

The experimental effectiveness of mathematical proof

Alexandre Miquel

November 2007

— DRAFT —

Abstract

Given a testable universal theory A (i.e. formed with universal formulæ built upon experimentally testable predicates), we show how to extract from a mathematical proof of inconsistency of A (formalized in classical second-order arithmetic) a physically effective algorithm that performs a finite set of experiments on A before stopping in finite time on a set of parameters experimentally falsifying an instance of a formula of the theory A . From this result, we deduce a method to extract, from a *mathematical proof* of $A \Rightarrow B$ (where A and B are testable universal theories) combined with an *experimental falsification* of B , a program of tests of A that eventually stops on an experimental falsification of the theory A , thus realizing the experimental version of the modus tollens.

1 An experimental arithmetic

In this section, we present a formalism which we call the *experimental arithmetic*. Basically, this formalism is an extension of second-order Peano arithmetic (PA2) with primitive predicate symbols (written p, q, r , etc.) expressing experimentally testable facts. By this, we mean that each generic atomic formula $p(x, y, z, \dots)$ associated to an experimental predicate symbol p comes with a ‘real world’ interpretation such as

The animal in box No. x is a cat

or

The concentration of alcohol in tube No. x is comprised between y and z percent

or even

The 2nd-coordinate of the magnetic field in the universe cube No. x has an average value comprised between y/w and z/w (in Tesla).

Since we work in arithmetic, sets of parameters of an experimental predicate p are naturally expressed as tuples of natural numbers, hence the need for more

or less clever codings to address the physical reality.¹ (For instance in the 3rd example, the use of a fixed sequence of universe cubes with discrete bounds to address portions of the space, and the introduction of a precision parameter w to express continuous quantities.) On the other hand, experimental predicates come with no special meaning as *mathematical* entities, that is, they come with no specific axiom or computational rule attached to them.

1.1 The language

The language of experimental arithmetic is defined from the following sets of symbols:

- An infinite set of first-order variables (written x, y, z , etc.), that is: variables denoting natural numbers.
- For each arity $n \geq 0$, an infinite set of second-order variables of arity n , (written X, Y, Z , etc.), that is: variables denoting n -ary relations over natural numbers.
- For each primitive recursive definition of a function—including zero and successor—a function symbol (written f, g, h , etc.) representing the function described by the given definition. In particular, we respectively denote by 0 and s the constant symbol representing zero and the unary function symbol representing the successor function. We will also freely use well-known symbols such as $+$ (addition), \times (multiplication), etc.
- A finite set of predicate symbols (written p, q, r , etc.) representing experimental predicates, each of them given with its arity.

The language of experimental arithmetic distinguishes two kinds of expressions: *numeric expressions* (written e, e_1, e' , etc.) that represent natural numbers; and *formulae* (written A, B, C , etc.) that represent facts—possibly true or false—built from the experimental predicates.

Numeric expressions The language of numeric expressions is inductively defined from the following two construction rules:

- If x is a first-order variable, then x is a numeric expression;
- If f is a n -ary function symbol and if e_1, \dots, e_n are n numeric expressions, then $f(e_1, \dots, e_n)$ is a numeric expression.

The set of free variables of a numeric expression e is written $FV(e)$. Given a numeric expression e , a variable x and a numeric expression e' , we write $e\{x := e'\}$ the numeric expression obtained by replacing in e all the occurrences of the variable x by e' .

For each closed numeric expression e , we call the *value of e* and write $\downarrow e$ the natural number computed by the expression e using the computation rules attached to the function symbols f contained in e .

¹We could easily enrich the language to express richer kinds of parameters such as relative integers, rational numbers, lists, trees, etc. However, the parameters of a testable predicate p should always remain discrete, independently from their representation.

Formulæ The language of formulæ is inductively defined from the following five construction rules:

- If p is a n -ary experimental predicate symbol and if e_1, \dots, e_n are n numeric expressions, then $p(e_1, \dots, e_n)$ is a formula;
- If X is a n -ary second-order variable and if e_1, \dots, e_n are n numeric expressions, then $X(e_1, \dots, e_n)$ is a formula;
- If A and B are formulæ, then $A \Rightarrow B$ is a formula;
- If x is a first-order variable and if B is a formula, then $\forall x B$ is a formula;
- If X is a n -ary second-order variable and if B is a formula, then $\forall X B$ is a formula.

The set of free (first- and second-order) variables of a formula A is written $FV(A)$. As usual, formulæ are considered up to α -conversion, regardless from the names of bound variables. Formulæ are equipped with two different operations of substitution:

- (*First-order substitution*) Given a formula A , a variable x and a numeric expression e , we write $A\{x := e\}$ the formula obtained by replacing in A all the free occurrences of the variable x by e , taking care of renaming bound (first-order) variables when needed in order to avoid variable captures. More generally, we denote by

$$A\{x_1 := e_1; \dots; x_n := e_n\}$$

the formula obtained by simultaneously replacing in the formula A all the free occurrences of variables x_i (among x_1, \dots, x_n) by the corresponding numeric expression e_i . (Again taking care of renaming bound variables when needed.)

- (*Second-order substitution*) Given a formula A , a n -ary second-order variable X , a list of n variables x_1, \dots, x_n and a formula B , we denote by

$$A\{X(x_1, \dots, x_n) := B\} \quad (\text{or } A\{X := \lambda x_1, \dots, x_n. B\})$$

the formula obtained by replacing in A all the atomic subformulæ of the form $X(e_1, \dots, e_n)$ (corresponding to free occurrences of X in A) by the formula $B\{x_1 := e_1; \dots; x_n := e_n\}$, while taking care of renaming bound (first- and second-order) variables when needed.

In this core language based on implication and (first- and second-order) universal quantification, it is possible to encode all the other constructions of logic: conjunction, disjunction, negation, (first- and second-order) existential quantification, Leibniz equality, etc. using well-known second-order encodings [1] (cf Fig. 1: abbreviations).

1.2 Deduction rules and derivations

The language defined above is equipped with the standard notion of provability in second order arithmetic (PA2), using a presentation based on intuitionistic natural deduction extended with Peirce's law to recover classical logic. Formally, we work with asymmetric sequents of the form

$$\Gamma \vdash A \quad (\text{'under the assumptions } \Gamma, A \text{ holds'})$$

where $\Gamma \equiv A_1, \dots, A_n$ is a finite list of formulæ used as hypotheses (the *subsequent*), and where A is a formula (the *consequent*).

