

Modified Realizability and Inductive Types

Andrea Asperti and Enrico Tassi

Dipartimento di Scienze dell'Informazione
Mura Anteo Zamboni 7, Bologna
aspersi@cs.unibo.it, tassi@cs.unibo.it

Abstract. In 1959, Kreisel introduced a notion of “modified” realizability that, among other things, provides an alternative technique to Gödel functional (dialectica) interpretation for establishing the connection between Peano Arithmetic and System T. While the dialectica interpretation has been widely studied in the literature, Kreisel’s technique, although remarkably simpler, has apparently been almost neglected (with the only exception of Troelstra). In this paper we give a modern presentation of modified realizability, and generalize it to arbitrary inductive types in a first order setting. This is part of a larger program, advocating the study of logical systems with primitive inductive types starting from weak, predicative logical frameworks and adding little by little small bits of logical power.

1 Introduction

The characterization of the provable recursive functions of Peano Arithmetic as the terms of system T is a well known result of Gödel [7, 8]. Although several authors acknowledge that the functional interpretation of the Dialectica paper is not among the major achievements of the author (see e.g. [5]), the result has been extensively investigated and there is a wide literature on the topic (see e.g. [20, 9, 5, 2], just to mention textbooks, and the bibliography therein).

A different, more neglected, but for many respects much more direct relation between Peano (or Heyting) Arithmetics and Gödel System T is provided by the so called *modified realizability*. Modified realizability was first introduced by Kreisel in [13] - although it will take you a bit of effort to recognize it in the few lines of paragraph 3.52 - and later in [14] under the name of generalized realizability. The name of modified realizability seems to be due to Troelstra [20] - who apparently disagreed Kreisel’s terminology but unfortunately failed in proposing a valid alternative; we shall reluctantly adopt this latter name to avoid further confusion. Modified realizability is a typed variant of realizability, essentially providing interpretations of HA^ω into itself: each theorem is realized by a typed function of system T, that also gives the actual computational content extracted from the proof. In spite of the simplicity and the elegance of the proof, it is extremely difficult to find a modern discussion of this result; the most recent exposition we are aware of is in the encyclopedic work by Troelstra [20] (pp.213-229) going back to thirty years ago. Even modern introductory books

to Type Theory and Proof Theory devoting much space to system T such as [6] and [21] surprisingly leave out this simple and illuminating result. Both the previous textbooks prefer to focus on higher order arithmetics and their relation with Girard's System F [4]. However, after a first period of enthusiasm, the impredicative encoding of inductive types in Logical Frameworks has shown several problems and limitations (see e.g. [22] pp.24-25) mostly solved by assuming inductive types as a primitive logical notion (leading, e.g., from the Calculus of Constructions to the Calculus of Inductive Constructions - CIC). Even the extraction algorithm of CIC, strictly based on realizability principles, and in a first time still oriented towards System F [17,18] has been recently rewritten [15] to take advantage of concrete types and pattern matching of ML-like languages. Unfortunately, systems like the Calculus of Inductive Constructions are so complex, from the logical point of view, to substantially prevent a really neat theoretical exposition (at present, there does not even exist a truly complete consistency proof covering all aspects of such systems); moreover, not everybody may be interested in all the features offered by these frameworks, from polymorphism to types depending on proofs. Our program is to restart the analysis of logical systems with primitive inductive types in a smooth way, starting from first order logic and adding little by little small bits of logical power. This paper is the first step in this direction.

2 Gödel system T

We shall use a variant of system T with three atomic types \mathbb{N} (natural numbers), \mathbb{B} (booleans) and $\mathbf{1}$ (a terminal object), and two binary type constructors \times (product) and \rightarrow (arrow type).

The terms of the language comprise the usual simply typed lambda terms with explicit pairs, plus the following additional constants:

$$\begin{array}{ll}
 * : \mathbf{1} & \\
 true : \mathbb{B}, \quad false : \mathbb{B} & D : A \rightarrow A \rightarrow \mathbb{B} \rightarrow A \\
 O : \mathbb{N}, \quad S : \mathbb{N} \rightarrow \mathbb{N} & R : A \rightarrow (A \rightarrow \mathbb{N} \rightarrow A) \rightarrow \mathbb{N} \rightarrow A
 \end{array}$$

Redexes comprise β -reduction, projections, and type specific reductions as reported in the following table.

$$\begin{array}{llll}
 \lambda x : U.M \ N \rightsquigarrow M[N/x] & (\beta) & M \rightsquigarrow * \quad \text{for any M of type } \mathbf{1} & (*) \\
 \pi_1 \langle M, N \rangle \rightsquigarrow M & (proj_1) & \pi_2 \langle M, N \rangle \rightsquigarrow N & (proj_2) \\
 D \ M \ N \ true \rightsquigarrow M & (D_{true}) & D \ M \ N \ false \rightsquigarrow N & (D_{false}) \\
 R \ M \ F \ 0 \rightsquigarrow M & (R_0) & R \ M \ F \ (S \ n) \rightsquigarrow F \ (R \ M \ F \ n) \ n & (R_S)
 \end{array}$$

Note that using the well known isomorphism $\mathbf{1} \rightarrow A \cong A$, $A \rightarrow \mathbf{1} \cong \mathbf{1}$ and $A \times \mathbf{1} \cong A \cong \mathbf{1} \times A$ (see [1], pp.231-239) we may always get rid of $\mathbf{1}$ (apart the trivial case). The terminal object does not play a major role in our treatment, but it allows to extract better algorithms. In particular we use it to realize atomic propositions, and stripping out the terminal object using the above isomorphisms gives a simple way of just keeping the truly informative part of the algorithms.

3 Heyting's arithmetics

In the following, we shall use Howard's formulae-as-types notion of construction [11] as a convenient way to give a compact, syntactical description of proofs as typed λ -terms. Terms corresponding to axiom schemata are "polymorphic terms" standing, in fact, for a whole family of different constants.

Terms $x, y, z, \dots, O, S(t), t_1 + t_2, t_1 \cdot t_2$.

