

Extracting reliable witnesses from classical realisers

Alexandre Miquel

February, 2008

(DRAFT)

1 Introduction

One of the most fascinating properties of intuitionistic logic (and more generally of constructive mathematics) is the *witness property*, that makes possible to extract an object x_0 such that $A(x_0)$ —the witness—from any proof of the formula $\exists x A(x)$ formalised in this setting. Unfortunately, this property does not hold in classical logic. For instance, the formula

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

(where C is an arbitrary closed formula) has an obvious classical proof (by excluded middle on C), but this proof gives no hint to compute the correct value of x , and the only way to find a suitable witness is to prove or disprove the formula C —which may be impossible if C is undecidable.

However, the witness property holds for classically provable existential formulæ in a particular case: when the quantification is numeric—that is, of the form $\exists^{\mathbb{N}} x A(x)$ —and when the quantified predicate $A(x)$ is decidable. In this case, a witness can be obtained by simply testing the property $A(x)$ on all numerals $x \in \mathbb{N}$ successively, stopping on the first one that meets the property.¹ But again in this case, the proof is useless: the only knowledge that the formula is true (in the standard model) is sufficient to perform the extraction process, independently from the proof.

In this paper, we reconsider the problem of extracting witnesses from classical proofs in the more general framework of Krivine realisability. In particular, we will show how to extract witnesses from classical realisers of existential formulæ, and study conditions to ensure that these witnesses are reliable.

2 The language λ_c

We start by introducing Krivine’s language λ_c whose programs will be used to realise the proofs of second-order arithmetic (cf sections 3 and 4).

¹In Kleene-style realisability, this procedure is actually the standard way to realise Markov principle $\neg\neg\exists^{\mathbb{N}} x A(x) \Rightarrow \exists^{\mathbb{N}} x A(x)$ (with $A(x)$ decidable).

2.1 Terms, stacks and processes

We consider three denumerable and pairwise disjoint sets of symbols: a set \mathcal{X} of variables (written x, y, z , etc.), a set \mathcal{C} of *constants* (written c, c', d , etc.) and a set \mathcal{B} of *stack constants* (written α, β, γ , etc.) The language λ_c distinguishes three syntactic categories: *terms* (or *programs*), *stacks* (or *evaluation contexts*) and *processes* (or *commands*).

Terms The (open) *terms* (notation: t, u , etc.) of the language λ_c are simply ordinary λ -terms with constants ranging over the set \mathcal{C} :

$$\text{TERMS} \quad t, u ::= x \mid \lambda x. t \mid tu \mid c \quad (c \in \mathcal{C})$$

The notions of free and bound variables are defined as usual, and the set of free variables of a term t is written $FV(t)$. The standard operation of (external) substitution is defined as usual and written $t\{x := u\}$. The set of all terms (resp. of all closed terms) is written $\Lambda(\mathcal{X})$ (resp. Λ).

Stacks *Stacks* (notation: π, π' , etc.) are finite ordered lists of closed terms terminated by a stack constant, that is:

$$\text{STACKS} \quad \pi ::= \alpha \mid t \cdot \pi \quad (\alpha \in \mathcal{B}, t \in \Lambda)$$

The set of all stacks is written Π .

Processes Finally, *processes* (notation: p, q , etc.) are simply pairs formed by a closed term t and a stack π , written $t \star \pi$. That is:

$$\text{PROCESSES} \quad p, q ::= t \star \pi \quad (t \in \Lambda, \pi \in \Pi)$$

The set of all processes is written $\Lambda \star \Pi$.

The constants \mathfrak{c} and k_π We distinguish several constants (in the set \mathcal{C}) which we give a special meaning, namely:

1. A constant $\mathfrak{c} \in \mathcal{C}$ that represents the operator *call/cc* (*call with current continuation*), and that will be used to realise Peirce's law.
2. For each stack $\pi \in \Pi$, a constant $k_\pi \in \mathcal{C}$ that represents the continuation associated to π . We assume that $k_\pi \neq \mathfrak{c}$ for all π , and that the correspondence $\pi \in \Pi \mapsto k_\pi \in \mathcal{C}$ is into.

On the other hand, we do not assume that the constants \mathfrak{c} and k_π exhaust the set \mathcal{C} of all constants. Instead, it is convenient to assume that the complement $\mathcal{C} \setminus \{\mathfrak{c}; k_\pi \mid \pi \in \Pi\}$ is not empty (and actually infinite) in order to pick new constants when needed, that we endow with specific evaluation rules.

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

$$\bar{n} = \underbrace{\bar{s}(\dots(\bar{s} \bar{0})\dots)}_n \quad (\text{where } \bar{0} = \lambda xy. x \text{ and } \bar{s} = \lambda nxy. y(nxy))$$

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 Evaluation

We also assume that the set $\Lambda \star \Pi$ of all processes is equipped with a binary relation of *evaluation*, written $p \succ p'$, that meets the following axioms:

$$\begin{array}{lll} \lambda x. t \star u \cdot \pi & \succ & t\{x := u\} \star \pi \\ tu \star \pi & \succ & t \star u \cdot \pi \\ \mathbf{c} \star t \cdot \pi & \succ & t \star \mathbf{k}_\pi \cdot \pi \\ \mathbf{k}_\pi \star t \cdot \pi' & \succ & t \star \pi \end{array}$$

In what follows, we write $p \succ^* p'$ the reflexive-transitive closure of the evaluation relation $p \succ p'$. Finally, given two terms t and t' , we write $t \succ^* t'$ when $t \star \pi \succ^* t' \star \pi$ for all stacks $\pi \in \Pi$.