The notations $FV(\Gamma)$, $\Gamma\{x := e\}$, $\Gamma\{X(x_1, \dots, x_n) := B\}$, etc. are extended to lists of formulæ $\Gamma = A_1, \dots, A_n$ by setting:

$$\begin{aligned} FV(A_1, \dots, A_n) &= FV(A_1) \cup \dots \cup FV(A_n) \\ (A_1, \dots, A_n)\{x := e\} &= A_1\{x := e\}, \dots, A_n\{x := e\} \end{aligned}$$

(And similarly for the notation $\Gamma\{X(x_1, \dots, x_n) := B\}$.)

The deduction rules of the system are given in Fig. 1. They consist of:

- The deduction rules of minimal intuitionistic second-order logic, namely: the axiom rule, plus the introduction and elimination rules of implication, first- and second-order universal quantification.
- A specific rule for Peirce's law, that entails all the other classical reasoning principles: the excluded middle ($A \vee \neg A$), double negation elimination ($\neg\neg A \Rightarrow A$) and *reductio ad absurdum*.

The axiom rule is extended with all the axioms of arithmetic, namely:

- The axioms expressing that the successor function is injective (Peano 3rd axiom) and that zero is not in its image (Peano 4th axiom). Notice that the formulation of these axioms relies on the encoding of equality and negation given in Fig. 1 ('abbreviations').
- The defining equalities of the function symbols f representing primitive recursive definitions of functions. Example of such equalities defining addition, multiplication, exponentiation, the factorial function, etc. are:

$$\begin{aligned} 0 + y &= y & 0 \times y &= 0 \\ s(x) + y &= s(x + y) & s(x) \times y &= (x \times y) + y \\ x \uparrow 0 &= s(0) & 0! &= s(0) \\ x \uparrow s(y) &= (x \uparrow y) \times x & s(x)! &= s(x) \times x! \end{aligned}$$

Once deduction rules have been defined, we can construct proofs by assembling instances of these rules such as in the following example:

$$\frac{\frac{\forall x (Y(x) \Rightarrow Z), Y(s(0)) \vdash \forall x (Y(x) \Rightarrow Z)}{\forall x (Y(x) \Rightarrow Z), Y(s(0)) \vdash Y(s(0)) \Rightarrow Z} \quad \frac{\forall x (X(x) \Rightarrow Y(z + x)), Y(s(0)) \vdash Y(s(0))}{\forall x (Y(x) \Rightarrow Z), Y(s(0)) \vdash Z}}{\forall x (Y(x) \Rightarrow Z), Y(s(0)) \vdash Z}$$

Formally, we call a *derivation* (or a *proof*) any finite tree d whose nodes are labelled with sequents, and such that for each node of d labelled with sequent S

and whose children are labelled with sequents S_1, \dots, S_n ($n \geq 0$), there is a deduction rule such that the inference $\frac{S_1 \dots S_n}{S}$ is an instance of this rule.

Given a derivation d , the sequent S that labels the root of d is called the *conclusion* of d , and d is called a *derivation of S* . When a sequent S has a derivation d , we say that S is *derivable*. In particular, when a sequent of the form $\vdash A$ (without assumption, A being a closed formula) is derivable, we say that the formula A is a *theorem* (of second-order arithmetic).

1.3 Arithmetic reasoning

The reader may have noticed that we presented axioms (cf Fig. 1) expressing that the successor function is injective (3rd Peano axiom) and non-surjective (4th Peano axiom), but that there is no axiom for the induction principle on natural numbers (5th Peano axiom). The reason is that in second-order logic, the induction principle can be obtained for free using a well-known trick [1] that relies on the strength of second-order universal quantification.

The trick is to define a class of ‘well-formed natural numbers’ (delimited by a predicate written $\text{Nat}(x)$) as the smallest class containing zero and closed under the successor function s . Formally, we introduce the abbreviation

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

We can easily show that the class of well-formed natural numbers contains zero and is closed under the successor function:

1. $\text{Nat}(0)$ (1st Peano axiom)
2. $\forall x (\text{Nat}(x) \Rightarrow \text{Nat}(s(x)))$ (2nd Peano axiom)

Using this predicate, we can relativise universal and existential first-order quantification to well-formed numerals using the following abbreviations:

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

It is then a simple exercise to check that induction holds provided we restrict universal quantification to the class of well-formed natural numbers:

$$\forall X (X(0) \Rightarrow \forall^{\mathbb{N}} y (X(y) \Rightarrow X(s(y))) \Rightarrow \forall^{\mathbb{N}} x X(x))$$

Using this principle, we easily show that all primitive recursive functions f map well-formed numerals to well-formed numerals

$$\forall^{\mathbb{N}} x_1 \dots \forall^{\mathbb{N}} x_n \text{Nat}(f(x_1, \dots, x_n))$$

and more generally that any numeric expression $e[x_1, \dots, x_n]$ with free variables x_1, \dots, x_n denotes a well-formed numeral provided its free variables x_1, \dots, x_n are restricted to the class of well-formed numerals:

$$\forall^{\mathbb{N}} x_1 \dots \forall^{\mathbb{N}} x_n \text{Nat}(e[x_1, \dots, x_n])$$

By systematically using the relativised first-order quantifications $\forall^{\mathbb{N}}$ and $\exists^{\mathbb{N}}$ instead of the ‘pure’ first-order quantifiers \forall and \exists , we can reason as in ordinary first-order arithmetic.

Syntax

(Numeric expressions)	$e, e' ::= x \mid f(e_1, \dots, e_n)$
(Formulæ)	$A, B ::= p(e_1, \dots, e_n) \mid X(e_1, \dots, e_n)$ $\mid A \Rightarrow B \mid \forall x A \mid \forall X A$

Abbreviations

(Contradiction)	$\perp \equiv \forall X X$
(Immediate truth)	$\top \equiv \forall X (X \Rightarrow X)$
(Negation)	$\neg A \equiv A \Rightarrow \perp$
(Conjunction)	$A \wedge B \equiv \forall X ((A \Rightarrow B \Rightarrow X) \Rightarrow X)$
(Disjunction)	$A \vee B \equiv \forall X ((A \Rightarrow X) \Rightarrow (B \Rightarrow X) \Rightarrow X)$
(1st order existence)	$\exists x A[x] \equiv \forall Y (\forall x (A[x] \Rightarrow Y) \Rightarrow Y)$
(2nd order existence)	$\exists X A[X] \equiv \forall Y (\forall X (A[X] \Rightarrow Y) \Rightarrow Y)$
(Leibniz equality)	$e_1 = e_2 \equiv \forall X (X(e_1) \Rightarrow X(e_2))$ $e_1 \neq e_2 \equiv \neg(e_1 = e_2)$

