

A Semantic Normalization Proof for System T

Lisa Allali¹ and Paul Brauner²

¹ École Polytechnique, INRIA & Région Ile de France allali@lix.polytechnique.fr

² INPL & LORIA paul.brauner@loria.fr

Abstract. Semantics methods have been used to prove cut elimination theorems for a long time. It is only recently that they have been extended to prove strong normalization results. For instance using the notion of super-consistency that is a semantic criterion for theories expressed in deduction modulo implying strong normalization. However, the strong normalization of System T has always been reluctant to such semantic methods. In this paper we give a semantic normalization proof of system T using the super consistency of some theory.

1 Introduction

When building a new theory, there are several criteria one wants this theory to meet. In particular the *cut elimination* property which means that the cut rule is redundant. This property ensures the consistency of the theory. In an intuitionistic framework (as it is the case in the present paper) it also gives the so-called *witness property*: if a proof is ending by the introduction of the existential quantifier one can exhibit a witness of this existence. *Normalization* of a theory is also a desirable property. It ensures the termination of the proof reduction process, gives a way to construct witnesses, and eases the potential implementation of the theory. In a system where all the cuts are captured by the reduction rules (like in the typed lambda calculus), cut elimination is a consequence of normalization.

Gentzen was the first to prove, with a syntactic method, the theorem of cut elimination in sequent calculus (Hauptsatz) [1]. More recently semantics approaches have been introduced (for instance in [2,3,4,5]) for building proof with respect to some notion of model that has been used by De Marco and Lipton [6] to prove cut elimination of the Intuitionistic Higher-Order Logic, and by Okada [7] for intuitionistic Linear Logic (first and higher-order). The bright side of semantic proofs is that they abstract from syntactic details intrinsic to the type system and concentrate on essential arguments. Therefore, they obediently adapt to language extensions and allow to characterize classes of theories that meet cut elimination property.

More recently, a link between such semantic methods and *normalization* results has been done in the frame of *deduction modulo* [8]. For every theory expressed as a rewrite system, this semantic criterion, called super-consistency, implies the normalization property of natural deduction considered modulo this theory.

A theory is said to be *classically consistent* if it has a model where propositions are valued in the algebra $\{0,1\}$. This notion extends to intuitionistic logic where a theory is said to be *intuitionistically consistent* if there exists a Heyting Algebra \mathcal{B} such

that the theory has a model where propositions are valued in \mathcal{B} . The algebra \mathcal{B} can alternatively be a complete ordered Pseudo Heyting Algebra (PHA) that we will define in the semantic tool section. If a theory bears a model not only for some but for *any* complete ordered PHA, this theory is said to be *super-consistent*. The main idea of super-consistency is that the algebras of reducibility candidates is such a complete ordered PHA. Thus, super-consistency abstracts over the notion of reducibility candidate. The abstraction done by super-consistency quantifying on any such algebra allows to take distance with reducibility candidates and permits to get rid of the details of the strong normalization proof.

This has led for instance to the semantic proofs that Heyting's arithmetic (*i.e.* Peano axioms considered in an intuitionistic setting) is not only consistent, but also verifies the strong normalization property [9,10]. Although systems with recursors enjoy *syntactic* proofs of strong normalization (*cf* Tait's proof for Gödel System T [11]), they have been reluctant to this super-consistency method so far.

Recursors are used to compute with inductive types on which today's proof-assistants heavily rely to express most theories. This is especially the case of the COQ proof assistant [12,13,14]. Also many recent programming languages with support for dependent types such as AGDA[15] include such recursors. Recursors allow the user to easily describe recursive functions and properties while ensuring the strong normalization of the system, which is proved (syntactically) once for all in the metatheory.

In this paper we propose to bridge the gap between deduction modulo and systems with recursors by exhibiting a very simple intermediate system. This system relies on *Fold-unfold* rules [16] which is a natural way to integrate a theory into natural deduction in the form of *inferences rules* instead of *axioms*. Fortunately, the terms of System T can be translated (with no complexity speed-up) by some terms of the Fold-Unfold calculus preserving the reduction relation. It is therefore sufficient to prove normalization of Fold-Unfold for this particular theory to obtain that of System T. That's where semantic methods come into play.

Indeed, a previous paper [17] has shown that deduction modulo and Fold-Unfold are equivalently normalizing for the same theory. In other words: the way a theory is injected into a deduction system is immaterial for the normalization property. We shall then use super-consistency arguments to ensure normalization of the Fold-Unfold system simulating System T reductions.

