply the universal realizer $t_0 \Vdash_{\text{NK}} \exists^{\mathbb{N}} x f(x) = 0$ to the term $\lambda xy. y (\text{stop } x)$ that inserts a ‘breakpoint’ y before returning x . If the first component x is a correct witness, then the second component y will vanish and let the program return the correct answer.

If the first component x is incorrect, then y will issue a backtrack, and this until a correct witness has been found.

We can now formalize this intuition as follows:

Proposition 5.3. *For all $\pi \in \Pi$, the process $t_0 \star (\lambda xy. y(\text{stop } x)) \cdot \pi$ evaluates (in a finite number of steps) to a process of the form $\text{stop} \star \hat{n} \cdot \pi$ for some natural number $n \in \mathbb{N}$ such that $f(n) = 0$.*

Proof. Let us take a stack $\pi \in \Pi$ and work in the pole defined by

$$\perp\!\!\!\perp = \{p : \exists n \in \mathbb{N} (f(n) = 0 \text{ and } p \succ^* \text{stop} \star \hat{n} \cdot \pi)\}.$$

Writing $S = \{\pi\}$, we easily check that $\text{stop} \Vdash_{\text{NK}} \{n\} \Rightarrow \dot{S}$ for all $n \in \mathbb{N}$ such that $f(n) = 0$ (from the very definition of $\perp\!\!\!\perp$ and S). Let us now show that the term $\lambda xy. y(\text{stop } x)$ realizes the formula $\forall x (\{x\} \Rightarrow f(x) = 0 \Rightarrow \dot{S})$. For that, consider an arbitrary element of the falsity value of this formula, that is: a stack of the form $\hat{n} \cdot u \cdot \pi$ for some $n \in \mathbb{N}$ and $u \in |f(n) = 0|$. We have

$$\lambda xy. y(\text{stop } x) \star \hat{n} \cdot u \cdot \pi \succ^* u \star (\text{stop } \hat{n}) \cdot \pi.$$

To show that $u \star (\text{stop } \hat{n}) \cdot \pi \in \perp\!\!\!\perp$, we distinguish two cases:

- $f(n) = 0$. In this case, we have $\text{stop } \hat{n} \Vdash_{\text{NK}} \dot{S}$ (using the ‘type’ we gave to stop), hence $(\text{stop } \hat{n}) \star \pi \in \perp\!\!\!\perp$ and thus $(\text{stop } \hat{n}) \cdot \pi \in \|\mathbf{1}\| = \|f(n) = 0\|$ (by Lemma 5.2). Therefore $u \star (\text{stop } \hat{n}) \cdot \pi \in \perp\!\!\!\perp$.
- $f(n) \neq 0$. In this case, we have $(\text{stop } \hat{n}) \cdot \pi \in \|\top \Rightarrow \perp\| = \|f(n) = 0\|$, hence $u \star (\text{stop } \hat{n}) \cdot \pi \in \perp\!\!\!\perp$.

In both cases we deduce that $\lambda xy. y(\text{stop } x) \star \hat{n} \cdot u \cdot \pi \in \perp\!\!\!\perp$ by anti-evaluation, which finishes the proof that $\lambda xy. y(\text{stop } x) \Vdash_{\text{NK}} \forall x (\{x\} \Rightarrow f(x) = 0 \Rightarrow \dot{S})$. From the latter we deduce that $(\lambda xy. y(\text{stop } x)) \cdot \pi \in \|\exists^{\mathbb{N}} x f(x) = 0\|$, so that $t_0 \star (\lambda xy. y(\text{stop } x)) \cdot \pi \in \perp\!\!\!\perp$. \square

Remark 5.4. The simple (and reliable) extraction procedure presented above returns a correct witness without keeping track of the intermediate witnesses proposed by the realizer t_0 . A simple way to display them during the computation is to introduce an instruction `print` such that

$$\text{print} \star \hat{n} \cdot u \cdot \pi \succ u \star \pi \quad (n \in \mathbb{N}, u \in \Lambda, \pi \in \Pi)$$

while printing the natural number n on some output device (the second part of the specification of `print` being purely informal). From the only evaluation rule of `print`, we easily check that $\text{print} \Vdash_{\text{NK}} \forall x (\{x\} \Rightarrow \mathbf{1})$. It is then a straightforward exercise to adapt the proof of Prop. 5.3 when the process $t_0 \star (\lambda xy. y(\text{stop } x)) \cdot \pi$ is replaced by the process $t_0 \star (\lambda xy. \text{print } xy(\text{stop } x)) \cdot \pi$ that ultimately does the same job—while printing the intermediate results.

In section 8.4 we shall reinterpret the witness extraction method of Prop. 5.3 through a well-suited negative translation.

5.3. Independence of the witness w.r.t. the stack π . It is easy to see that the witness computed by the process $t_0 \star (\lambda xy. y(\text{stop } x)) \cdot \pi$ (in the sense of Prop. 5.3) does not actually depend on the stack π , provided we make the following ‘closed world’ assumptions:

- (1) The relation of (one step) evaluation \succ is defined as the union of the rules (GRAB), (PUSH), (CALL/CC), (RESUME), (SUCC), (REC-0) and (REC-S) (cf Sections 3.1 and 4.1). In particular, evaluation is deterministic.

- (2) The term t_0 contains no continuation constant k_π , that is: t_0 is a *proof-like term* according to the terminology of [19]. Note that this condition is automatically fulfilled when t_0 is a proof-term built in system PA2⁺.
- (3) The term **stop** is an extra instruction (with no evaluation rule).

To prove the desired independence result, we define an operation of *stack extension* for terms, stacks and processes as follows¹⁰. Given a fixed stack π_0 , we denote by $t\{\diamond := \pi_0\}$ (resp. $\pi\{\diamond := \pi_0\}$, $p\{\diamond := \pi_0\}$) the term t (resp. the stack π , the process p) in which every occurrence of the stack bottom \diamond is replaced by the stack π_0 , including inside continuation constants.

Formally, these operations are defined by:

$$\begin{aligned}
x\{\diamond := \pi_0\} &\equiv x \\
(\lambda x . t)\{\diamond := \pi_0\} &\equiv \lambda x . t\{\diamond := \pi_0\} \\
(tu)\{\diamond := \pi_0\} &\equiv t\{\diamond := \pi_0\}u\{\diamond := \pi_0\} \\
\kappa\{\diamond := \pi_0\} &\equiv \kappa \\
(k_\pi)\{\diamond := \pi_0\} &\equiv k_{(\pi\{\diamond := \pi_0\})} \\
\diamond\{\diamond := \pi_0\} &\equiv \pi_0 \\
(t \cdot \pi)\{\diamond := \pi_0\} &\equiv t\{\diamond := \pi_0\} \cdot \pi\{\diamond := \pi_0\} \\
(t \star \pi)\{\diamond := \pi_0\} &\equiv t\{\diamond := \pi_0\} \star \pi\{\diamond := \pi_0\}
\end{aligned} \tag{\kappa \in \mathcal{K}}$$

Note that when t is a proof-like term, we have $t\{\diamond := \pi_0\} \equiv t$. From assumption (1) we immediately get:

Lemma 5.5. *If $p \succ p'$, then $p\{\diamond := \pi_0\} \succ p'\{\diamond := \pi_0\}$ (for all $\pi_0 \in \Pi$).*

Proof. By case analysis on the evaluation rule using (1).

(The same result holds if we replace \succ by \succ^* .)

Let us now assume that t_0 is a proof-like term (assumption (2)) that is a universal realizer of the formula $\exists^{\mathbb{N}}x f(x) = 0$. From Prop. 5.3, we know that there is some $n \in \mathbb{N}$ such that $f(n) = 0$ and

$$t_0 \star (\lambda xy . y(\mathbf{stop} x)) \cdot \diamond \succ^* \mathbf{stop} \star \hat{n} \cdot \diamond .$$

But if we apply Lemma 5.5 with an arbitrary stack π , we thus get

$$t_0 \star (\lambda xy . y(\mathbf{stop} x)) \cdot \pi \succ^* \mathbf{stop} \star \hat{n} \cdot \pi$$

(using the fact that t_0 is a proof-like term, so that $t_0\{\diamond := \pi\} \equiv t_0$).

Since evaluation is deterministic and since the instruction **stop** has no evaluation rule, the answer produced by Prop. 5.3 with an arbitrary stack π is unique, and it is the same as if we take the stack $\pi \equiv \diamond$.

5.3.1. Adding other instructions. The property of independence of the witness w.r.t. the stack π crucially depends on the fact that evaluation is deterministic and substitutive w.r.t. the stack constant \diamond (in the sense of Lemma 5.5). However, it is sometimes useful to consider instructions whose evaluation rules break Lemma 5.5 (without breaking determinism of evaluation). An example of such an instruction is the instruction **quote** with the evaluation rule

$$\mathbf{quote} \star t \cdot \pi \succ t \star \hat{n}_\pi \cdot \pi ,$$

¹⁰For an account of the possible uses of this technique, see [10].

where n_π is the *code* of the stack π according to a fixed bijection between natural numbers and stacks. (Such an instruction is introduced in [17] to realize several forms of the axiom of choice.) If t_0 uses such an instruction, then the witness provided by Prop. 5.3 may actually depend on the stack π .

5.4. Extraction in the decidable case. The witness extraction procedure we presented in section 5.2 for Σ_1^0 -formulae can be generalized to any existential formula $\exists^{\mathbb{N}}x A(x)$ provided the predicate $A(x)$ is decidable, using a decision function expressed as a λ_c -term.

Formally, a *decision function* for the predicate $A(x)$ is a term $d_A \in \Lambda_c$ such that for all $n \in \mathbb{N}$, $u, v \in \Lambda_c$ and $\pi \in \Pi$ we have

$$d_A \star \hat{n} \cdot u \cdot v \cdot \pi \quad \succ^* \quad \begin{cases} u \star \pi & \text{if } \mathcal{M} \models A(n) \\ v \star \pi & \text{if } \mathcal{M} \not\models A(n) \end{cases}$$

(writing \mathcal{M} the full standard model of PA2). Intuitively, a decision function for the predicate $A(x)$ is a closed λ_c -term d_A such that for every natural number $n \in \mathbb{N}$, the applied term $d_A \hat{n}$ acts as a boolean value indicating whether the formula $A(n)$ holds or not in the full standard model of PA2.

Extracting a witness in this case also requires another ingredient to repudiate the wrong witnesses proposed by the realizer t_0 . Formally, we call a *function of conditional refutation* of the predicate $A(x)$ any term $r_A \in \Lambda_c$ such that

$$r_A \Vdash_{\text{NK}} \{n\} \Rightarrow \neg A(n)$$

for all $n \in \mathbb{N}$ such that $\mathcal{M} \not\models A(n)$. Intuitively, the purpose of a function of conditional refutation r_A is to provide a counter-realizer $t_A \hat{n} \Vdash_{\text{NK}} \neg A(n)$ that we shall oppose to the realizer $u \Vdash_{\text{NK}} A(n)$ coming with any wrong witness proposed by the realizer t_0 . Such terms r_A can be built for a very large class of formulae as we shall see in section 5.5.

Using the decision function d_A and the function of conditional refutation r_A , we now get a simple algorithm to perform witness extraction from a universal realizer $t_0 \Vdash_{\text{NK}} \exists^{\mathbb{N}}x A(x)$:

- (1) Extract $n \in \mathbb{N}$ and $u \Vdash_{\text{NK}} A(n)$ from the universal realizer t_0 .
- (2) Check whether $A(n)$ is true or not, using the decision function d_A .
 - If $A(n)$ is true, then return n (using the ‘stop’ instruction).
 - If $A(n)$ is false, then execute the realizer $r_A \hat{n} u \Vdash_{\text{NK}} \perp$ to backtrack.

In the language λ_c , this procedure is implemented by applying the universal realizer t_0 to the λ_c -term $\lambda xy. d_A x (\text{stop } x) (r_A x y)$ that does the expected job:

Proposition 5.6. *Let d_A and r_A be respectively a decision function and a function of conditional refutation for the predicate $A(x)$, and let t_0 be a universal realizer of the formula $\exists^{\mathbb{N}}x A(x)$. Then for all $\pi \in \Pi$, the process*

$$t_0 \star (\lambda xy. d_A x (\text{stop } x) (r_A x y)) \cdot \pi$$

evaluates (in a finite number of steps) to a process of the form $\text{stop} \star \hat{n} \cdot \pi$ for some natural number $n \in \mathbb{N}$ such that $A(n)$ is true in the full standard model.

