A direct computational interpretation of second-order arithmetic via update recursion

Valentin Blot

A direct computational interpretation of second-order arithmetic via update recursion

Valentin Blot
Inria, Laboratoire Méthodes Formelles, Université Paris-Saclay

ABSTRACT

Second-order arithmetic has two kinds of computational interpretations: via Spector's bar recursion of via Girard's polymorphic lambda-calculus. Bar recursion interprets the negative translation of the axiom of choice which, combined with an interpretation of the negative translation of the excluded middle, gives a computational interpretation of the negative translation of the axiom scheme of comprehension. It is then possible to instantiate universally quantified sets with arbitrary formulas (second-order elimination). On the other hand, polymorphic lambda-calculus interprets directly second-order elimination by means of polymorphic types. The present work aims at bridging the gap between these two interpretations by interpreting directly second-order elimination through update recursion, which is a variant of bar recursion.

ACM Reference Format:
Valentin Blot. 2022. A direct computational interpretation of second-order arithmetic via update recursion. In *37th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) (LICS '22), August 2–5, 2022, Haifa, Israel.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3531130.3532458

1 INTRODUCTION

In this introduction we recall the existing interpretations of second-order arithmetic, we present the outline of our contribution, and we mention some related work.

1.1 Bar recursive interpretations

The usual route, summarized in figure 1, for interpreting second-order arithmetic via bar recursion consists of a derivation of the axiom scheme of comprehension through a combination of the excluded middle with the axiom of countable choice, followed by a negative translation into first-order intuitionistic arithmetic extended with the negative translation of the axiom of choice. Finally, this system is interpreted with the help of bar recursion that gives computational content to the negative translation of the axiom of countable choice (or to the double-negation shift which, together with the axiom of countable choice, implies intuitionistically the negative translation of the axiom of countable choice).

We review in this section the various existing bar recursive interpretations of second-order arithmetic.

Spector's original bar recursion. Gödel's dialectica interpretation [9] provides a computational interpretation of arithmetic into System T — simply-typed lambda-calculus with natural numbers and a recursor. Spector extends this interpretation to an interpretation of second-order arithmetic [14] by extending Gödel's System T with a bar recursion operator. Spector uses this bar recursion operator to interpret what he calls the F axiom scheme, but which is nowadays known as double-negation shift (DNS). Composing his interpretation with a negative translation from classical arithmetic with countable choice into intuitionistic arithmetic with countable choice and DNS, he obtains an interpretation of classical arithmetic with countable choice and therefore of second-order arithmetic.

Berardi-Bezem-Coquand's demand-driven operator. Berardi, Bezem and Coquand define an operator [3] (usually called BBC) inspired by Spector's bar recursion but that has a more direct and intuitive behavior. Also, contrary to Spector's interpretation that extends Gödel's Dialectica, their interpretation extends Kreisel's modified realizability [10]. The BBC operator is demand-driven in the sense that it computes step by step finite approximations of an ideal infinite object, the order of these steps being dictated by the environment in which the operator executes. They use this operator to provide a computational interpretation of (a principle equivalent to) the negative translation of the axiom of countable choice. Their interpretation is therefore slightly more direct than Spector's in the sense that it interprets the negative translation of the axiom of countable choice rather than DNS.

Berger-Oliva's modified bar recursion. Berger and Oliva define an operator [6] that more closely resembles Spector's bar recursion, but their interpretation extends Kreisel's modified realizability, as Berardi-Bezem-Coquand's. Moreover, they follow more closely the lead of Spector by interpreting DNS, from which the negative translation of the axiom of countable choice can be derived.

Streicher and Krivine's bar recursion in classical realizability. Streicher gives a presentation of Krivine's classical realizability in the setting of categorical realizability, as a subtopos $\mathcal{K}$ of the relative realizability topos $\mathcal{E}$ induced by the model of coherence spaces and stable maps and a well-chosen set of proof-like elements [15]. Using bar induction and the interpretation of bar recursion as a stable map between appropriate coherence spaces, he shows that $\mathcal{E}$ validates the double-negation shift principle, and because $\mathcal{E}$ also validates countable choice, he obtains that the classical realizability topos $\mathcal{K}$ validates countable choice.

Krivine uses the BBC operator to realize the negative translation of the axiom of countable choice [11].

1.2 Interpretation of second-order arithmetic via system F

The second kind of interpretation of second-order arithmetic is via Girard's representation theorem [8] that relies on a translation of second-order arithmetic into Girard-Reynolds polymorphic lambda-calculus [7, 13] (system F). This interpretation uses a version of second-order arithmetic that is obtained from first-order arithmetic by adding a quantifier on *predicates*, rather than the axiom schema

LICS '22, August 2–5, 2022, Haifa, Israel
2022. ACM ISBN 978-1-4503-9351-5/22/08.
https://doi.org/10.1145/3531130.3532458

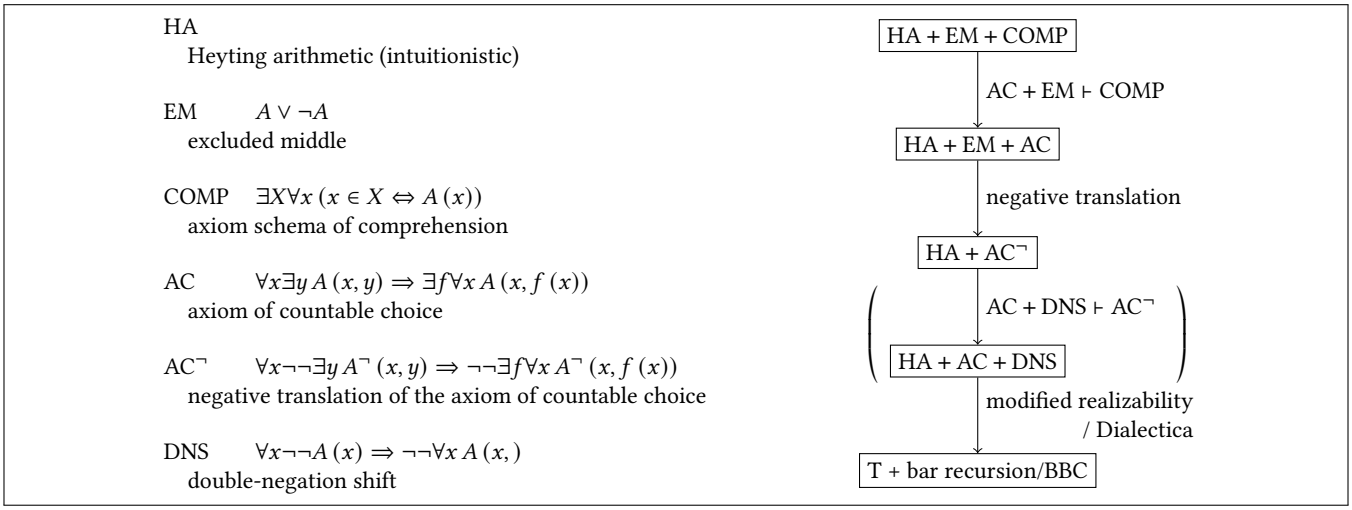

Figure 1: Bar recursive interpretations of second-order arithmetic

of comprehension. This means that set variables can be instantiated with arbitrary predicates written in the language of the logic: this is the second-order elimination rule: $\forall X\,A \Rightarrow A\,[x.B/X]$, where B is any formula and $A\,[x.B/X]$ is A where every atomic subformula $X\,(t)$ is replaced with $B\,[t/x]$. The interpretation of this rule in polymorphic lambda-calculus carries no computational content: it is interpreted with the instantiation of a polymorphic program of type $\forall X\,T$ with $X := U$ to obtain a program of type $T\,[U/X]$, where T and U are derived from A and B. This apparent simplicity of the interpretation is however counterbalanced at the meta-level by an impredicative normalization proof.

1.3 Contributions

As explained above, the bar recursive interpretations of second-order arithmetic involve several logical transformations prior to the concrete interpretation by programs. Conversely, the polymorphic interpretation exploits a strong correspondence between quantification on predicates and polymorphic types. In the present paper we make a crucial step towards the unravelling of a similar strong correspondence in the case of bar recursion. Indeed, we interpret directly the same system of second-order arithmetic as the one used in Girard's representation theorem, via a bar recursive interpretation of the elimination of quantification on predicates.

We define a very generic notion of realizability value and prove a general result about the behavior of the variant of bar recursion we use. This general result proves to be flexible enough so that it can be used for proving both termination of the programming language and correctness of the realizability interpretation.

1.4 Related work

Berger defines a variant of the bar recursion operator that he calls "update recursion" [5]. This variant is closely related to the BBC operator, in the sense that it builds finite approximations of an ideal infinite object step-by-step and on demand. Berger proves that this operator realizes a principle called "update induction" and shows that the BBC operator can indeed be implemented via update recursion. Berger's operator is the variant of bar recursion that most closely fits our needs, so we use (a slight variant of) it in the present work. However, we show directly that our version of update recursion realizes second-order elimination, without going through update induction.

Aschieri, Berardi and Birolo extend lambda-calculus with a variant of delimited exceptions and use interactive realizability to interpret intuitionistic first-order arithmetic extended with a restricted form of excluded middle `EM1` on Σ^0_1 formulas [2]. Powell uses a similar idea in the context of Gödel's Dialectica interpretation to define the concept of learning algorithms [12], that build approximations to witnesses of decidable formulas. He then transforms a countable sequence of learning algorithms into a single algorithm and proves that it can be used to interpret arithmetical comprehension on Σ^0_1 formulas. Our work can be seen as an extension of these works, where we relax the decidability requirement and provide computational content to the full comprehension axiom instead of its restriction to Σ^0_1 formulas.