Formulas $\perp, P \wedge Q, P \vee Q, P \rightarrow Q, \exists x.P, \forall x.P$

Logical Axioms

$$\begin{array}{ll}
 \text{left} : P \wedge Q \rightarrow P & \text{right} : P \wedge Q \rightarrow Q \\
 \text{or_in_l} : P \rightarrow P \vee Q & \text{or_in_r} : P \rightarrow Q \vee P \\
 \text{conj} : P \rightarrow Q \rightarrow P \wedge Q & \text{ex_intro} : \forall x.(P \rightarrow \exists x.P) \\
 \text{false_ind} : \perp \rightarrow Q & \\
 \text{or_ind} : (P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow P \vee Q \rightarrow R & \\
 \text{ex_ind} : (\forall x.P(x) \rightarrow Q) \rightarrow \exists x.P(x) \rightarrow Q & (x \text{ not free in } Q)
 \end{array}$$

Arithmetical Axioms

$$\begin{array}{ll}
 \text{discr} : \forall x.0 = S(x) \rightarrow \perp & \text{injS} : \forall x, y.S(x) = S(y) \rightarrow x = y \\
 \text{plus_O} : \forall x.x + 0 = x & \text{plus_S} : \forall x, y.x + S(y) = S(x + y) \\
 \text{times_O} : \forall x.x \cdot 0 = 0 & \text{times_S} : \forall x, y.x \cdot S(y) = x + (x \cdot y) \\
 \text{nat_ind} : P(0) \rightarrow (\forall x.P(x) \rightarrow P(S(x))) \rightarrow \forall x.P(x) &
 \end{array}$$

Inference Rules

$$\begin{array}{ll}
 \Gamma, x : P, \Delta \vdash x : P \quad (\text{Proj}) & \Gamma \vdash ax : AX \quad (\text{Const}) \\
 \\
 \frac{\Gamma, x : P \vdash M : Q}{\Gamma \vdash \lambda x : P.M : P \rightarrow Q} \quad (\rightarrow_i) & \frac{\Gamma \vdash M : P \rightarrow Q \quad \Gamma \vdash N : P}{\Gamma \vdash MN : Q} \quad (\rightarrow_e) \\
 \\
 \frac{\Gamma \vdash M : P}{\Gamma \vdash \lambda x : \mathbb{N}.M : \forall x.P} \quad (\forall_i) & \frac{\Gamma \vdash M : \forall x.P}{\Gamma \vdash Mt : P[t/x]} \quad (\forall_e)
 \end{array}$$

where $ax : AX$ is any logical or arithmetical axiom in the previous lists.

4 Extraction

In this section we define a mapping $\llbracket \cdot \rrbracket$ from proofs in HA to terms in system T. The first step is to define a translation from formulae to types:

$$\begin{array}{ll}
 \llbracket P \rrbracket = \mathbf{1} \text{ if } P \text{ is atomic} & \llbracket P \wedge Q \rrbracket = \llbracket P \rrbracket \times \llbracket Q \rrbracket \\
 \llbracket P \rightarrow Q \rrbracket = \llbracket P \rrbracket \rightarrow \llbracket Q \rrbracket & \llbracket \forall x.P \rrbracket = \mathbb{N} \rightarrow \llbracket P \rrbracket \\
 \llbracket P \vee Q \rrbracket = \mathbb{B} \times (\llbracket P \rrbracket \times \llbracket Q \rrbracket) & \llbracket \exists x.P \rrbracket = \mathbb{N} \times \llbracket P \rrbracket
 \end{array}$$

Terms t of the logical systems are mapped in the same term of system T, after having suitably defined $+$ and \cdot by primitive recursion.

The following table provides the proofs to terms mapping for axioms and structured proofs.

$$\begin{aligned}
\llbracket \lambda x : P.M \rrbracket &= \lambda x : \llbracket P \rrbracket . \llbracket M \rrbracket & \llbracket MN \rrbracket &= \llbracket M \rrbracket \llbracket N \rrbracket \\
\llbracket \lambda x : \mathbb{N}.M \rrbracket &= \lambda x : \mathbb{N} . \llbracket M \rrbracket & \llbracket Mt \rrbracket &= \llbracket M \rrbracket t \\
\llbracket left \rrbracket &= \pi_1 & \llbracket right \rrbracket &= \pi_2 \\
\llbracket or_in_l \rrbracket &= \lambda x : \llbracket P \rrbracket . \langle true, \langle x, \perp_{\llbracket Q \rrbracket} \rangle \rangle & \llbracket or_in_r \rrbracket &= \lambda y : \llbracket Q \rrbracket . \langle false, \langle \perp_{\llbracket P \rrbracket}, y \rangle \rangle \\
\llbracket conj \rrbracket &= \lambda x : \llbracket P \rrbracket . \lambda y : \llbracket Q \rrbracket . \langle x, y \rangle & \llbracket ex_intro \rrbracket &= \lambda x : \mathbb{N} . \lambda f : \llbracket P \rrbracket . \langle x, f \rangle \\
\llbracket discr \rrbracket &= \lambda_- : \mathbb{N} . \lambda_- : \mathbf{1} . * & \llbracket injS \rrbracket &= \lambda_- : \mathbb{N} . \lambda_- : \mathbb{N} . \lambda_- : \mathbf{1} . * \\
\llbracket plus_O \rrbracket &= \llbracket times_O \rrbracket = \lambda_- : \mathbb{N} . * & \llbracket plus_S \rrbracket &= \llbracket times_S \rrbracket = \lambda_- : \mathbb{N} . \lambda_- : \mathbb{N} . * \\
\llbracket false_ind \rrbracket &= \lambda_- : \mathbf{1} . \perp_{\llbracket Q \rrbracket} & \llbracket nat_ind \rrbracket &= R \\
\llbracket ex_ind \rrbracket &= \lambda f : (\mathbb{N} \rightarrow \llbracket P \rrbracket \rightarrow \llbracket Q \rrbracket) . \lambda p : \mathbb{N} \times \llbracket P \rrbracket . f (\pi_1 p) (\pi_2 p) \\
\llbracket or_ind \rrbracket &= \lambda f : \llbracket P \rightarrow R \rrbracket . \lambda g : \llbracket Q \rightarrow R \rrbracket . \lambda z : \llbracket P \vee Q \rrbracket . \\
& \quad D (f (\pi_1 (\pi_2 z))) (g (\pi_2 (\pi_2 z))) (\pi_1 z)
\end{aligned}$$

The previous translation requires the definition of a canonical element \perp_T for any type T of system \mathbb{T} , defined as follows

$$\perp_{\mathbf{1}} = * \quad \perp_B = true \quad \perp_N = 0 \quad \perp_{U \times V} = \langle \perp_U, \perp_V \rangle \quad \perp_{U \rightarrow V} = \lambda_- : U . \perp_V$$

Lemma 1. *For all predicate $P(x)$ and for all terms t_1 and t_2 , $\llbracket P(t_1) \rrbracket = \llbracket P(t_2) \rrbracket$*

Proof. Trivial, since the definition of $\llbracket \cdot \rrbracket$ does not look inside atomic formulas.

In spite of using the boolean type \mathbb{B} , we could have enriched system \mathbb{T} with a sum type $P + Q$, interpreting or_in_l and or_in_r with the two injections, and or_ind with elimination by cases. Our approach is closer to the standard realizability interpretation.