Proof. Let us take a stack $\pi \in \Pi$ and work in the pole defined by

$$\perp = \{p : \exists n \in \mathbb{N} (\mathcal{M} \models A(n) \text{ and } p \succ^* \text{stop} \star \hat{n} \cdot \pi)\}.$$

Writing $S = \{\pi\}$, we easily check that $\text{stop} \Vdash_{\text{NK}} \{n\} \Rightarrow \dot{S}$ for all $n \in \mathbb{N}$ such that $\mathcal{M} \models A(n)$ (from the very definition of \perp and S). Let us now show that the term

$\lambda xy . d_A x (\text{stop } x) (r_A x y)$ realizes the formula $\forall x (\{x\} \Rightarrow A(x) \Rightarrow \dot{S})$. For that, consider an arbitrary element of the falsity value of this formula, that is: a stack of the form $\hat{n} \cdot u \cdot \pi$ for some $n \in \mathbb{N}$ and $u \in |A(n)|$. We have

$$\lambda xy . d_A x (\text{stop } x) (r_A x y) \star \hat{n} \cdot u \cdot \pi \succ^* d_A \star \hat{n} \cdot (\text{stop } \hat{n}) \cdot (r_A \hat{n} u) \cdot \pi.$$

To show that $d_A \star \hat{n} \cdot (\text{stop } \hat{n}) \cdot (r_A \hat{n} u) \cdot \pi \in \perp$, we distinguish two cases:

- $\mathcal{M} \models A(n)$. In this case we have

$$d_A \star \hat{n} \cdot (\text{stop } \hat{n}) \cdot (r_A \hat{n} u) \cdot \pi \succ^* (\text{stop } \hat{n}) \star \pi \succ \text{stop} \star \hat{n} \cdot \pi \in \perp$$

(using the fact that $\text{stop} \Vdash_{\text{NK}} \{n\} \Rightarrow \dot{S}$ when $\mathcal{M} \models A(n)$), from which we get $d_A \star \hat{n} \cdot (\text{stop } \hat{n}) \cdot (r_A \hat{n} u) \cdot \pi \in \perp$ by anti-evaluation.

- $\mathcal{M} \not\models A(n)$. In this case we have

$$d_A \star \hat{n} \cdot (\text{stop } \hat{n}) \cdot (r_A \hat{n} u) \cdot \pi \succ^* (r_A \hat{n} u) \star \pi \succ^* r_A \star \hat{n} \cdot u \cdot \pi \in \perp$$

since $\hat{n} \cdot u \cdot \pi \in \|\{n\} \Rightarrow A(n) \Rightarrow \perp\|$ and $r_A \in \|\{n\} \Rightarrow A(n) \Rightarrow \perp\|$ from the definition that r_A is a function of conditional refutation. By anti-evaluation we get: $d_A \star \hat{n} \cdot (\text{stop } \hat{n}) \cdot (r_A \hat{n} u) \cdot \pi \in \perp$.

In both cases we deduce that $\lambda xy . d_A x (\text{stop } x) (r_A x y) \star \hat{n} \cdot u \cdot \pi \in \perp$ by anti-evaluation, which finishes the proof that $\lambda xy . d_A x (\text{stop } x) (r_A x y)$ realizes the formula $\forall x (\{x\} \Rightarrow A(x) \Rightarrow \dot{S})$ in the pole \perp . From the latter, we immediately deduce that $(\lambda xy . d_A x (\text{stop } x) (r_A x y)) \cdot \pi \in \|\exists^{\mathbb{N}} x f(x) = 0\|$, from which we conclude that $t_0 \star (\lambda xy . d_A x (\text{stop } x) (r_A x y)) \cdot \pi \in \perp$. \square

5.4.1. *The particular case of Σ_1^0 -formulae.* In the case where the predicate $A(x)$ is of the form $A(x) \equiv f(x) = 0$ for some primitive recursive function symbol f , it is easy to implement a decision function d_A from a λ_c -term that actually computes f . Such a function d that tests whether $f(n) = 0$ for a given argument $n \in \mathbb{N}$ can even be characterized in terms of realizability as follows:

Lemma 5.7. *Let f be a primitive recursive function symbol. For every term $d \in \Lambda_c$, the following assertions are equivalent:*

- (1) d decides the predicate $A(x) \equiv f(x) = 0$;
- (2) $d \Vdash_{\text{NK}} \forall Z \forall^{\mathbb{N}} x (Z(0) \Rightarrow \forall y Z(s(y)) \Rightarrow Z(f(x)))$.

Proof. 1. \Rightarrow 2. Easily follows from the evaluation rules of the term d .

2. \Rightarrow 1. Let $n \in \mathbb{N}$, $u, v \in \Lambda_c$ and $\pi \in \Pi$. We distinguish two cases:

- $f(n) = 0$. We let $\perp = \{p \succ^* u \star \pi\}$ and define a function $F : \mathbb{N} \rightarrow \mathfrak{P}(\pi)$ by $F(0) = \{\pi\}$ and $F(p) = \emptyset$ for all $p > 0$. We easily check that $u \in |\dot{F}(0)|$, $v \in |\forall y \dot{F}(s(y))|$ and $\pi \in \|\dot{F}(f(n))\|$, hence $d \star \hat{n} \cdot u \cdot v \cdot \pi \in \perp$.
- $f(n) \neq 0$. We let $\perp = \{p \succ^* v \star \pi\}$ and define a function $F : \mathbb{N} \rightarrow \mathfrak{P}(\pi)$ by $F(0) = \emptyset$ and $F(p) = \{\pi\}$ for all $p > 0$. We again check that $u \in |\dot{F}(0)|$, $v \in |\forall y \dot{F}(s(y))|$ and $\pi \in \|\dot{F}(f(n))\|$, hence $d \star \hat{n} \cdot u \cdot v \cdot \pi \in \perp$. \square

The function of conditional refutation for the predicate $A(x) \equiv f(x) = 0$ is even easier to build: simply take the constant function $r_A \equiv \lambda z. z?$ (where $?$ is any λ_c -term possibly depending on z), using the fact that $\vdash_{\text{NK}} \lambda z. z? : f(n) \neq 0$ for all natural numbers n such that $f(n) \neq 0$. (Note that the term $\lambda z. z?$ does not depend on n .) In this case, the function of conditional refutation r_A can be replaced by the *conditional refutation* $\lambda z. z?$ that is a universal realizer of the formula $f(n) \neq 0$ for all natural numbers n such that $f(n) \neq 0$.

Given a term $d_f \in \Lambda_c$ that decides the predicate $f(x) = 0$, we can thus perform witness extraction from a universal realizer $t_0 \Vdash_{\text{NK}} \exists^{\mathbb{N}} x f(x) = 0$ using the process:

$$t_0 \star (\lambda xy. d_f x (\text{stop } x) (y?)) \star \pi$$

(whose second branch has been simplified). Note that this process is slightly more complex than the process presented in Prop. 5.3 that does not even need to consider a decision function to perform witness extraction from t_0 .

5.5. Existence of functions of conditional refutation. The existence of a function of conditional refutation can be shown for a wide class of predicates, and in particular for every predicate $A(x)$ that is expressed in the language of first-order arithmetic such as defined in the end of section 3.3 (replacing $\forall x (\text{nat}(x) \Rightarrow P)$ by $\forall^{\mathbb{N}} x P$ in the corresponding BNF).

Let us first recall that:

Proposition 5.8. *For every $k \geq 0$, there exists a closed proof-term R_k such that for every formula of the form*

$$A \equiv \exists^{\mathbb{N}} y_1 \forall^{\mathbb{N}} z_1 \cdots \exists^{\mathbb{N}} y_k \forall^{\mathbb{N}} z_k f(y_1, z_1, \dots, y_k, z_k) \neq 0,$$

if $\mathcal{M} \models A$, then $R_k \Vdash_{\text{NK}} A$.

Proof. The existence of such a proof-term R_k is an immediate consequence of Theorem 21 (p. 14) in [19]. Note that R_k only depends on k . \square

We also check that:

Proposition 5.9 (Existence of the prenex form). *If $A(x_1, \dots, x_p)$ is a formula of first-order arithmetic depending on p first-order variables x_1, \dots, x_p , then there exists a natural number $k \geq 0$ and a function symbol f of arity $p + 2k$ such that the formula*

$$A'(x_1, \dots, x_p) \equiv \exists^{\mathbb{N}} y_1 \forall^{\mathbb{N}} z_1 \cdots \exists^{\mathbb{N}} y_k \forall^{\mathbb{N}} z_k f(x_1, \dots, x_p, y_1, z_1, \dots, y_k, z_k) \neq 0$$

is logically equivalent to $A(x_1, \dots, x_p)$, in the sense that there are closed proof-terms u_1, u_2 such that

$$\begin{aligned} \vdash_{\text{NK}} u_1 : \forall^{\mathbb{N}} x_1 \cdots \forall^{\mathbb{N}} x_p (A(x_1, \dots, x_p) \Rightarrow A'(x_1, \dots, x_p)) \\ \vdash_{\text{NK}} u_2 : \forall^{\mathbb{N}} x_1 \cdots \forall^{\mathbb{N}} x_p (A'(x_1, \dots, x_p) \Rightarrow A(x_1, \dots, x_p)). \end{aligned}$$

Proof. This theorem is the reformulation in the type system of Fig. 1 and 3 of the existence of prenex forms in first-order arithmetic. \square

From Prop. 5.8 and 5.9 we deduce the following:

Proposition 5.10 (Existence of a conditional refutation). *If $A(x)$ is a formula of first-order arithmetic that only depends on a first-order variable x , then the predicate $A(x)$ has a function of conditional refutation r_A .*

Proof. From Prop. 5.9, there exists a formula

$$A'(x) \equiv \exists^{\mathbb{N}} y_1 \forall^{\mathbb{N}} z_1 \cdots \exists^{\mathbb{N}} y_k \forall^{\mathbb{N}} z_k f(x, y_1, z_1, \dots, y_k, z_k) \neq 0$$

with closed proof-terms u_1, u_2 such that:

$$\vdash_{\text{NK}} u_1 : \forall^{\mathbb{N}} x (A(x) \Rightarrow A'(x)) \quad \text{and} \quad \vdash_{\text{NK}} u_2 : \forall^{\mathbb{N}} x (A'(x) \Rightarrow A(x)).$$

It suffices to take $r_A \equiv \lambda x. u_2 x R_k$ (by Prop. 5.8). □

5.6. The method of the kamikaze. The witness extraction procedure presented in section 5.4 depends on two components: a function d_A deciding the predicate $A(x)$, and another function r_A that conditionally refutes the predicate $A(x)$. The critical component here is the decision function d_A , since the function r_A can be constructed for a wider class of formulæ, i.e. for all arithmetic formulæ (cf section 5.5).

In the case where we have a function of conditional refutation r_A but no decision function for the predicate $A(x)$ —typically, when $A(x)$ is a non atomic arithmetic formula—we can still extract a possibly infinite sequence of ‘witness proposals’ from the universal realizer $t_0 \Vdash_{\text{NK}} \forall^{\mathbb{N}} x A(x)$ by systematically repudiating every proposed witness using the function of conditional refutation r_A .

This extraction method, which we call the *method of the kamikaze*, consists to apply the universal realizer $t_0 \Vdash_{\text{NK}} \exists^{\mathbb{N}} x A(x)$ to the term $\lambda xy. \text{print } x (r_A xy)$ (using the ‘print’ instruction introduced in Remark 5.4), thus implementing in the language λ_c the following algorithm:

- (1) Extract $n \in \mathbb{N}$ and $u \Vdash_{\text{NK}} A(n)$ from the universal realizer t_0 .
- (2) Print n on some output device.
- (3) Try to backtrack by executing $r_A \hat{n} u$.

The crucial point here is that there is no warranty that the piece of code executed at step 3 will actually issue a backtrack, since we do not know whether $\neg A(n)$ is true. The only invariant we can ensure is the following: as long as the proposed witness n is incorrect, the refutation function r_A is applied in agreement with its specification, so that step 3. will issue a backtrack. But as soon as a correct witness n has been reached, the current process becomes ill-typed, and then anything may happen: the process may enter an infinite loop (possibly displaying other numbers) as it may crash, for instance due to a stack underflow (by evaluating an abstraction or one of the instructions α , κ_π , **print** in front of an empty stack), or due to the fact that **print** is evaluated in front of a stack which does not start with a primitive numeral.