Deduction rules

(Axiom)	$\overline{\Gamma \vdash A} \quad (A \in \Gamma \cup \mathcal{A})$
(\Rightarrow -intro,-elim)	$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \quad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$
(\forall^1 -intro,-elim)	$\frac{\Gamma \vdash B}{\Gamma \vdash \forall x B} \quad (x \notin FV(\Gamma)) \quad \frac{\Gamma \vdash \forall x B}{\Gamma \vdash B\{x := e\}}$
(\forall^2 -intro,-elim)	$\frac{\Gamma \vdash B}{\Gamma \vdash \forall X B} \quad (X \notin FV(\Gamma)) \quad \frac{\Gamma \vdash \forall X B}{\Gamma \vdash B\{X(x_1, \dots, x_n) := A\}}$
(Peirce's law)	$\overline{\overline{\Gamma \vdash ((A \Rightarrow B) \Rightarrow A) \Rightarrow A}}$

where \mathcal{A} denotes the set of all axioms of arithmetic, that consists of:

- The axiom of injectivity: $\forall x \forall y (s(x) = s(y) \Rightarrow x = y)$
 - The axiom of non confusion: $\forall x s(x) \neq 0$
 - The defining equalities attached to the function symbols f
 (that recursively define the corresponding primitive recursive functions)
-
-

Figure 1: Language and deduction rules of PA2

1.4 Experimental formulæ

Inside the language of formulæ, we distinguish three classes of formulæ representing experimental statements, namely: the class of *elementary formulæ*, the class of *testable formulæ* and the class of *testable universal formulæ*.

Elementary formulæ. We call an *elementary formula* any (open or closed) formula E_0 that has of the following three forms:

$$\begin{aligned} (\textit{Experimentation}) \quad E_0 &\equiv p(e_1, \dots, e_n) \\ (\textit{Falsity}) \quad E_0 &\equiv \perp \equiv \forall X X \\ (\textit{Equality test}) \quad E_0 &\equiv e_1 = e_2 \equiv \forall X (X(e_1) \Rightarrow X(e_2)) \end{aligned}$$

Intuitively, elementary formulæ are atomic formulæ built from experimental predicates in a broader sense that includes the equality test (seen as a purely mathematical experiment). We include falsity \perp (representing the experiment that always fails) just for convenience, since the testable equality $0 = 1$ would have done the same job as well.

Testable formulæ. They are inductively built from the following three construction rules:

- If E_0 is an elementary formula, then E_0 is a testable formula.
- If E_0 is an elementary formula and if E is a testable formula, then the formula $E_0 \Rightarrow E$ is a testable formula.
- If E_0 is an elementary formula and if E is a testable formula, then the formula $\neg E_0 \Rightarrow E$ is a testable formula.

In other words, a testable formula is a formula of the form

$$E \equiv \pm E_1 \Rightarrow \dots \Rightarrow \pm E_k \Rightarrow E_0,$$

where E_0, \dots, E_k are elementary formulæ (here, the notation $\pm E_i$ refers to one of both formulæ E_i or $\neg E_i$ indifferently). From the point of view of classical logic, a testable statement is thus nothing but a (disjunctive) clause formed from elementary statements:

$$E \Leftrightarrow \mp E_1 \vee \dots \vee \mp E_k \vee E_0.$$

Closed testable formulæ are called *testable singular formulæ*

Testable universal formulæ. Finally, we call a *testable universal formula* any closed formula of the form

$$U \equiv \forall^{\mathbb{N}} x_1 \dots \forall^{\mathbb{N}} x_n E[x_1, \dots, x_n]$$

where $E[x_1, \dots, x_n]$ is a testable formula whose free variables occur among the set of variables $\{x_1, \dots, x_n\}$. Notice that each testable singular formula (as well as each closed elementary formula) is a testable universal formula with a degree of universality (i.e. the number of $\forall^{\mathbb{N}}$) equal to zero.

Testable universal formulæ are testable in the sense that all their instances (corresponding to all possible combinations of parameters in \mathbb{N}^n) can be tested

separately. Of course, a testable universal formula cannot be verified globally—which would require an infinite number of tests—but it can be falsified with a single test corresponding to a single set of parameters.

Epistemologically, the notion of testable universal formula approximately corresponds to (and is actually directly inspired from) K. R. Popper’s notion of empirical (or falsifiable) statement [5].

(Elementary formulæ)	$E_0 ::= \perp \mid p(e_1, \dots, e_n) \mid e_1 = e_2$
(Testable formulæ)	$E ::= E_0 \mid E_0 \Rightarrow E \mid \neg E_0 \Rightarrow E$
(Testable universal formulæ)	$U ::= \forall^{\mathbb{N}} x_1 \dots \forall^{\mathbb{N}} x_n E[x_1, \dots, x_n]$

Figure 2: The language of experimental formulæ

1.5 The experimental theory and the tools to test it

The notion of experimental theory We call an *experimental theory* any finite list U_1, \dots, U_ℓ of testable universal formulæ. In practice, such a theory will be formed with three kinds of statements:

- (1) Currently accepted scientific laws (according to a given theory)
- (2) Initial conditions (of a particular experiment)
- (3) Observed effects (of the experiment)

Typically, statements of the first kind will be formalised as testable universal formulæ whereas statements of the second and third kinds will be formalised as testable singular formulæ.

The situation we are interested in occurs when the experimental theory is contradictory. In practice, the contradiction arises when an observed effect (3) does not match a prediction derived from the initial conditions (2) using the universal laws given by the theory (1). The aim of this paper is to present an effective procedure that selects particular instances of the formulæ U_1, \dots, U_ℓ and test them successively until one of them fails—typically, an instance of one of the universal statements of the accepted theory.

In what follows, we assume given an experimental theory U_1, \dots, U_ℓ .

Test functions To test the experimental theory, we assume that each experimental predicate symbol p comes with a test function

$$\mathbf{Val}(p) : \mathbb{N}^n \rightarrow \{0; 1\}$$

which maps each n -tuple of parameters $(x_1, \dots, x_n) \in \mathbb{N}^n$ to a boolean value representing the success (1) or the failure (0) of the experiment associated to the atomic formula $p(x_1, \dots, x_n)$.

From an experimental point of view, we assume that the computation of $\mathbf{Val}(p)(x_1, \dots, x_n)$ can be achieved using a finite amount of resources for all tuples of parameters $(x_1, \dots, x_n) \in \mathbb{N}^n$. In particular, we assume that this computation can always be done (at least potentially) in finite time.