3 Realising second-order logic formulæ

3.1 The language of second-order arithmetic

The language of second-order arithmetic is made of two kinds of expressions: *numeric expressions* (a.k.a. first-order terms) and *formulæ*, whose syntax is the following:

$$\begin{array}{ll} \text{NUMERIC EXPR.} & e, e' ::= x \mid f(e_1, \dots, e_n) \\ \text{FORMULÆ} & A, B ::= X(e_1, \dots, e_n) \mid A \Rightarrow B \mid \forall x B \mid \forall X B \end{array}$$

Here we use letters x, y, z etc. (resp. X, Y, Z , etc.) for first-order variables (resp. second-order variables). We assume given function symbols f, g, h , etc. for all primitive recursive (definitions of) functions. These function symbols include the constant symbol 0 (zero) and the unary function symbol s (successor). Given a closed numeric expression e , we write $\downarrow e$ the natural number denoted by e (using the plain interpretation of the function symbols which appear in e).

Other connectives ($\perp, \top, \wedge, \vee$), quantifiers (first- and second-order \exists) and Leibniz equality are defined using the well-known second-order encodings:

$$\begin{array}{ll} \perp & \equiv \forall X X \\ \top & \equiv \forall X (X \Rightarrow X) \\ \neg A & \equiv A \Rightarrow \perp \\ A \wedge B & \equiv \forall X ((A \Rightarrow B \Rightarrow X) \Rightarrow X) \\ A \vee B & \equiv \forall X ((A \Rightarrow X) \Rightarrow (B \Rightarrow X) \Rightarrow X) \\ \exists x A(x) & \equiv \forall Y (\forall x (A(x) \Rightarrow Y) \Rightarrow Y) \\ \exists X A(X) & \equiv \forall Y (\forall X (A(X) \Rightarrow Y) \Rightarrow Y) \\ x = y & \equiv \forall Z (Z(x) \Rightarrow Z(y)). \end{array}$$

²Such a definition of Church numerals is necessary to make possible the use of storage operators we will introduce in subsection 4.3.

3.2 Preliminaries

The saturated set \perp The construction of the realisability model of second-order arithmetic is parameterised by a set of processes $\perp \subseteq \Lambda \star \Pi$ which is *saturated* in the sense that

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

(The set \perp is also said to be *closed under anti-evaluation*.) Intuitively, \perp represents a set of accepting processes (w.r.t. some correctness criterion), and the condition of saturation expresses that each processes that evaluates to an accepting process is itself an accepting process. Given a set of stacks $S \subset \Pi$ (resp. a set of closed terms $T \subset \Lambda$) we write

$$\begin{aligned} S^\perp &= \{t \in \Lambda \mid \forall \pi \in S \quad t \star \pi \in \perp\} \\ (\text{resp. } T^\perp &= \{\pi \in \Pi \mid \forall t \in T \quad t \star \pi \in \perp\}). \end{aligned}$$

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 $F = \rho(X) : \mathbb{N}^n \rightarrow \mathfrak{P}(\Pi)$.

3.3 Interpreting formulæ

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

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

(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:

$$\begin{aligned} \|\dot{F}(e_1, \dots, e_n)\| &= F(\downarrow e_1, \dots, \downarrow e_n) \\ \|A \Rightarrow B\| &= |A| \cdot \|B\| = \{t \cdot \pi \mid t \in |A|, \pi \in \|B\|\} \\ \|\forall x B\| &= \bigcup_{n \in \mathbb{N}} \|B\{x := n\}\| \\ \|\forall X B\| &= \bigcup_{F: \mathbb{N}^n \rightarrow \mathfrak{P}(\Pi)} \|B\{X := \dot{F}\}\| \end{aligned}$$

(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 (w.r.t. the saturated set \perp) and write $t \Vdash_\perp A$ when $t \in |A|_\perp$. (We will frequently drop the subscript \perp and write $t \Vdash A$ for $t \Vdash_\perp A$ when the saturated set \perp is clear from the context.) Finally, we say that t is a *universal realiser* of A and write $t \Vdash\!\!\Vdash A$ when $t \Vdash_\perp A$ for all saturated sets \perp .

The standard model as a degenerated case The interpretation defined above depends on the parameterising saturated set of processes \perp . A degenerated (but nonetheless interesting) case is when $\perp = \emptyset$. In this particular case, it is easy to see that there are only two different truth values: the full truth value Λ , which is the orthogonal of the empty set of stacks, and the empty truth value \emptyset , which is the orthogonal of any nonempty set of stacks. We then recover the boolean interpretation of all connectives and quantifiers, in the sense that:

Fact 1 — *If $\perp = \emptyset$, then for all closed formulæ A :*

$$|A|_\perp = \begin{cases} \Lambda & \text{if } A \text{ is true} \\ \emptyset & \text{if } A \text{ is false} \end{cases} \quad (\text{in the full standard model})$$

By ‘full standard model’, we mean the model where numeric expressions are interpreted by natural numbers and where *nary* predicate symbols are interpreted by all the possible *nary* relations.