Of course, the interest of the method is that the process that performs the blind extraction of the successive witnesses proposed by the universal realizer t_0 cannot go wrong until a correct witness has been reached. We can actually even show that this process eventually reaches a correct witness:

Proposition 5.11. *If t_0 is a universal realizer of $\exists^{\mathbb{N}}x A(x)$ and if r_A is a function of conditional refutation of the predicate $A(x)$, then for all stacks $\pi \in \Pi$ the process*

$$t_0 \star (\lambda xy. \text{print } x (r_A x y)) \cdot \pi$$

evaluates (in a finite number of steps) to a process of the form $\text{print } \star \hat{n} \cdot u \cdot \pi$, where $u \in \Lambda_c$ and where $n \in \mathbb{N}$ is such that $A(n)$ holds in the full standard model.

Proof. Let us take a stack $\pi \in \Pi$ and work in the pole defined by

$$\perp\!\!\!\perp = \{p : \exists n \in \mathbb{N} \exists u \in \Lambda_c (\mathcal{M} \models A(n) \wedge p \succ^* \text{print } \star \bar{n} \cdot u \cdot \pi)\}.$$

Set $S = \{\pi\}$. We first want to show that $\text{print } \star \hat{n} \cdot (r_A \hat{n} v) \cdot \pi \in \perp\!\!\!\perp$ for all $n \in \mathbb{N}$ and for all $v \in |A(n)|$. We distinguish the following two cases:

- $\mathcal{M} \models A(n)$. In this case we have $\text{print } \star \hat{n} \cdot (r_A \hat{n} v) \cdot \pi \in \perp\!\!\!\perp$ from the very definition of the pole $\perp\!\!\!\perp$.
- $\mathcal{M} \not\models A(n)$. In this case, we have

$$\text{print } \star \hat{n} \cdot (r_A \hat{n} v) \cdot \pi \succ (r_A \hat{n} v) \star \pi \succ^* r_A \star \hat{n} \cdot v \cdot \pi \in \perp\!\!\!\perp$$

from our assumption on r_A combined with the fact that $\mathcal{M} \not\models A(n)$. Hence we get $\text{print } \star \hat{n} \cdot (r_A \hat{n} v) \cdot \pi \in \perp\!\!\!\perp$ by anti-evaluation.

From this result we easily get

$$\lambda xy. \text{print } x (r_A x y) \Vdash_{\text{NK}} \forall x (\{x\} \Rightarrow A(x) \Rightarrow \dot{S})$$

and finally: $t_0 \star (M \lambda xy. \text{print } x (r_A x y)) \cdot \pi \in \perp\!\!\!\perp$. □

Let us note that the above proof relies in an essential way in the definition of a pole $\perp\!\!\!\perp$ that is not closed under evaluation, thus reflecting the fact that the process which performs kamikaze extraction is correct *up to some point* during evaluation. After this point has been reached—that is: when a correct witness has been printed—the realizability model gives us no invariant anymore about the execution of the current process, so that anything may happen.

6. AN EXAMPLE BASED ON THE MINIMUM PRINCIPLE

In this section, we give an example of witness extraction in the Σ_1^0 -case.

An important aspect of the witness extraction procedure described in Prop. 5.3 is that the universal realizer $t_0 \Vdash_{\text{NK}} \exists^{\mathbb{N}}x f(x) = 0$ does not need to be a proof-term in the sense of the type system of PA2⁺—it just needs to be a universal realizer in the sense of classical realizability. Indeed, the naive method that consists to extract the λ_c -term from the proof *as is* tends to produce highly inefficient code. On the other hand, many useful arithmetic lemmas have universal realizers that are much more compact (and much more efficient) than the realizers that would come from official proofs.

For this reason, it is reasonable to isolate such lemmas during the extraction process, and to replace their official proof-terms (i.e. coming from derivations in PA2⁺) by universal realizers built by hand. In what follows, we shall illustrate this point with the minimum principle.

6.1. Notations. In PA2^+ , it is convenient to define the ordering relation $x \leq y$ from Leibniz equality by letting

$$x \leq y \equiv \text{minus}(x, y) = 0,$$

where minus is the binary primitive recursive function defined by the equations

$$\begin{aligned} \text{minus}(x, 0) &= x \\ \text{minus}(0, s(y)) &= 0 \\ \text{minus}(s(x), s(y)) &= \text{minus}(x, y) \end{aligned}$$

Given a unary primitive recursive function symbol f , we express that f is a function from natural numbers to natural numbers with the formula

$$\text{Fun}(f) \equiv \forall^{\mathbb{N}} x \text{ nat}'(f(x)) \equiv \forall x (\{x\} \Rightarrow \forall Z ((\{f(x)\} \Rightarrow Z) \Rightarrow Z)).$$

It is easy to check that universal realizers of the formula $\text{Fun}(f)$ are precisely the closed λ_c -terms that compute the function f , namely:

Lemma 6.1. *Given a term $t \in \Lambda_c$, the following assertions are equivalent:*

- (1) For all $u \in \Lambda_c$, $\pi \in \Pi$: $t \star \widehat{n} \cdot u \cdot \pi \succ^* u \star \widehat{f(n)} \cdot \pi$ (i.e. t computes f)
- (2) $t \Vdash_{\text{NK}} \text{Fun}(f)$ (i.e. t universally realizes $\text{Fun}(f)$)

Proof. 1. \Rightarrow 2. immediately follows from the definitions of classical realizability.

2. \Rightarrow 1. Let us assume that $t \Vdash_{\text{NK}} \text{Fun}(f)$, and fix $n \in \mathbb{N}$, $u \in \Lambda_c$ and $\pi \in \Pi$. We define the pole $\perp = \{p : p \succ^* u \star \widehat{f(n)} \cdot \pi\}$ and the falsity value $S = \{\pi\}$, from which we easily check that $u \Vdash_{\text{NK}} \{f(n)\} \Rightarrow \dot{S}$. From our initial assumption, we have $t \Vdash_{\text{NK}} \{n\} \Rightarrow (\{f(n)\} \Rightarrow \dot{S}) \Rightarrow \dot{S}$, and thus $t \star \widehat{n} \cdot u \cdot \pi \in \perp$. \square

Finally, we use the shorthand $\langle x; y \rangle \equiv \lambda z. z x y$ to denote order pairs in λ_c , keeping in mind that this construction can be used to prove (or realize) both conjunctions and numeric existential quantifications.

6.2. The functional minimum principle. We now want to build a universal realizer of the formula expressing that a function from natural numbers to natural numbers reaches its minimum:

$$\text{MinPrinc} \equiv \text{Fun}(f) \Rightarrow \exists^{\mathbb{N}} x \forall^{\mathbb{N}} y (f(x) \leq f(y))$$

(Note that the premise $\text{Fun}(f)$ is crucial to prove/realize the result.) Since this formulation of the minimum principle is (classically) provable in PA2^+ , we could take any proof-term of it as a universal realizer. In this case however, it is much more interesting to build a universal realizer by hand.

For that, let us take a closed λ_c -term test_le that performs the comparison of two primitive natural numbers, in the sense that

$$\text{test_le} \star \widehat{n} \cdot \widehat{m} \cdot u \cdot v \cdot \pi \succ^* \begin{cases} u \star \pi & \text{if } n \leq m \\ v \star \pi & \text{otherwise} \end{cases}$$

for all $n, m \in \mathbb{N}$, $u, v \in \Lambda_c$ and $\pi \in \Pi$. (It is a straightforward exercise of programming to implement such a term in λ_c .)

Now, let us consider a closed λ_c -term min_aux such that

$$\begin{aligned} \text{min_aux} \star f \cdot k \cdot n \cdot m \cdot \pi \succ^* \\ \langle n, \lambda n'. f n' (\lambda m'. \text{test_le } m m' \mathbf{I} (k (\text{min_aux } f k n' m'))) \rangle \cdot \pi \end{aligned}$$

for all $f, k, n, m \in \Lambda_c$ and $\pi \in \Pi$. Intuitively, such a λ_c -term `min_aux` is a recursive function that takes the following arguments:

- A realizer $f \Vdash_{\text{NK}} \text{Fun}(f)$ (i.e. an implementation of f)
- A continuation $k \Vdash_{\text{NK}} \neg \exists^{\mathbb{N}} x \forall^{\mathbb{N}} y (f(x) \leq f(y))$ for backtracking.
- The current witness proposal n .
- The image $m = f(n)$ of the current witness proposal. (We keep this argument across the recursive call to avoid recomputing it later.)

When it is called with these arguments, the function `min_aux` returns an ordered pair $\langle n, h \rangle$ whose first component is the current witness proposal n , and whose second component is a function

$$h \equiv \lambda n'. f n' (\lambda m'. \text{test_le } m m' \mathbf{I} (k (\text{min_aux } f k n' m')))$$

that takes a natural number n' , computes its image $m' = f(n')$ and compares it with m . In the case where $m \leq m'$, the function h returns the identity term \mathbf{I} , which is an obvious realizer of $f(n) \leq f(n')$. In the case where $m' < m$, the function f backtracks using the continuation k , and recursively calls `min_aux` with n' as the new witness proposal, and m' as its image by f .

Note that there are several ways to implement the term `min_aux` in λ_c . For instance, we can let

$$\text{min_aux} \equiv \mathbf{Y} (\lambda r f k n m. \langle n, \lambda n'. f n' (\lambda m'. \text{test_le } m m' \mathbf{I} (k (r f k n' m')) \rangle),$$

where $\mathbf{Y} \equiv (\lambda yz. z(yy))(\lambda yz. z(yy))$ is Turing's fixpoint combinator¹¹; or we can simply introduce `min_aux` as an extra instruction with the desired evaluation rule. Whatever the way we implement `min_aux`, we can check that:

Lemma 6.2. *Writing $E \equiv \exists^{\mathbb{N}} x \forall^{\mathbb{N}} y f(x) \leq f(y)$, we have:*

$$\text{min_aux} \Vdash_{\text{NK}} \forall x (\text{Fun}(f) \Rightarrow \neg E \Rightarrow \{x\} \Rightarrow \{f(x)\} \Rightarrow E).$$

Proof. Fix a pole \perp and two realizers $f \Vdash_{\text{NK}} \text{Fun}(f)$ and $k \Vdash_{\text{NK}} \neg E$, and consider the property $\text{IH}(m)$ defined by

$$\begin{aligned} \text{IH}(m) : & \text{ for all } n \in \mathbb{N} \text{ s.t. } f(n) = m, \text{ for all } \pi \in \|E\| \\ & \text{ we have: } \text{min_aux} \star f \cdot k \cdot \widehat{n} \cdot \widehat{m} \cdot \pi \in \perp. \end{aligned}$$

We want to prove $\text{IH}(m)$ by well-founded induction on m . For that, let us fix $m \in \mathbb{N}$, assume that $\text{IH}(m')$ for all $m' < m$, and take $n \in \mathbb{N}$ such that $f(n) = m$ and $\pi \in \|E\|$. From the evaluation rules of `test_le`, we can derive that

$$\text{test_le } \widehat{m} \widehat{m'} \mathbf{I} (k (\text{min_aux } f k n' m')) \Vdash_{\text{NK}} f(n) \leq f(n')$$

for all $n', m' \in \mathbb{N}$ such that $m' = f(n')$, distinguishing cases depending on whether $m \leq m'$ or $m' < m$, and using the induction hypothesis $\text{IH}(m')$ in the second case. From this we successively get

$$\begin{aligned} & \lambda m'. \text{test_le } \widehat{m} \widehat{m'} \mathbf{I} (k (\text{min_aux } f k n' m')) \Vdash_{\text{NK}} \forall y (\{f(y)\} \Rightarrow f(n) \leq f(y)) \\ & \lambda n'. f n' (\lambda m'. \text{test_le } \widehat{m} \widehat{m'} \mathbf{I} (k (\text{min_aux } f k n' m'))) \Vdash_{\text{NK}} \forall^{\mathbb{N}} y (f(n) \leq f(y)) \\ & \langle n, \lambda n'. f n' (\lambda m'. \text{test_le } \widehat{m} \widehat{m'} \mathbf{I} (k (\text{min_aux } f k n' m'))) \rangle \Vdash_{\text{NK}} E \\ & \langle n, \lambda n'. f n' (\lambda m'. \text{test_le } \widehat{m} \widehat{m'} \mathbf{I} (k (\text{min_aux } f k n' m'))) \rangle \star \pi \in \perp \end{aligned}$$

hence `min_aux` $\star f \cdot k \cdot \widehat{n} \cdot \widehat{m} \cdot \pi \in \perp$, by anti-evaluation. \square

¹¹We use Turing's fixpoint combinator rather than Church's, since Turing's combinator is better suited for the call-by-name strategy.