From a mathematical point of view, we make the only assumption that the function $\mathbf{Val}(p) : \mathbb{N}^n \rightarrow \{0; 1\}$ is total. In particular:

1. We do not assume that the function $\mathbf{Val}(p) : \mathbb{N}^n \rightarrow \{0; 1\}$ is decidable, or even semi-decidable. Effectiveness has to be understood here in the (informal) sense of physics, not in the (formal) sense of recursion theory. In this work, we never need the assumption that physically computable functions are Turing computable (Church's thesis).
2. More important, we do not assume that the set of test functions $\mathbf{Val}(p)$ constitutes a model of the experimental theory, which means that we do not assume that the experimental theory is valid. The extracted program of tests has to be correct independently from the validity of the experimental theory—it must perform correct computations even from wrong experimental assumptions.

Evaluating testable singular formulæ From the test functions $\mathbf{Val}(p)$ we define a generalised test function $E \mapsto \mathbf{Val}(E)$ that associates a boolean value $\mathbf{Val}(E) \in \{0; 1\}$ to each testable singular formula E , indicating whether the experiment associated to E succeeds or fails. Formally, the boolean value $\mathbf{Val}(E) \in \{0; 1\}$ is recursively defined on the structure of E by the equations

$$\begin{aligned} \mathbf{Val}(\perp) &= 0 \\ \mathbf{Val}(e_1 = e_2) &= \begin{cases} 1 & \text{if } \downarrow e_1 = \downarrow e_2 \\ 0 & \text{otherwise} \end{cases} \\ \mathbf{Val}(p(e_1, \dots, e_n)) &= \mathbf{Val}(p)(\downarrow e_1, \dots, \downarrow e_n) \\ \mathbf{Val}(E_0 \Rightarrow E) &= \sup(1 - \mathbf{Val}(E_0), \mathbf{Val}(E)) \\ \mathbf{Val}(\neg E_0 \Rightarrow E) &= \sup(\mathbf{Val}(E_0), \mathbf{Val}(E)) \end{aligned}$$

Of course, if we assume that the test functions $\mathbf{Val}(p)$ are physically computable, then the evaluation function $E \mapsto \mathbf{Val}(E)$ of testable singular formulæ is physically computable too, using a simple recursive procedure ultimately based on the test functions $\mathbf{Val}(p)$, the equality test, and on the computation rules of primitive recursive functions inside numeric expressions.

In what follows, we will say that a tuple $(\nu_1, \dots, \nu_n) \in \mathbb{N}^n$ is a *falsification* of a testable universal formula of the form

$$U \equiv \forall^{\mathbb{N}} x_1 \cdots \forall^{\mathbb{N}} x_n E[x_1, \dots, x_n],$$

when $\mathbf{Val}(E[\nu_1, \dots, \nu_n]) = 0$.

2 Extracting a program from a proof

2.1 The programming language

The programming language we use here is an extension of Krivine's language λ_c [4] with a small set of extra instructions to test the experimental theory. This

language is defined from three kinds of syntactic entities:

- *Terms* (written t, u , etc.), that represent programs defined independently from their evaluation context. From the point of view of logic, terms are the computational equivalents of proofs.
- *Stacks* (written π, π' , etc.), that represent evaluation contexts of programs. Formally, stacks are defined as finite lists of closed terms (representing function arguments) of the form $\pi = u_1 \cdot \dots \cdot u_n \cdot \diamond$, where \diamond denotes the bottom of the stack. From the point of view of logic, stacks are the computational equivalents of *counter-proofs* (or *refutations*). Notice that stacks are only formed with closed terms, and thus are closed objects (i.e. with no free occurrence of a term variable).
- *Processes*, written $t \star \pi$, that are just formed by putting a term t in front of a stack π . From the point of view of logic, processes represent the interaction (i.e. the ‘debate’) between a proof and a counter-proof within a contradictory situation.

The formal definition of terms, stacks and processes of the programming language is given in Fig. 3. The set of all closed terms is written \mathcal{T} whereas the set of all (closed) stacks is written Π .

Given two terms t, u and a term variable ξ , we write $t\{\xi := u\}$ the term obtained by replacing in t each free occurrence of the term variable ξ by the term u . In practice, we will only use this operation when u is a closed term. Term substitution immediately generalises to a notion of parallel substitution written $t\{\xi_1 := u_1; \dots; \xi_n := u_n\}$. Notice that when u_1, \dots, u_n are closed terms, parallel substitution may be computed sequentially by the equation

$$t\{\xi_1 := u_1; \dots; \xi_n := u_n\} = (\dots t\{\xi_1 := u_1\} \dots)\{\xi_n := u_n\}.$$

Syntax of terms The language of terms includes all the constructions of Krivine’s language λ_c , namely:

- The usual constructions of the λ -calculus: variables (written ξ, ζ, ϕ , etc.), function abstractions ($\lambda\xi.t$) and function applications (tu). From the point of view of logic, these constructions correspond to intuitionistic reasoning principles, and thus suffice to extract the computational contents of all proofs of HA2, the intuitionistic fragment of PA2.
- Specific instructions to manipulate continuations: the constant \mathfrak{c} (*call with current continuation*, or *call/cc*, for short) and for each stack π , a constant k_π representing the corresponding continuation. From the point of view of logic, the constant \mathfrak{c} (that generates constants k_π during the evaluation process) implements Pierce’s law, from which we recover all classical logic, and thus the full strength of PA2.

This core language is extended with the following extra instructions:

- For each formula U of the experimental theory, a constant test_U that tests an instance of U (whose parameters are given as arguments) and whose evaluation depends on the outcome of the experiment (cf Fig. 3).

Syntax of the language

Terms	$t, u ::= \xi \mid \lambda\xi.t \mid tu \mid k_\pi \mid \mathbf{c} \mid \text{test}_U \mid \text{stop}$	
Stacks	$\pi ::= \diamond \mid t \cdot \pi$	(t, π closed)
Processes	$p ::= t \star \pi$	

Evaluation rules (except test instructions)

(PUSH)	$tu \star \pi \succ t \star u \cdot \pi$	
(GRAB)	$\lambda\xi.t \star u \cdot \pi \succ t\{\xi := u\} \star \pi$	
(CALL/CC)	$\mathbf{c} \star t \cdot \pi \succ t \star k_\pi \cdot \pi$	
(RESTORE)	$k_\pi \star t \cdot \pi' \succ t \star \pi$	
	$\text{stop} \star \pi \not\succeq$	

Evaluation rules of test instructions test_U

$U[x_1, \dots, x_n] \equiv \forall^{\mathbb{N}}x_1 \cdots \forall^{\mathbb{N}}x_n E[x_1, \dots, x_n]$
 $E[x_1, \dots, x_n] \equiv L_1[x_1, \dots, x_n] \Rightarrow \cdots \Rightarrow L_k[x_1, \dots, x_n] \Rightarrow R[x_1, \dots, x_n]$
 $\bar{\nu}_1, \dots, \bar{\nu}_n \equiv$ Church numerals associated to the numbers ν_1, \dots, ν_n