5 Modified realizability

The realizability relation is a relation $f \mathcal{R} P$ where $f : \llbracket P \rrbracket$ (we assume P is a closed formula). In particular:

- $\neg(* \mathcal{R} \perp)$
- $* \mathcal{R} (t_1 = t_2)$ iff $t_1 = t_2$ is true
- $\langle f, g \rangle \mathcal{R} (P \wedge Q)$ iff $f \mathcal{R} P$ and $g \mathcal{R} Q$
- $\langle true, \langle f, g \rangle \rangle \mathcal{R} (P \vee Q)$ iff $f \mathcal{R} P$
- $\langle false, \langle f, g \rangle \rangle \mathcal{R} (P \vee Q)$ iff $g \mathcal{R} Q$
- $f \mathcal{R} (P \rightarrow Q)$ iff for any m such that $m \mathcal{R} P$, $(f m) \mathcal{R} Q$
- $f \mathcal{R} (\forall x.P)$ iff for any natural number n $(f n) \mathcal{R} P[n/x]$
- $\langle n, g \rangle \mathcal{R} (\exists x.P)$ iff $g \mathcal{R} P[n/x]$

With the notation \underline{n} , where n is a natural number, we indicate the corresponding term of HA.

Lemma 2. *For any axiom $ax : Ax$, $\llbracket ax \rrbracket : \llbracket Ax \rrbracket$ and $\llbracket ax \rrbracket \mathcal{R} Ax$.*

Proof. Typing is an easy check, so we focus on realizability, starting from the most interesting cases.

nat.ind. We must prove that the recursion schema R realizes the induction principle. To this aim we must prove that for any a and f such that $a \mathcal{R} P(0)$ and $f \mathcal{R} \forall x.(P(x) \rightarrow P(S(x)))$, and any natural number n , $(Rafn) \mathcal{R} P(\underline{n})$.

We proceed by induction on n .

If $n = 0$, $(Raf0) = a$ and by hypothesis $a \mathcal{R} P(0)$.

Suppose by induction that $(Rafn) \mathcal{R} P(\underline{n})$, and let us prove that the relation still holds for $n + 1$. By definition $(Raf(n+1)) = fn(Rafn)$, and since $f \mathcal{R} \forall x.(P(x) \rightarrow P(S(x)))$, $(fn(Rafn)) \mathcal{R} P(S(\underline{n})) = P(\underline{n+1})$.

ex.ind. We must prove that

$$ex.ind \mathcal{R} (\forall x : (Px) \rightarrow Q) \rightarrow (\exists x : (Px)) \rightarrow Q$$

Following the definition of \mathcal{R} we have to prove that given

$f \mathcal{R} \forall x : ((Px) \rightarrow Q)$ and $p \mathcal{R} \exists x : (Px)$, then $ex.ind f p \mathcal{R} Q$.

p is a couple $\langle n_p, g_p \rangle$ such that $g_p \mathcal{R} P[n_p/x]$, while f is a function such that for all n and for all $m \mathcal{R} P[n/x]$ then $f n m \mathcal{R} Q$ (note that x is not free in Q so $[n/x]$ affects only P).

Expanding the definition of *ex.ind*, *left* and *right* we obtain $f n_p g_p$ that we know is in relation \mathcal{R} with Q since $g_p \mathcal{R} P[n_p/x]$.

ex.intro. We must prove that

$$\lambda x : \mathbb{N} . \lambda f : \llbracket P \rrbracket . \langle x, f \rangle \mathcal{R} \forall x . (P \rightarrow \exists x . P(x))$$

that is for each n $ex.intro n \mathcal{R} (P[n/x] \rightarrow \exists x . P(x))$.

Again by definition of \mathcal{R} we have to prove that, given $m \mathcal{R} P[n/x]$, $ex.intro n m \mathcal{R} \exists x . P(x)$. Expanding the definition of *ex.intro* we have $\langle n, m \rangle \mathcal{R} \exists x . P(x)$ that is true since $m \mathcal{R} P[n/x]$.

or.ind. We have to prove that for any $f \mathcal{R} P \rightarrow R$, any $g \mathcal{R} Q \rightarrow R$ and any $z \mathcal{R} P \vee Q$,

$$D (f (\pi_1 (\pi_2 z))) (g (\pi_2 (\pi_2 z))) (\pi_1 z)$$

realizes R . We have two cases: either z is $\langle true, \langle p, q \rangle \rangle$ and $p \mathcal{R} P$, or z is $\langle false, \langle p, q \rangle \rangle$ and $q \mathcal{R} Q$. In the first case, $D (f p) (g q) true \rightsquigarrow (f p)$ that realizes R by assumption. Similarly in the other case.

or.in.l. We have to prove that

$$\lambda x : \llbracket P \rrbracket . \langle true, \langle x, \perp_{\llbracket Q \rrbracket} \rangle \rangle \mathcal{R} P \rightarrow P \vee Q$$

that is, for each $m \mathcal{R} P$, $\langle true, \langle m, \perp_{\llbracket Q \rrbracket} \rangle \rangle \mathcal{R} P \vee Q$, that holds by definition.

or.in.r. Analogous to *or.in.l.*

left. We have to prove that $\pi_1 \mathcal{R} P \wedge Q \rightarrow P$, that is equal to proving that for each $m \mathcal{R} P \wedge Q$ then $\pi_1 m \mathcal{R} P$. m must be a couple $\langle p, q \rangle$ such that $p \mathcal{R} P$ and $q \mathcal{R} Q$. So we conclude that $\pi_1 m$ reduces to p that is in relation \mathcal{R} with P .

right. Analogous to *left.*

conj. We have to prove that

$$\lambda x : \llbracket P \rrbracket . \lambda y : \llbracket Q \rrbracket . \langle x, y \rangle \mathcal{R} P \rightarrow Q \rightarrow P \wedge Q$$

Following the definition of \mathcal{R} we have to show that for each $m \mathcal{R} P$ and for each $n \mathcal{R} Q$ then $(\lambda x : \llbracket P \rrbracket . \lambda y : \llbracket Q \rrbracket . (x, y)) m n \mathcal{R} P \wedge Q$.