Now we can set:

$$\text{min_princ} \equiv \lambda f . f \widehat{0} (\lambda m . \mathbf{c} (\lambda k . \text{min_aux } f k \widehat{0} m)).$$

Intuitively, this function takes an implementation of f , computes the image $m = f(0)$, captures the current continuation as k and then calls `min_aux` with 0 as the initial witness proposal (accompanied with its image $m = f(0)$).

Combining Lemma 6.2 and the property of adequacy (Prop. 3.10) with the derivable judgment

$$\begin{aligned} z : \forall x (\text{Fun}(f) \Rightarrow \neg E \Rightarrow \{x\} \Rightarrow \{f(x)\} \Rightarrow E) \\ \vdash_{\text{NK}} \lambda f . f \widehat{0} (\lambda m . \mathbf{c} (\lambda k . z f k \widehat{0} m)) : \text{MinPrinc} \end{aligned}$$

we immediately deduce that:

$$\text{min_princ} \Vdash_{\text{NK}} \text{Fun}(f) \Rightarrow \exists^{\mathbb{N}} x \forall^{\mathbb{N}} y (f(x) \leq f(y)).$$

6.3. A Σ_1^0 -consequence of the minimum principle. Let f and g be two functions from natural numbers to natural numbers. The minimum principle gives a simple argument to show the existence of a natural number x such that $f(x) \leq f(g(x))$, which is to take a point x where f reaches its minimum. In $\text{PA}2^+$, the argument is formalized as follows:

$$\begin{aligned} z : \text{MinPrinc}, f : \text{Fun}(f), g : \text{Fun}(g) \vdash_{\text{NK}} \\ z f (\lambda n h . \langle n, \check{g} h \rangle) : \exists^{\mathbb{N}} x (f(x) \leq f(g(x))) \end{aligned}$$

Considering implementations $\check{f} \Vdash_{\text{NK}} \text{Fun}(f)$ and $\check{g} \Vdash_{\text{NK}} \text{Fun}(g)$ of the functions f and g , we thus get a universal realizer of the following Σ_1^0 -formula:

$$\text{min_princ } \check{f} (\lambda n h . \langle n, \check{g} n h \rangle) \Vdash_{\text{NK}} \exists^{\mathbb{N}} x (f(x) \leq f(g(x)))$$

By Prop. 5.3, we know that the process

$$p_0 \equiv \text{min_princ } \check{f} (\lambda n h . \langle n, \check{g} n h \rangle) \star (\lambda x y . y (\text{stop } x)) \cdot \diamond$$

computes the desired witness (which depends of course on f and g).

6.4. Executing λ_c -code. Fig. 4 illustrates the execution of the above process p_0 in the particular case where f and g are given by

$$f(x) = |x - 1000| \quad \text{and} \quad g(x) = 2x + 1.$$

The process p_0 was executed using the `jivaro` head reduction machine [23], a small interpreter of Krivine's λ_c -calculus extended with many built-in primitives (mainly for arbitrary-precision arithmetic and string manipulation). We slightly altered the code of p_0 in order to print intermediate witness proposals, so that the actual code of p_0 is

$$p_0 \equiv \text{min_princ } \check{f} (\lambda n h . \langle n, \check{g} n h \rangle) \star (\lambda x y . \text{print } x y (\text{stop } x)) \cdot \diamond$$

where `print` is the instruction mentioned in Remark 5.4 p. 20.

As shown in the input script of Fig. 4, each component of the process p_0 is introduced as a new instruction given with its evaluation rule (using the command `Define`). Note that such definitions may be (mutually) recursive, which is the case here for the instructions `min_aux` and `min_snd`. The interest of using named instructions rather than anonymous λ_c -terms is that we can more easily track when each piece of the code comes into head position during execution.

<u>Input script</u>		
<code>Define I x = x ;;</code>		<code>(* Identity *)</code>
<code>Define pair x y z = z x y ;;</code>		<code>(* Pairing *)</code>
<code>Define test_le = int_le ;;</code>		<code>(* Alias for int_le primitive *)</code>
<code>(* Realizing the minimum principle *)</code>		
<code>Define min_aux f k n m = pair n (min_snd f k m) ;;</code>		
<code>Define min_snd f k m n' = f n' (\m' test_le m m' I (k (min_aux f k n' m')))) ;;</code>		
<code>Define min_princ f = f 0 (\n (callcc (\k min_aux f k 0 n))) ;;</code>		
<code>(* Take f(x) = n - 1000 and g(x) = 2x + 1 *)</code>		
<code>Define f n = int_le n 1000 (int_minus 1000 n) (int_minus n 1000) ;;</code>		
<code>Define g n = int_mult 2 n (\m int_succ m) ;;</code>		
<code>(* Universal realizer of \existsN x, f(x) <= f(g(x)) *)</code>		
<code>Define realizer = min_princ f (\n\h pair n (g n h)) ;;</code>		
<code>Trace On ;;</code>		
<code>(* Perform Sigma^0_1 witness extraction & print intermediate witnesses *)</code>		
<code>Eval realizer ; (\x\y print x y (stop x)) ;;</code>		
<u>Output</u>	<u>Evaluation statistics (instruction calls)</u>	
0	@ (PUSH)	419
1	λ (GRAB)	68
3		
7	int_le	23
15	pair	22
31	f	12
63	int_minus	12
127	g	11
255	int_mult	11
511	int_succ	11
1023		
0.01 s: stopped	min_aux	11
Final state: stop ; 1023	min_snd	11
	print	11
	test_le	11
	k_π (RESTORE)	10
	I	1
	callcc (SAVE)	1
	min_princ	1
	realizer	1
	stop	1

Figure 4: Example of witness extraction using the `jivaro` machine

The output given in Fig. 4 shows that during its execution, the process p_0 successively tries the following guesses for x :

$$x_0 = 0, \quad x_1 = 1, \quad x_2 = 3, \quad x_3 = 7, \quad x_5 = 15, \quad x_6 = 31, \\ x_7 = 63, \quad x_8 = 127, \quad x_9 = 255, \quad x_{10} = 511, \quad x_{11} = 1023.$$

Since the last guess ($x_{11} = 1023$) is a solution of the problem, the execution stops on the final state $\text{stop} \star \widehat{1023} \cdot \diamond$, with the form predicted by Prop. 5.3.

The choice of this particular sequence of guesses is explained as follows.

During the execution of the process p_0 , the proof of $\exists^{\mathbb{N}}x(f(x) \leq f(g(x)))$ uses the guess x_i produced by the minimum principle as well as the accompanying justification of the formula $\forall^{\mathbb{N}}y(f(x_i) \leq f(y))$ to build a realizer of $f(x_i) \leq f(g(x))$. But when the latter is executed, it invokes the accompanying justification, that actually compares the values of x_i and $g(x_i)$ by f . In the case where $f(g(x_i)) < f(x_i)$, the guess x_i was wrong, and the accompanying justification backtracks to the point where the minimum principle was invoked (using an embedded continuation k_π). When restarted, the minimum principle can then propose $x_{i+1} = g(x_i)$ as a new guess. As a consequence, the process p_0 produces its guesses $x_i = g^i(0)$ by iterating the function g until $f(x_i) \leq f(g(x_i))$.¹²

Note that this behavior is the same as the one we observe when treating the same example using Friedman’s method or its refinements [3]. Of course, this similarity is not a coincidence since Friedman’s translation is actually hard-wired in Krivine’s semantics (as already pointed out in [25]), and we shall come back to this point with more details in sections 7 and 8.

6.4.1. *Evaluation statistics.* Fig. 4 also provides some statistics giving how many times each instruction has been called during evaluation.

Not surprisingly, the most frequent operations are PUSH (419 times) and GRAB (68 times), the asymmetry between these coming from the fact that stack arguments are not only consumed by abstractions (GRAB), but also by the instructions used by the program, which may be primitive (`callcc`, `int_le`, etc.) or defined by the user (`pair`, `min_aux`, etc.)

We can also see that our hand-made implementation of the minimum principle is optimal: the number of calls to the function f as well as the number of comparisons of images (by f) of guesses (using the instruction `test_le`) are both minimal. Moreover, the `callcc` instruction is called once during the whole execution, thus creating a unique continuation constant k_π (where $|\pi| = 2$) that is used exactly 10 times (RESTORE), that is: once for each backtrack.

We also tested this example by replacing the hand-made realizer of the minimum principle with an actual proof of it (in PA2⁺). The observed behavior remains the same, but the proof-term is much bigger and its execution is quite inefficient, mainly due to the arithmetic reasoning involved in the induction underlying the proof of the principle. (In the hand-made realizer, induction is performed at the meta-theoretic level, and thus has no cost during execution.) We can also notice that depending the way we use classical logic in the proof of the minimum principle, the corresponding proof-term may invoke several times the `call/cc` instruction, or only once as in the hand-made realizer.

7. INTUITIONISTIC SECOND-ORDER ARITHMETIC

We now define a type system for intuitionistic second-order arithmetic (HA2), as well as a realizability model that closely follows the traditional Brouwer-Heyting-Kolmogorov interpretation. As in [25], we introduce a primitive form of conjunction (as a Cartesian product) and primitive forms of first- and second-order existential quantification (as infinitary unions).

¹²Note that although each guess x_i claims to be a point where f reaches its minimum (until the context proves it wrong and forces backtrack), none of them—including the last one—is such a point, since f takes its minimum for $x = 1000$.

7.1. The language of formulæ. The language of arithmetic expressions of HA2 is the same as for PA2 (Fig. 1), and it is equipped with the congruence $e \cong e'$ generated from the same equations (cf section 2.2). The language of formulæ is now the following:

$$\begin{array}{l} \text{Formulæ} \quad A, B ::= \text{null}(e) \mid \text{nat}(e) \mid X(e_1, \dots, e_k) \\ \quad \quad \quad \mid A \Rightarrow B \mid \forall x A \mid \forall X A \\ \quad \quad \quad \mid A \wedge B \mid \exists x A \mid \exists X A \end{array}$$

To the language of formulæ of PA2 (Fig. 1) we add:

- A new predicate symbol $\text{nat}(e)$ to give a type to the Peano-style numerals we shall introduce in the language of proof-terms.
- A primitive conjunction $A \wedge B$ that we shall interpret in the intuitionistic realizability model as a type of pairs.
- Primitive forms of first- and second-order existential quantification that will be interpreted in the model as infinitary unions (as in [25]).

In this setting, the units \top and \perp are defined with the shorthands $\top \equiv \exists ZZ$ and $\perp \equiv \forall ZZ$, whereas numeric quantifications are defined as

$$\begin{aligned} \forall^{\mathbb{N}}x A(x) &\equiv \forall x (\text{nat}(x) \Rightarrow A(x)) \\ \exists^{\mathbb{N}}x A(x) &\equiv \exists x (\text{nat}(x) \wedge A(x)) \end{aligned}$$

7.1.1. The congruence $A \cong A'$. The congruence $A \cong A'$ over the class of formulæ of HA2 is defined from the defining equations of the primitive recursive function symbols of the signature, plus the three equations

$$\text{null}(0) \cong \top \equiv \exists ZZ \quad \text{null}(s(e)) \cong \perp \equiv \forall ZZ$$

and

$$(\exists v A(v)) \Rightarrow B \cong \forall v (A(v) \Rightarrow B)$$

where v is any first- or second-order variable that does not occur free in B . We shall see that the second equation is not only consistent with the interpretation of existential quantifications as infinitary unions (cf section 7.4), but that it is also crucial to establish Prop. 8.6.

7.2. A type system for intuitionistic second-order arithmetic. We introduce an intuitionistic (and more traditional) proof system based on a judgment of the form $\Gamma \vdash_{\text{NJ}} t : A$, where the proof-term t is now formed in the pure λ -calculus enriched with the following constants: **pair** (pairing), **fst** (first projection), **snd** (second projection), **0** (zero), **s** (successor) and **rec** (recursor). In what follows we shall write $\langle t; u \rangle$ for the application **pair** $t u$, and denote by Λ the set of all closed proof-terms. Typing contexts are simply defined here as finite ordered lists of declarations of the form $\Gamma \equiv x_1 : A_1, \dots, x_n : A_n$ where x_1, \dots, x_n are pairwise distinct proof-variables.

The class of derivable judgments $\Gamma \vdash_{\text{NJ}} t : A$ is inductively defined from the rules of inference of Fig. 5 (writing $\forall^{\mathbb{N}}x A(x) \equiv \forall x (\text{nat}(x) \Rightarrow A(x))$). Note that there is no elimination rule for first- and second-order primitive existential quantification, since the desired elimination can be performed using the conversion rule $\forall v (A(v) \Rightarrow B) \cong (\exists v A(v)) \Rightarrow B$ (where $v \notin FV(B)$).