1.5 Outline of the paper

In the first section, we present the system of second-order arithmetic used in Girard's representation theorem, where the usual comprehension axiom is replaced with the equivalent notion of quantification over predicates.

In the next section we define our programming language together with its operational and denotational semantics.

In the final section, we define our general notion of realizability value and prove a very generic result about update recursion, then we prove normalization of our programming language using "propositional" realizability values a.k.a. reducibility candidates. Then we prove adequacy of our interpretation via realizability values that depend on natural numbers and sets of natural numbers. Finally, we validate this interpretation by proving an extraction result from proofs of Π^0_2 formulas of second-order arithmetic.

2 SECOND-ORDER ARITHMETIC

We define in this section the theory of second-order arithmetic for which we provide a computational interpretation in the following sections.

This theory is the one used in Girard's representation theorem with quantifications over predicates, as witnessed by the $\forall 2e$ rule, where $A\,[x.B/X]$ is the formula A where every instance of an atom of the form $X\,(t)$ for some term t is replaced with the formula $B\,[t/x]$.

Second-order logic has two kinds of variables: $x, y, \dots$ denote first-order (number) variables and $X, Y, \dots$ denote second-order (set) variables. The terms of second-order arithmetic are built from number variables, 0, successor, addition and multiplication:

$$t, u \;::=\; x \mid 0 \mid S\,t \mid t + u \mid t \times u$$

Formulas are built from atomic formulas (set variables applied to some term), implication and first- and second-order universal quantification:

$$A, B \;::=\; X\,(t) \mid A \Rightarrow B \mid \forall x\,A \mid \forall X\,A$$

The formal system of second-order arithmetic (rules and axioms) is defined in figure 2. Our proof trees are annotated with proof terms for convenience, but we do not consider any kind of reduction on these proof terms.

As usual, other logical connectors such as equality, conjunction, disjunction and first- and second-order quantification can be encoded in second-order logic. In the present paper we only use the following encodings:

$$X \;\equiv\; X\,(0) \qquad t = u \;\equiv\; \forall X\,(X\,(t) \Rightarrow X\,(u))$$
$$\bot \;\equiv\; \forall X\,X \qquad \neg A \;\equiv\; A \Rightarrow \bot \qquad t \neq u \;\equiv\; \neg\,(t = u)$$

In particular, while formally all our predicate variables are unary, we encode the nullary ones via an arbitrary instantiation at 0. This technical choice allows for a more uniform treatment in the realizability interpretation. Note also that since we have terms for addition and multiplication, we could also encode predicate variables of arbitrary arity, but we do not need them in the present work.

3 THE PROGRAMMING LANGUAGE

In this section we describe the programming language in which we interpret proofs made in second-order arithmetic. We first define its syntax, typing rules and operational semantics, and then we define its denotational semantics in complete partial orders.

3.1 Syntax, typing and operational semantics

This programming language is an extension of simply-typed lambda-calculus with primitive natural numbers, sum types, a unit type and an update recursion operator ur.

The types are as follows, where ι denotes the type of natural numbers:

$$\sigma, \tau \;::=\; \iota \mid 1 \mid \sigma \to \tau \mid \sigma + \tau$$

The typing rules and operational semantics are given in figure 3, where the notation $\lambda\_.M$ is a shorthand for $\lambda x.M$ when x is not free in M, and the construct $\_\,\langle \_ \mapsto \rangle$ in the reduction rule for update recursion is syntactic sugar for the update of a notion of partial function that we define now. These partial functions on natural numbers, that the update recursion operator relies on, are encoded as functions from natural numbers to a sum type $\sigma + 1$, that we use as an option type. We consider that such a partial function $M : \iota \to \sigma + 1$ is defined at $n \in \mathbb{N}$ if $M\,\overline{n} \rightsquigarrow^* \{N \mid\}$ for some $N : \sigma$, and is undefined at $n \in \mathbb{N}$ if $M\,\overline{n} \rightsquigarrow^* \{\mid \star\}$. Consequently, the function with empty domain is defined as:

$$\epsilon \;\equiv\; \lambda\_.\,\{\mid \star\}$$

and if M is some partial function, then $M\,\langle N_1 \mapsto N_2 \rangle$ denotes its update with value N_2 at N_1, which is defined as:

$$M\,\langle N_1 \mapsto N_2 \rangle \;\equiv\; \lambda x.\mathtt{if}\; x = N_1\; \mathtt{then}\; \{N_2 \mid\}\; \mathtt{else}\; (M\,x)$$

where:

$$\mathtt{if}\; M = N\; \mathtt{then}\; P_1\; \mathtt{else}\; P_2 \;\equiv$$
$$(M\,?_\iota\; (\lambda u.u\,?_\iota\, P_1\,(\lambda y\_.P_2))\,(\lambda xzu.u\,?_\iota\, P_2\,(\lambda y\_.z\,y)))\;N$$

is an operation deciding equality on natural numbers.

We now comment on the update recursion operator. By looking at the type of the recursor, one can understand update recursion as an operator turning a program of type:

$$(\iota \to \sigma + (\sigma \to \iota)) \to \iota$$

into a program of type:

$$(\iota \to \sigma + 1) \to \iota$$

That is, if some M takes as input a sequence of elements which are either of type σ or of type $\sigma \to \iota$, then $\mathsf{ur}\,M$ takes simply as input a partial function to σ. The reduction rule of ur shows how this happens: if M needs the value of its argument at some point n, then:

- either the partial function given as argument to $\mathsf{ur}\,M$ is defined at n, in which case $\mathsf{ur}\,M$ provides that value to M, as a value of type σ,
- or the argument to $\mathsf{ur}\,M$ is undefined at n, in which case $\mathsf{ur}\,M$ provides indirectly to M a value of type $\sigma \to \iota$ by reading the input N of type σ from M, and triggering a recursive call with an updated partial function having value N at n.

Our version of update recursion is a bit different from Berger's [5]. The first difference is that we use sum types, while Berger encodes them with booleans and products. The second difference is that while the first argument in our version is of type:

$$(\iota \to \sigma + (\sigma \to \iota)) \to \iota$$

the first argument in Berger's version is (with our notations) of type:

$$(\iota \to \sigma + 1) \to (\iota \to \sigma \to \iota) \to \iota$$

Finaly, Berger's update recursion satisfies (with our notations) the following equation:

$$\psi\,M\,N = M\,N\,(\lambda xy.N\,x\,?_+\,(\lambda\_.0)\,(\lambda\_.\psi\,M\,(N\,\langle x \mapsto y\rangle)))$$

Berger's version and ours are however interdefinable:

$$\mathsf{ur}\,M = \psi\,(\lambda uv.M\,(\lambda x.u\,x\,?_+\,(\lambda y.\,\{y \mid\})\,(\lambda\_.\,\{\mid v\,x\})))$$
$$\psi\,M = \mathsf{ur}(\lambda u.M\,(\lambda x.u\,x\,?_+\,(\lambda y.\,\{y \mid\})\,(\lambda\_.\,\{\mid \star\}))$$
$$(\lambda x.u\,x\,?_+\,(\lambda\_\_.0)\,(\lambda v.v)))$$

$$\frac{}{\Gamma \vdash \mathrm{sn0} : \forall x\,(S\,x \neq 0)} \qquad \frac{}{\Gamma \vdash \mathrm{add0} : \forall x\,(x + 0 = x)} \qquad \frac{}{\Gamma \vdash \mathrm{mul0} : \forall x\,(x \times 0 = 0)}$$

$$\frac{}{\Gamma \vdash \mathrm{sinj} : \forall x \forall y\,(S\,x = S\,y \Rightarrow x = y)} \qquad \frac{}{\Gamma \vdash \mathrm{addS} : \forall x \forall y\,(x + S\,y = S\,(x + y))} \qquad \frac{}{\Gamma \vdash \mathrm{mulS} : \forall x \forall y\,(x \times S\,y = x + x \times y)}$$

$$\frac{}{\Gamma \vdash \mathrm{ind} : \forall X\,(X\,(0) \Rightarrow \forall x\,(X\,(x) \Rightarrow X\,(S\,x)) \Rightarrow \forall x\,X\,(x))}$$

axioms of second-order arithmetic

$$Ax\ \frac{}{\Gamma \vdash p : A}\ (p{:}A) \in \Gamma \qquad Dne\ \frac{}{\Gamma \vdash \mathrm{dne}_A : \neg\neg A \Rightarrow A} \qquad \Rightarrow i\ \frac{\Gamma, p : A \vdash \pi : B}{\Gamma \vdash \lambda p.\pi : A \Rightarrow B} \qquad \Rightarrow e\ \frac{\Gamma \vdash \pi_1 : A \Rightarrow B \qquad \Gamma \vdash \pi_2 : A}{\Gamma \vdash \pi_1\,\pi_2 : B}$$

$$\forall i\ \frac{\Gamma \vdash \pi : A}{\Gamma \vdash \lambda x.\pi : \forall x\,A}\ x \notin FV(\Gamma) \qquad \forall e\ \frac{\Gamma \vdash \pi : \forall x\,A}{\Gamma \vdash \pi\,t : A\,[t/x]} \qquad \forall 2i\ \frac{\Gamma \vdash \pi : A}{\Gamma \vdash \lambda X.\pi : \forall X\,A}\ X \notin FV(\Gamma) \qquad \forall 2e\ \frac{\Gamma \vdash \pi : \forall X\,A}{\Gamma \vdash \pi\,(x.B) : A\,[x.B/X]}$$

rules of second-order logic

Figure 2: Second-order arithmetic

$$\frac{}{\Gamma \vdash x : \sigma}\ (x{:}\sigma) \in \Gamma \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}$$

$$\frac{}{\Gamma \vdash \overline{n} : \iota}\ n \in \mathbb{N} \qquad \frac{\Gamma \vdash M : \iota}{\Gamma \vdash \mathsf{S}\,M : \iota} \qquad \frac{\Gamma \vdash M : \iota}{\Gamma \vdash M\,?_\iota : \sigma \to (\iota \to \sigma \to \sigma) \to \sigma}$$