This is the same of $\langle m, n \rangle \mathcal{R} P \wedge Q$ that is verified since $m \mathcal{R} P$ and $n \mathcal{R} Q$. *false_ind.* We have to prove that $\perp_{\llbracket Q \rrbracket} \mathcal{R} \perp \rightarrow Q$. Trivial, since there is no $m \mathcal{R} \perp$.

discr. Since there is no n such that $0 = Sn$ is true, $\text{discr } n \mathcal{R} 0 = S \underline{n} \rightarrow \perp$ for each n .

injS. We have to prove that for each n_1 and n_2

$$(\lambda_- : \mathbb{N} . \lambda_- : \mathbb{N} . \lambda_- : \mathbf{1} . *) n_1 n_2 \mathcal{R} (S(x) = S(y) \rightarrow x = y)[n_1/x][n_2/y].$$

We assume that $m \mathcal{R} S(n_1) = S(n_2)$ and we have to show that $(\lambda_- : \mathbb{N} . \lambda_- : \mathbf{1} . *) n_1 n_2 m$ that reduces to $*$ is in relation \mathcal{R} with $n_1 = n_2$. Since in the standard model of natural numbers $S(n_1) = S(n_2)$ implies $n_1 = n_2$ we have that $* \mathcal{R} n_1 = n_2$.

plus.O. Since in the standard model for natural numbers 0 is the neutral element for addition $\lambda_- : \mathbb{N} . * \mathcal{R} \forall x . x + 0 = x$.

plus.S. In the standard model of natural numbers the addition of two numbers is the operation of counting the second starting from the first. So

$$\lambda_- : \mathbb{N} . \lambda_- : \mathbb{N} . * \mathcal{R} \forall x, y . x + S(y) = S(x + y)$$

times.O. Since in the standard model for natural numbers 0 is the absorbing element for multiplication $\lambda_- : \mathbb{N} . * \mathcal{R} \forall x . x \cdot 0 = 0$.

times.S. In the standard model of natural numbers the multiplications of two numbers is the operation of adding the first to himself a number of times equal to the second number. So

$$\lambda_- : \mathbb{N} . \lambda_- : \mathbb{N} . * \mathcal{R} \forall x, y . x + S(y) = S(x + y)$$

To make the statement of the next theorem more readable we adopt the following notation: \overrightarrow{B} has to be intended as B_1, \dots, B_n , and $x : \overrightarrow{\mathbb{N}}$ is a shortcut for $x_1 : \mathbb{N}, \dots, x_m : \mathbb{N}$.

Theorem 1 For any provable sequent $\overrightarrow{B} \vdash M : P$ with free variables in \overrightarrow{x} ,

$$\lambda x : \overrightarrow{\mathbb{N}} . \lambda b : \overrightarrow{\llbracket B \rrbracket} . \llbracket M \rrbracket \mathcal{R} \forall \overrightarrow{x} . \overrightarrow{B} \rightarrow P$$

Proof. The proof is by induction on M ; we only consider the case of (\rightarrow_e) (the other cases are trivial or similar).

(\rightarrow_e) . We know by hypothesis that for all $n : \mathbb{N}$, and any $\overrightarrow{m} \mathcal{R} \overrightarrow{B}$,

$$\llbracket M \rrbracket [\overrightarrow{n} / \overrightarrow{x}; \overrightarrow{m} / \overrightarrow{b}] \mathcal{R} P \rightarrow Q$$

and similarly,

$$\llbracket N \rrbracket [\overrightarrow{n} / \overrightarrow{x}; \overrightarrow{m} / \overrightarrow{b}] \mathcal{R} P$$

Hence $(\llbracket M \rrbracket [\overrightarrow{n} / \overrightarrow{x}; \overrightarrow{m} / \overrightarrow{b}] \llbracket N \rrbracket [\overrightarrow{n} / \overrightarrow{x}; \overrightarrow{m} / \overrightarrow{b}])$, that is equivalent to $(\llbracket M \rrbracket \llbracket N \rrbracket) [\overrightarrow{n} / \overrightarrow{x}; \overrightarrow{m} / \overrightarrow{b}]$, realizes Q .

Corollary 1. *For any proof M of a Π_2 -formula $\forall x \exists y. P(x, y)$, $\llbracket M \rrbracket : \mathbb{N} \rightarrow \mathbb{N} \times \mathbf{1} \equiv \mathbb{N} \rightarrow \mathbb{N}$, and $P(m, \llbracket M \rrbracket m)$ is true.*

By the previous corollary, $\llbracket \cdot \rrbracket$ allows to extract the computational content of the proof.

Corollary 2. *(Troelstra [20]). The provably recursive functions of Peano Arithmetics are exactly the functions of system T .*

Proof. In one direction, it amounts to prove that the normalization proof for terms of system T may be expressed in Peano Arithmetics. In the other direction, if a function is provably total it means that for a suitable encoding \underline{n} the function it is possible to prove $\vdash \forall x \exists y. T(\underline{n}, x, y)$, where T is the well known Kleene's predicate. This is a Π_2 -formula and Peano and Heyting arithmetics have the same expressive power on the Π_2 -fragment (see e.g. [19]), so it is possible to extract a term t of system T such that, for any m , $T(n, m, t m)$ is true, and $U \circ t$ is the desired function (U is the result-extracting function associated with T).

Corollary 3. *Heyting arithmetic is consistent.*

Proof. If we have a proof $\vdash p : \perp$ then $\llbracket p \rrbracket \mathcal{R} \perp$, but this is impossible for definition of realizability¹.

The previous result also holds for Peano arithmetic, since \perp is a Π_2 -formula.

Example Let us prove the following principle of well founded induction:

$$(\forall m. (\forall p. p < m \rightarrow P p) \rightarrow P m) \rightarrow \forall n. P n$$

In the following proof we shall make use of proof-terms, since we finally wish to extract the computational content; we leave to reader the easy check that the proof object describes the usual and natural proof of the statement.

We assume to have already proved the following lemmas (having trivial realizers):

$$L : \forall p, q. p < q \rightarrow q \leq 0 \rightarrow \perp \quad M : \forall p, q, n. p < q \rightarrow q \leq (Sn) \rightarrow p \leq n$$

Let us assume $h : \forall m. (\forall p. p < m \rightarrow P p) \rightarrow P m$. We prove by induction on n that $\forall q. q \leq n \rightarrow P q$. For $n = 0$, we get a proof of $P q$ by

$$B \equiv \lambda q. \lambda h_0 : q \leq 0. h q (\lambda p. \lambda k : p < q. \text{false_ind } (L p q k h_0))$$

In the inductive case, we must prove that, for any n ,

$$(\forall q. q \leq n \rightarrow P q) \rightarrow (\forall q. q \leq Sn \rightarrow P q)$$

¹ The proof implicitly uses the fact that the functions of system T are well defined total mathematical functions, i.e. the fact that the system is strongly normalizing

Assume $h_1 : \forall q. q \leq n \rightarrow Pq$ and $h_2 : q \leq S n$. Let us prove $\forall p. p < q \rightarrow P p$.
 If $h_3 : p < q$ then $(M p q n h_3 h_2) : p \leq n$, hence $h_1 p (M p q n h_3 h_2) : P p$.
 In conclusion, the proof of the inductive case is

$$I \equiv \lambda n. \lambda h_1 : \forall q. q \leq n \rightarrow P q. \lambda q. \lambda h_2 : q \leq S n. \\ h q (\lambda p. \lambda h_3 : p < q. h_1 p (M p q n h_3 h_2))$$

(where h is free in I). The full proof is

$$\lambda h : \forall m. (\forall p. p < m \rightarrow P p) \rightarrow P m. \lambda m. \text{nat_ind } B \ I \ m \ m \ (le_n \ m)$$

where le_n is a proof that $\forall n. n \leq n$, and the free P in the definition of nat_ind is instantiated with $\forall m. m \leq m \rightarrow P m$.