1. Case where $\mathbf{Val}(L_i[\nu_1, \dots, \nu_n]) = 0$ ($1 \leq i \leq k$)

$\text{test}_U \star \bar{\nu}_1 \cdots \bar{\nu}_n \cdot \pi \succ \lambda\xi_1 \cdots \xi_k. \xi_i \star \pi$	(if L_i positive)
$\text{test}_U \star \bar{\nu}_1 \cdots \bar{\nu}_n \cdot \pi \succ \lambda\xi_1 \cdots \xi_k. \xi_i \mathbf{I} \star \pi$	(if L_i negative)
 2. Case where $\mathbf{Val}(R[\nu_1, \dots, \nu_n]) = 1$

$\text{test}_U \star \bar{\nu}_1 \cdots \bar{\nu}_n \cdot \pi \succ \lambda\xi_1 \cdots \xi_k. \mathbf{I} \star \pi$
 3. Case where $\mathbf{Val}(E[\nu_1, \dots, \nu_n]) = 0$

$\text{test}_U \star \bar{\nu}_1 \cdots \bar{\nu}_n \cdot \pi \succ \text{stop} \star \bar{U} \cdot \bar{\nu}_1 \cdots \bar{\nu}_n \cdot \pi$
-
-

Figure 3: The language λ_c with test instructions

- A constant **stop** with no associated evaluation rule, and whose only purpose is to report the failure of a test of the experimental theory.

Following the terminology of Krivine, we call a *quasi-proof* any term which is built only from variables, abstractions, applications and the constant \mathfrak{c} . In other words, a quasi-proof is a term that contains neither the continuation constants k_π nor the extra constants **test_U** or **stop**. (The terminology of a ‘quasi-proof’ comes from the fact that programs extracted from proofs in pure classical second-order arithmetic are all of this form.)

Church numerals To every natural number $n \in \mathbb{N}$ we associate a closed term written \bar{n} (‘Church numeral n ’) and defined by

$$\bar{n} = \underbrace{\bar{s}(\dots(\bar{s} \bar{0})\dots)}_n \quad (\text{where } \bar{0} = \lambda\xi\phi.\xi \text{ and } \bar{s} = \lambda\nu\xi\phi.\phi(\nu\xi\phi))$$

Notice that with this definition, Church numeral \bar{n} is not in normal form in the sense of λ -calculus (except when $n = 0$).²

2.2 The relation of evaluation

The set of all processes is equipped with a binary relation of (*one step*) *evaluation* written $p \succ p'$, whose rules are given in Fig. 3. These rules comprise the usual evaluations rules of abstraction (GRAB), application (PUSH) and of the constants \mathfrak{c} (CALL/CC) and k_π (RESTORE), plus specific evaluation rules for the constants **test_U** we will describe with more details in subsection 3.4.

Notice that evaluation is deterministic, at least for a program that does not use test instructions: for each process $t \star \pi$ where t is not a test instruction, there is at most one process $t' \star \pi'$ such that $(t \star \pi) \succ (t' \star \pi')$. In many cases of course, there is no such process, typically when t is an abstraction (or a constant \mathfrak{c} or k_π) and when the stack π is empty, or simply when t is **stop**.

On the other hand, the evaluation rules of test instructions **test_U** such as presented in Fig. 3 are not deterministic (strictly speaking) since the three cases that define the outcome of the evaluation are not mutually exclusive: case 1 may happen for several values of k simultaneously, whereas case 2 may happen simultaneously with case 1. (Only case 3 is exclusive from cases 1 and 2.) However, it is always possible to define a priority between case 1 and case 2 (and inside case 1) in order to make the evaluation process completely deterministic. We will not discuss further the choice of making evaluation of test instructions deterministic or not: this is simply irrelevant in the realisability model (section ???), where both alternatives are supported indifferently.

In what follows, we write \succ^* the reflexive-transitive closure of \succ .

2.3 Extracting a program from a proof

The extraction mechanism actually consists of two extraction functions:

- A generic extraction function $d \mapsto d^*$ that extracts an *open* term d^* from a derivation d of an arbitrary sequent $\Gamma \vdash A$ in PA2. The term d^* is actually a quasi-proof (containing no test instruction) whose free variables represent the hypotheses in Γ .

²Such a definition is necessary to use the storage operators we will introduce in ???.

- A specific extraction function $d \mapsto \hat{d}$ (defined from the latter) that extracts a *closed* term \hat{d} from a derivation of the sequent $U_1, \dots, U_\ell \vdash \perp$ (where U_1, \dots, U_ℓ is the experimental theory). Unlike the (open) term d^* , the (closed) term \hat{d} contains test instructions corresponding to the experimental hypotheses that are used to derive the contradiction. This term is ready to be evaluated against the empty stack.

The generic extraction function The generic extraction function $d \mapsto d^*$ is recursively defined on the structure of the derivation using the equations of Fig. 4. To define this function, we first need to associate a distinct variable ξ_A to each formula A that appears as an hypothesis of a sequent somewhere in the derivation, while keeping the names consistent across deduction steps. (The map $A \mapsto \xi_A$ is naturally defined bottom-up in the derivation d .)

$$\left(\overline{\Gamma \vdash A} \right)^* = \begin{cases} \xi_A & \text{if } A \in \Gamma \\ \lambda \xi . \xi & \text{if } A \in \mathcal{A} \setminus \{4\text{th Peano axiom}\} \\ \lambda \xi . \xi (\lambda \phi . \phi) & \text{if } A = 4\text{th Peano axiom} \end{cases}$$

$$\left(\frac{\begin{array}{c} \vdots d \\ \Gamma, A \vdash B \end{array}}{\Gamma \vdash A \Rightarrow B} \right)^* = \lambda \xi_A . d^* \qquad \left(\frac{\begin{array}{cc} \vdots d_1 & \vdots d_2 \\ \Gamma \vdash A \Rightarrow B & \Gamma \vdash A \end{array}}{\Gamma \vdash B} \right)^* = d_1^* d_2^*$$

$$\left(\frac{\begin{array}{c} \vdots d \\ \Gamma \vdash A \end{array}}{\Gamma \vdash \forall x B} \right)^* = d^* \qquad \left(\frac{\begin{array}{c} \vdots d \\ \Gamma \vdash \forall x B \end{array}}{\Gamma \vdash \forall B \{x := e\}} \right)^* = d^*$$

$$\left(\frac{\begin{array}{c} \vdots d \\ \Gamma \vdash A \end{array}}{\Gamma \vdash \forall X B} \right)^* = d^* \qquad \left(\frac{\begin{array}{c} \vdots d \\ \Gamma \vdash \forall X B \end{array}}{\Gamma \vdash \forall B \{X(x_1, \dots, x_n) := A\}} \right)^* = d^*$$

$$\left(\overline{\Gamma \vdash ((A \Rightarrow B) \Rightarrow A) \Rightarrow A} \right)^* = \mathbf{\alpha}$$

Figure 4: Extraction of a term d^* from a derivation d in PA2

The specific extraction function Given a derivation d (in PA2) of the sequent $U_1, \dots, U_\ell \vdash \perp$ expressing that the experimental theory is contradictory, we let

$$\hat{d} := d^* \{ \xi_1 := M_1 \text{test}_{U_1}; \dots; \xi_\ell := M_n \text{test}_{U_1} \}$$

where ξ_1, \dots, ξ_ℓ are the term variables associated to the experimental hypotheses U_1, \dots, U_ℓ , respectively, and where M_1, \dots, M_n are closed quasi-proofs (called *storage operators*) we will define and discuss in subsection ???.