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \{M \mid\} : \sigma + \tau} \qquad \frac{\Gamma \vdash M : \tau}{\Gamma \vdash \{\mid M\} : \sigma + \tau} \qquad \frac{\Gamma \vdash M : \sigma + \tau}{\Gamma \vdash M\,?_+ : (\sigma \to \nu) \to (\tau \to \nu) \to \nu}$$

$$\frac{}{\Gamma \vdash \star : 1} \qquad \frac{}{\Gamma \vdash \mathsf{ur} : ((\iota \to \sigma + (\sigma \to \iota)) \to \iota) \to (\iota \to \sigma + 1) \to \iota}$$

typing derivations

$$(\lambda x.M)\,N \rightsquigarrow M\,[N/x] \qquad \mathsf{ur}\,M\,N \rightsquigarrow M\,(\lambda x.N\,x\,?_+\,(\lambda y.\,\{y \mid\})\,(\lambda\_.\,\{\mid \lambda y.\mathsf{ur}\,M\,(N\,\langle x \mapsto y\rangle)\})\})$$

$$\mathsf{S}\,\overline{n} \rightsquigarrow \overline{n+1} \qquad \overline{0}\,?_\iota\,N_1\,N_2 \rightsquigarrow N_1 \qquad \overline{n+1}\,?_\iota\,N_1\,N_2 \rightsquigarrow N_2\,\overline{n}\,(\overline{n}\,?_\iota\,N_1\,N_2)$$

$$\{M \mid\}?_+\,N_1\,N_2 \rightsquigarrow N_1\,M \qquad \{\mid M\}?_+\,N_1\,N_2 \rightsquigarrow N_2\,M$$

$$E\,[M] \rightsquigarrow E\,[N] \quad \text{if } M \rightsquigarrow N \quad \text{where } E\,[\_] ::= E\,[\_]\,M \mid \mathsf{S}\,E\,[\_] \mid E\,[\_]\,?_\iota \mid E\,[\_]\,?_+$$

operational semantics

Figure 3: The programming language

3.2 Denotational semantics: complete partial orders

The termination of the update recursion operator depends crucially on the continuity of programs: if a program turns an infinite sequence into a value of type ι, then it can only depend on a finite part of its input sequence. The intuition is that such a program converges in a finite number of steps to a final value, and therefore during this computation it can only evaluate its input at a finite number of positions. The denotational model of complete partial orders does provide a general framework to talk about continuity of programs in this sense.

Moreover, it will be convenient, for proving correctness of update recursion, to talk about ideal sequences which are limits of finite sequences, but may not be expressible as programs. Such ideal sequences may even be non-computable. If $(a_n)_{n \in \mathbb{N}}$ is a sequence of elements of some complete partial order D, then the function that turns $n \in \mathbb{N}$ into a_n is itself an element of the complete partial order of functions from natural numbers to D, so the model of complete partial orders is very suited for handling such ideal sequences.

We now turn to the basic definitions about complete partial orders:

Definition 3.1. A partially ordered set D is directed if it is non-empty and:

$$\forall x, y \in D,\ \exists z \in D,\ x \leq z \text{ and } y \leq z$$

A complete partial order (cpo) is a partial order D:

- which has a least element $\bot$,
- such that every directed subset $X \subseteq D$ has a least upper bound $\bigvee X \in D$.

A continuous function between cpos is a function that preserves directed sets and their suprema:

Definition 3.2. If D and E are cpos, a map $f : D \longrightarrow E$ is continuous if for all directed $X \subseteq D$, $f(X) \subseteq E$ is directed and:

$$f\left(\bigvee X\right) = \bigvee f(X)$$

The following proposition is a fundamental result about cpos that permits the interpretation of functional programming languages in the model of cpos. Its proof can be found, e.g. in [1].

Proposition 3.3. *The set of pointwise ordered continuous maps from cpo D to cpo E is a cpo that we denote with $D \longrightarrow E$.*

Moreover, the componentwise ordering on products of cpos turns the model of cpos into a cartesian closed category, and therefore into a model of λ-calculus (see [1] for details).

After these general definitions about the model of cpos, we move to its use for our programming language.

We start with the operation of lifting of a cpo, which adds a least element to an existing cpo: if A is a partially ordered set we let $A_\bot$ be the set $A \cup \{\bot\}$ equipped with the order on A extended with $\bot \leq a$ for all $a \in A_\bot$. If A is a set then it is implicitly equipped with the discrete ordering. Liftings of such sets are used to interpret base types.

We associate to each type σ a cpo $[\sigma]$ as follows:

$$[\iota] = \mathbb{N}_\bot \qquad [\sigma \to \tau] = [\sigma] \to [\tau]$$
$$[1] = \{\star\}_\bot \qquad [\sigma + \tau] = ([\sigma] \uplus [\tau])_\bot$$

where we choose the same symbol $\star$ to denote the program on the unit type and its interpretation. If $a \in A$ and $b \in B$ we write $\{a \mid\}$ for $\{a \mid\} \in A \uplus B$ and $\{\mid b\}$ for $\{\mid b\} \in A \uplus B$. Similarly we write $\{A \mid\} = \{\{a \mid\} \mid a \in A\}$ and $\{\mid B\} = \{\{\mid b\} \mid b \in B\}$.

Each program $x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash M : \sigma$ is interpreted as a continuous function $[M]$ from $[\sigma_1] \times \ldots [\sigma_n]$ (equipped with the componentwise ordering) to $[\sigma]$ as described in figure 4.

We now prove that the operation deciding equality between natural numbers, the empty function and the update of a partial function do have the intended semantics.

Proposition 3.4. *We have:*

$$\left[\begin{array}{r}\texttt{if } M = N \texttt{ then } P_1 \\ \texttt{else } P_2\end{array}\right] = \begin{cases} \bot & \text{if } [M] = \bot \text{ or } [N] = \bot \\ [P_1] & \text{if } [M] = [N] \neq \bot \\ [P_2] & \text{if } \bot \neq [M] \neq [N] \neq \bot \end{cases}$$

$$[\epsilon]\,(a) = \{\mid \star\}$$

$$[M\,\langle N_1 \mapsto N_2\rangle]\,(a) = \begin{cases} \bot & \text{if } a = \bot \\ [\{N_2 \mid\}] & \text{if } a = [N_1] \\ [M]\,(a) & \text{if } a \neq [N_1] \text{ and } a \neq \bot \end{cases}$$

Proof. For the first item: if $[M] = \bot$ then:

$$[\texttt{if } M = N \texttt{ then } P_1 \texttt{ else } P_2] = \bot$$

otherwise define:

$$d_0 = [\lambda u.u\,?_\iota\,P_1\,(\lambda y\_.P_2)]$$
$$d_{n+1} = [\lambda xzu.u\,?_\iota\,P_2\,(\lambda y\_.z\,y)]\,(n)\,(d_n)$$

so that:

$$[\texttt{if} M = N \texttt{ then } P_1 \texttt{ else } P_2] = d_{[M]}\,([N])$$

We then prove by induction on $n \in \mathbb{N}$ that $d_n\,(\bot) = \bot, d_n\,(n) = [P_1]$ and $d_n\,(m) = [P_2]$ if $m \neq n$, from which the result follows. The second item is immediate and the third follows easily from the first by case analysis. □

We now prove that the interpretation of update recursion satisfies the recursive equation implied by its reduction rule:

Lemma 3.5. *For all $\vec{a}, b$:*

$$\mathfrak{F}\,([\mathtt{ur}]\,(\vec{a})\,(b)) = [\mathtt{ur}]\,(\vec{a})\,(b)$$

Proof. Indeed, $\mathfrak{F}$ is continuous so:

$$\begin{aligned}\mathfrak{F}\,([\mathtt{ur}]\,(\vec{a})\,(b)) &= \mathfrak{F}\left(\bigvee_{n\in\mathbb{N}} \mathfrak{F}^n\,(\bot)\right) = \bigvee_{n\in\mathbb{N}} \mathfrak{F}\,(\mathfrak{F}^n\,(\bot)) \\ &= \bigvee_{n\in\mathbb{N}} \mathfrak{F}^{n+1}\,(\bot) = \bot \vee \bigvee_{n\in\mathbb{N}} \mathfrak{F}^{n+1}\,(\bot) \\ &= \bigvee_{n\in\mathbb{N}} \mathfrak{F}^n\,(\bot) = [\mathtt{ur}]\,(\vec{a})\,(b)\end{aligned}$$ □

The following proposition asserts that the denotational semantics of our programming language is correct with respect to its operational semantics.

Proposition 3.6. *If $M \rightsquigarrow N$ then $[M] = [N]$.*

Proof. By case on the reduction $M \rightsquigarrow N$, using the previous lemma for the case of update recursion. □

The following result, computational adequacy, allows to "go back" from the denotational model to the syntactic language on base types. It asserts that if some program on a base type is interpreted as a value, then it must reduce to that value. It will be particularly useful for proving that extracted programs do compute the intended values. The result is standard, usually proved via logical relations, the interested reader can refer to [1].

Proposition 3.7. *If $M : \iota$ and $N : 1$ are closed programs such that $[M] = n$ for some $n \in \mathbb{N}$ and $[N] = \star$, then $M \rightsquigarrow^* \overline{n}$ and $N \rightsquigarrow^* \star$.*

In the following we will often drop the interpretation brackets and use programs with parameters, that is, we will write $M\,[\vec{a}/\vec{x}]$ instead of $[M]\,(\vec{a})$.