<u>The language of HA2</u>	
Formulæ	$A, B ::= X(e_1, \dots, e_k) \mid \text{null}(e) \mid \text{nat}(e)$ $\mid A \Rightarrow B \mid \forall x A \mid \forall X A \mid \exists x A \mid \exists X A$
Proof-terms	$t, u ::= x \mid \lambda x. t \mid tu \mid \text{pair}$ $\mid \text{fst} \mid \text{snd} \mid 0 \mid s \mid \text{rec}$
Contexts	$\Gamma ::= \emptyset \mid \Gamma, x : A$
<u>The congruence $A \cong A'$</u>	
$\text{null}(0) \cong \top$	$\text{null}(s(x)) \cong \perp$
	$(\exists v A) \Rightarrow B \cong \forall v (A \Rightarrow B) \quad (v \notin FV(B))$
<u>Abbreviations</u>	
$\top \equiv \exists Z Z$	$\forall^{\mathbb{N}} x A(x) \equiv \forall x (\text{nat}(x) \Rightarrow A(x))$
$\perp \equiv \forall Z Z$	$\exists^{\mathbb{N}} x A(x) \equiv \exists x (\text{nat}(x) \wedge A(x))$
	$e = e' \equiv \forall Z (Z(e) \Rightarrow Z(e'))$
<u>Typing rules</u>	
$\overline{\Gamma \vdash_{\text{NJ}} x : A} \quad (x:A) \in \Gamma$	$\frac{\Gamma \vdash_{\text{NJ}} t : A}{\Gamma \vdash_{\text{NJ}} t : A'} \quad A \cong A'$
$\overline{\Gamma \vdash_{\text{NJ}} \text{pair} : A \Rightarrow B \Rightarrow A \wedge B}$	
$\overline{\Gamma \vdash_{\text{NJ}} \text{fst} : A \wedge B \Rightarrow A}$	$\overline{\Gamma \vdash_{\text{NJ}} \text{snd} : A \wedge B \Rightarrow B}$
$\overline{\Gamma \vdash_{\text{NJ}} 0 : \text{nat}(0)}$	$\overline{\Gamma \vdash_{\text{NJ}} s : \forall^{\mathbb{N}} x \text{nat}(s(x))}$
$\overline{\Gamma \vdash_{\text{NJ}} \text{rec} : \forall Z (Z(0) \Rightarrow \forall^{\mathbb{N}} y (Z(y) \Rightarrow Z(s(y))) \Rightarrow \forall^{\mathbb{N}} x Z(x))}$	
$\frac{\Gamma, x : A \vdash_{\text{NJ}} t : B}{\Gamma \vdash_{\text{NJ}} \lambda x. t : A \Rightarrow B}$	$\frac{\Gamma \vdash_{\text{NJ}} t : A \Rightarrow B \quad \Gamma \vdash_{\text{NJ}} u : A}{\Gamma \vdash_{\text{NJ}} tu : B}$
$\frac{\Gamma \vdash_{\text{NJ}} t : A}{\Gamma \vdash_{\text{NJ}} t : \forall x A} \quad x \notin FV(\Gamma)$	$\frac{\Gamma \vdash_{\text{NJ}} t : \forall x A}{\Gamma \vdash_{\text{NJ}} t : A\{x := e\}}$
$\frac{\Gamma \vdash_{\text{NJ}} t : A}{\Gamma \vdash_{\text{NJ}} t : \forall X A} \quad X \notin FV(\Gamma)$	$\frac{\Gamma \vdash_{\text{NJ}} t : \forall X A}{\Gamma \vdash_{\text{NJ}} t : A\{X(x_1, \dots, x_k) := B\}}$
$\frac{\Gamma \vdash_{\text{NJ}} t : A\{x := e\}}{\Gamma \vdash_{\text{NJ}} t : \exists x A}$	$\frac{\Gamma \vdash_{\text{NJ}} t : A\{X(x_1, \dots, x_k) := B\}}{\Gamma \vdash_{\text{NJ}} t : \exists X A}$

Figure 5: Intuitionistic second-order arithmetic (HA2)

The type system of HA2 is expressive enough to provide typable proof-terms for all the theorems of intuitionistic second-order arithmetic. (The specific axioms of arithmetic are treated the same way as in PA2.)

7.3. Weak reduction and inner reduction. Proof-terms of HA2 are equipped with a binary relation of one-step *weak reduction* written $t \succ_w t'$ and defined from the rules

$$\frac{}{(\lambda x. t)u \succ_w t\{x := u\}} \quad \frac{}{\text{rec } u_0 u_1 0 \succ_w u_0} \quad \frac{}{\text{rec } u_0 u_1 (\text{s } t) \succ_w u_1 t} \quad \frac{}{\text{rec } u_0 u_1 t \succ_w u_0 t}$$

$$\frac{}{\text{fst } \langle t_1; t_2 \rangle \succ_w t_1} \quad \frac{}{\text{snd } \langle t_1; t_2 \rangle \succ_w t_2} \quad \frac{t \succ_w t'}{tu \succ_w t'u} \quad \frac{u \succ_w u'}{tu \succ_w tu'}$$

Note that weak reduction is allowed both in the left- and right hand-side of applications, but not below λ -abstraction (i.e. we disable the ξ -rule of λ -calculus). We write \succ_w^* the reflexive-transitive closure of one step weak reduction.

Lemma 7.1. *If $t \succ_w t'$, then $t\{x := u\} \succ_w t'\{x := u\}$ (for all terms u).*

Proof. By induction on the derivation of $t \succ_w t'$. □

Complementarily to the notion of weak reduction, we also define a relation of *inner reduction* written $t \succ_i t'$ from the rules:

$$\frac{t \succ_w t'}{\lambda x. t \succ_i \lambda x. t'} \quad \frac{t \succ_i t'}{tu \succ_i t'u} \quad \frac{u \succ_i u'}{tu \succ_i t'u'} \quad \frac{t \succ_i t'}{\lambda x. t \succ_i \lambda x. t'}$$

The reflexive-transitive closure of the relation of inner reduction is written \succ_i^* while its reflexive-symmetric-transitive closure is written $=_i$.

The union of both relations \succ_w and \succ_i is the ordinary relation of one step reduction, written \succ . By the standard method of parallel reductions we get:

Proposition 7.2. *The relation \succ is confluent.*

We now want to deduce from this proposition a result of confluence for weak reduction modulo inner reductions. For that, we first need to show that inner reductions can be postponed, in this sense that any finite sequence of (weak and inner) reductions can be decomposed into a finite sequence of weak reductions followed by a finite sequence of inner reductions. Following Takahashi [28], we shall prove this result by introducing a notion of *parallel inner reduction*, written $t \succ_I t'$ and defined from the rules:

$$\frac{}{t \succ_I t} \quad \frac{t \succ_w t'}{\lambda x. t \succ_I \lambda x. t'} \quad \frac{t \succ_I t' \quad u \succ_I u'}{tu \succ_I t'u'} \quad \frac{t \succ_I t'}{\lambda x. t \succ_I \lambda x. t'}$$

From this definition it is clear that $(\succ_i) \subseteq (\succ_I) \subseteq (\succ_i^*)$, so that $(\succ_i^*) = (\succ_I^*)$. We first check that parallel inner reduction enjoys the expected property of substitutivity:

Lemma 7.3. *If $t \succ_I t'$ and $u \succ_I u'$, then $t\{x := u\} \succ_I t'\{x := u'\}$.*

Proof. By induction on the derivation of $t \succ_I t'$. □

Proposition 7.4. *If $t \succ_I t' \succ_w u$, then $t \succ_w^+ u_0 \succ_I u$ for some term u_0 .*

Proof. By induction on the derivation of $t' \succ_w u$. □

Corollary 7.5 (Postponement). *If $t \succ^* u$, then $t \succ_w^* u_0 \succ_i^* u$ for some u_0 .*

Proof. We first show that if $t \succ_I t' \succ_w^* u$, then $t \succ_w^* u_0 \succ_I u$ for some u_0 , by induction on the number of reduction steps in $t' \succ_w^* u$ using Prop. 7.4. From this we deduce the desired property by induction on the number of reduction steps in $t \succ^* u$, using the fact that $(\succ_I^*) = (\succ_i^*)$. \square

From Prop. 7.2 and Corollary 7.5 we immediately get:

Proposition 7.6 (Confluence of \succ_w modulo $=_i$). *It $t \succ_w^* t_1$ and $t \succ_w^* t_2$, then there are terms t'_1 and t'_2 such that $t_1 \succ_w^* t'_1$, $t_2 \succ_w^* t'_2$ and $t'_1 =_i t'_2$.*

7.4. The intuitionistic realizability model. We now build a simple realizability model for the type system defined above, in which formulæ are interpreted as *saturated sets* of terms, that is, as sets of closed proof-terms $S \subseteq \Lambda$ such that both conditions $t \succ_w t'$ and $t' \in S$ imply $t \in S$. The set of all saturated sets is written **SAT**.

Here, a *valuation* is a function ρ whose domain is a finite set of (first- and second-order) variables, such that:

- $\rho(x) \in \mathbb{N}$ for every first-order variable $x \in \text{dom}(\rho)$;
- $\rho(X) : \mathbb{N}^k \rightarrow \mathbf{SAT}$ for every k -ary second-order variable $X \in \text{dom}(\rho)$.

Parametric expressions, formulæ and contexts are defined as before. Every closed parametric formula $A[\rho]$ is interpreted as a saturated set $\llbracket A[\rho] \rrbracket \in \mathbf{SAT}$ that is defined by the expected equations

$$\begin{aligned} \llbracket X(e_1, \dots, e_k)[\rho] \rrbracket &= \rho(X)(\mathbf{Val}(e_1[\rho]), \dots, \mathbf{Val}(e_k[\rho])) \\ \llbracket \text{null}(e)[\rho] \rrbracket &= \begin{cases} \Lambda & \text{if } \mathbf{Val}(e[\rho]) = 0 \\ \emptyset & \text{if } \mathbf{Val}(e[\rho]) \neq 0 \end{cases} \\ \llbracket \text{nat}(e)[\rho] \rrbracket &= \{t \in \Lambda : t \succ_w^* s^n 0, \text{ where } n = \mathbf{Val}(e[\rho])\} \\ \llbracket (A \Rightarrow B)[\rho] \rrbracket &= \{t \in \Lambda : \forall u \in \llbracket A[\rho] \rrbracket tu \in \llbracket B[\rho] \rrbracket\} \\ \llbracket (A \wedge B)[\rho] \rrbracket &= \{t \in \Lambda : \exists t_1 \in \llbracket A[\rho] \rrbracket \exists t_2 \in \llbracket B[\rho] \rrbracket t \succ_w^* \langle t_1; t_2 \rangle\} \\ \llbracket (\forall x A)[\rho] \rrbracket &= \bigcap_{n \in \mathbb{N}} \llbracket A[\rho; x \leftarrow n] \rrbracket & \llbracket (\forall X A)[\rho] \rrbracket &= \bigcap_{F: \mathbb{N}^k \rightarrow \mathbf{SAT}} \llbracket A[\rho; X \leftarrow F] \rrbracket \\ \llbracket (\exists x A)[\rho] \rrbracket &= \bigcup_{n \in \mathbb{N}} \llbracket A[\rho; x \leftarrow n] \rrbracket & \llbracket (\exists X A)[\rho] \rrbracket &= \bigcup_{F: \mathbb{N}^k \rightarrow \mathbf{SAT}} \llbracket A[\rho; X \leftarrow F] \rrbracket \end{aligned}$$

In what follows, we shall write $t \Vdash_{\text{NJ}} A[\rho]$ for $t \in \llbracket A[\rho] \rrbracket$.

Lemma 7.7. *If A and A' are two formulæ of HA2 such that $A \cong A'$, then for all valuations ρ closing A and A' we have $\llbracket A[\rho] \rrbracket = \llbracket A'[\rho] \rrbracket$.*

7.5. Adequacy. Given a substitution σ and a closed parametric context $\Gamma[\rho]$, we write $\sigma \Vdash_{\text{NJ}} \Gamma[\rho]$ when $\text{dom}(\Gamma) \subseteq \text{dom}(\sigma)$ and $\sigma(x) \Vdash_{\text{NJ}} A[\rho]$ for all $(x : A) \in \Gamma$. We say that:

- A judgment $\Gamma \vdash_{\text{NJ}} t : A$ is *sound* when for all valuations ρ and for all substitutions σ such that $\sigma \Vdash_{\text{NJ}} \Gamma[\rho]$, we have $t[\sigma] \Vdash_{\text{NJ}} A[\rho]$.