The following theorem shows that the evaluation of the program \hat{d} (in front of the empty stack) performs finitely many tests on the experimental theory

(corresponding to the steps where a test instruction test_U is evaluated) before it finds a tuple of parameters invalidating one of the hypotheses of the experimental theory:

Theorem 1 — *If d is a derivation of the sequent $U_1, \dots, U_\ell \vdash \perp$ (in PA2), then the evaluation of the process $\hat{d} \star \diamond$ eventually reaches a state of the form*

$$\text{stop} \star \bar{k} \cdot \bar{\nu}_1 \cdots \bar{\nu}_{n_k} \cdot \pi$$

where k is the index of an hypothesis of the experimental theory

$$U_k \equiv \forall^{\mathbb{N}} x_1 \cdots \forall^{\mathbb{N}} x_{n_k} E_k[x_1, \dots, x_{n_k}] \quad (1 \leq k \leq \ell)$$

and where $(\nu_1, \dots, \nu_{n_k})$ is a falsification of U_k .

The next section is devoted to the construction of a realisability model justifying the above constructions, and to the proof of Theorem 1.

3 The realisability model

3.1 Preliminaries

The saturated set $\perp\!\!\!\perp$ The construction of the realisability model is parameterised by a set of processes $\perp\!\!\!\perp$ which is *saturated* in the sense that

$$\text{If } p \succ p' \text{ and } p' \in \perp\!\!\!\perp, \text{ then } p \in \perp\!\!\!\perp$$

(The set $\perp\!\!\!\perp$ is also said to be *closed under anti-evaluation*.)

In the sequel, we will need a particular choice of $\perp\!\!\!\perp$, namely, the set $\perp\!\!\!\perp_0$ formed by all processes that eventually reach the **stop** in evaluation position:

$$\perp\!\!\!\perp_0 = \{p \mid \exists \pi \ p \succ^* \text{stop} \star \pi\}.$$

Extending the language of formulæ To define the interpretation function, we need to enrich the language of formulæ by adding a new predicate symbol \dot{F} of arity n for each function $F : \mathbb{N}^n \rightarrow \mathfrak{P}(\Pi)$ that associates a set of stacks (a ‘falsity value’) to each n -tuple of natural numbers:

$$A, B ::= \dots \mid \dot{F}(e_1, \dots, e_n) \quad (F : \mathbb{N}^n \rightarrow \mathfrak{P}(\Pi))$$

(While doing this extension, the cardinality of the language of formulæ jumps from the denumerable to the power of continuum.)

Valuations We call a *valuation* any function ρ that associates a natural number $\rho(x) \in \mathbb{N}$ to each first-order variable x , and that also associates a function $\rho(X) : \mathbb{N}^n \rightarrow \mathfrak{P}(\Pi)$ to each n -ary second-order variable X .

Given an open formula A (defined in the initial language) and a valuation ρ , we write $A[\rho]$ the closed formula of the extended language obtained by replacing in the formula A each free occurrence of a first-order variable x by the natural number $\rho(x) \in \mathbb{N}$ (seen as a numeric expression), and by replacing each free occurrence of a n -ary second-order variable X by the predicate symbol \dot{F} associated to the function $F = \rho(X) : \mathbb{N}^n \rightarrow \mathfrak{P}(\Pi)$.

3.2 The interpretation function

Let \perp be an arbitrary saturated set. To each closed formula of the extended language we associate two sets, namely:

- A set of closed terms $|A|$ that we call the *truth value* of A .
- A set of stacks $\|A\|$ that we call the *falsity value* of A .

The truth value $|A|$ of a closed formula A is indirectly defined from the falsity value $\|A\|$ by the equation

$$|A| = \|A\|^\perp = \{t \in \mathcal{T} \mid \forall \pi \in \|A\| \ t \star \pi \in \perp\}.$$

(Literally: the truth value of A is the set of all closed terms that are able to interact with all the refutation attempts given by the falsity value $\|A\|$.)

Notice that the operation $S \mapsto S^\perp$ that maps each falsity value to the corresponding truth value is contravariant: the larger the falsity value, the smaller the truth value, and vice-versa.

The falsity value of a closed formula A (of the extended language) is inductively defined from the following equations:

$$\begin{aligned} \|\dot{F}(e_1, \dots, e_n)\| &= F(\downarrow e_1, \dots, \downarrow e_n) \\ \|p(e_1, \dots, e_n)\| &= \begin{cases} \{t \cdot \pi \mid t \star \pi \in \perp\} & \text{if } \mathbf{Val}(p)(\downarrow e_1, \dots, \downarrow e_n) = 1 \\ \Pi & \text{if } \mathbf{Val}(p)(\downarrow e_1, \dots, \downarrow e_n) = 0 \end{cases} \\ \|A \Rightarrow B\| &= |A| \cdot \|B\| \\ \|\forall X A\| &= \bigcup_{F: \mathbb{N}^n \rightarrow \mathfrak{P}(\Pi)} \|A\{X := \dot{F}\}\| \end{aligned}$$

(Remember that $\downarrow e$ denotes the value of the closed numeric expression e .)

Since both sets $|A|$ and $\|A\|$ associated to each formula A depend on the saturated set of processes \perp , we will sometimes write them $|A|_\perp$ and $\|A\|_\perp$, respectively, in order to prevent any confusion.

Given a closed term t and a closed formula A , we say that t *realises* A and write $t \Vdash A$ when $t \in |A|$ (relatively to a fixed saturated set \perp). In the case where $\perp = \perp_0$, we write $t \Vdash_0 A$ for $t \in |A|_{\perp_0}$.