As we shall see in the forthcoming subsection, many formulæ A (and in particular, all the provable formulæ) can be given a universal realiser $t \Vdash\!\!\Vdash A$. From Fact 1 it is clear that

Fact 2 — *Every formula A that has a universal realiser is true in the full standard model.*

3.4 Adequacy

The notion of realisability is connected with the ordinary relation of typing via the following lemma:

Lemma 3 (Adequacy) — *If the judgement $x_1 : A_1, \dots, x_n : A_n \vdash t : B$ is derivable according to the rules of Fig. 1, then for all valuations ρ and for all realisers $u_1 \Vdash A_1[\rho], \dots, u_n \Vdash A_n[\rho]$ we have $t\{x_1 := u_1; \dots; x_n := u_n\} \Vdash B[\rho]$.*

	$\overline{\Gamma \vdash x : A} \quad ((x:A) \in \Gamma)$
(Axiom)	
(\Rightarrow)	$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x. t : A \Rightarrow B} \quad \frac{\Gamma \vdash t : A \Rightarrow B \quad \Gamma \vdash t : A}{\Gamma \vdash tu : B}$
(\forall^1)	$\frac{\Gamma \vdash t : B}{\Gamma \vdash t : \forall x B} \quad (x \notin FV(\Gamma)) \quad \frac{\Gamma \vdash t : \forall x B}{\Gamma \vdash t : B\{x := e\}}$
(\forall^2)	$\frac{\Gamma \vdash t : B}{\Gamma \vdash t : \forall X B} \quad (X \notin FV(\Gamma)) \quad \frac{\Gamma \vdash t : \forall X B}{\Gamma \vdash t : B\{X := \lambda x_1 \dots x_n. A\}}$
(Peirce's law)	$\overline{\Gamma \vdash \mathbf{c} : ((A \Rightarrow B) \Rightarrow A) \Rightarrow A}$

Figure 1: Typing rules of classical second-order logic

In the particular case where both the formula A and the term t are closed, the statement $\vdash t : A$ (typing) entails $t \Vdash A$ (universal realisability). As we shall see in the sequel, realisability is much stronger than typing.

4 Realising arithmetic formulæ

From Lemma 3, it is clear that any formula A which is provable in classical second-order logic has at least a uniform realiser $t \Vdash A$, which is extracted from the proof by reading each deduction rule as a typing rule of Fig. 1 (following the Curry-Howard correspondence).³ In this section, we aim to extend this result to all reasoning principles of arithmetic.

4.1 Realising equalities and inequalities

Before tackling the problem of induction, let us see how arithmetical equalities and inequalities can be realised. For that, we first check that:

Fact 4 — *Given a fixed saturated set \perp :*

1. $\|\perp\| = \|\forall X X\| = \Pi$
2. $\|\top\| = \|\forall X (X \Rightarrow X)\| = \{u \cdot \pi \mid (u \star \pi) \in \perp\}$

³Remember that classical reasoning principles can be deduced from Peirce's law.

3. For all closed individuals e_1 and e_2 :

$$\|e_1 = e_2\| = \begin{cases} \{u \cdot \pi \mid (u \star \pi) \in \perp\} & \text{if } \downarrow e_1 = \downarrow e_2 \\ \Lambda \cdot \Pi & \text{otherwise} \end{cases}$$

(writing $\Lambda \cdot \Pi = \{u \cdot \pi \mid u \in \Lambda, \pi \in \Pi\}$).

From this, we immediately deduce the following:

Lemma 5 — Given two closed numeric expressions e_1 and e_2 :

1. If $\downarrow e_1 = \downarrow e_2$, then: $\lambda z . z \Vdash e_1 = e_2$
2. If $\downarrow e_1 \neq \downarrow e_2$, then: $\lambda z . zu \Vdash \neg(e_1 = e_2)$ (for any term u)

Lemma 6 (Realising Peano 3rd and 4th axioms)

1. $\lambda z . z \Vdash \forall x \forall y (s(x) = s(y) \Rightarrow x = y)$
2. $\lambda z . zu \Vdash \forall x \neg(s(x) = 0)$ (for any term u)

Lemma 7 (Uniform equalities) — Let $e_1[x_1, \dots, x_n]$ and $e_2[x_1, \dots, x_n]$ be two numeric expressions with free variables x_1, \dots, x_n . If $\downarrow e_1[p_1, \dots, p_n] = \downarrow e_2[p_1, \dots, p_n]$ for all $p_1, \dots, p_n \in \mathbb{N}$, then

$$\lambda z . z \Vdash \forall x_1 \cdots \forall x_n e_1[x_1, \dots, x_n] = e_2[x_1, \dots, x_n].$$

In particular, the lemma above shows that all the defining equations of primitive recursive functions—such as $\forall x \forall y (x + s(y) = s(x + y))$ —are uniformly realised by the identity function. However this result is much more general and also deals with equalities that do not come from a direct computation—typically, the commutativity of addition $\forall x \forall y (x + y = y + x)$.