4 REALIZABILITY

In this section we define a general notion of realizability value. We first use it to prove normalization of our programming language. We then define the interpretation of proofs in our programming language and we prove the correctness of this interpretation with respect to realizability values associated to formulas. Finally, we prove an extraction theorem from proofs of Π^0_2 theorems.

$$[x_i]\,(\vec{a}) = a_i \qquad [\lambda x.M]\,(\vec{a})\,(b) = [M]\,(\vec{a}, b) \qquad [M\,N]\,(\vec{a}) = [M]\,(\vec{a})\,([N]\,(\vec{a})) \qquad [\star]\,(\vec{a}) = \star$$

$$[\overline{n}]\,(\vec{a}) = n \qquad [\mathsf{S}\,M]\,(\vec{a}) = [M]\,(\vec{a}) + 1 \qquad [M\,?_\iota]\,(\vec{a})\,(b)\,(c) = d_{[M](\vec{a})} \text{ where } \begin{cases} d_\perp = \perp \\ d_0 = b \\ d_{n+1} = c\,(n)\,(d_n) \end{cases}$$

$$[\{M \mid\}]\,(\vec{a}) = \{[M]\,(\vec{a}) \mid\} \qquad [\{\mid M\}]\,(\vec{a}) = \{\mid [M]\,(\vec{a})\} \qquad [M\,?_+]\,(\vec{a})\,(b)\,(c) = \begin{cases} \perp & \text{if } [M]\,(\vec{a}) = \perp \\ b\,(d) & \text{if } [M]\,(\vec{a}) \text{ is some } \{d \mid\} \\ c\,(d) & \text{if } [M]\,(\vec{a}) \text{ is some } \{\mid d\} \end{cases}$$

$$[\mathsf{ur}]\,(\vec{a})\,(b) = \bigvee_{n\in\mathbb{N}} \mathfrak{F}^n\,(\perp) \qquad \text{where} \qquad \mathfrak{F} = [\lambda zrf.z(\lambda x.f\,x\,?_+\,(\lambda y.\,\{y \mid\})\,(\lambda\_.\,\{\mid \lambda y.r\,(f\,\langle x \mapsto y\rangle)\})\})]\,(\vec{a})\,(b)$$

Figure 4: Denotational semantics in complete partial orders

4.1 Realizability values

In this section we define a general notion of realizability value that is just a subset of the cpo interpretation of a type. We then prove some properties on these realizability values and we prove a general result on the behavior of update recursion with respect to arbitrary realizability values.

Definition 4.1. A realizability value $\mathcal{A}$ on a type σ is a subset of its interpretation $[\sigma]$. We define for each realizability values $\mathcal{A}$ on σ and $\mathcal{B}$ on τ the following realizability values on $\sigma \to \tau$ and $\sigma + \tau$:

$$\mathcal{A} \to \mathcal{B} = \{f \in [\sigma \to \tau] \mid \forall a \in \mathcal{A},\, f\,(a) \in \mathcal{B}\}$$
$$\mathcal{A} + \mathcal{B} = \{\mathcal{A} \mid\} \cup \{\mid \mathcal{B}\}$$

As for subtyping, $\to$ is contravariant on the left and covariant on the right with respect to inclusion. We also have some commutation rules with respect to unions and intersections.

Lemma 4.2. *For every realizability values $\mathcal{A}_1$, $\mathcal{A}_2$ on σ and $\mathcal{B}_1$, $\mathcal{B}_2$ on τ, if $\mathcal{A}_2 \subseteq \mathcal{A}_1$ and $\mathcal{B}_1 \subseteq \mathcal{B}_2$ then:*

$$\mathcal{A}_1 \to \mathcal{B}_1 \subseteq \mathcal{A}_2 \to \mathcal{B}_2$$

Il $(\mathcal{A}_e)_{e\in E}$ is a family of realizability values and $\mathcal{B}$ is a realizability value, then:

$$\bigcap_{e\in E} (\mathcal{A}_e \to \mathcal{B}) = \bigcup_{e\in E} \mathcal{A}_e \to \mathcal{B} \qquad \bigcap_{e\in E} (\mathcal{B} \to \mathcal{A}_e) = \mathcal{B}_e \to \bigcap_{e\in E} \mathcal{A}_e$$
$$\bigcup_{e\in E} (\mathcal{A}_e \to \mathcal{B}) \subseteq \bigcap_{e\in E} \mathcal{A}_e \to \mathcal{B} \qquad \bigcup_{e\in E} (\mathcal{B} \to \mathcal{A}_e) \subseteq \mathcal{B} \to \bigcup_{e\in E} \mathcal{A}_e$$

Proof. For the first item, let $f \in \mathcal{A}_1 \to \mathcal{B}_1$ and let $a \in \mathcal{A}_2$. Then $a \in \mathcal{A}_1$ so $f\,(a) \in \mathcal{B}_1 \subseteq \mathcal{B}_2$. The second item follows from the basic rules of quantifiers. □

The following proposition describes very precisely the behavior of update recursion on arbitrary realizability values. We will use it both for proving normalization of our programming language, and for proving correctness of our interpretation of the $\forall 2e$ rule of second-order arithmetic. Our proof is largely inspired by [4].

Proposition 4.3. *Let $(\mathcal{A}_n)_{n\in\mathbb{N}}$ be a family of realizability values on σ and let $\mathcal{B} \subseteq \mathbb{N}$. Then:*

$$\mathsf{ur} \in \left(\bigcap_{n\in\mathbb{N}} (\{n\} \to \mathcal{A}_n + (\mathcal{A}_n \to \mathcal{B})) \to \mathcal{B}\right) \to \bigcap_{n\in\mathbb{N}} (\{n\} \to \mathcal{A}_n + \{\star\}) \to \mathcal{B}$$

Proof. Remark that $\bigcap_{n\in\mathbb{N}} (\{n\} \to \mathcal{A}_n + \{\star\})$ is a realizability value on the type $\iota \to \sigma + 1$ of partial functions on σ that we presented earlier. We define a preorder $\preccurlyeq$ on this realizability value, which corresponds to the usual order between partial functions, as follows:

$$f \preccurlyeq g \quad \text{iff} \quad \forall n \in \mathbb{N},\, f\,(n) \in \{\mathcal{A}_n \mid\} \Rightarrow g\,(n) = f\,(n)$$

Be careful that this preorder $\preccurlyeq$ is not the cpo order $\leq$. It is a different preorder that we only use in the proof of the current proposition.

Let $a \in \bigcap_{n\in\mathbb{N}} (\{n\} \to \mathcal{A}_n + (\mathcal{A}_n \to \mathcal{B})) \to \mathcal{B}$ and let:

$$E = \left\{ f \in \bigcap_{n\in\mathbb{N}} (\{n\} \to \mathcal{A}_n + \{\star\}) \;\middle|\; \mathsf{ur}\,a\,f \notin \mathcal{B} \right\}$$

We prove that every non-empty chain of $(E, \preccurlyeq)$ has an upper bound but $(E, \preccurlyeq)$ has no maximal element. Therefore by Zorn's lemma E must be empty, which proves the result.

Every non-empty chain of $(E, \preccurlyeq)$ has an upper bound. Let $F \subseteq E$ be a non-empty chain. Define $\curlyvee F$ by $\curlyvee F\,(\perp) = \perp$ and:

$$\text{if } \forall f \in F,\, f\,(n) = \{\mid \star\} \text{ then } \curlyvee F\,(n) = \{\mid \star\}$$
$$\text{if } \exists f \in F,\, f\,(n) \in \{\mathcal{A}_n \mid\} \text{ then } \curlyvee F\,(n) = f\,(n)$$

Note that since F is a chain, the value in the second case is unique so $\curlyvee F$ is well defined. By construction, $\curlyvee F$ is an upper bound of F in $\bigcap_{n\in\mathbb{N}} (\{n\} \to \mathcal{A}_n + \{\star\})$, so we are left to prove $\curlyvee F \in E$. Suppose for the sake of contradiction that $\curlyvee F \notin E$, that is, $\mathsf{ur}\,a\,\curlyvee F \in \mathcal{B}$. Define f_I for finite $I \subseteq \mathbb{N}$ by:

$$f_I\,(\perp) = \perp \qquad f_I\,(n) = \begin{cases} \curlyvee F\,(n) & \text{if } n \in I \\ \perp & \text{otherwise} \end{cases}$$

Then for the cpo order $\leq$ on $[\iota \to \sigma + 1]$:

- $\{f_I \mid I \subseteq \mathbb{N} \text{ finite}\}$ is directed
- $\curlyvee f = \bigvee \{f_I \mid I \subseteq \mathbb{N} \text{ finite}\}$

By monotonicity of $\mathsf{ur}\ a$ we have for all I:

$$\mathsf{ur}\ a\ f_I \leq \mathsf{ur}\ a \vee F \quad \text{so} \quad \mathsf{ur}\ a\ f_I \in \{\bot, \mathsf{ur}\ a \vee F\}$$

and then by continuity of $\mathsf{ur}\ a$ there must exist I_0 such that:

$$\mathsf{ur}\ a\ f_{I_0} = \mathsf{ur}\ a \vee F$$

By definition of $\vee F$ and non-emptiness of F, for each $i \in I_0$ there exists $f_i \in F$ such that:

$$f_i\ (i) = \vee F\ (i) = f_{I_0}\ (i)$$

Then since F is a chain and I_0 is finite, there exists $f \in F$ such that $f_i \preccurlyeq f$ for every $i \in I_0$. Finally, for each $n \in \mathbb{N}$:

- if $n \notin I_0$ then $f_{I_0}\ (n) = \bot \leq f\ (n)$
- if $n \in I_0$ then $f_n\ (n) = \vee F\ (n) = f_{I_0}\ (n)$ and:
  - if $f_n\ (n) \in \{\mathcal{A}_n \mid\}$ then $f\ (n) = f_n\ (n)$ since $f_n \preccurlyeq f$, and therefore $f_{I_0}\ (n) = f\ (n)$
  - if $\vee F\ (n) = \{\mid \star\}$ then $f\ (n) = \{\mid \star\}$ by definition of $\vee F$ since $f \in F$, and therefore $f_{I_0}\ (n) = f\ (n)$

therefore $f_{I_0} \leq f$ and $\mathsf{ur}\ a \vee F = \mathsf{ur}\ a\ f_{I_0} \leq \mathsf{ur}\ a\ f$ by monotonicity of $\mathsf{ur}\ a$. But $\mathsf{ur}\ a \vee F \in \mathcal{B}$ so $\mathsf{ur}\ a \vee F \neq \bot$, which means that $\mathsf{ur}\ a\ f = \mathsf{ur}\ a \vee F \in \mathcal{B}$, contradicting $F \subseteq E$.