3.3 Realising the proofs of PA2

The rules given in Fig. 4 define an extraction function that transforms any proof done in PA2 (cf Fig. 1) into a term d^* . This generic extraction mechanism is correct w.r.t. the realisability model in the following sense:

Lemma 1 (Adequacy) — *Let $A_1, \dots, A_n \vdash B$ be a derivable sequent of PA2 (using the rules of Fig. 1) and let d be a derivation of this sequent. For all saturated sets \perp , for all valuations ρ and for all closed terms u_1, \dots, u_n such that $u_1 \Vdash A_1, \dots, u_n \Vdash A_n$ (relatively to the set \perp), we have*

$$d^* \{\xi_1 := u_1; \dots; \xi_n := u_n\} \Vdash B$$

where ξ_1, \dots, ξ_n are the term variables associated to the assumptions A_1, \dots, A_n in the extraction process.

Proof. The proof is done by structural induction on the derivation d . The cases corresponding to purely logical rules is standard (see for instance [2, 4]). Regarding the axioms of PA2, it suffices to check that the term $\mathbf{I} = \lambda\xi.\xi$ is a realiser of the third Peano axiom (injectivity of successor) and of all defining equalities of primitive recursive function symbols, whereas the term $\lambda\xi.\xi\mathbf{I}$ is a realiser of the fourth Peano axiom (non-confusion). \square

3.4 Evaluating test instructions

Storage operators Let us first enrich the language of formulæ with a new syntactic construct $\{e\} \Rightarrow B$ (where e is a numeric expression and where B a formula) whose falsity value is defined by

$$\|\{e\} \Rightarrow B\| = \overline{\downarrow e} \cdot \|B\|$$

(in the case where both e and B are closed). The truth value $|\{e\} \Rightarrow B|$ is defined from the falsity value $\|\{e\} \Rightarrow B\|$ the usual way: $|\{e\} \Rightarrow B| = \|\{e\} \Rightarrow B\|^\perp$.

Given a natural number n , we call a *storage operator* of arity n any closed quasi-proof M_n such that for all formulæ $A[x_1, \dots, x_n]$ (possibly) depending on n first-order variables x_1, \dots, x_n , we have

$$\begin{aligned} M_n \Vdash & \forall x_1 \cdots \forall x_n (\{x_1\} \Rightarrow \cdots \Rightarrow \{x_n\} \Rightarrow A[x_1, \dots, x_n]) \\ & \Rightarrow \forall^{\mathbb{N}} x_1 \cdots \forall^{\mathbb{N}} x_n A[x_1, \dots, x_n] \end{aligned}$$

Intuitively, a storage operator M_n is a ‘wrapper’ that transforms any program t whose computational behaviour is specified only when its arguments are fully developed Church numerals (that we can understand as ‘intuitionistic integers’) into a program whose behaviour is specified when its arguments are arbitrary realisers of the predicate $\mathbf{Nat}(x)$ (that we can understand as ‘classical integers’), using the very same formula.

Krivine showed [2, 4] that such terms M_n exist for all arities $n \geq 0$. For instance, we can take

$$\begin{aligned} M_0 & \equiv \lambda\phi.\phi \\ M_1 & \equiv \lambda\phi.\lambda\nu.\nu\phi(\lambda\psi.\lambda\xi.\psi(\bar{s}\xi))\bar{0} \\ M_n & \equiv \lambda\phi.M_1(\lambda\nu.M_{n-1}(\phi\nu)) \quad (n \geq 2) \end{aligned}$$

How test instructions work Consider a testable universal formula

$$U \equiv \forall^{\mathbb{N}} x_1 \cdots \forall^{\mathbb{N}} x_n E[x_1, \dots, x_n],$$

that belongs to the experimental theory, writing

$$E[x_1, \dots, x_n] \equiv L_1[x_1, \dots, x_n] \Rightarrow \cdots \Rightarrow L_k[x_1, \dots, x_n] \Rightarrow R[x_1, \dots, x_n],$$

where each formula $L_i[x_1, \dots, x_n]$ is either elementary or the negation of an elementary formula, and where the formula $R[x_1, \dots, x_n]$ is elementary.

The evaluation mechanism of the test instruction \mathbf{test}_U associated to the universal formula U is the following: each time the instruction \mathbf{test}_U comes into evaluation position in front of a stack starting with n Church numerals $\bar{\nu}_1, \dots, \bar{\nu}_n$, the instruction tests the formula $E[\nu_1, \dots, \nu_n]$ by computing its truth value (using the functions $\mathbf{Val}(p)$ associated to the experimental predicates).

When the test succeeds, the instruction test_U consumes the n top-most arguments of the stacks and evaluates to a quasi-proof t that realises the formula $E[\nu_1, \dots, \nu_n]$. This quasi-proof may depend on the way the implication $E[\nu_1, \dots, \nu_n]$ is satisfied:

- We can take $t \equiv \lambda \xi_1 \cdots \xi_k . \xi_i \mathbf{I}$ when a premise $L_i[\nu_1, \dots, \nu_n]$ is falsified for some $i \in [1..k]$ (Fig. 3, item 1).
- Or we can take $t \equiv \lambda \xi_1 \cdots \xi_k . \mathbf{I}$ when the conclusion $R[\nu_1, \dots, \nu_n]$ is satisfied (Fig. 3, item 2).

Notice that in both cases, the quasi-proof t is a realiser of the testable formula $E[\nu_1, \dots, \nu_n]$ independently from the choice of the saturated set \perp .

When the test fails, the instruction test_U does not consume the n top-most arguments of the stack, but pushes on the stack the Church numeral (written \bar{U}) of the index of the formula U in the experimental theory, and evaluates to the instruction stop . Of course, the term $\text{stop } \bar{U} \bar{\nu}_1 \cdots \bar{\nu}_n$ does not realise the formula $E[\nu_1, \dots, \nu_n]$ for an arbitrary choice of the saturated set \perp , but it realises this formula in the case where $\perp = \perp_0$, from the very definition of this set. The latter remark explains why the instruction test_U is correct only w.r.t. the saturated set \perp_0 .

Lemma 2 — *For every formula U of the experimental theory, one has:*

$$\text{test}_U \Vdash_0 \forall x_1 \cdots \forall x_n (\{x_1\} \Rightarrow \cdots \Rightarrow \{x_n\} \Rightarrow E[x_1, \dots, x_n]).$$

Proof. Let $\nu_1, \dots, \nu_n \in \mathbb{N}$ and $\pi \in \|E(\nu_1, \dots, \nu_n)\|$. We have to show that $\text{test}_U \star \bar{\nu}_1 \cdots \bar{\nu}_n \cdot \pi \in \perp_0$. We distinguish two cases:

1. Either $\mathbf{Val}(E[\nu_1, \dots, \nu_n]) = 1$. In this case, we have

$$\text{test}_U \star \bar{\nu}_1 \cdots \bar{\nu}_n \cdot \pi \succ t \cdot \pi,$$

where t is a quasi-proof given by the first item of Fig. 3 (depending on an hypothesis of the testable formula $E[\nu_1, \dots, \nu_n]$ is falsified or its conclusion is satisfied). In all cases, we easily check that $t \Vdash E[\nu_1, \dots, \nu_n]$, hence we have $t \star \pi \in \perp_0$, and thus $\text{test}_U \star \bar{\nu}_1 \cdots \bar{\nu}_n \cdot \pi \in \perp_0$ by anti-reduction.