4.2 Induction principle

In second-order arithmetic, the induction principle is given for free using a standard trick, which 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

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

By purely logical means, it is easy to 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:

$$\begin{aligned} \forall^{\text{nat}} x A[x] &\equiv \forall x (\text{nat}(x) \Rightarrow A[x]) \\ \exists^{\text{nat}} x A[x] &\equiv \forall X (\forall x (\text{nat}(x) \Rightarrow A(x) \Rightarrow X) \Rightarrow X) \end{aligned}$$

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

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

Extracting realisers from the corresponding proofs, we get the following:

Fact 8 — *Using the abbreviations $\bar{0} = \lambda xy . x$, $\bar{s} = \lambda nxy . y(nxy)$ and $\bar{n} = \bar{s}^n \bar{0}$ (Church numeral n , for all $n \in \mathbb{N}$), we have:*

1. $\bar{n} \Vdash \text{nat}(n)$ for all $n \in \mathbb{N}$
2. $\bar{s} \Vdash \forall x (\text{nat}(x) \Rightarrow \text{nat}(s(x)))$

Combining induction with the defining equalities of primitive recursive function symbols, we can check that all primitive recursive functions f map well-formed numerals to well-formed numerals, and more generally that any numeric expression denotes a well-formed numeral provided its free variables are restricted to the class of well-formed numerals:

Lemma 9 (Totality of primitive recursive functions) — *For all function symbols f of arity n , the formula*

$$\forall^{\text{nat}} x_1 \dots \forall^{\text{nat}} x_n \text{nat}(f(x_1, \dots, x_n))$$

is provable in second-order arithmetic, and thus has a uniform realiser.

Lemma 10 (Totality of numeric expressions) — *For all numeric expressions $e[x_1, \dots, x_n]$ with free variables x_1, \dots, x_n , the formula*

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

is provable in second-order arithmetic, and thus has a uniform realiser.

4.3 Storage operators

To understand the notion of a storage operator, it is necessary to 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

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

(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: $|\{e\} \Rightarrow B| = \|\{e\} \Rightarrow B\|^\perp$.

We call a *storage operator* (for natural numbers) any closed quasi-proof M such that

$$M \Vdash \forall x \forall Z ((\{x\} \Rightarrow Z) \Rightarrow \text{nat}(x) \Rightarrow Z)$$

Intuitively, a storage operator is a term M that transforms any function f which is able to produce a value of some type from a given Church numeral \bar{n} (in its developed form) into a function $M f$ which is able to produce a value of the same type from any realiser $t \Vdash \text{nat}(n)$. It is easy to build such an operator, for instance by taking $M \equiv \lambda f n . n f (\lambda h x . h (\bar{s} x)) \bar{0}$ [2, 4].

Storage operators have an interesting computational behaviour which is given by the following lemma:

Lemma 11 — *If M is a storage operator, then for all terms u , for all stacks π , for all $n \in \mathbb{N}$ and for all universal realisers $t \Vdash \mathbf{nat}(n)$, we have:*

$$M \star u \cdot t \cdot \pi \succ^* u \star \bar{n} \cdot \pi.$$

Proof. Consider the set $S = \{\pi\} \subseteq \Pi$ and let \perp be the set of all processes p such that $p \succ^* u \star \bar{n} \cdot \pi$. By construction of \perp and S we have $u \Vdash \{n\} \Rightarrow \dot{S}$ (for this choice of \perp), hence $M_1 \star u \cdot t \cdot \pi \in \perp$. \square

As shown by the above lemma, a storage operator evaluates any universal realiser $t \Vdash \mathbf{nat}(n)$ into the corresponding Church numeral \bar{n} before passing it to the desired function. Storage operators can thus be understood as a way to perform call-by-value inside the call-by-name discipline of λ_c . Also notice that this lemma entails that, when the evaluation is deterministic, the sets of universal realisers of the formulæ $\mathbf{nat}(n)$ (for $n \in \mathbb{N}$) are pairwise disjoint.⁴

By subtyping, a storage operator M can also be given the following types:

$$\begin{aligned} M &\Vdash \forall x \forall Z ((\{x\} \Rightarrow Z) \Rightarrow \mathbf{nat}(x) \Rightarrow Z) \\ M &\Vdash \forall Z \forall x ((\{x\} \Rightarrow Z(x)) \Rightarrow \mathbf{nat}(x) \Rightarrow Z(x)) \\ M &\Vdash \forall Z (\forall x (\{x\} \Rightarrow Z(x)) \Rightarrow \forall^{\mathbf{nat}} x Z(x)) \end{aligned}$$

With the latter specification, a storage operator M_n can be seen as a wrapper that transforms any function f whose computational behaviour is specified only when its argument is a fully developed Church numeral (that we can understand as an ‘intuitionistic integer’) into a function whose behaviour is specified when its argument is an arbitrary realiser of the predicate $\mathbf{nat}(x)$ (that we can understand as a ‘classical integer’).

Storage operators of arity k More generally, we say that a quasi-proof M_k is a *storage operator of arity k* when

$$M_k \Vdash \forall x_1 \dots \forall x_k \forall X \left((\{x_1\} \Rightarrow \dots \Rightarrow \{x_k\} \Rightarrow X) \Rightarrow (\mathbf{nat}(x_1) \Rightarrow \dots \Rightarrow \mathbf{nat}(x_k) \Rightarrow X) \right).$$

Such operators can be defined for all arities $k \geq 0$ by setting

$$\begin{aligned} M_0 &\equiv \lambda f . f \\ M_1 &\equiv \lambda f . \lambda n . n f (\lambda h . \lambda x . h (\bar{s} x)) \bar{0} \\ M_k &\equiv \lambda f . M_1(\lambda n . M_{k-1}(f n)). \end{aligned} \quad (k \geq 2)$$

(Of course, the properties discussed above immediately generalise to storage operators of arbitrary arity.)