The semantics arguments being clearly identified, we then generalize the whole reasoning to a system with recursors *à la* System T for a certain family of mutually inductive types. This confirms the validity of our approach and provides the foundations of a possible implementation (this system has actually already been played with in *lemuridae*, a toy proof assistant for deduction modulo). System T being rather a programming language than a deduction system, we finally discuss the extension of the method to predicate logic. Simulation through Fold-Unfold is still valid. On the semantic side, the System T being in some way a impoverished version of arithmetic, its super-consistency [9] fits our needs. We finally show how our method adapts to deduction system with rewriting on first-order terms.

2 Framework

2.1 Deduction Systems

Natural Deduction. Our starting point is natural deduction for predicate logic. The proof-term language is given by the following grammar, whose constructs are respectively typed by the usual typing rules (Ax), ($\Rightarrow I$), ($\Rightarrow E$), ($\wedge I$), ($\wedge E_1$), ($\wedge E_2$), ($\vee I_1$), ($\vee I_2$), ($\vee E$), ($\perp E$), ($\forall I$), ($\forall E$), ($\exists I$) and ($\exists E$) (see [18] for instance).

$$\pi ::= \alpha \mid \lambda\alpha.\pi \mid (\pi \pi') \mid \langle \pi, \pi' \rangle \mid \text{fst}(\pi) \mid \text{snd}(\pi) \mid i(\pi) \mid j(\pi) \mid (\delta \pi_1 \alpha.\pi_2 \beta.\pi_3) \mid I \mid (\delta_{\perp} \pi) \mid \lambda x.\pi \mid (\pi t) \mid \langle t, \pi \rangle \mid (\delta_{\exists} \pi x.\alpha.\pi')$$

The variables x, y, \dots are variables of the first-order theory while α, β, \dots are proof variables. Notice that the variables α, β and x are bound in $\lambda\alpha.\pi$, $\lambda x.\pi$, $(\delta \pi_1, \alpha.\pi_2, \beta.\pi_3)$ and $(\delta_{\exists} \pi, x.\alpha.\pi')$. As usual, the process of cut elimination is modeled by (generalized) β -reduction, whose rules are reminded below.

$$\begin{array}{ll} (\lambda\alpha.\pi_1 \pi_2) \triangleright \pi_1\{\alpha := \pi_2\} & \delta i(\pi_1) \alpha.\pi_2 \beta.\pi_3 \triangleright \pi_2\{\alpha := \pi_1\} \\ \text{fst}(\langle \pi_1, \pi_2 \rangle) \triangleright \pi_1 & \delta j(\pi_1) \alpha.\pi_2 \beta.\pi_3 \triangleright \pi_3\{\beta := \pi_1\} \\ \text{snd}(\langle \pi_1, \pi_2 \rangle) \triangleright \pi_2 & \delta_{\exists} \langle t, \pi_1 \rangle x.\alpha.\pi_2 \triangleright \pi_2\{x := t, \alpha := \pi_1\} \\ (\lambda x.\pi t) \triangleright \pi\{x := t\} & \end{array}$$

Deduction Modulo (DM). Deduction Modulo is a formalism that aims at distinguishing reasoning from computation in proofs. Modern type systems feature a so-called *conversion rule* which allows to identify propositions which are equal modulo beta-conversion.

$$(\text{CONV}) \frac{\Gamma \vdash t : T \quad T \equiv_{\beta} T'}{\Gamma \vdash t : T'}$$

A side-effect of this rule is to allow *computation* inside the proof, the computation being performed by proof reduction. The idea of deduction modulo [19] is to study the phenomenon of computation inside a proof in a simpler framework: predicate logic, where propositions are identified by a congruence. The congruence depends on the theory. It is usually defined as the symmetric and transitive closure of a rewrite system over first-order terms and propositions. Therefore, the typing rules of deduction modulo are those of natural deduction, modulo the congruence. This may be explicit by a rephrasing of the inference rules, as shown for the implication elimination below.

$$(\Rightarrow E) \frac{\Gamma \vdash C \quad \Gamma \vdash A}{\Gamma \vdash B} C \equiv A \Rightarrow B$$

The other rules of deduction modulo are build in the same way upon natural deduction (see figure 1 in appendix for the full system). In this paper, we restrict ourselves to theories expressed by proposition rewrite systems defined as follows. We call them *computational theories*.

Definition 1 (Proposition rewrite system). We call proposition rewrite rule every rule $R : P \rightarrow \varphi$ rewriting atomic propositions P into propositions φ build upon the language of predicate logic. Moreover, we suppose that $\mathcal{FV}(\varphi) \subseteq \mathcal{FV}(P)$. We define a proposition rewrite system as an orthogonal, hence confluent set of proposition rewrite rules.