- An inference rule $\frac{P_1 \dots P_n}{C}$ (where P_1, \dots, P_n and C are typing judgments) is *sound* when the soundness of its premises P_1, \dots, P_n (in the above sense) implies the soundness of its conclusion C .

Proposition 7.8 (Adequacy). *The typing rules of Fig. 5 are sound.*

From this result, we immediately get:

Proposition 7.9 (Witness property). *If $\vdash_{\text{NJ}} t : \exists^{\mathbb{N}} x \text{ null}(f(x))$, then there are a number $n \in \mathbb{N}$ and a closed term u such that $f(n) = 0$ and $t \succ_w^* \langle s^n 0; u \rangle$.*

Proof. From the definition of the denotation of the formula $\exists^{\mathbb{N}} x \text{ null}(f(x)) \equiv \exists x (\text{nat}(x) \wedge \text{null}(f(x)))$ in the realizability model, we know that there are a number $n \in \mathbb{N}$ and a term $u \in \Lambda$ such that $t \succ_w^* \langle s^n 0; u \rangle$ and $u \in \llbracket \text{null}(f(n)) \rrbracket$. Which means that the denotation $\llbracket \text{null}(f(n)) \rrbracket$ is inhabited, so that $f(n) = 0$ (by definition of the interpretation of the predicate null). \square

8. THE NEGATIVE TRANSLATION

8.1. Translating formulæ. We now define a negative translation of the formulæ of PA2⁺ (Fig. 1 and 3) into formulæ of HA2 (Fig. 5). We do not consider the usual double negation translation, but Streicher and Oliva’s negative translation [25], that is designed to mimic Krivine’s realizability in intuitionistic logic. Technically, this translation is parameterized by a fixed formula R (of HA2) that is intended to represent the pole \perp . In what follows, we write $\neg_R A$ as a shorthand for $A \Rightarrow R$.

Every formula A of PA2⁺ is translated as two formulæ of HA2, written $A^{\neg\neg}$ and A^\perp . Intuitively, the intuitionistic formula A^\perp represents the type of stacks facing a classical proof of A ; it is mainly built using the connective \wedge (representing the operation of conjoining) and from the two primitive forms of existential quantifications in HA2 (corresponding to universal quantification from the point of view of stacks). The intuitionistic formula $A^{\neg\neg}$ —that represents the type of classical proofs of A —is uniformly defined by $A^{\neg\neg} \equiv \neg_R A^\perp$. Formally:

Definition 8.1 (Definition of the negative translation). The formula A^\perp is defined by induction on A by the equations

$$\begin{aligned} (X(e_1, \dots, e_k))^\perp &\equiv X(e_1, \dots, e_k) & (\text{null}(e))^\perp &\equiv \text{null}(\text{neg}(e)) \\ (A \Rightarrow B)^\perp &\equiv A^{\neg\neg} \wedge B^\perp & (\forall x A)^\perp &\equiv \exists x A^\perp \\ (\{e\} \Rightarrow B)^\perp &\equiv \text{nat}(e) \wedge B^\perp & (\forall X A)^\perp &\equiv \exists X A^\perp \end{aligned}$$

(using the unary function ‘neg’ defined in section 2.1), whereas the formula $A^{\neg\neg}$ is defined as $A^{\neg\neg} \equiv \neg_R A^\perp \equiv A^\perp \Rightarrow R$.

Remark 8.2. Notice that through this translation, we have

$$(\forall v A(v))^{\neg\neg} \equiv \exists v A(v)^\perp \Rightarrow R \cong \forall v (A(v)^\perp \Rightarrow R) \equiv \forall v (A(v)^{\neg\neg}),$$

using the specific commutation rules of HA2. These conversions are crucial for the translation of the introduction and elimination rules of first- and second-order universal quantifications in the proof of Prop. 8.6.

We first check that the translations $A \mapsto A^\perp$ and $A \mapsto A^{\neg\neg}$ are substitutive:

Lemma 8.3 (Substitutivity). *For all arithmetic expressions e and for all formulae A and B of $PA2^+$:*

- (1) $(A\{x := e\})^\perp \equiv A^\perp\{x := e\}$
- (2) $(A\{x := e\})^{\neg\neg} \equiv A^{\neg\neg}\{x := e\}$
- (3) $(A\{X(x_1, \dots, x_k) := B\})^\perp \equiv A^\perp\{X(x_1, \dots, x_k) := B^\perp\}$
- (4) $(A\{X(x_1, \dots, x_k) := B\})^{\neg\neg} \equiv A^{\neg\neg}\{X(x_1, \dots, x_k) := B^\perp\}$

Proof. Item 1 is proved by induction on A , and item 2 immediately follows from item 1 (since $A^{\neg\neg} \equiv A^\perp \Rightarrow R$). The same for item 3 and item 4. \square

It is a simple exercise to check that:

Lemma 8.4. *If $A \cong A'$ ($PA2^+$), then $A^\perp \cong A'^\perp$ and $A^{\neg\neg} \cong A'^{\neg\neg}$ ($HA2$).*

Proof. This is obvious for the defining equations of function symbols (that are the same in both systems) since the translation does not affect arithmetic expressions. We only have to check that

$$(\text{null}(s(e)))^\perp \equiv \text{null}(\text{neg}(s(e))) \cong \text{null}(0) \cong \exists ZZ \equiv (\forall ZZ)^\perp \equiv (\perp)^\perp.$$

The rest of the proof proceeds by a straightforward induction. \square

We finally extend the translation $A \mapsto A^{\neg\neg}$ to a translation $\Gamma \mapsto \Gamma^{\neg\neg}$ that transforms any context Γ of $PA2^+$ into a context $\Gamma^{\neg\neg}$ of $HA2$. This translation is defined by induction on the length of Γ as follows:

$$\begin{aligned} \emptyset^{\neg\neg} &\equiv \emptyset \\ (\Gamma, x : A)^{\neg\neg} &\equiv \Gamma^{\neg\neg}, x : A^{\neg\neg} \\ (\Gamma, x : \{e\})^{\neg\neg} &\equiv \Gamma^{\neg\neg}, x : \text{nat}(e). \end{aligned}$$

8.2. CPS-translating terms and stacks. To define the translation of proof-terms, we introduce the convenient shorthand

$$\text{let } \langle x; y \rangle = u \text{ in } t \equiv (\lambda xy. t) (\text{fst } u) (\text{snd } u) \quad (\text{'destructing let'})$$

We first define a translation $t \mapsto t^*$ from proof-terms of $PA2^+$ (Fig. 1 and 3) into proof-terms of $HA2$ (Fig. 5). We will later extend this translation to continuation constants k_π and stacks. Formally:

Definition 8.5 (Translation of proof-terms). We associate to every proof-term of $PA2^+$ a proof-term t^* of $HA2$ that is inductively defined by:

$$\begin{aligned} x^* &\equiv x \\ (tu)^* &\equiv \lambda k. t^* \langle u^*; k \rangle \\ (\lambda x. t)^* &\equiv \lambda k. \text{let } \langle x; k' \rangle = k \text{ in } t^* k' \\ (\mathfrak{c})^* &\equiv \lambda k. \text{let } \langle x; k' \rangle = k \text{ in } x \langle (\lambda k''. \text{let } \langle y; _ \rangle = k'' \text{ in } y k'); k' \rangle \\ (\hat{n})^* &\equiv s^n 0 \\ (s)^* &\equiv \lambda k. \text{let } \langle x; k' \rangle = k \text{ in} \\ &\quad \text{let } \langle y; k'' \rangle = k' \text{ in } y \langle s x; k'' \rangle \\ (\text{rec})^* &\equiv \lambda k. \text{let } \langle z_0; k' \rangle = k \text{ in} \\ &\quad \text{let } \langle z_1; k'' \rangle = k' \text{ in} \\ &\quad \text{let } \langle x; k''' \rangle = k'' \text{ in } \text{rec } z_0 (\lambda x' y k_0. z_1 \langle x'; \langle \lambda k_1. y k_1; k_0 \rangle \rangle) x k''' \end{aligned}$$

Notice how the destructing `let` is used to mimic the destruction of the stack represented by the continuation variable k . Also note that the translation of the constant \widehat{n} does not start with a continuation abstraction $\lambda k \dots$, which reflects the fact that this construct is not intended to appear in head position.

We can now prove the following:

Proposition 8.6 (Correctness w.r.t. typing). *If $\Gamma \vdash_{\text{NK}} t : A$ in $PA2^+$, then $\Gamma^{\neg\neg} \vdash_{\text{NJ}} t^* : A^{\neg\neg}$ (in the sense of Fig. 5).*

Proof. By induction on the derivation, distinguishing cases according to the last applied rule. We first treat the cases of the rules of Fig. 1.

- **Axiom.Immediate**, since $x^* \equiv x$.
- **Conversion**. Immediately follows from Lemma 8.4.
- \Rightarrow -intro. The desired judgment comes from the following derivation:

$$\frac{\frac{\frac{\frac{\frac{\vdots \text{ (IH)}}{\Gamma, x : A^{\neg\neg} \vdash_{\text{NJ}} t^* : B^{\neg\neg}}{\Gamma^{\neg\neg}, k : A^{\neg\neg} \wedge B^\perp, x : A^{\neg\neg}, k' : B^\perp \vdash_{\text{NJ}} t^* : B^\perp \Rightarrow R}}{\Gamma^{\neg\neg}, k : A^{\neg\neg} \wedge B^\perp, x : A^{\neg\neg}, k' : B^\perp \vdash_{\text{NJ}} t^* k' : R}}{\Gamma^{\neg\neg}, k : A^{\neg\neg} \wedge B^\perp \vdash_{\text{NJ}} \text{let } \langle x; k' \rangle = k \text{ in } t^* k' : R}}{\Gamma^{\neg\neg} \vdash_{\text{NJ}} \underbrace{\lambda k. \text{let } \langle x; k' \rangle = k \text{ in } t^* k' : (A \Rightarrow B)^{\neg\neg}}_{(\lambda x. t)^*}}$$

(In the derivation above, we omit obvious branches and indicate uses of the admissible rule of weakening with a double bar.)

- \Rightarrow -elim. The desired judgment comes from the following derivation:

$$\frac{\frac{\frac{\frac{\vdots \text{ (IH)}}{\Gamma^{\neg\neg} \vdash_{\text{NJ}} t^* : (A \Rightarrow B)^{\neg\neg}}{\Gamma^{\neg\neg}, k : B^\perp \vdash_{\text{NJ}} t^* : A^{\neg\neg} \wedge B^\perp \Rightarrow R}}{\Gamma^{\neg\neg}, k : B^\perp \vdash_{\text{NJ}} t^* \langle u^*; k \rangle : A^{\neg\neg} \wedge B^\perp}}{\frac{\frac{\frac{\frac{\vdots \text{ (IH)}}{\Gamma^{\neg\neg} \vdash_{\text{NJ}} u^* : A^{\neg\neg}}{\Gamma^{\neg\neg}, k : B^\perp \vdash_{\text{NJ}} u^* : A^{\neg\neg}}}{\Gamma^{\neg\neg}, k : B^\perp \vdash_{\text{NJ}} t^* \langle u^*; k \rangle : R}}{\Gamma^{\neg\neg} \vdash_{\text{NJ}} \underbrace{\lambda k. t^* \langle u^*; k \rangle : B^{\neg\neg}}_{(tu)^*}}}$$

- \forall -intro (1st order). The desired judgment comes from the derivation

$$\frac{\frac{\frac{\frac{\vdots \text{ (IH)}}{\Gamma^{\neg\neg} \vdash_{\text{NJ}} t^* : A^{\neg\neg}}{\Gamma^{\neg\neg} \vdash_{\text{NJ}} t^* : \forall x (A^{\neg\neg})}}{\Gamma^{\neg\neg} \vdash_{\text{NJ}} t^* : (\forall x A)^{\neg\neg}} \text{ (Remark 8.2)}}{\Gamma^{\neg\neg} \vdash_{\text{NJ}} t^* : (\forall x A)^{\neg\neg}}$$

- \forall -elim (1st order). The desired judgment comes from the derivation

$$\frac{\frac{\frac{\frac{\frac{\vdots \text{ (IH)}}{\Gamma^{\neg\neg} \vdash_{\text{NJ}} t^* : (\forall x A)^{\neg\neg}}{\Gamma^{\neg\neg} \vdash_{\text{NJ}} t^* : \forall x (A^{\neg\neg})}}{\Gamma^{\neg\neg} \vdash_{\text{NJ}} t^* : A^{\neg\neg} \{x := e\}} \text{ (Remark 8.2)}}{\Gamma^{\neg\neg} \vdash_{\text{NJ}} t^* : (A \{x := e\})^{\neg\neg}} \text{ (Lemma 8.3)}}{\Gamma^{\neg\neg} \vdash_{\text{NJ}} t^* : (A \{x := e\})^{\neg\neg}}$$