⁴This is no more the case with a non-deterministic evaluation. For instance if we add in the language of realisers a constant \dagger with two evaluation rules

$$\dagger \star t \cdot u \cdot \pi \succ t \cdot \pi \quad \text{and} \quad \dagger \star t \cdot u \cdot \pi \succ u \cdot \pi,$$

then it is straightforward to check that the quasi-proof $\dagger \bar{0} \bar{1}$ is both a universal realiser of $\mathbf{nat}(0)$ and of $\mathbf{nat}(1)$.

4.4 Conditional realisers

There are many situations where we can build a universal realiser t of a (closed) formula A under the only assumption that this formula is true (in the standard model). This situation typically occurs when the particular shape of A is sufficient to predict a realiser of A —provided A is true. For instance, we know that any equality $e_1 = e_2$ is (universally) realised by the term $\lambda z . z$ *provided* this equality is true, and that any inequality $e_1 \neq e_2$ is (universally) realised by the term $\lambda z . zz$ *provided* this inequality is true. This remark naturally leads to the following definition:

Given a closed formula A (of the enriched language), we say that a closed term t *conditionally realises* the formula A and write $t \Vdash^\circ A$ when the truth of A in the standard model implies that t is a universal realiser of A . When $t \Vdash^\circ A$, we say that t is a *conditional realiser* of A .

With this notation, we thus have $\lambda z . z \Vdash^\circ e_1 = e_2$ and $\lambda z . zz \Vdash^\circ e_1 \neq e_2$ for all closed numeric expressions e_1 and e_2 . (This property immediately generalises to all Π_1^0 formulæ.) The existence of conditional realisers actually extends to the whole (first-order) arithmetical hierarchy:

Theorem 12 — *For each closed formula of the form*

$$A \equiv \forall^{\text{nat}} x_1 \exists^{\text{nat}} y_1 \cdots \forall^{\text{nat}} x_k \exists^{\text{nat}} y_k f(x_1, y_1, \dots, x_k, y_k) = 0$$

there exists a quasi-proof t such that $t \Vdash^\circ A$.

Proof. This is an immediate consequence of Theorem 14 p. 15 [4] (where the property is proved for a slightly larger class of formulæ). \square

This theorem is fundamental, since it expresses that true formulæ (w.r.t. the standard model) and universally realisable formulæ actually coincide in the first-order fragment of the language of second-order arithmetic.

5 The witness property

5.1 The naive extraction mechanism

In intuitionistic realisability *à la* AF2, it is straightforward to extract a witness from a realiser t of the formula $\exists^{\text{nat}} x A[x]$, since the term $t(\lambda xy . x)$ (that extracts the first component of the pair t) actually computes the desired witness.

In classical realisability, this is no more the case. As in AF2, the term $t(\lambda xy . x)$ (constructed from a classical realiser $t \Vdash \exists^{\text{nat}} x A[x]$) realises the formula $\text{nat} \equiv \forall X (X \Rightarrow (X \Rightarrow X) \Rightarrow X)$, but unlike what happens in AF2, classical realisers of the formula nat do not generally reduce to Church numerals (such terms have a richer structure that may involve continuations). To extract a numeral, it is necessary to use a storage operator as follows:

Proposition 13 — *Given a closed formula of the form $\exists^{\text{nat}} x A[x]$, for all universal realisers $t \Vdash \exists^{\text{nat}} x A[x]$, for all closed terms u and for all stacks π , there exists a natural number $n \in \mathbb{N}$ such that*

$$\text{extr}_0 \star t \cdot u \cdot \pi \succ^* u \star \bar{n} \cdot \pi$$

where $\text{extr}_0 \equiv \lambda xy . x (M(\lambda z . yz))$ and where M is a (unary) storage operator.

Proof. Given a universal realiser $t \Vdash \exists^{\text{nat}} x A[x]$, a closed term u and a stack π , let \perp be the saturated set formed by all processes p such that $p \succ^* u \star \bar{n} \cdot \pi$ for some $n \in \mathbb{N}$, and write $S = \{\pi\}$. We have (for this choice of \perp):

- $u \Vdash \forall x (\{x\} \Rightarrow \dot{S})$ (by definition of \perp and S)
- $\lambda z_-. uz \Vdash \forall x (\{x\} \Rightarrow A[x] \Rightarrow \dot{S})$ (by adequacy)
- $M(\lambda z_-. uz) \Vdash \forall x (\text{nat}(x) \Rightarrow A[x] \Rightarrow \dot{S})$ (by def. of M)
- $t(M(\lambda z_-. uz)) \Vdash \dot{S}$ (from the type of t)
- $t(M(\lambda z_-. uz)) \star \pi \in \perp$ (by definition of S)

From the definition of \perp , we deduce that $t(M(\lambda z_-. uz)) \star \pi \succ^* u \star \bar{n} \cdot \pi$ for some $n \in \mathbb{N}$, hence $\text{extr}_0 \star t \cdot u \cdot \pi \succ^2 t(M(\lambda z_-. uz)) \star \pi \succ^* u \star \bar{n} \cdot \pi$. \square

Notice that in the most general case, the natural number n may depend on both the term u and the stack π . (This is typically the case when the realiser $t \Vdash \exists^{\text{nat}} x A[x]$ is implemented using non strictly classical instructions, such as the instruction ‘quote’ [4].) However, it can be shown that the extracted numeral n only depends on the realiser t in the situation where:

1. The realiser is implemented only using the constructions of the pure λ -calculus and the constant \mathfrak{c} (call/cc).
2. The evaluation of λ -abstraction, application and the constants \mathfrak{c} and \mathfrak{k}_π does not involve other rules than those given in subsection 2.2 (i.e. evaluation is deterministic on these constructions).