2. Either $\mathbf{Val}(E[\nu_1, \dots, \nu_n]) = 0$. In this case, we immediately get

$$\text{test}_U \star \bar{\nu}_1 \cdots \bar{\nu}_n \cdot \pi \succ \text{stop} \star \bar{U} \cdot \bar{\nu}_1 \cdots \bar{\nu}_n \cdot \pi \in \perp_0. \quad \square$$

Corollary 3 — *For every formula U of the experimental theory, we have:*

$$M_n \text{test}_U \Vdash_0 \forall^{\mathbb{N}} x_1 \cdots \forall^{\mathbb{N}} x_n E[x_1, \dots, x_n]$$

(where M_n is a storage operator of arity n).

Proof. Immediately follows from the latter proposition and the definition of storage operators. \square

3.5 Correctness of the specific extraction function

We can now give the

Proof of Theorem 1. Consider a derivation d of the sequent $U_1, \dots, U_\ell \vdash \perp$ (in PA2). From Lemma 1 (Adequacy) we know that

$$d^* \{ \xi_1 := u_1; \dots; \xi_\ell := u_\ell \} \Vdash \perp$$

for all realisers $u_1 \Vdash U_1, \dots, u_\ell \Vdash U_\ell$ (and for an arbitrary choice of the saturated set \perp). If we now take $\perp = \perp_0$ and let $u_i = M_{n_i} \text{test}_{U_i}$ (for all $1 \leq i \leq \ell$), we immediately deduce from Corollary 3 that

$$\hat{d} = d^* \{ \xi_1 := M_{n_1} \text{test}_{U_1}; \dots; \xi_\ell := M_{n_\ell} \text{test}_{U_\ell} \} \Vdash_0 \perp.$$

Which means that $\hat{d} \star \pi \in \perp_0$ for all stacks $\pi \in \|\perp\| = \Pi$, and in particular for $\pi = \diamond$. From the definition of \perp , there exists a stack π such that

$$\hat{d} \star \diamond \succ^* \text{stop} \star \pi.$$

But since by construction the term \hat{d} does not contain the instruction **stop** (only the constants \mathfrak{c} and test_U may appear in \hat{d}), the final state $\text{stop} \star \pi$ necessarily comes from a failing experiment (Fig. 3, item 3), so that the stack π is actually of the form

$$\pi \equiv \bar{k} \cdot \bar{\nu}_1 \cdots \bar{\nu}_{n_k} \cdot \pi',$$

where ν_1, \dots, ν_{n_k} is a tuple of parameters falsifying the testable universal formula U_k ($1 \leq k \leq \ell$). \square

3.6 The experimental modus tollens

Let us now consider an experimental theory formed by testable universal formulæ U_1, \dots, U_ℓ , as well as a testable universal formula

$$V \equiv \forall^{\mathbb{N}} x_1 \cdots \forall^{\mathbb{N}} x_n F[x_1, \dots, x_n]$$

such that $U_1, \dots, U_\ell \vdash V$ is derivable in PA2.

Theorem 2 (Experimental modus tollens) — *From a derivation (in PA2) of the sequent $U_1, \dots, U_\ell \vdash V$ and a falsification $(\nu_1, \dots, \nu_n) \in \mathbb{N}^n$ of the formula V , it is possible to construct a term t such that the evaluation of the process $t \star \diamond$ eventually reaches a final state that gives a falsification of one of the formulæ U_1, \dots, U_ℓ in the sense of Theorem 1.*

Proof. Let us write the formula $F[x_1, \dots, x_n]$ as

$$F[x_1, \dots, x_n] \equiv L_1[x_1, \dots, x_n] \Rightarrow \cdots \Rightarrow L_k[x_1, \dots, x_n] \Rightarrow R[x_1, \dots, x_n],$$

where each formula $L_i[x_1, \dots, x_n]$ is either elementary or the negation of an elementary formula, and where the formula $R[x_1, \dots, x_n]$ is elementary. From the derivation of the sequent $U_1, \dots, U_\ell \vdash V$, we easily build (in PA2) a derivation d of the sequent

$$U_1, \dots, U_\ell, L_1[\nu_1, \dots, \nu_n], \dots, L_k[\nu_1, \dots, \nu_n], \neg R[\nu_1, \dots, \nu_n] \vdash \perp$$

By applying Theorem 1 to the above sequent (whose hypotheses are testable universal formulæ), we know that the process $\hat{d} \star \diamond$ eventually reaches a finite state that gives a falsification (in the sense of Theorem 1) of one of the hypotheses of the sequent, that is

1. either a falsification of one of the formulæ U_1, \dots, U_ℓ ;
2. either a falsification of one of the formulæ $L_i[\nu_1, \dots, \nu_n]$;
3. either a falsification of the formula $\neg R[\nu_1, \dots, \nu_n]$.

But since $\mathbf{Val}(F[\nu_1, \dots, \nu_n]) = 0$, we know that

$$\mathbf{Val}(L_i[\nu_1, \dots, \nu_n]) = 1 \quad \text{and} \quad \mathbf{Val}(R[\nu_1, \dots, \nu_n]) = 0 \quad (1 \leq i \leq k)$$

Hence cases 2 and 3 are impossible, and case 1 is the only possible case. Therefore the falsification coming from the evaluation of $\hat{d} \star \diamond$ is a falsification of one of the formulæ U_1, \dots, U_ℓ . \square

References

- [1] J.-Y. Girard, Y. Lafont, P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.
- [2] J.-L. Krivine. *A general storage theorem for integers in call-by-name lambda-calculus*. Th. Comp. Sc., 129, p. 79-94 (1994).
- [3] J.-L. Krivine. *Typed lambda-calculus in classical Zermelo-Fraenkel set theory*. Arch. Math. Log., 40, 3, p. 189-205 (2001).
- [4] J.-L. Krivine. *Realizability in classical logic*. Course notes of a series of lectures given in the University of Marseille, may 2004 (last revision: july 2005). To appear in *Panoramas et synthèses*, Société Mathématique de France.
- [5] K. R. Popper. *The Logic of Scientific Discovery*. 1959. (Routledge, 1992)