- $\{-\} \Rightarrow$ -elim-2. The desired judgment comes from the derivation:

$$\frac{\frac{\frac{\vdots \text{ (IH)}}{\Gamma^{\neg\neg} \vdash_{\text{NJ}} t^* : (\{n\} \Rightarrow B)^{\neg\neg}}{\Gamma^{\neg\neg}, k : B^\perp \vdash_{\text{NJ}} t^* : \text{nat}(n) \wedge B^\perp \Rightarrow R} \quad \frac{\Gamma^{\neg\neg}, k : B^\perp \vdash_{\text{NJ}} s^n 0 : \text{nat}(n)}{\Gamma^{\neg\neg}, k : B^\perp \vdash_{\text{NJ}} \langle s^n 0; k \rangle : \text{nat}(n) \wedge B^\perp}}{\frac{\Gamma^{\neg\neg}, k : B^\perp \vdash_{\text{NJ}} t^* \langle s^n 0; k \rangle : R}{\Gamma^{\neg\neg} \vdash_{\text{NJ}} \underbrace{\lambda k . t^* \langle s^n 0; k \rangle}_{(t \hat{n})^*} : B^{\neg\neg}}}$$

The cases of s and rec are left to the reader. \square

8.2.1. *Extending the translation to the full λ_c -calculus.* We now extend the translation $t \mapsto t^*$ defined on the proof-terms of PA2^+ into a full translation of the λ_c -calculus. For that, we close the set of instructions \mathcal{K} by letting

$$\mathcal{K} = \{\mathfrak{c}; s; \text{rec}; \text{stop}\} \cup \{\hat{n} : n \in \mathbb{N}\},$$

and we close the relation of evaluation \succ by defining it as the union of the rules (GRAB), (PUSH), (CALL/CC), (RESUME), (SUCC), (REC-0) and (REC-S).

Formally, we define by mutual induction on t and π two translations $t \mapsto t^*$ (where t now ranges over all terms of the λ_c -calculus) and $\pi \mapsto \pi^*$ by adding to the equations of Def. 8.5 the following:

$$\begin{aligned} (\mathfrak{k}_\pi)^* &\equiv \lambda k . \text{let } \langle x; _ \rangle = k \text{ in } x \pi^* & (\diamond)^* &\equiv 0 \\ \text{stop}^* &\equiv \lambda z . z & (t \cdot \pi)^* &\equiv \langle t^*; \pi^* \rangle \end{aligned}$$

Stacks are translated here in the obvious way, that is: as finite lists. Note that the bottom of the stack \diamond could be translated by any closed term as well: it has no evaluation rule, and it is not involved in the type system of PA2 . On the other hand, we translate stop as the identity term, and this choice will be important in the analysis of section 8.4.

Finally, processes are translated by letting:

$$(t \star \pi)^* \equiv t^* \pi^* .$$

8.3. Simulation of evaluation by weak reduction. The expected property would be that each evaluation step $t_1 \star \pi_2 \succ t_2 \star \pi_2$ in λ_c corresponds to one or several weak reduction steps $t_1^* \pi_1^* \succ_w^+ t_2^* \pi_2^*$ through the CPS-translation. Although this works for almost all the evaluation rules (application, abstraction, call/cc, continuation and successor), the property does not hold for the rule (REC-S) so that we need to refine a little bit more.

Proposition 8.7 (One step simulation). *If $t_1 \star \pi_1 \succ t_2 \star \pi_2$ (one step evaluation in λ_c), then $t_1^* \pi_1^* \succ_w^+ t_2^* u$ (weak reduction) for some term $u =_i \pi_2^*$. Moreover, for all rules but (SUCC-S), we have $u \equiv \pi_2^*$.*

Proof. We distinguish cases according to the applied rule. The cases of abstraction, application, call/cc and continuation constants are easy—they do not involve inner conversion steps—and standard [25], so that we do not treat them here. Let us consider the evaluation rules dealing with primitive numerals.

- Rule (SUCC) We have:

$$\begin{aligned} (\mathbf{s} \star \widehat{n} \cdot u \cdot \pi)^* &\equiv \mathbf{s}^* \langle \mathbf{s}^n \mathbf{0}; \langle u^*; \pi^* \rangle \rangle \\ \gamma_w^* & \quad u^* \langle \mathbf{s}^{n+1} \mathbf{0}; \pi^* \rangle \equiv (u \star \widehat{n+1} \cdot \pi)^* \end{aligned}$$

- Rule (REC-0) We have:

$$\begin{aligned} (\mathbf{rec} \star u_0 \cdot u_1 \cdot \widehat{0} \cdot \pi)^* &\equiv \mathbf{rec}^* \langle u_0^*; \langle u_1^*; \langle \mathbf{0}; \pi^* \rangle \rangle \rangle \\ \gamma_w^* & \quad \mathbf{rec} u_0^* T[u_1^*] \mathbf{0} \pi^* \\ \gamma_w^* & \quad u_0^* \pi^* \equiv (u_0 \star \pi)^*, \end{aligned}$$

writing $T[z] \equiv \lambda x' y k_0 . z \langle x'; \langle \lambda k_1 . y k_1; k_0 \rangle \rangle$.

- Rule (REC-S) We have:

$$\begin{aligned} (\mathbf{rec} \star u_0 \cdot u_1 \cdot \widehat{n+1} \cdot \pi)^* &\equiv \mathbf{rec}^* \langle u_0^*; \langle u_1^*; \langle \mathbf{s}^{n+1} \mathbf{0}; \pi^* \rangle \rangle \rangle \\ \gamma_w^* & \quad \mathbf{rec} u_0^* T[u_1^*] (\mathbf{s}^{n+1} \mathbf{0}) \pi^* \\ \gamma_w^* & \quad u_1^* \langle \mathbf{s}^n \mathbf{0}; \langle \lambda k_1 . \mathbf{rec} u_0^* T[u_1^*] (\mathbf{s}^n \mathbf{0}) k_1; \pi^* \rangle \rangle \end{aligned}$$

Moreover:

$$\begin{aligned} & u_1^* \langle \mathbf{s}^n \mathbf{0}; \langle \lambda k_1 . \mathbf{rec} u_0^* T[u_1^*] (\mathbf{s}^n \mathbf{0}) k_1; \pi^* \rangle \rangle \\ =_i & u_1^* \langle \mathbf{s}^n \mathbf{0}; \langle \lambda k_1 . (\mathbf{rec} u_0 u_1 \widehat{n})^* k_1; \pi^* \rangle \rangle \\ =_i & u_1^* \langle \mathbf{s}^n \mathbf{0}; \langle (\mathbf{rec} u_0 u_1 \widehat{n})^*; \pi^* \rangle \rangle \equiv (u_1 \star \widehat{n} \cdot (\mathbf{rec} u_0 u_1 \widehat{n}) \cdot \pi)^* \end{aligned}$$

(In the second last line, we mimic η -reduction with an inner reduction step, using the fact that the term $(\mathbf{rec} u_0 u_1 \widehat{n})^*$ is an abstraction). \square

Corollary 8.8 (Grand simulation). *If $t_1 \star \pi_1 \succ^* t_2 \star \pi_2$ (evaluation in λ_c), then $t_1^* \pi_1^* \succ_w^* u$ (weak reduction) for some term $u =_i t_2^* \pi_2^*$.*

Proof. By induction on the number of evaluation steps, using Prop. 8.7 and Corollary 7.5 for the induction case. \square

8.4. The negative interpretation of classical witness extraction. Let us now reinterpret the classical witness extraction method described in section 5.2 through the negative translation defined above.

For that, let us consider a closed classical proof term t_0 such that

$$\vdash_{\text{NK}} t_0 : \exists^{\mathbb{N}} x f(x) = 0$$

(where $\exists^{\mathbb{N}} x f(x) = 0 \equiv \forall Z (\forall x (\{x\} \Rightarrow f(x) = 0 \Rightarrow Z) \Rightarrow Z)$), and let us analyze the behavior of the process

$$p_0 \equiv t_0 \star (\lambda xy . y (\mathbf{stop} x)) \cdot \diamond$$

that performs witness extraction (by Prop. 5.3) through the negative translation of section 8.1 and the CPS-translation of section 8.2. (We can end p_0 with the empty stack from the results of section 5.3.)

In the sequel, we write $u \equiv \lambda xy . y (\mathbf{stop} x)$.

8.4.1. *Typing the process $(p_0)^*$.* From Prop. 8.6 we get

$$\vdash_{\text{NJ}} t_0^* : \forall Y \left(\forall x (\text{nat}(x) \wedge (f(x) = 0)^{\neg\neg} \wedge Y \Rightarrow R) \wedge Y \Rightarrow R \right)$$

(writing conjunction right-associative), so that by instantiating Y with \top :

$$\vdash_{\text{NJ}} t_0^* : \forall x (\text{nat}(x) \wedge (f(x) = 0)^{\neg\neg} \wedge \top \Rightarrow R) \wedge \top \Rightarrow R.$$

Let us now fix the pole R by letting $R \equiv \exists^{\mathbb{N}} x \text{null}(f(x))$. Since $\text{stop}^* \equiv \lambda z . z$, we can give it the type

$$\vdash_{\text{NJ}} \text{stop}^* : \forall x (\text{nat}(x) \wedge \text{null}(f(x)) \Rightarrow R),$$

which is precisely the introduction rule of numeric existential quantification. (Remember that $R \equiv \exists^{\mathbb{N}} x \text{null}(f(x)) \equiv \exists x (\text{nat}(x) \wedge \text{null}(f(x)))$.)

Thanks to this, we can typecheck through the CPS-translation all the constituents of the term u . We first have

$$x : \text{nat}(x) \vdash_{\text{NJ}} (\text{stop } x)^* : \text{null}(f(x)) \Rightarrow R.$$

Moreover:

$$y : (f(x) = 0)^{\neg\neg} \vdash_{\text{NJ}} y : \forall Z \left((Z(f(x)) \Rightarrow R) \wedge Z(0) \Rightarrow R \right)$$

(using the definition of Leibniz equality, the axiom rule and the conversion rule), so that by instantiating $Z(x)$ with $\text{null}(x)$ we get

$$y : (f(x) = 0)^{\neg\neg} \vdash_{\text{NJ}} y : (\text{null}(f(x)) \Rightarrow R) \wedge \top \Rightarrow R$$

We thus have

$$x : \text{nat}(x), y : (f(x) = 0)^{\neg\neg} \vdash_{\text{NJ}} (y (\text{stop } x))^* : \top \Rightarrow R$$

and finally:

$$\vdash_{\text{NJ}} u^* : \forall x (\text{nat}(x) \wedge (f(x) = 0)^{\neg\neg} \wedge \top \Rightarrow R)$$

We can now typecheck the term $(u \cdot \diamond)^*$

$$\vdash_{\text{NJ}} (u \cdot \diamond)^* \equiv \langle u^*; 0 \rangle : (\text{nat}(x) \wedge (f(x) = 0)^{\neg\neg} \wedge \top \Rightarrow R) \wedge \top,$$

so that $\vdash_{\text{NJ}} p_0^* \equiv t_0^* \langle u^*; 0 \rangle : R$.

This shows that the CPS-translation of the process described in Prop. 5.3 is actually an intuitionistic proof (in HA2) of the formula $R \equiv \exists^{\mathbb{N}} x \text{null}(f(x))$. From Prop. 7.9, we thus know that the term $(p_0)^*$ weakly reduces to a pair whose first component is the desired witness.

Through the CPS-translation defined in section 8.2, the extraction method described in section 5.2 thus amounts to transform a classical proof-term t_0 of the formula $\exists^{\mathbb{N}} x f(x) = 0$ into an intuitionistic proof-term $t_0^* \langle u^*; 0 \rangle$ of the same formula (up to the coding details of numeric existential quantification and of the predicate expressing the nullity of its argument). Here we can see the essential ingredients of Friedman's transformation:

- The use of a negative translation to transform a classical proof t_0 of a Σ_1^0 -formula into an intuitionistic proof t_0^* of a more complex formula.
- The choice of the return formula R (the pole), that is precisely defined as the formula we want to prove intuitionistically.
- The key use of the introduction rule of numeric existential quantification (here via the term $\text{stop}^* \equiv \lambda z . z$) to return the desired result.