On the other hand, the natural number n extracted by this method is not necessarily a correct witness of the existential statement, in this sense that we do not necessarily have $A[n]$. This is typically the case when the existential statement has been proved using some classical reasoning principle:

Counter-example Let C be an arbitrary closed formula, and consider the predicate $A[x]$ defined by

$$A[x] \equiv (x = 0 \wedge C) \vee (x = 1 \wedge \neg C)$$

(using the abbreviations defined in subsection 3.1). Using the adequacy lemma, we easily build a universal realiser of the existential formula $\exists^{\text{nat}} x A[x]$ from the following simple realisers:

$$\begin{array}{lll}
\mathbf{I} & \equiv & \lambda z. z \quad \Vdash \quad \forall x (x = x) \\
\text{pair} & \equiv & \lambda xyp. pxy \quad \Vdash \quad \forall X \forall Y (X \Rightarrow Y \Rightarrow X \wedge Y) \\
& & \Vdash \quad \forall X \forall^{\text{nat}} x_0 (X(x_0) \Rightarrow \exists^{\text{nat}} x X(x)) \\
\text{inl} & \equiv & \lambda xfg. fx \quad \Vdash \quad \forall X \forall Y (X \Rightarrow X \vee Y) \\
\text{inr} & \equiv & \lambda yfg. gy \quad \Vdash \quad \forall X \forall Y (Y \Rightarrow X \vee Y) \\
\\
\text{em} & \equiv & \mathfrak{c}(\lambda k. \text{inr}(\lambda x. k(\text{inl } x))) \quad \Vdash \quad \forall X (X \vee \neg X) \\
T_1 & \equiv & \lambda x. \text{pair } \bar{1} (\text{inl}(\text{pair } \mathbf{I} x)) \quad \Vdash \quad C \Rightarrow \exists^{\text{nat}} x A[x] \\
T_0 & \equiv & \lambda y. \text{pair } \bar{0} (\text{inr}(\text{pair } \mathbf{I} y)) \quad \Vdash \quad \neg C \Rightarrow \exists^{\text{nat}} x A[x] \\
T & \equiv & \text{em } T_1 T_0 \quad \Vdash \quad \exists^{\text{nat}} x A[x]
\end{array}$$

To extract a witness using the method described above, it suffices to evaluate the process $\text{extr}_0 \star T \cdot c \cdot \alpha$ (where c is an inert constant and α an arbitrary stack constant), and we get:

$$\text{extr}_0 \star T \cdot \text{exit} \cdot \alpha \succ^* \text{exit} \star \bar{0} \cdot \alpha.$$

Of course, the witness $n = 0$ we obtain by this method (which is correct only when C is false) is definitely not related to the truth or falsity of the formula C , but it just reflects the strategy of the proof: first try the alternative $\neg C$, and then backtrack to C in case there is a problem.

Immediate existence However, there is at least a case where the naive extraction method always gives a correct witness: when the universal realiser $t \Vdash \exists^{\text{nat}} x A[x]$ corresponds to an introduction rule of the existential quantifier:

$$\frac{t_0 \Vdash \text{nat}(n) \quad t_1 \Vdash A[n]}{t \equiv (\lambda z . z t_0 t_1) \Vdash \exists^{\text{nat}} x A[x]}$$

We then immediately check that:

Proposition 14 — *If a closed formula universal $\exists^{\text{nat}} x A[x]$ has a universal realiser t of the form $t \equiv \lambda z . z t_0 t_1$ where $t_0 \Vdash \text{nat}(n)$ for some $n \in \mathbb{N}$ and $t_1 \Vdash A[n]$, then for all terms u and for all stacks π we have $\text{extr}_0 \star t \cdot u \cdot \pi \succ^* u \star \bar{n} \cdot \pi$.*

Proof. The process $\text{extr}_0 \star t \cdot u \cdot \pi$ evaluates as follows:

$$\begin{aligned} \text{extr}_0 \star t \cdot u \cdot \pi &\succ^* t \star M(\lambda z . . uz) \cdot \pi \\ &\succ^* M \star \lambda z . . uz \cdot t_0 \cdot t_1 \cdot \pi \quad (\text{since } t \equiv \lambda z . z t_0 t_1) \\ &\succ^* \lambda z . . uz \star \bar{n} \cdot t_1 \cdot \pi \quad (\text{by Lemma 11}) \\ &\succ^* u \star \bar{n} \cdot \pi \quad \square \end{aligned}$$

5.2 Extraction with a decidable predicate

We now consider the situation where the predicate $A[x]$ is *decidable*. Formally, we assume that there is a λ_c -term d_A that *decides* the formula $A[x]$ in the sense that for all $n \in \mathbb{N}$ we have

$$d_A \bar{n} \succ^* \begin{cases} \lambda xy . x & \text{if } A[n] \text{ is true} \\ \lambda xy . y & \text{if } A[n] \text{ is false} \end{cases} \quad (\text{in the full standard model}).$$

The existence of a decision procedure for the predicate $A[x]$ (given as a λ_c -term d_A) naturally suggests an algorithm to extract a reliable witness from a universal realiser of the formula $\exists^{\text{nat}} x A[x]$ by the method of trial/error:

1. Extract a (possibly false) witness n using the method of subsection 5.1.
2. Using the decision function d_A , check whether the proposed witness n is correct or not (i.e. do we really have $A[n]$?)
3. If the proposed witness n is correct, return it. If not, oppose a proof of $\neg A[n]$ to the justification that $A[n]$ (that should come with the witness) in order to backtrack to step 1, and to get a new witness.