$(E, \preccurlyeq)$ **has no maximal element.** Suppose for the sake of contradiction that f is a maximal element of E. Then $\mathsf{ur}\ a\ f \notin \mathcal{B}$, so by lemma 3.5:

$$a(\lambda x.f\ x\ ?_+\ (\lambda y.\ \{y \mid\})\ (\lambda\_.\ \{\mid \lambda y.\mathsf{ur}\ a\ (f\ \langle x \mapsto y\rangle)\})) = \mathsf{ur}\ a\ f \notin \mathcal{B}$$

but then by hypothesis on a we have:

$$\lambda x.f\ x\ ?_+\ (\lambda y.\ \{y \mid\})\ (\lambda\_.\ \{\mid \lambda y.\mathsf{ur}\ a\ (f\ \langle x \mapsto y\rangle)\}) \notin \bigcap_{n\in\mathbb{N}} (\{n\} \to \mathcal{A}_n + (\mathcal{A}_n \to \mathcal{B}))$$

so there exists $n \in \mathbb{N}$ such that:

$$f\ n\ ?_+\ (\lambda y.\ \{y \mid\})\ (\lambda\_.\ \{\mid \lambda y.\mathsf{ur}\ a\ (f\ \langle n \mapsto y\rangle)\}) \notin \mathcal{A}_n + (\mathcal{A}_n \to \mathcal{B})$$

Since $f\ n \in \mathcal{A}_n + \{\star\}$, either there exists $b \in \mathcal{A}_n$ such that $f\ n = \{b \mid\}$, or $f\ n = \{\mid \star\}$. In the first case, we obtain $\{b \mid\} \notin \mathcal{A}_n + (\mathcal{A}_n \to \mathcal{B})$, which is a contradiction. Therefore $f\ n = \{\mid \star\}$ and we have:

$$\{\mid \lambda y.\mathsf{ur}\ a\ (f\ \langle n \mapsto y\rangle)\} \notin \mathcal{A}_n + (\mathcal{A}_n \to \mathcal{B})$$

so there exists $b \in \mathcal{A}_n$ such that $\mathsf{ur}\ a\ (f\ \langle n \mapsto b\rangle) \notin \mathcal{B}$ and then $f\ \langle n \mapsto b\rangle \in E$. But $f \preccurlyeq f\ \langle n \mapsto b\rangle$ and $f\ \langle n \mapsto b\rangle\ n = \{b \mid\}$ by 3.4, so since $f\ n = \{\mid \star\}$ we also have $f\ \langle n \mapsto b\rangle \not\preccurlyeq f$, contradicting the maximality of f in E. □

4.2 Normalization

We now use the general notion of realizability value to prove the normalization of our programming language. For that, we define for each type σ the realizability value $|\sigma|$ on σ as follows:

$$|\iota| = \mathbb{N} \qquad |\sigma \to \tau| = |\sigma| \to |\tau|$$
$$|1| = \{\star\} \qquad |\sigma + \tau| = |\sigma| + |\tau|$$

These realizability values are the semantic counterparts of the usual reducibility candidates, so they satisfy the following property:

Lemma 4.4. *For every type σ, there exists M such that $[M] \in |\sigma|$ and every such M reduces to a normal form.*

Proof. We prove the results by induction on σ. We have $[0] \in |\iota|$ and $[\star] \in |1|$. If we have $[M] \in |\tau|$ then $[\lambda\_.M] \in |\sigma \to \tau|$ and $[\{\mid M\}] \in |\sigma + \tau|$.

Normalization on $|\iota|$ and $|1|$ is a consequence of computational adequacy (proposition 3.7).

If we have $[M] \in |\sigma \to \tau|$ but M diverges, then for N such that $[N] \in |\sigma| \neq \emptyset$, $M\ N$ diverges as well, contradicting the induction hypothesis on τ.

If we have $[M] \in |\sigma + \tau|$ but M diverges, then $M\ ?_+\ (\lambda\_.\star)\ (\lambda\_.\star)$ diverges as well, but since $|\sigma + \tau| = \{|\sigma| \mid\} \cup \{\mid |\tau|\}$, we have $[M\ ?_+\ (\lambda\_.\star)\ (\lambda\_.\star)] = \star \in |1|$, contradicting computational adequacy. □

We are now ready to prove that each program belongs to the realizability value associated to its type:

Theorem 4.5. *For each typing derivation:*

$$x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash M : \tau$$

if:

$$(a_1, \ldots, a_n) \in |\sigma_1| \times \ldots \times |\sigma_n|$$

then:

$$M\ [a_1, \ldots, a_n / x_1, \ldots, x_n] \in |\tau|$$

Proof. The proof proceeds by induction on the structure of M. The cases x, $\lambda x.M$, $M\ N$, $\overline{n}$, $\mathsf{S}\ M$, $\{M \mid\}$, $\{\mid M\}$, and $\star$ are straightforward.

For recursion, let $M : \iota$ be such that $[M] \in |\iota| = \mathbb{N}$, and let $a \in |\sigma|$ and $b \in |\iota \to \sigma \to \sigma|$. We define $d_0 = a$ and $d_{n+1} = b\ (n)\ (d_n)$ and prove by induction on $n \in \mathbb{N}$ that $d_n \in |\sigma|$. Indeed, $d_0 = a \in |\sigma|$ and $d_{n+1} = b\ (n)\ (d_n) \in |\sigma|$ since $n \in |\iota|$ and $d_n \in |\sigma|$ by induction hypothesis.

For case analysis, let $M : \sigma + \tau$ be such that $[M] \in |\sigma + \tau| = |\sigma| + |\tau|$ and let $b_1 \in |\sigma \to \nu|$ and $b_2 \in |\tau \to \nu|$. If $[M] \in \{|\sigma| \mid\}$ then there exists $c \in |\sigma|$ such that $[M] = \{c \mid\}$ and therefore we get $[M\ ?_+]\ (b_1)\ (b_2) = b_1\ (c) \in |\nu|$, and if $[M] \in \{\mid |\tau|\}$ then there exists $c \in |\tau|$ such that $[M] = \{\mid c\}$ and therefore we get $[M\ ?_+]\ (b_1)\ (b_2) = b_2\ (c) \in |\nu|$.

Finally, for update recursion, this is a consequence of 4.3 with $\mathcal{A}_n = |\sigma|$ and $\mathcal{B} = |\iota| = \mathbb{N}$, since by lemma 4.2:

$$\bigcap_{n\in\mathbb{N}} (\{n\} \to |\sigma| + (|\sigma| \to |\iota|)) = \bigcup_{n\in\mathbb{N}} \{n\} \to |\sigma| + (|\sigma| \to |\iota|) = |\iota| \to |\sigma| + (|\sigma| \to |\iota|)$$

$$\bigcap_{n\in\mathbb{N}} (\{n\} \to |\sigma| + \{\star\}) = \bigcup_{n\in\mathbb{N}} \{n\} \to |\sigma| + |1| = |\iota| \to |\sigma| + |1|$$

□

From this theorem and the previous lemma we get normalization of our programming language:

Corollary 4.6. *If M is a closed program, then M is normalizing.*

4.3 Realizability interpretation

We now define the interpretation of second-order arithmetic proofs as programs of our programming language. Our interpretation follows the line of modified realizability and associates to each

proof of a formula A in second-order arithmetic a program which type $A^\dagger$ is derived from the formula. The mapping is defined as follows:

$$X\,(t)^\dagger \equiv \iota \qquad (A \Rightarrow B)^\dagger \equiv A^\dagger \to B^\dagger$$
$$(\forall x\, A)^\dagger \equiv \iota \to A^\dagger \qquad (\forall X\, A)^\dagger \equiv A^\dagger$$

We interpret $X\,(t)$ with type ι in order to be able to extract natural numbers from proofs in theorem 4.14. In case we aren't interested in extraction then we can interpret $X\,(t)$ with an arbitrary discrete non-empty type.

Before interpreting proofs as programs, we have to interpret terms t of the logic as programs $t^\dagger$ of type ι. The free variables of $t^\dagger$ are the first-order variables of t and they are of type ι. This interpretation is straightforward, since addition and multiplication are easily implemented with the following programs:

$$M + N \equiv N\,?\,M\,(\lambda xz.\mathsf{S}\,z) \qquad M \times N \equiv N\,?\,0\,(\lambda xz.M + z)$$

Finally, the proof of a sequent:

$$p_1 : A_1, \dots, p_n : A_n \vdash \pi : A$$

is interpreted as a program $\pi^\dagger$ which typing derivation's conclusion is of the form:

$$\vec{x} : \iota, p_1 : A_1^\dagger, \dots, p_n : A_n^\dagger \vdash \pi^\dagger : A^\dagger$$

where $\vec{x}$ are the free first-order variables of $A_1, \dots, A_n, A$. The interpretation of logical rules and arithmetical axioms is given in figure 5. Most of these interpretations are standard and the novelty here lies in the interpretation of the $\forall 2e$ rule. The interpretation of this rule consists in an instance of update recursion applied to a program inductively defined over the formula under the second-order quantification.