From the previous proof, after stripping terminal objects, and a bit of eta-contraction to make the term more readable, we extract the following term (types are omitted):

$$R' \equiv \lambda f. \lambda m. R (\lambda n. f \ n \ (\lambda q. *)) (\lambda n \lambda g \lambda q. f \ q \ g) \ m \ m$$

The intuition of this operator is the following: suppose to have a recursive definition $hq = F[h]$ where $q : \mathbb{N}$ and $F[h] : A$. This defines a functional $f : \lambda q. \lambda g. F[g] : N \rightarrow (N \rightarrow A) \rightarrow A$, such that (morally) h is the fixpoint of f . For instance, in the case of the Fibonacci function, f is

$$\text{fibonacci} \equiv \lambda q. \lambda g. \text{if } q = 0 \text{ then } 1 \text{ else if } q = 1 \text{ then } 1 \text{ else } g(q-1) + g(q-2)$$

So f builds a new approximation of h from the previous approximation h taken as input. R' precisely computes the m th-approximation starting from a dummy function $(\lambda q. *_A)$. Alternatively, you may look at g as the “history” (course of values) of h for all values less or equal to q ; then f extends g to $q+1$.

As an example, the computation of $R' \text{ fibonacci } 2$ is reported in Figure ?? . Note that the second argument of fibonacci is always a method to calculate all the previous values of fibonacci .

$$\begin{aligned} R' \text{ fibonacci } 2 &\rightsquigarrow R (\lambda n. \text{fibonacci } n \ (\lambda q. *)) (\lambda n \lambda g \lambda q. \text{fibonacci } q \ g) \ 2 \ 2 \\ &\rightsquigarrow (\lambda n \lambda g \lambda q. \text{fibonacci } q \ g) \ 1 \ (R (\lambda n. \text{fibonacci } n \ (\lambda q. *)) (\lambda n \lambda g \lambda q. \text{fibonacci } q \ g) \ 1) \ 2 \\ &\rightsquigarrow \lambda q. \text{fibonacci } q \ (R (\lambda n. \text{fibonacci } n \ (\lambda q. *)) (\lambda n \lambda g \lambda q. \text{fibonacci } q \ g) \ 1) \ 2 \\ &\rightsquigarrow \lambda q. \text{fibonacci } q \ ((\lambda n \lambda g \lambda q. \text{fibonacci } q \ g) \ 0 \\ &\quad (R (\lambda n. \text{fibonacci } n \ (\lambda q. *)) (\lambda n \lambda g \lambda q. \text{fibonacci } q \ g) \ 0)) \ 2 \\ &\rightsquigarrow \lambda q. \text{fibonacci } q \ (\lambda q. \text{fibonacci } q \ (R (\lambda n. \text{fibonacci } n \ (\lambda q. *)) (\lambda n \lambda g \lambda q. \text{fibonacci } q \ g) \ 0)) \ 2 \\ &\rightsquigarrow \lambda q. \text{fibonacci } q \ (\lambda q. \text{fibonacci } q \ (\lambda n. \text{fibonacci } n \ (\lambda q. *))) \ 2 \\ &\rightsquigarrow \text{fibonacci } 2 \ (\lambda q. \text{fibonacci } q \ (\lambda n. \text{fibonacci } n \ (\lambda q. *))) \\ &\rightsquigarrow (\lambda q. \text{fibonacci } q \ (\lambda n. \text{fibonacci } n \ (\lambda q. *))) \ 1 + (\lambda q. \text{fibonacci } q \ (\lambda n. \text{fibonacci } n \ (\lambda q. *))) \ 0 \\ &\rightsquigarrow \text{fibonacci } 1 \ (\lambda n. \text{fibonacci } n \ (\lambda q. *)) + \text{fibonacci } 0 \ (\lambda n. \text{fibonacci } n \ (\lambda q. *)) \\ &\rightsquigarrow 1 + 1 \\ &\rightsquigarrow 2 \end{aligned}$$

6 Inductive types

In the sequel, we adopt the vector notation to make things more readable. \vec{m} has to be intended as $m_1 \dots m_n$ where n may be equal to 0 (we use $m_1 \vec{m}$ when we want to give a name to the first m and assert $n > 0$). If the vector notation is used inside an arrow type it has a slightly different meaning, $A \rightarrow \vec{B} \rightarrow C$ is a shortcut for $A \rightarrow B_1 \rightarrow \dots \rightarrow B_n \rightarrow C$.

6.1 Inductive types

An inductive type is a datatype freely generated by a given set of constructors c_1, \dots, c_n . Following [22] and [18] we use the notation

$$Ind(X)\{c_1 : C(X); \dots; c_n : C(X)\}$$

to denote such an inductive type², where

$$C(X) ::= X \quad | \quad T \rightarrow C(X) \quad | \quad X \rightarrow C(X)$$

and T is any other inductive type. Typical examples of inductive types are:

$$\begin{aligned} \mathbb{B} &= Ind(X)\{true : X; false : X\} \\ \mathbb{N} &= Ind(X)\{0 : X; S : X \rightarrow X\} \\ Pos &= Ind(X)\{one : X; next : \mathbb{B} \rightarrow X \rightarrow X\} \end{aligned}$$

Note that we do not allow higher-order types in argument position of constructors.

We shall collectively call Ind the class of inductive types, and use the identifier T to denote an arbitrary element of this class.

In order to avoid logical problems with existential quantification over empty types, we add the requirement of having at least one not recursive constructor in any inductive type $A = Ind(X)$, namely to have a constructor

$$c : \vec{T} \rightarrow X$$

(where \vec{T} can be empty). In this case, the canonical element of A is

$$\perp_A = c \ \overline{\perp_T}$$

6.2 Extensions to the logic framework

To talk about arbitrary inductive types we have to extend our logical language to a multi-sorted first-order logical framework. The language of term is extended with all constructors, and (first order) quantification is allowed over any inductive type T , i.e. we have $\forall x : T.A$ and $\exists x : T.A$. Then rules 4 and 5 of the $\llbracket \cdot \rrbracket$ definition are replaced by $\llbracket \forall x : T.P \rrbracket = T \rightarrow \llbracket P \rrbracket$ and $\llbracket \exists x : T.P \rrbracket = T \times \llbracket P \rrbracket$.

² For readability reasons, we prefer to give a label to each constructor instead of using the much more verbose notation $Constr(n, Ind(X)\{\dots\})$ to indicate the n^{th} constructor

For each inductive type we will describe the formation rules and the corresponding induction principle schema.

Symmetrically we have to extend System T with arbitrary inductive types and we will see how their recursors are defined in the following sections.