Putting aside the implementation details (and the question of termination), the informal description above shows that we need an extra ingredient to implement step 3, namely: a way to build a realiser of the formula $\neg A[n]$ in the case where $A[n]$ is false. For this reason, we assume given a λ_c -term r_A such that $r_A \bar{n}$ conditionally realises $\neg A[n]$ for all $n \in \mathbb{N}$ (cf subsection 4.4).

Formally, the algorithm depicted above is implemented by the term

$$\text{extr}_1 \equiv \lambda v w x y . x (M (\lambda z z' . v z (y z) (w z z')))$$

(where M is a unary storage operator) whose computational behaviour is described by the following proposition:

Proposition 15 — *Given terms t_A , r_A and t such that*

1. t_A decides the predicate $A[x]$;
2. $r_A \bar{n} \Vdash^\circ \neg A[n]$ for all $n \in \mathbb{N}$;
3. $t \Vdash \exists^{\text{nat}} x A[x]$;

for all terms u and for all stacks π there is a number $n \in \mathbb{N}$ such that

$$\text{extr}_1 \star d_A \cdot r_A \cdot t \cdot u \cdot \pi \succ^* u \star \bar{n} \cdot \pi$$

and such that the formula $A[n]$ is true (in the standard model).

Proof. Let \perp be the saturated set formed by all processes p that evaluate to a process of the form $u \star \bar{n} \cdot \pi$ where n is a natural number such that $A[n]$ holds (in the standard model), and write $S = \{\pi\}$. We now want to show that

$$\lambda z z' . d_A z (u z) (r_A z z') \Vdash \forall x (\{x\} \Rightarrow A[x] \Rightarrow \dot{S})$$

(for this choice of \perp). For that, take an arbitrary $n \in \mathbb{N}$, a realiser $s \Vdash A[n]$, and let us show that $\lambda z z' . d_A z (u z) (r_A z z') \star \bar{n} \cdot s \cdot \pi \in \perp$. We distinguish the following two cases:

- Either $A[n]$ is true. In this case, we have

$$\begin{aligned} & \lambda z z' . d_A z (u z) (r_A z z') \star \bar{n} \cdot s \cdot \pi \\ \Upsilon^* & d_A \bar{n} \star u \bar{n} \cdot r_A \bar{n} s \cdot \pi \\ \Upsilon^* & \lambda x y . x \star u \bar{n} \cdot r_A \bar{n} s \cdot \pi && \text{(since } A[n] \text{ is true)} \\ \Upsilon^* & u \star \bar{n} \cdot \pi \in \perp && \text{(by def. of } \perp) \end{aligned}$$

- Either $A[n]$ is false. In this case, we have

$$\begin{aligned} & \lambda z z' . d_A z (u z) (r_A z z') \star \bar{n} \cdot s \cdot \pi \\ \Upsilon^* & d_A \bar{n} \star u \bar{n} \cdot r_A \bar{n} s \cdot \pi \\ \Upsilon^* & \lambda x y . y \star u \bar{n} \cdot r_A \bar{n} s \cdot \pi && \text{(since } A[n] \text{ is false)} \\ \Upsilon^* & r_A \bar{n} \star s \cdot \pi \in \perp && \text{(since } r_A \bar{n} \Vdash \neg A[n] \text{ and } s \Vdash A[n]) \end{aligned}$$

Hence we have $\lambda z z' . d_A z (u z) (r_A z z') \Vdash \forall x (\{x\} \Rightarrow A[x] \Rightarrow \dot{S})$. From this, we get $M (\lambda z z' . d_A z (u z) (r_A z z')) \Vdash \forall x (\text{nat}(x) \Rightarrow A[x] \Rightarrow \dot{S})$ and thus

$$t (M (\lambda z z' . d_A z (u z) (r_A z z'))) \Vdash \dot{S}.$$

Since $\pi \in S$, we finally have:

$$\text{extr}_1 \star d_A \cdot r_A \cdot t \cdot u \cdot \pi \succ^* t(M(\lambda z z'. d_A z(u z)(r_A z z'))) \star \pi \in \perp. \quad \square$$

Remark. It is interesting to notice that the recursive nature of the informal algorithm presented above is not reflected in the (rather direct) implementation style of the term

$$t(M(\lambda z z'. d_A z(u z)(r_A z z')))$$

that ultimately does the extraction job. However, the possibility of recursion is actually hidden in the classical realiser $t \Vdash \exists^{\text{nat}} x A[x]$ which is given the control of the process. By capturing its continuation, the realiser t can thus replay the top-level application as many times as needed. (In practice, the continuation that implements the recursive call is hidden in the realiser $s \Vdash A[n]$ which is passed as z' to the function r_A .)