4.4 Realizability semantics

We now have to prove that this interpretation is correct with respect to the logic. In order to define what we mean by correct, we associate to each formula a realizability value that represents (the interpretations of) the programs which are correct with respect to the formula.

A valuation for a term t (resp. a formula A) is a mapping of first-order variables of t (resp. A) to elements of $\mathbb{N}$ and second-order variables of A to elements of $\mathcal{P}\,(\mathbb{N})$. As we did for programs, instead of pairs (t, v) (resp. (A, v)) of a term (resp. formula) and a valuation for it, we rather use terms (resp. formulas) with parameters, that is, terms (resp. formulas) where free variables are replaced with their assignment through v. The value $|t|$ in $\mathbb{N}$ of a closed term with parameters t is defined using the standard interpretations of S, $+$ and $\times$ in $\mathbb{N}$.

Lemma 4.7. *If t is a closed term with parameters, then $\left[t^\dagger\right] = |t|$.*

Proof. The proof is by induction on t, using that $\left[0^\dagger\right] = 0$, $\left[S\,t^\dagger\right] = \left[t^\dagger\right] + 1$ and the following facts:

$$\left[t^\dagger + u^\dagger\right] = \left[t^\dagger\right] + \left[u^\dagger\right] \qquad \left[t^\dagger \times u^\dagger\right] = \left[t^\dagger\right] \times \left[u^\dagger\right] \qquad \square$$

We are now ready to define the realizability value associated to each formula with parameters. The whole realizability semantics is parameterized by a set $\perp\!\!\!\perp \subseteq \mathbb{N}$ that will be the set of realizers of the formula $\bot$. Allowing for a non-empty set of realizers of $\bot$ is a well-known technique for keeping computational content from classical proofs. Indeed, if $\bot$ has no realizers then for any formula A, either A has realizers, in which case $\neg A$ has no realizer, or A has no realizer, in which case everything is a realizer of $\neg A$. Therefore $\neg\neg A$ has no computational content and we cannot hope to get anything interesting when eliminating double negation on it.

For each closed formula with parameters A we define the realizability value $|A| \subseteq \left|A^\dagger\right|$ of A on $A^\dagger$ as follows:

$$|X\,(t)| = \begin{cases} \mathbb{N} & \text{if } |t| \in X \\ \perp\!\!\!\perp & \text{if } |t| \notin X \end{cases} \qquad |A \Rightarrow B| = |A| \to |B|$$
$$|\forall x\, A| = \bigcap_{x \in \mathbb{N}} (\{x\} \to |A|) \qquad |\forall X\, A| = \bigcap_{X \subseteq \mathbb{N}} |A|$$

4.5 Adequacy

We now prove the main result of our realizability interpretation. The adequacy theorem asserts that the program interpreting a given proof belongs to the realizability value of the formula proven.

Theorem 4.8. *For each proof in second-order arithmetic:*

$$p_1 : A_1, \dots, p_n : A_n \vdash \pi : A$$

such that $FV\,(A_1, \dots, A_n, A) \subseteq \{x_1, \dots, x_m, X_1, \dots, X_k\}$, if:

$$(x_1, \dots, x_m) \in \mathbb{N}^m$$
$$(X_1, \dots, X_k) \in (\mathcal{P}\,(\mathbb{N}))^k$$
$$(a_1, \dots, a_n) \in |A_1| \times \dots \times |A_n|$$

then:

$$\pi^\dagger\,[a_1, \dots, a_n / p_1, \dots, p_n] \in |A|$$

The proof of this theorem proceeds by induction on π. We split it into three parts: adequacy for the axioms, adequacy for the first-order intuitionistic part, and adequacy for the *Dne* and $\forall 2e$ rules.

Adequacy for the axioms. In the case of axioms, the correctness of our interpretation is mostly straightforward. As usual, we use recursion on natural numbers to interpret induction.

Lemma 4.9. *If $\pi : A$ is an axiom of second-order arithmetic then:*

$$\pi^\dagger \in |A|$$

Proof. By cases. For axioms add0, addS, mul0 and mulS, this is a consequence of the fact that if t and u are terms with parameters such that $|t| = |u|$, then $\lambda p.p \in |t = u|$. Indeed, if $|t| = |u|$ then for all $X \subseteq \mathbb{N}$, $|X\,(t)| = |X\,(u)|$, so $|t = u| = |\forall X\,(X \Rightarrow X)| = \perp\!\!\!\perp \to \perp\!\!\!\perp$.

For sinj, let $x, y \in \mathbb{N}$. If $x = y$ then $|S\,x = S\,y| = \perp\!\!\!\perp \to \perp\!\!\!\perp = |x = y|$, so $\lambda p.p \in |S\,x = S\,y \Rightarrow x = y|$. If $x \neq y$ then there exists $X \subseteq \mathbb{N}$ such that $x \in X$ but $y \notin X$, so $|x = y| = \mathbb{N} \to \perp\!\!\!\perp$. Similarly, $|S\,x = S\,y| = \mathbb{N} \to \perp\!\!\!\perp$, and therefore $\lambda p.p \in |S\,x = S\,y \Rightarrow x = y|$.

For sn0, let $x \in \mathbb{N}$. Since $|S\,x| = x + 1 \neq 0$, we have as before $|S\,x = 0| = \mathbb{N} \to \perp\!\!\!\perp$ and so $|S\,x \neq 0| = (\mathbb{N} \to \perp\!\!\!\perp) \to \perp\!\!\!\perp$. We then obtain $\lambda p.p\,0 \in |S\,x \neq 0|$.

For ind, let $X \subseteq \mathbb{N}$, $a \in |X\,(0)|$ and $b \in |\forall x\,(X\,(x) \Rightarrow X\,(S\,x))|$. We prove by induction that for all $x \in \mathbb{N}$, $x\,?_\iota\,a\,b \in |X\,(x)|$. Indeed, $0\,?_\iota\,a\,b = a \in |X\,(0)|$ and if $x\,?_\iota\,a\,b \in |X\,(x)|$ then $x + 1\,?_\iota\,a\,b = b\,x\,(x\,?_\iota\,a\,b) \in |X\,(x+1)|$ since $x\,?_\iota\,a\,b \in |X\,(x)|$ by induction hypothesis. $\square$

$$p^\dagger \equiv p \qquad (\lambda p.\pi)^\dagger \equiv \lambda p.\pi^\dagger \qquad (\pi_1\,\pi_2)^\dagger \equiv \pi_1^\dagger\,\pi_2^\dagger \qquad (\lambda x.\pi)^\dagger \equiv \lambda x.\pi^\dagger \qquad (\pi\,t)^\dagger \equiv \pi^\dagger\,t^\dagger \qquad (\lambda X.\pi)^\dagger \equiv \pi^\dagger$$

$(\mathsf{dne}_A)^\dagger \equiv \mathsf{dne}_{A^\dagger}$ where $\vdash \mathsf{dne}_\sigma : ((\sigma \to \iota) \to \iota) \to \sigma$ is defined by:

$$\mathsf{dne}_\iota \equiv \lambda p.p\,(\lambda q.q)$$
$$\mathsf{dne}_{\sigma\to\tau} \equiv \lambda pq.\mathsf{dne}_\tau\,(\lambda r.p\,(\lambda s.r\,(s\,q)))$$

$$(\pi\,(x.B))^\dagger \equiv \mathsf{dne}_{(A[x.B/X])^\dagger}\left(\lambda p.\mathsf{ur}\left(\lambda r.p\left(\varphi^1_{A,B,X}\,\pi^\dagger\right)\right)\epsilon\right) \text{ where for } \vec{x} = FV(A),$$

$$\vec{x} : \vec{\iota}, r : \iota \to B^\dagger + \left(B^\dagger \to \iota\right) \vdash \varphi^1_{A,x.B,X} : A^\dagger \to (A\,[x.B/X])^\dagger$$
$$\vec{x} : \vec{\iota}, r : \iota \to B^\dagger + \left(B^\dagger \to \iota\right) \vdash \varphi^2_{A,x.B,X} : (A\,[x.B/X])^\dagger \to A^\dagger$$
are defined by:

$$\varphi^1_{X(t),x.B,X} \equiv r\,t^\dagger\,?_+\,(\lambda qp.q)\,(\lambda q.\mathsf{exf}_{B^\dagger}) \qquad \varphi^1_{Y(t),x.B,X} \equiv \lambda p.p \qquad \varphi^1_{\forall\,y\,A,x.B,X} \equiv \lambda py.\varphi^1_{A,B,X}\,(p\,y) \qquad \varphi^1_{\forall\,Y\,A,x.B,X} \equiv \varphi^1_{A,B,X}$$
$$\varphi^2_{X(t),x.B,X} \equiv r\,t^\dagger\,?_+\,(\lambda qp.0)\,(\lambda q.q) \qquad \varphi^2_{Y(t),x.B,X} \equiv \lambda p.p \qquad \varphi^2_{\forall\,y\,A,x.B,X} \equiv \lambda py.\varphi^2_{A,B,X}\,(p\,y) \qquad \varphi^2_{\forall\,Y\,A,x.B,X} \equiv \varphi^2_{A,B,X}$$
if $Y \neq X$

$$\varphi^1_{A_1\Rightarrow A_2,x.B,X} \equiv \lambda pq.\varphi^1_{A_2,x.B,X}\left(p\left(\varphi^2_{A_2,x.B,X}\,q\right)\right)$$
$$\varphi^2_{A_1\Rightarrow A_2,x.B,X} \equiv \lambda pq.\varphi^2_{A_2,x.B,X}\left(p\left(\varphi^1_{A_2,x.B,X}\,q\right)\right)$$

$$\vdash \mathsf{exf}_\sigma : \iota \to \sigma$$
$$\mathsf{exf}_\sigma \equiv \lambda p.\mathsf{dne}_\sigma\,(\lambda\_.p)$$

$$\mathrm{add0}^\dagger \equiv \lambda xp.p \qquad \mathrm{addS}^\dagger \equiv \lambda xyp.p \qquad \mathrm{mul0}^\dagger \equiv \lambda xp.p \qquad \mathrm{mulS}^\dagger \equiv \lambda xyp.p$$

$$\mathrm{sn0}^\dagger \equiv \lambda xp.p\,0 \qquad \mathrm{sinj}^\dagger \equiv \lambda xyp.p \qquad \mathrm{ind}^\dagger \equiv \lambda pqx.x\,?_\iota\,p\,q$$

Figure 5: Realizability interpretation of second-order arithmetic

Adequacy for the first-order intuitionistic part. We now prove adequacy for the system without the *Dne* and ∀2*e* rules. As for the axioms, this result is straightforward.