The definition of the realizability relation is \mathcal{R} is modified substituting each occurrence of \mathbb{N} with a generic inductive type T .

6.3 Induction principle

The induction principle for an inductive type X and a predicate Q has the following type

$$X_{ind} = \overrightarrow{\Delta\{C(X), c\}} \rightarrow \forall t : X. Q(t)$$

Δ takes a constructor type $C(X)$ and a term c (initially c is a constructor of X , and $c : C(X)$) and is defined by recursion as follows:

$$\begin{aligned} \Delta\{X, c\} &= Q(c) \\ \Delta\{T \rightarrow C(X), c\} &= \forall m : T. \Delta\{C(X), c\ m\} \\ \Delta\{X \rightarrow C(X), c\} &= \forall t : X. Q(t) \rightarrow \Delta\{C(X), c\ t\} \end{aligned}$$

6.4 Recursor

The type of the recursor R_X on an inductive type X is

$$\overrightarrow{\square\{C(X)\}} \rightarrow X \rightarrow \alpha$$

\square is defined by recursion on the constructor type $C(X)$.

$$\begin{aligned} \square\{X\} &= \alpha \\ \square\{T \rightarrow C(X)\} &= T \rightarrow \square\{C(X)\} \\ \square\{X \rightarrow C(X)\} &= X \rightarrow \alpha \rightarrow \square\{C(X)\} \end{aligned}$$

We say that

$$R_X \overrightarrow{f} (c_i \overrightarrow{m}) \rightsquigarrow \nabla\{C(X)_i, f_i, \overrightarrow{m}\}$$

∇ takes a constructor type $C(X)$, a term f (of type $\square\{C(X)\}$) and is defined by recursion as follows:

$$\begin{aligned} \nabla\{X, f, _ \} &= f \\ \nabla\{T \rightarrow C(X), f, m_1 \overrightarrow{m}\} &= \nabla\{C(X), f\ m_1, \overrightarrow{m}\} \\ \nabla\{X \rightarrow C(X), f, m_1 \overrightarrow{m}\} &= \nabla\{C(X), f\ m_1 (R_X \overrightarrow{f} m_1), \overrightarrow{m}\} \end{aligned}$$

We assume $R_X \overrightarrow{f} (c_i \overrightarrow{m})$ is well typed, so in the first case we can omit \overrightarrow{m} since it is an empty sequence.

Example 1. The three inductive types \mathbb{B} , \mathbb{N} and Pos of section 6.1 generate the following induction principles and recursors.

$$\begin{aligned}
\mathbb{B}_{ind} &= Q(true) \rightarrow Q(false) \rightarrow \forall x : \mathbb{B}.Q(x) \\
R_{\mathbb{B}} : A \rightarrow A \rightarrow \mathbb{B} \rightarrow A \\
R_{\mathbb{B}} M N true &\rightsquigarrow M \\
R_{\mathbb{B}} M N false &\rightsquigarrow N \\
\mathbb{N}_{ind} &= Q(0) \rightarrow (\forall t : \mathbb{N}.Q(t) \rightarrow Q(S t)) \rightarrow \forall x : \mathbb{N}.Q(x) \\
R_{\mathbb{N}} : A \rightarrow (\mathbb{N} \rightarrow A \rightarrow A) \rightarrow \mathbb{N} \rightarrow A \\
R_{\mathbb{N}} M f O &\rightsquigarrow M \\
R_{\mathbb{N}} M f (S t) &\rightsquigarrow (f t (R_{\mathbb{N}} M f t)) \\
Pos_{ind} &= Q(one) \rightarrow (\forall m : \mathbb{B}. \forall t : Pos.Q(t) \rightarrow Q(next m t)) \rightarrow \forall x : Pos.Q(x) \\
R_{Pos} : A \rightarrow (\mathbb{B} \rightarrow Pos \rightarrow A \rightarrow A) \rightarrow Pos \rightarrow A \\
R_{Pos} M f one &\rightsquigarrow M \\
R_{Pos} M f (next m t) &\rightsquigarrow (f m t (R_{Pos} M f t))
\end{aligned}$$

6.5 Realizability of the induction principle

Once we have inductive types and their induction principle we want to show that the recursor R_X realizes X_{ind} , that is that R_X has type $\llbracket X_{ind} \rrbracket$ and is in relation \mathcal{R} with X_{ind} .

Theorem 2 $R_X : \llbracket X_{ind} \rrbracket$

Proof. For each predicate $Q(t)$ there is a type α in system \mathbb{T} such that $\llbracket Q(t) \rrbracket = \alpha$, moreover by Lemma 1 α does not depend on the term t . By definition of $\llbracket \cdot \rrbracket$ and X_{ind} we have

$$\begin{aligned}
\llbracket X_{ind} \rrbracket &= \overrightarrow{\llbracket \Delta\{C(X), c\} \rrbracket} \rightarrow \forall t : X.Q(t) = \overrightarrow{\llbracket \Delta\{C(X), c\} \rrbracket} \rightarrow \llbracket \forall t : X.Q(t) \rrbracket = \\
&= \overrightarrow{\llbracket \Delta\{C(X), c\} \rrbracket} \rightarrow X \rightarrow \llbracket Q(t) \rrbracket = \overrightarrow{\llbracket \Delta\{C(X), c\} \rrbracket} \rightarrow X \rightarrow \alpha
\end{aligned}$$

The type of R_X is $\overrightarrow{\square\{C(X)\}} \rightarrow X \rightarrow \alpha$, so we are left to prove that, for any constructor c , $\llbracket \Delta\{C(X), c\} \rrbracket = \square\{C(X)\}$; this is done by induction on the type $C(X)$. We have three cases:

$C(X) = X$. We have to prove that $\llbracket Q(c) \rrbracket = \alpha$ that follows by assumption.

$C(X) = T \rightarrow C(X)$.

$$\begin{aligned}
\llbracket \Delta\{T \rightarrow C(X), c\} \rrbracket &= \\
&= \llbracket \forall m : T. \Delta\{C(X), c m\} \rrbracket = T \rightarrow \llbracket \Delta\{C(X), c m\} \rrbracket \stackrel{*}{=} T \rightarrow \square\{C(X)\} = \\
&= \square\{T \rightarrow C(X)\}
\end{aligned}$$

$C(X) = X \rightarrow C(X)$.

$$\begin{aligned}
\llbracket \Delta\{X \rightarrow C(X), c\} \rrbracket &= \\
&= \llbracket \forall t : X. Q(t) \rightarrow \Delta\{C(X), c t\} \rrbracket = X \rightarrow \llbracket Q(t) \rightarrow \Delta\{C(X), c m\} \rrbracket = \\
&= X \rightarrow \alpha \rightarrow \llbracket \Delta\{C(X), c m\} \rrbracket \stackrel{*}{=} X \rightarrow \alpha \rightarrow \square\{C(X)\} = \\
&= \square\{X \rightarrow C(X)\}
\end{aligned}$$

We marked with * steps justified by the inductive hypothesis.