5.3 Forcing the production of witnesses

In the case where no decision procedure can be given for the predicate $A[x]$, it is tempting to replace in the latter extraction algorithm the decision function d_A by a function that systematically repudiates the witness in order to force the production of a (possibly infinite) stream of potential witnesses, with the hope that there is at least a correct witness among them. (Of course, we still need a term r_A such that $r_A \bar{n} \Vdash \neg A[n]$ for all $n \in \mathbb{N}$.)

To implement this idea, we first need to extend the language of realisers with an extra instruction ‘print’ whose informal operational semantics is the following: pop the top element of the stack, print it (via some device), and then continue the computation by putting the next element of the stack into head position. Formally, the instruction print is thus endowed with the following evaluation rule:

$$\text{print} \star t \cdot u \cdot \pi \succ u \star \pi.$$

Notice that this evaluation rule does not describe the intended visual side effect of the instruction (such an effect definitely stands outside the scope of any mathematical description), so that strictly speaking, the instruction print actually behaves as the term $\lambda xy. y$ —it is the same thing with a different name.⁵

Once this instruction has been introduced, we can set

$$\text{extr}_2 \equiv \lambda w x. x(M(\lambda z z'. \text{print } z(w z z')))$$

and check that a correct witness is eventually produced during the execution of the program $\text{extr}_2 r_A t$, where t is a uniform realiser of $\exists^{\text{nat}} x A[x]$:

Proposition 16 — *If r_A is a term such that $r_A \bar{n} \Vdash \neg A[n]$ for all $n \in \mathbb{N}$, then for all universal realisers $t \Vdash \exists^{\text{nat}} x A[x]$ and for all stacks π there exists a number $n \in \mathbb{N}$ and a stack π' such that*

$$\text{extr}_2 \star r_A \cdot t \cdot \pi \succ^* \text{print} \star \bar{n} \cdot \pi'$$

and such that the formula $A[n]$ is true (in the standard model).

⁵Of course, this difference will be fruitfully exploited in the definition of the set \perp in the proof of Prop. 16.

Proof. Let \perp be the set of all processes that eventually evaluate to a process of the form $\text{print} \star \bar{n} \cdot \pi'$, where n is a numeral such that $A[n]$ holds, and where π' is an arbitrary stack. We now want to show that

$$\lambda z z' . \text{print } z (r_A z z') \Vdash \forall x (\{x\} \Rightarrow A[x] \Rightarrow \perp)$$

(for this choice of \perp). For that, take an arbitrary $n \in \mathbb{N}$, a realiser $s \Vdash A[n]$ and an arbitrary stack π' , and let us show that $\lambda z z' . \text{print } z (r_A z z') \star \bar{n} \cdot s \cdot \pi' \in \perp$. We distinguish the following two cases:

- Either $A[n]$ is true. In this case we have

$$\begin{array}{l} \lambda z z' . \text{print } z (r_A z z') \star \bar{n} \cdot s \cdot \pi' \\ \gamma^* \text{print} \star \bar{n} \cdot (r_A \bar{n} s) \cdot \pi' \in \perp \quad (\text{by def. of } \perp) \end{array}$$

- Either $A[n]$ is false. In this case we have

$$\begin{array}{l} \lambda z z' . \text{print } z (r_A z z') \star \bar{n} \cdot s \cdot \pi' \\ \gamma^* \text{print} \star \bar{n} \cdot (r_A \bar{n} s) \cdot \pi' \\ \gamma r_A \bar{n} \star s \cdot \pi' \in \perp \quad (\text{since } r_A \bar{n} \Vdash \neg A[n] \text{ and } s \Vdash A[n]) \end{array}$$

Hence we have $\lambda z z' . \text{print } z (r_A z z') \Vdash \forall x (\{x\} \Rightarrow A[x] \Rightarrow \perp)$. From this, we get $M(\lambda z z' . \text{print } z (r_A z z')) \Vdash \forall x (\text{nat}(x) \Rightarrow A[x] \Rightarrow \perp)$ and thus

$$t(M(\lambda z z' . \text{print } z (r_A z z'))) \Vdash \perp.$$

Therefore: $\text{extr}_2 \star r_A \cdot t \cdot \pi \succ^* t(M(\lambda z z' . \text{print } z (r_A z z'))) \star \pi \in \perp$. \square

In practice, the extracted program $\text{extr}_2 r_A t$ may have a finite or an infinite execution, and may print finitely or infinitely many potential witnesses. The only thing we know is that a correct witness will eventually appear among the terms that are displayed by the instruction `print`. (But if the predicate $A[x]$ is undecidable, there is no way to know which one is correct). In particular, one should notice that once a correct witness n has been found and printed during the evaluation of the process $\text{extr}_2 \star r_1 \cdot t \cdot \pi$, the sequel of the execution enjoys no correctness invariant anymore:

$$\begin{array}{l} \dots \\ \gamma^* \text{print} \star \bar{n} \cdot (r_A \bar{n} s) \cdot \pi' \quad (\in \perp \text{ since } A[n]) \\ \gamma r_A \bar{n} \star s \cdot \pi' \in \perp \quad (\text{no correctness invariant}) \\ \gamma^* ??? \end{array}$$

(This is due to the fact that our correctness invariant \perp is not closed under evaluation.) Anything may thus happen: the execution may be infinite (while printing finitely or infinitely many terms) or it may be finite (if at some point an abstraction or a constant runs out of argument).

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.