Lemma 4.10. *For each proof in second-order arithmetic without the Dne and ∀2e rules:*

$$p_1 : A_1, \ldots, p_n : A_n \vdash \pi : A$$

such that $FV(A_1, \ldots, A_n, A) \subseteq \{x_1, \ldots, x_m, X_1, \ldots, X_k\}$, *if:*

$$(x_1, \ldots, x_m) \in \mathbb{N}^m$$
$$(X_1, \ldots, X_k) \in (\mathcal{P}(\mathbb{N}))^k$$
$$(a_1, \ldots, a_n) \in |A_1| \times \ldots \times |A_n|$$

then:

$$\pi^\dagger\,[a_1, \ldots, a_n/p_1, \ldots, p_n] \in |A|$$

Proof. The proof is by induction on π. Most of the cases are straightforward. For the ∀*e* rule we use lemma 4.7. □

Adequacy for the Dne and ∀2e rules. This section contains the main result: correctness of our interpretation of second-order elimination. First, we prove adequacy for double-negation elimination:

Lemma 4.11. *For each formula A with parameters:*

$$\mathsf{dne}_{A^\dagger} \in |\neg\neg A \Rightarrow A|$$
$$\mathsf{exf}_{A^\dagger} \in |\bot \Rightarrow A|$$

Proof. We proceed by induction on the structure of A for $\mathsf{dne}_{A^\dagger}$.

For $X(t)$, let $a \in |\neg\neg X(t)|$. If $|t| \in X$ then $|X(t)| = \mathbb{N}$, so we have $\mathsf{dne}_{X(t)^\dagger}(a) \in |X(t)|$. If $|t| \notin X$, then $|X(t)| = \perp\!\!\!\perp$ so we have $a \in (\perp\!\!\!\perp \to \perp\!\!\!\perp) \to \perp\!\!\!\perp$ and therefore:

$$\mathsf{dne}_{X(t)^\dagger}(a) = a\,(\lambda p.p) \in \perp\!\!\!\perp = |X(t)|$$

$A \Rightarrow B$ and $\forall x\,A$ are consequences of lemma 4.10.

For $\forall X\,A$, let $a \in |\neg\neg\forall X\,A|$ and let $X \subseteq \mathbb{N}$. Then we have that $|\forall X\,A| = \bigcap_{X\subseteq\mathbb{N}} |A| \subseteq |A|$, so by property of $\to$ with respect to inclusion, $|\neg\neg\forall X\,A| \subseteq |\neg\neg A|$, so $a \in |\neg\neg A|$ and therefore:

$$\mathsf{dne}_{(\forall X\,A)^\dagger}\,a = \mathsf{dne}_{A^\dagger}\,a \in |A|$$

For $\mathsf{exf}_{A^\dagger}$, if $a \in |\bot|$ then $\lambda\_.a \in |\neg\neg A|$ by lemma 4.10 and therefore $\mathsf{exf}_{A^\dagger}(a) = \mathsf{dne}_{A^\dagger}(\lambda\_.a) \in |A|$. □

Before proving adequacy for the ∀2*e* rule, we prove the following lemma about our inductively defined φ^1 and φ^2:

Lemma 4.12. *If:*

$$a \in \bigcap_{x\in\mathbb{N}} (\{x\} \to |B| + |\neg B|)$$
$$X = \{x \in \mathbb{N} \mid a\,x \in \{|B|\,|\}\}$$

then:

$$\varphi^1_{A,B,X}\,[a/r] \in |A \Rightarrow A\,[x.B/X]|$$
$$\varphi^2_{A,B,X}\,[a/r] \in |A\,[x.B/X] \Rightarrow A|$$

Proof. We proceed by induction on A. In the $X\,(t)$ case we need to prove:

$$a\,t^\dagger\,?_+\,(\lambda qp.q)\,\left(\lambda q.\mathtt{exf}_{B^\dagger}\right) \in |X\,(t) \Rightarrow B\,[t/x]|$$
$$a\,t^\dagger\,?_+\,(\lambda qp.0)\,(\lambda q.q) \qquad \in |B\,[t/x] \Rightarrow X\,(t)|$$

If $a\,t^\dagger = \{b\,|\,\}$ for some $b \in |B\,[t/x]|$ then $|X\,(t)| = \mathbb{N}$ by definition of X and lemma 4.7, so:

$$\lambda p.b \in \mathbb{N} \to |B\,[t/x]| = |X\,(t) \Rightarrow B\,[t/x]|$$
$$\lambda p.0 \in |B\,[t/x]| \to \mathbb{N} = |B\,[t/x] \Rightarrow X\,(t)|$$

If $a\,t^\dagger = \{\,|\,b\}$ for some $b \in |\neg B\,[t/x]|$ then $|X\,(t)| = \perp\!\!\!\perp$ by definition of X and lemma 4.7, so:

$$\mathtt{exf}_{B^\dagger} \in \perp\!\!\!\perp \to |B\,[t/x]| = |X\,(t) \Rightarrow B\,[t/x]|$$
$$b \in |B\,[t/x]| \to \perp\!\!\!\perp = |B\,[t/x] \Rightarrow X\,(t)|$$

The cases $Y\,(t)$ for $Y \neq X$, $A_1 \Rightarrow B_2$ and $\forall y\,A$ are consequences of 4.10, and the case $\forall Y\,A$ is a consequence of:

$$|\forall Y\,(C \Rightarrow D)| \subseteq |\forall Y\,C \Rightarrow \forall Y\,D| \qquad \square$$

Finally, we prove adequacy for the $\forall 2e$ rule:

Proposition 4.13. *For each proof in second-order arithmetic without the $\forall 2e$ rule:*

$$p_1 : A_1, \ldots, p_n : A_n \vdash \pi : \forall X\,A$$

such that $FV\,(A_1, \ldots, A_n, \forall X\,A) \subseteq \{x_1, \ldots, x_m, X_1, \ldots, X_k\}$, if:

$$(x_1, \ldots, x_m) \in \mathbb{N}^m$$
$$(X_1, \ldots, X_k) \in (\mathcal{P}\,(\mathbb{N}))^k$$
$$(a_1, \ldots, a_n) \in |A_1| \times \ldots \times |A_n|$$

then:

$$(\pi\,(x.B))^\dagger\,[a_1, \ldots, a_n/p_1, \ldots, p_n] \in |A\,[x.B/X]|$$

Proof. Let $M = \pi^\dagger\,[a_1, \ldots, a_n/p_1, \ldots, p_n] \in |\forall X\,A|$ and let $a \in |\neg A\,[x.B/X]|$. We have to prove:

$$\mathsf{ur}\left(\lambda r.a\left(\varphi^1_{A,B,X}\,M\right)\right)\,\epsilon \in |\perp|$$

We apply proposition 4.3 with $\mathcal{A}_n = |B\,[n/x]|$ and $\mathcal{B} = |\perp|$ so we have to prove:

$$\lambda r.a\left(\varphi^1_{A,B,X}\,M\right) \in \bigcap_{x\in\mathbb{N}} (\{x\} \to |B| + |\neg B|) \to |\perp|$$
$$\epsilon \in \bigcap_{x\in\mathbb{N}} (\{x\} \to |B| + \{\star\})$$

This second property is immediate since for all $n \in \mathbb{N}$, $\epsilon\,n = \{\,|\,\star\}$ by proposition 3.4. For the first property, let:

$$b \in \bigcap_{x\in\mathbb{N}} (\{x\} \to |B| + |\neg B|)$$

so we are left to prove $a\left(\varphi^1_{A,B,X}\,[b/r]\;M\right) \in |\perp|$, and since we have $a \in |\neg A\,[x.B/X]|$ it suffices to prove:

$$\varphi^1_{A,B,X}\,[b/r]\;M \in |A\,[x.B/X]|$$

which is a consequence of lemma 4.12 since:

$$b \in \bigcap_{x\in\mathbb{N}} (\{x\} \to |B| + |\neg B|)$$

and $M \in |\forall X\,A| \subseteq |A\,[\{x \in \mathbb{N}\,|\,b\,x \in \{B\,|\,\}\}\,/X]|$. $\square$

We now give an intuition about the computational behavior of our interpretation of the $\forall 2e$ rule.