Theorem 3 $R_X \mathcal{R} X_{ind}$

Proof. To prove that $R_X \mathcal{R} X_{ind}$ we must prove that for any f_i such that $f_i \mathcal{R} \Delta\{C(X)_i, c_i\}$, and each $t : X$, $R_X \vec{f} t \mathcal{R} Q(t)$.

We proceed by induction on the structure of t .

Suppose $t = c_i \vec{m}$ for some constructor c_i , so that $R_X \vec{f} t = R_X \vec{f} (c_i \vec{m}) \rightsquigarrow \nabla\{C(X)_i, f_i, \vec{m}\}$. We now prove by induction on the structure of $C(X)_i$ that if $a \mathcal{R} \Delta\{C(X)_i, b\}$ then $\nabla\{C(X)_i, a, \vec{m}\} \mathcal{R} Q(b \vec{m})$.

$C(X)_i = X$. In this case, \vec{m} is empty, $\nabla\{C(X)_i, a, \vec{m}\} = a$ that realizes $\Delta\{C(X)_i, b\} = Q(b)$.

$C(X)_i = T \rightarrow C(X)$. In this case we have

$$\begin{aligned} \nabla\{T \rightarrow C(X), a, m_1 \vec{m}\} &= \nabla\{C(X), a m_1, \vec{m}\} \\ \Delta\{T \rightarrow C(X), b\} &= \forall t : T. \Delta\{C(X), b t\} \end{aligned}$$

By hypothesis we have $a \mathcal{R} \forall t : T. \Delta\{C(X), b t\}$, so that, $a m_1 \mathcal{R} \Delta\{C(X)_i, b m_1\}$, by definition of realizability. Hence, by inductive hypothesis, $\nabla\{C(X)_i, a m_1, \vec{m}\} \mathcal{R} Q(b m_1 \vec{m})$.

$C(X)_i = X \rightarrow C(X)$. In this case $\nabla\{X \rightarrow C(X), a, m_1 \vec{m}\}$ is equal to $\nabla\{C(X), a m_1 (R_X \vec{f} m_1), \vec{m}\}$ by definition, and that $\Delta\{X \rightarrow C(X), b\}$ is equal to $\forall x : X. Q(x) \rightarrow \Delta\{C(X), b x\}$.

By hypothesis $a \mathcal{R} \forall x : X. Q(x) \rightarrow \Delta\{C(X), b x\}$, and by the outer inductive hypothesis $R_X \vec{f} m_1 \mathcal{R} Q(m_1)$. So, $a m_1 (R_X \vec{f} m_1) \mathcal{R} \Delta\{C(X)_i, b m_1\}$ and by the (inner) inductive hypothesis,

$$\nabla\{C(X)_i, a m_1 (R_X \vec{f} m_1), \vec{m}\} \mathcal{R} Q(b m_1 \vec{m}).$$

7 Strong normalization of extended system T

Strong normalization for system T is a well known result[6] that can be easily extended to System T with this kind of inductive types. The first thing we have to do is to extend the definition of neutral term to the terms not of the form $\langle u, v \rangle, \lambda x. u, c_i \vec{m}$.

In conformity with the proof in [6], we call $\nu(t)$ the length of the longest reduction path from t and $\ell(t)$ the number of symbols in the normal form of t .

For an inductive type $Ind(X)\{c_1 : C(X); \dots; c_n : C(X)\}$ we have to prove that for each i , given a proper sequence of reducible arguments \vec{m} and \vec{f} , $(c_i \vec{m})$ and $R_X \vec{f} (c_i \vec{m})$ are reducible.

First the simple case of constructors. If the constructor c_i takes no arguments then it is already in normal form. If it takes m_1, \dots, m_n reducible arguments, then $\nu(c_i \vec{m}) = \max\{\nu(m_1), \dots, \nu(m_n)\}$ and so $c_i \vec{m}$ is strongly normalizable thus reducible for the definition of reducibility for base types.

To show that $R_X \vec{f} (c_i \vec{m})$ is reducible we can use **(CR 3)** from [6] that states that if t is neutral and every t' obtained by executing one redex of t is reducible, then t is reducible.

Now we have to show that each term that can be obtained by a reduction step is reducible. We can proceed by induction on $\Sigma\nu(f_i) + \nu(c_i \vec{m}) + \ell(c_i \vec{m})$ since we know by hypothesis that \vec{f} and $(c_i \vec{m})$ are reducible and consequently strongly normalizing.

The base case is when c_i takes no arguments and \vec{f} are normal. In this case the only redex we can compute is $R_X \vec{f} c_i \rightsquigarrow f_i$ that is reducible by hypothesis. The interesting inductive case is when \vec{m} and \vec{f} are normal, so the only reduction step we can execute is $R_X \vec{f} (c_i \vec{m}) \rightsquigarrow f_i \vec{m} (R_X \vec{f} n)$ where \vec{n} are the recursive arguments of c_i (here we wrote the recursive calls as the last parameters of f_i just to lighten notation). Since $\ell(n_j)$ is less than $\ell(c_i \vec{m})$ for every j we can apply the inductive hypothesis and state that $R_X \vec{f} n_j$ is reducible. Then by definition of reducibility of the arrow types and by the hypothesis that f_i and \vec{m} are reducible, we obtain that $f_i \vec{m} (R_X \vec{f} n)$ is reducible. All other cases, when we execute a redex in \vec{m} or \vec{f} , are straightforward applications of the induction hypothesis.

8 Consistent extensions

Realizability provides first of all a technique to prove consistency of logical systems (see Corollary 3), and to investigate consistent extensions (or conversely to conclude underivability of certain formulae from the fact that they are not realizable). For instance (see [20]), Markov's principle

$$(M_{PR}) \quad \neg\neg\exists x.A \rightarrow \exists x.A$$

for A (primitive) recursive is not validated by modified realizability, while on the other hand it validates the independence of premiss schemata

$$(IP) \quad (\neg A \rightarrow \exists x.B) \rightarrow \exists x.(\neg A \rightarrow B)$$

By the results in the previous sections (we also need termination), we have a very simple argument for proving that first order logic extended by inductive types and the corresponding induction principles is consistent. We may also add any other realizable formula without losing consistency, in particular the axioms asserting that all constructors are *injective* and *different* from each other:

$$\begin{aligned} (inj) \quad & \forall \vec{x}, \vec{y}. c \vec{x} = c \vec{y} \rightarrow x_i = y_i \\ (discr) \quad & \forall \vec{x}, \vec{y}. c_i \vec{x} \neq c_j \vec{y} \quad (\text{for all } c_i \neq c_j) \end{aligned}$$

These are the axioms required to guarantee that every *term* model build on constructors of inductive types is isomorphic to the free model³ (of course, this does not exclude the existence of different *non standard* models). Similarly, all

³ See e.g. [3] for an interesting discussion of these axioms in the context of inductionless induction.

valid equations of System T with arbitrary inductive types are trivially realized, and thus consistent with the logical system. From this point of view, the most natural generalization consists in extending quantifiers to all finite types, including all terms and all equations of the generalized System T. Systems of these kind are usually identified by adding a ω superscript to the corresponding first order variant (HA^ω , PA^ω , and so on). It is worth to mention that while PA is really painful if used for proving arithmetical properties, PA^ω , although being a conservative extension of PA, is already much more usable. The possibility of extending the language with arbitrary datatypes such as lists, trees, and so on increases in a sensible way the flexibility and actual usability of the language (especially if used to prove properties of programs).