9. CONCLUSION

9.1. From BHK semantics to Krivine’s semantics. Friedman’s extraction method consists to transform a classical proof of an existential formula into an intuitionistic proof of the same formula. Its main drawback is that the underlying CPS-transformation makes the resulting program much bigger than the originating proof, and more difficult to understand.

For this reason, much attention has been devoted to the optimization of the extracted code. The typical approach is *refined program extraction* [3], that relies on a clever analysis of formulæ in order to minimize the insertion of negations during the translation. In practice, such an approach gives much better programs than Friedman’s method, typically when considering examples such as the one we treated in section 6. However, the approach of refined program extraction ultimately remains intuitionistic, since the extracted program is built and analyzed according to the traditional Brouwer-Heyting-Kolmogorov (BHK) semantics, using the tools of intuitionistic realizability (i.e. the mathematical expression of BHK semantics).

In this paper, we have proposed another approach, which is to extract a program from a classical proof directly, by interpreting classical reasoning principles with control operators. The price to pay is that the computational meaning of the extracted program cannot be analyzed within the traditional BHK semantics anymore. For that, we proposed to use Krivine’s theory of classical realizability, that constitutes a genuine alternative to BHK semantics for classical logic.

9.1.1. A negative semantics for classical programs. It has already been pointed out [25] that Friedman’s negative translation is hard-wired in Krivine’s semantics. Taking the notations of section 8, we get the correspondence:

Krivine’s semantics	Friedman’s translation
The pole \perp	The formula R
Falsity value $\ A\ $	Formula A^\perp
Truth value $ A = \ A\ ^\perp$	Formula $A^{\neg\neg} \equiv A^\perp \Rightarrow R$
Classical proof-term t	CPS-translated term t^*

In this paper, we have shown that the correspondence can be lifted up at the level of the witness extraction methods, in the sense that the natural extraction method that comes with Krivine’s machinery (section 5.2) is, up to a CPS-translation, the same as Friedman’s (provided we use Streicher and Oliva’s negative translation instead of the traditional not-not translation).

However, Krivine’s semantics is more subtle than the composition of a negative translation with the standard BHK interpretation, since the way it is formulated makes the negative translation implicit. Thanks to this, it is possible to reason about classical programs directly. We illustrated this point with the witness extraction procedures presented in section 5 and with the construction by hand of a universal realizer of the minimum principle in section 6.

It should also be noted that Krivine’s semantics is compatible with the usual deduction rules of intuitionistic logic, as soon as they are formulated in FA2 style [14]. In particular, the typing rules of Fig. 1 but the last one (Peirce’s law) are the usual Curry-style typing rules of intuitionistic minimal logic, formulated the usual way. Any intuitionistic proof-term that is well typed in FA2 is not only correct w.r.t. the usual intuitionistic realizability

semantics of FA2 [14], but it is also correct according to Krivine’s semantics. The latter departs from the traditional BHK semantics only when classical reasoning is involved.

9.2. Executing extracted programs. In our approach, extracted programs are not ordinary λ -terms, but λ -terms with control operators that need to be evaluated according to a strict call-by-name discipline. Specific tools are thus required to execute them¹³.

The main implementation difficulty comes from stacks, whose machine representation has to be carefully designed in order to avoid unnecessary duplications when call/cc is executed. Fortunately, control operators have been introduced in programming languages long before the discovery of their connection with classical logic, and we can benefit from the many implementation strategies that have been proposed since. To illustrate this, we shall discuss two of them. (For a survey on the different ways to implement control, see [4].)

9.2.1. Stacks as chained lists. The simplest way to implement stacks is to represent them as heap-allocated chained lists of closures. With this representation, call/cc comes for free since it simply consists to make a copy of the current stack pointer. The main advantage of this method is that it naturally maximizes the possibility of sharing large final segments of stacks. Its main drawback is that each PUSH operation requires the allocation of a small block on the heap, which block is subject to later garbage collection. Moreover, the resulting fragmentation of the stack may considerably degrade cache performance.

However, the simplicity of this approach makes it well-suited for a small interpreter intended to quickly test small examples. This is this design that is currently used in the *jivaro* machine [23].

9.2.2. The stack/heap model. In the perspective of implementing a real compiler of λ_c -programs, a more realistic representation of stacks is given by the stack/heap model [4], where the logical stack is physically split in two parts:

- A *stack cache*, that consists of a mutable array of closures representing the topmost part of the logical stack. This stack cache lies in a fixed zone of the memory, and works almost as the ordinary system stack.
- A *far stack*, that represents the rest of the logical stack in the heap as a chained list of non mutable blocks containing stack chunks. As for all the other heap-allocated blocks, the stack chunks of the far stack may be shared and are subject to garbage collection.

The underlying idea of the stack/heap approach is that during execution, almost all the operations on the logical stack take place in the stack cache, the manipulation of the far stack being exceptional. Pushing an argument onto the stack proceeds in the stack cache as usual, as well as grabbing the topmost element. The difference is that in the latter case, the cache may underflow, in which case we need to refill the cache by copying the contents of the first block of the far stack. (After copying, the far stack pointer should point to the next block). With this approach, call/cc only needs to make a copy of the stack cache into a newly heap-allocated block. The pointer to this newly allocated block then becomes the corresponding continuation constant as illustrated in Fig. 6. Restoring a formerly saved

¹³We cannot directly use the interpreters and compilers dedicated to popular functional programming languages such as LISP, Caml, SML or Haskell, since these tools implement either the call-by-value discipline or the call-by-need discipline.

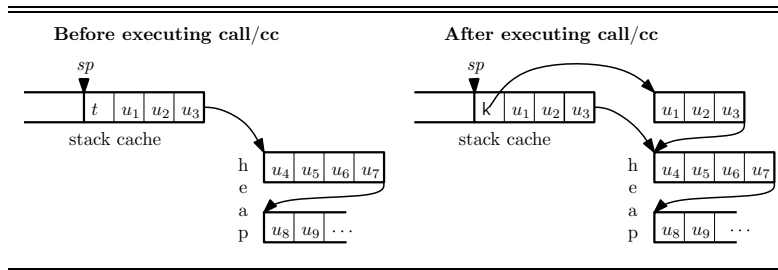


Figure 6: Execution of call/cc in the stack/heap model

stack thus consists to clear the stack cache and to let the far stack pointer point to the first continuation block. (The stack cache will then be automatically refilled with the next GRAB operation.)

The interest of this approach is that call/cc only needs to copy the part of the stack that has been used since the last execution of a control operator. The authors of [4] consider that this approach is a *zero overhead* approach, in the sense that it adds a negligible overhead to the most frequent operations PUSH and GRAB, compared with the traditional single-chunk stack model.

9.2.3. On the frequency of control in classical proofs. The above discussion about implementation issues raises a strong argument in favor of using λ_c for interpreting classical proofs. A quick look at the classical proofs of well-known theorems shows that classical reasoning is definitely not used with the same frequency as intuitionistic reasoning. Purely intuitionistic reasoning (introduction/elimination of connectives and quantifiers, induction...) appears everywhere, whereas classical reasoning principles are only used at some few strategic places. In some sense, one can consider that actual mathematics are more quasi-intuitionistic than really classical. The execution trace (Fig. 4 p. 30) of the example of section 6 makes the comparison more dramatic, since it shows that intuitionistic operations are executed several hundreds of times while classical operations are executed a dozen of times (call/cc being invoked only once).

These figures suggest that a good execution policy for classical proofs should concentrate all the execution overhead induced by the presence of classical reasoning to the classical operations themselves (that are the less frequent ones) while keeping ordinary intuitionistic operations (the most frequent ones) as fast as possible. But this is precisely what the λ_c -calculus does, especially when executed in the stack/heap model described above. On the other hand, using a negative translation—even optimized—adds a non negligible execution overhead to all the intuitionistic operations of the proof, just to remove the need of introducing specific operators for classical reasoning.

9.3. Classical extraction in Coq. The ideas presented in this paper have been implemented in a classical extraction module for Coq developed by the author [22]. On the theoretical side, this implementation is based on an extension of Krivine’s classical realizability model to the calculus of constructions with universes [21]. This module permits the extraction of a λ_c -term from any classical proof formalized in Coq—provided classical logic is only allowed in the impredicative sort Prop. It also proposes witness extraction facilities based on the techniques presented in section 5.

This module automatically performs several optimizations in the extracted code. For instance, Coq unary numerals (as well as the corresponding arithmetic operators) are automatically translated into the primitive numerals discussed in section 4. Similar optimizations are introduced to change the representation of other inductively defined data-types such as ordering. Moreover, the extractor proposes predefined optimized realizers for many theorems of Coq’s standard library, following the spirit of what we did in section 6 with the minimum principle. In this way, the user can formalize classical proofs using the tools provided by Coq’s standard library while benefiting from many optimizations that are allowed by the theory of classical realizability.

However, these hand-made optimized realizers are provided only for a little fragment of Coq’s standard library, and there is currently no general mechanism to generate them on the fly. Future work includes the design of a general theory for realizer optimization in the framework of classical realizability, following the spirit of refined program extraction [3].

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] U. Berger, W. Buchholz, and H. Schwichtenberg. Refined program extraction from classical proofs. *Annals of Pure and Applied Logic*, 114(1-3):3–25, 2002.
- [4] W. D. Clinger, A. Hartheimer, and E. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, 1999.
- [5] P.-L. Curien and H. Herbelin. The duality of computation. In *ICFP*, pages 233–243, 2000.
- [6] H. Friedman. Classically and intuitionistically provably recursive functions. *Higher Set Theory*, 669:21–28, 1978.
- [7] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.
- [8] K. Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12:280–287, 1958.
- [9] T. Griffin. A formulae-as-types notion of control. In *Principles Of Programming Languages (POPL’90)*, pages 47–58, 1990.
- [10] M. Guillermo. *Jeux de réalisabilité en arithmétique classique*. PhD thesis, Université Paris 7, 2008.
- [11] U. Kohlenbach. *Applied Proof Theory: Proof Interpretations and their Use in Mathematics*. Springer, 2008.
- [12] G. Kreisel. On the interpretation of non-finitist proofs, part I. *Journal of Symbolic Logic*, 16:241–267, 1951.
- [13] G. Kreisel. On the interpretation of non-finitist proofs, part II: Interpretation of number theory. *Journal of Symbolic Logic*, 17:43–58, 1952.
- [14] J. L. Krivine. *Lambda-calculus, types and models*. Masson, 1993.
- [15] J.-L. Krivine. A general storage theorem for integers in call-by-name lambda-calculus. *Th. Comp. Sc.*, 129:79–94, 1994.
- [16] J.-L. Krivine. Typed lambda-calculus in classical Zermelo-Fraenkel set theory. *Arch. Math. Log.*, 40(3):189–205, 2001.
- [17] J.-L. Krivine. Dependent choice, ‘quote’ and the clock. *Th. Comp. Sc.*, 308:259–276, 2003.
- [18] J.-L. Krivine. Structures de réalisabilité, RAM et ultrafiltre sur \mathbb{N} . Manuscript, available on the author’s web page, 2008.
- [19] J.-L. Krivine. Realizability in classical logic. In interactive models of computation and program behaviour. *Panoramas et synthèses*, 27, 2009.
- [20] P. Letouzey. A new extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *TYPES*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219. Springer, 2002.

- [21] A. Miquel. Classical program extraction in the calculus of constructions. In J. Duparc and T. A. Henzinger, editors, *CSL*, volume 4646 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 2007.
- [22] A. Miquel. The classical extraction module for Coq. <http://perso.ens-lyon.fr/alexandre.miquel/kextraction/>, 2009.
- [23] A. Miquel. The Jivaro head reduction machine for the λ_c -calculus. <http://perso.ens-lyon.fr/alexandre.miquel/jivaro/>, 2009.
- [24] A. Miquel. Relating classical realizability and negative translation for existential witness extraction. In P.-L. Curien, editor, *TLCA*, volume 5608 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2009.
- [25] P. Oliva and T. Streicher. On Krivine’s realizability interpretation of classical second-order arithmetic. *Fundam. Inform.*, 84(2):207–220, 2008.
- [26] M. Parigot. Proofs of strong normalisation for second order classical natural deduction. *J. Symb. Log.*, 62(4):1461–1479, 1997.
- [27] H. Schwichtenberg. Minimal logic for computable functionals. Technical report, Mathematisches Institut der Universität München, 2005.
- [28] M. Takahashi. Parallel reductions in lambda-calculus. *J. Symb. Comput.*, 7(2):113–123, 1989.
- [29] The Coq Development Team. The coq proof assistant reference manual – version v8.2. Technical report, INRIA, 2009.