The first argument that we give to update recursion for realizing the $\forall 2e$ rule is some:

$$a \in \bigcap_{x\in\mathbb{N}} (\{x\} \to |B| + |\neg B|) \to |\perp|$$

This realizer is built from the inductively defined $\varphi^1_{A,x.B,X}$ and $\varphi^2_{A,x.B,X}$ and goes down in the structure of A to find every occurence of $X\,(t)$ for some term t. For each such occurence, a replaces $X\,(t)$ with $B\,[t/x]$ depending on its input at point $t^\dagger$, and on whether $X\,(t)$ occurs positively (φ^1) or negatively (φ^2) in A:

- if the input of a at point $t^\dagger$ is some $\{b\,|\,\}$ then:
  - if $X\,(t)$ occurs positively then a uses $\lambda\_.b$ as a realizer of $\top \Rightarrow B\,[t/x]$
  - if $X\,(t)$ occurs negatively then a uses a trivial realizer of $B\,[t/x] \Rightarrow \top$
- if the input of a at point $t^\dagger$ is some $\{\,|\,b\}$ then:
  - if $X\,(t)$ occurs positively then a uses $\mathtt{exf}_{B^\dagger}$ as a realizer of $\perp \Rightarrow B\,[t/x]$
  - if $X\,(t)$ occurs negatively then a uses b as a realizer of $B\,[t/x] \Rightarrow \perp$

We now explain the meaning of the second argument of update recursion, on which recursion happens. Each partial function:

$$f \in \bigcap_{x\in\mathbb{N}} (\{x\} \to |B| + \{\star\})$$

can be understood as a "current knowledge" about the ns at which we have a realizer of $B\,(n)$. For a given f, if $f\,n$ is some $\{b\,|\,\}$ then the "current knowledge" is that b is a realizer for $B\,(n)$. Otherwise if $f\,n$ is $\{\,|\,\star\}$ then we know nothing about $B\,(n)$ yet. At the beginning of the computation we have no information at all so the second argument is the "empty knowledge" ϵ.

Update recursion then has to provide a with some realizer of $\bigcap_{x\in\mathbb{N}} (\{x\} \to |B| + |\neg B|)$, that is, a "perfect knowledge" that provides for each n either a realizer of $B\,(n)$, or a realizer of $\neg B\,(n)$. The way it approximates such a perfect knowledge is as follows: if n is such that $f\,n = \{b\,|\,\}$ for some $b \in |B\,(n)|$, then the "current knowledge" already contains a realizer for $B\,(n)$, so update recursion can simply return it. Otherwise, update recursion has to provide either a realizer of $B\,(n)$, or a realizer of $\neg B\,(n)$. At that point update recursion chooses to build a realizer of $\neg B\,(n)$ by reading from a a realizer b of $B\,(n)$ and making a recursive call with an extended knowledge that contains b as a realizer of $B\,(n)$. Inside the recursive call, if a reads again its input at point n then update recursion will use the new current knowledge and answer with this same b that a provided before the recursive call.

This recursive process will eventually stop at some point because a being continuous, it can only look at the "current knowledge" at a finite number of points.

4.6 Extraction

The adequacy theorem gives us the possibility of extracting programs from proofs in second-order arithmetic, as witnessed by the following extraction theorem:

Theorem 4.14. *Each proof in second-order arithmetic of a closed formula of the form* $\forall x \forall X\ (\forall y\ (t = 0 \Rightarrow X) \Rightarrow X)$ *(which is the usual second-order encoding of* $\forall x \exists y\ t = 0$*) can be extracted to a program* $M : \iota \to \iota$ *such that for all* $x \in \mathbb{N}$ *there exists* $y \in \mathbb{N}$ *such that* $M\,\overline{x} \rightsquigarrow^* \overline{y}$ *with* $|t| = 0$.

Proof. Let π be a proof of $\forall x \exists y\ t = 0$. The adequacy lemma gives us for each $x \in \mathbb{N}$, $\pi^\dagger\ x \in |\forall y\ (t = 0 \Rightarrow \emptyset) \Rightarrow \emptyset|$. Fix now:

$$\perp\!\!\!\perp = \{y \in \mathbb{N} \mid |t| = 0\}$$

We now prove that $\lambda yp.p\ y \in |\forall y\ (t = 0 \Rightarrow \emptyset)|$. Let $y \in \mathbb{N}$ and let $a \in |t = 0| = |\forall Y\ (Y\ (t) \Rightarrow Y\ (0))|$. In particular for $Y = \mathbb{N} \setminus \{0\}$, $a \in |Y\ (t)| \to \perp\!\!\!\perp$. There are two cases:

- If $|t| = 0$, then $y \in \perp\!\!\!\perp = |Y\ (t)|$ by definition of $\perp\!\!\!\perp$,
- if $|t| \neq 0$ then $|Y\ (t)| = \mathbb{N}$ so $y \in |Y\ (t)|$.

In both cases $a\ y \in \perp\!\!\!\perp$. Therefore we get:

$$\lambda yp.p\ y \in |\forall y\ (t = 0 \Rightarrow \emptyset)|$$

so:

$$\pi^\dagger\ x\ (\lambda yp.p\ y) \in \perp\!\!\!\perp = \{y \in \mathbb{N} \mid |t| = 0\}$$

We can therefore define:

$$M = \lambda x.\pi^\dagger\ x\ (\lambda yp.p\ y)$$

and we have by proposition 3.7 that for all $x \in \mathbb{N}$ there exists $y \in \mathbb{N}$ such that $M\,\overline{x} \rightsquigarrow^* \overline{y}$ and $|t| = 0$. □

5 CONCLUSION AND FUTURE WORKS

We have defined a bar recursive interpretation of second-order arithmetic presented as arithmetic with quantification on predicates rather than the equivalent axiom scheme of comprehension. This presentation of second-order arithmetic is the one that most closely reflects the typing rules of polymorphic λ-calculus, and as such we made a step towards a comparison of the two families of interpretations of second-order arithmetic: bar recursion and system F. As a future work we would like to deepen the understanding of the connection between these two principles by comparing the computational behavior of programs extracted from a single proof via the two techniques.

Another aspect that we would like to study is wether it is possible to use control operators in the interpretation of the $\forall 2e$ rule. Indeed, there is a strong connection between the negative translation of proofs and the continuation-passing style (cps) translation of programs, the latter being the Curry-Howard equivalent of the former. Calculi with control features have been designed to interpret classical proofs directly. Most of these calculi contain a notion of duality that corresponds on the logical side to the duality between a formula and its negation, and on the computational side to a call-by-name or a call-by-value evaluation strategy. During the computation of an approximation to the "perfect knowledge" mentioned in the previous section, update recursion has an asymmetric behavior that consists in building a realizer of $\neg B$ by reading a realizer of B and making a recursive call with a new knowledge extended with this new realizer. This behavior corresponds to the call-by-name interpretation of the excluded middle under a cps translation. We would therefore like to have a version of update recursion that uses control operators and can be translated either to the current version through a call-by-name cps translation, or to a dual version through a call-by-value cps translation. Moreover, control operators can capture context and restore it at a later point. We would like to explore the possibility of using this property to define more intuitive versions of our φ^1 and φ^2 that could act on all instances of $X\ (t)$ in a formula through context capture.

REFERENCES

[1] Roberto Amadio and Pierre-Louis Curien. 1998. *Domains and Lambda-Calculi.* Cambridge Tracts in Theoretical Computer Science, Vol. 46. Cambridge University Press.

[2] Federico Aschieri, Stefano Berardi, and Giovanni Birolo. 2013. Realizability and Strong Normalization for a Curry-Howard Interpretation of HA + EM1. In *Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2-5, 2013, Torino, Italy*, Vol. 23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 45–60.

[3] Stefano Berardi, Marc Bezem, and Thierry Coquand. 1998. On the Computational Content of the Axiom of Choice. *Journal of Symbolic Logic* 63, 2 (1998), 600–622.

[4] Ulrich Berger. 2001. The Berardi-Bezem-Coquand-functional in a domain-theoretic setting. (2001). http://www-compsci.swan.ac.uk/~csulrich/ftp/bbc.ps.gz.

[5] Ulrich Berger. 2004. A Computational Interpretation of Open Induction. In *19th IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 326.

[6] Ulrich Berger and Paulo Oliva. 2005. Modified bar recursion and classical dependent choice. In *Logic Colloquium '01 (Lecture Notes in Logic, Vol. 20)*. Springer-Verlag, 89–107.

[7] Jean-Yves Girard. 1971. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *2nd Scandinavian Logic Symposium*. North-Holland, 63–69.

[8] Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Ph. D. Dissertation. Université Paris 7.

[9] Kurt Gödel. 1958. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica* 12, 3-4 (1958), 280–287.

[10] Georg Kreisel. 1959. Interpretation of analysis by means of constructive functionals of finite types. In *Constructivity in mathematics: Proceedings of the colloquium held at Amsterdam, 1957 (Studies in Logic and the Foundations of Mathematics)*. North-Holland Publishing Company, 101–128.

[11] Jean-Louis Krivine. 2016. Bar Recursion in Classical Realisability: Dependent Choice and Continuum Hypothesis. In *25th EACSL Annual Conference on Computer Science Logic*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 25:1–25:11.

[12] Thomas Powell. 2016. Gödel's functional interpretation and the concept of learning. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*. ACM, 136–145.

[13] John Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium, Paris, April 9-11, 1974 (Lecture Notes in Computer Science)*. Springer, 408–423.

[14] Clifford Spector. 1962. Provably recursive functionals of analysis: a consistency proof of analysis by an extension of principles in current intuitionistic mathematics. In *Recursive Function Theory: Proceedings of Symposia in Pure Mathematics*, Vol. 5. American Mathematical Society, 1–27.

[15] Thomas Streicher. 2017. A Classical Realizability Model arising from a Stable Model of Untyped Lambda Calculus. *Logical Methods in Computer Science* 13, 4 (2017).