Introducing functional terms in the logical signatures, one could also consider the possibility of weakening some restrictions on induction types, and notably the requirement for constructors to have arguments of atomic types (obviously imposing suitable positivity requirements for the recursive case).

A well known example (see e.g. [16, 22]) is the type of ordinals o .

$$\begin{aligned}
o &\equiv \text{Ind}(X)\{O : X; S : X \rightarrow X; \text{lim} : (\mathbb{N} \rightarrow X) \rightarrow X\} \\
o_{\text{ind}} &: Q(O) \rightarrow (\forall x : o.(Q(x) \rightarrow Q(Sx))) \rightarrow \\
&\quad (\forall f : \mathbb{N} \rightarrow o.(\forall n : \mathbb{N}.Q(fn)) \rightarrow Q(\text{lim } f)) \rightarrow \forall x : o.Q(x) \\
R_o &: \alpha \rightarrow (o \rightarrow \alpha \rightarrow \alpha) \rightarrow ((\mathbb{N} \rightarrow o) \rightarrow (\mathbb{N} \rightarrow \alpha) \rightarrow \alpha) \rightarrow o \rightarrow \alpha \\
R_o \text{ m g h } O &\rightsquigarrow m \\
R_o \text{ m g h } (S x) &\rightsquigarrow g x (R_o \text{ m g h } x) \\
R_o \text{ m g h } (\text{lim } f) &\rightsquigarrow h f (\lambda x : \mathbb{N}.R_o \text{ m g h } (f x))
\end{aligned}$$

Both the proofs of normalization and the proof of realizability require techniques that go beyond the simple structural recursion used so far (the problem is the recursive call on $(f x)$ in the reduction rule for $(\text{lim } f)$; see [22], section 3.4 for a detailed discussion). This is related to the fact that, on the contrary of the induction principles for inductive types considered so far, o_{ind} , which is a principle of transfinite induction, strictly extends the expressive power of PA.

9 Conclusions

In this paper we have given a modern presentation of Kreisel's *modified realizability*, generalizing it to arbitrary (first order) inductive types. In particular, we associate to each inductive type an elimination principle and a recursor, and prove that the former is realized by the latter (in the spirit of [10]).

Extending first order arithmetic with inductive types gives an elegant instrument for working with common (inductive) objects like lists or trees without the burden of coding them into natural numbers. However, the logical power of the system is not increased and the consistency of the system is still provable with simple techniques, providing a simple and powerful tool, particularly valuable from a didactical viewpoint.

The natural prosecution of the work is to consider extensions to the logical framework along the many directions of type theory (higher-order inductive

types, dependent types, polymorphism and so on) adding little by little small bits of logical power and suitably weighing the cost and the benefit of each extension.

References

1. A.Asperti, G.Longo. Categories, Types and Structures. Foundations of Computing, Cambridge University press, 1991.
2. J.Avigad, S.Feferman. Gödel's Functional ("Dialectica") Interpretation. Handbook of Proof Theory, S.R.Buss (ed). Elsevier, Amsterdam, 1998
3. H.Comon. Inductionless Induction. Handbook of Automated Reasoning, A.Robinson and A.Voronkov eds., pp. 913-962, MIT-Press & Elsevier, 2001.
4. J.Y.Girard. The system F of variable types, fifteen years later. Theoretical Computer Science 45, 1986.
5. G.Y.Girard. Proof Theory and Logical Complexity. Bibliopolis, Napoli, 1987.
6. G.Y.Girard, Y.Lafont, P.Taylor. Proofs and Types. Cambridge Tracts in Theoretical Computer Science 7.Cambridge University Press, 1989.
7. K.Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. Dialectica, 12, pp.34-38, 1958.
8. K.Gödel. Collected Works. Vol.II, Oxford University Press, 1990.
9. J.R.Hindley, J. P. Seldin. Introduction to Combinators and Lambda-calculus, Cambridge University Press, 1986.
10. W.A.Howard. Functional interpretation of bar induction by bar recursion. Compositio Mathematica 20, pp.107-124. 1968.
11. W.A.Howard. The formulae-as-types notion of constructions. in J.P.Seldin and J.R.Hindley editors, to H.B.Curry: Essays on Combinatory Logic, Lambda calculus and Formalism. Academic Press, 1980.
12. S.C.Kleene. On the interpretation of intuitionistic number theory. Journal of Symbolic Logic, n.10, pp.109-124, 1945.
13. G.Kreisel. Interpretation of analysis by means of constructive functionals of finite type. In. A.Heyting ed. *Constructivity in mathematics*. North Holland, Amsterdam, 1959.
14. G.Kreisel. On weak completeness of intuitionistic predicate logic. Journal of Symbolic Logic 27, pp. 139-158. 1962.
15. P.Letouzey. Programmation fonctionnelle certifiée; l'extraction de programmes dans l'assistant Coq. Ph.D. Thesis, Université de Paris XI-Orsay, 2004.
16. P.Martin-Löf. Intuitionistic Type Theory. Bibliopolis, Napoli, 1984.
17. C.Paulin-Mohring. Extraction de programme dans le Calcul de Constructions. Ph.D. Thesis, Université de Paris 7, 1987.
18. C.Paulin-Mohring. Extracting F_ω programs from proofs in the Calculus of Constructions. In proc. of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, January, ACM Press 1989.
19. K.Schütte. Proof Theory. Grundlehren der mathematischen Wissenschaften 225, Springer Verlag, Berlin, 1977.
20. A.S.Troelstra. Metamathematical Investigation of Intuitionistic Arithmetic and Analysis. Lecture Notes in Mathematics 344, Springer Verlag, Berlin, 1973.
21. A.S.Troelstra, H.Schwichtenberg. Basic Proof Theory. Cambridge Tracts in Theoretical Computer Science 43.Cambridge University Press, 1996.
22. B.Werner. Une Théorie des Constructions Inductives. Ph.D.Thesis, Université de Paris 7, 1994.