We shall call $DM_{\mathcal{R}}$ the deduction modulo system parametrized by the rewrite system \mathcal{R} . The proof-terms are left unchanged w.r.t. natural deduction, only the types are identified.

Example 1 (Equality on naturals). Let us consider the rewrite system \mathcal{R} formed by the proposition rewrite rule $R_{\text{eq}} : S(x) = S(y) \rightarrow x = y$. Then the type $(100 = 100) \Rightarrow (0 = 0)$ has $\lambda\alpha.\alpha$ as a proof in $DM_{\mathcal{R}}$ with only one step of reasoning but 100 steps of rewriting that are transparent in the proof:

$$\begin{array}{c} (Ax) \frac{}{\alpha : 100 = 100 \vdash \alpha : 0 = 0} (100 = 100) \equiv (0 = 0) \\ (\Rightarrow I) \frac{}{\vdash \lambda\alpha.\alpha : (100 = 100) \Rightarrow (0 = 0)} \end{array}$$

Cut Elimination and Deduction Modulo. In the frame of Natural Deduction, a cut is defined as an introduction followed by an elimination of the same connector. A cut-free proof always ends by an introduction rule, which ensures the disjunction property (resp. the witness property). Namely, if one can prove $A \vee B$ (resp. $\exists x P(x)$), the last inference rule of a cut-free proof being necessarily $(\vee I_1)$ or $(\vee I_2)$ (resp. $(\exists I)$), one can pick a proof of either A or B (resp. a witness of the existence of such an x) at the head of the last proof step.

Those nice features are lost as soon as axioms are used to define a theory. Indeed the property of the last rule being an introduction one disappears, so do the disjunction and witness properties. In order to recover it, the notion of cut has to be extended: as an example, the typical way of getting round this problem in arithmetic is to introduce *induction cuts* whenever the induction axiom scheme is used. A cut-free proof in arithmetic is then not only free of cuts w.r.t. their intro-elim characterization, but also w.r.t. the additional *ad-hoc* definition.

A major interest of Deduction Modulo is that when a theory can be entirely described by a set of rewrite rules (as it had been done for arithmetic [10,9], set theory [20] and Higher Order Logic [21] for instance) the notion of cut doesn't need any specific extension: it remains the same as for Natural Deduction, and the disjunction and witness properties are still ensured.

A drawback of Deduction Modulo is that proof reduction may not terminate anymore, depending on the theory defined by \mathcal{R} . This is the case for very simple (and even consistent) theories like the one defined by $P \rightarrow (P \Rightarrow P)$, which allows to type $(\lambda x.(x x)) (\lambda x.(x x))$ [8]. On the other hand, the strong normalization of $DM_{\mathcal{R}}$ implies the consistency of \mathcal{R} . Therefore, finding criteria which entails the strong normalization of deduction modulo is an active research topic, which has lead to elegant proofs of normalization for arithmetic and set theory [10,9,20].

Natural Deduction with Folding and Unfolding Rules (FU). Consider a propositional rewrite system \mathcal{R} . For each rewrite rule $R : P \rightarrow \varphi$, we may add Prawitz' *unfolding* and *folding* rules [22] respectively replacing P by its definition φ and φ by its name P . To do so, we enrich the proof-terms language of natural deduction with two constructs:

$$\mathcal{T} = \dots \mid R \pi \mid \mathfrak{R} \pi$$

which are typed by the following inference rules.

$$\text{(UNFOLD}_R) \frac{\Gamma \vdash \pi : P}{\Gamma \vdash R \pi : \varphi} \quad \text{(FOLD}_R) \frac{\Gamma \vdash \pi : \varphi}{\Gamma \vdash \mathfrak{R} \pi : P}$$

It is possible in this system to perform some cut on the proposition P . We therefore introduce the following reduction rule which eliminates these cuts.

$$R (\mathfrak{R} \pi) \triangleright \pi$$

Let us remark that, as in deduction modulo, some proof-terms of this system may not terminate, even for consistent theories. This is the case for the theory defined by the rewrite rule $R : P \rightarrow (P \Rightarrow P)$ where the following looping term has type P .

$$\begin{aligned} & \lambda \alpha. (R \alpha \alpha) (\mathfrak{R} \lambda \alpha. (R \alpha \alpha)) \\ & \triangleright R (\mathfrak{R} \lambda \alpha. (R \alpha \alpha)) (\mathfrak{R} \lambda \alpha. (R \alpha \alpha)) \\ & \triangleright \lambda \alpha. (R \alpha \alpha) (\mathfrak{R} \lambda \alpha. (R \alpha \alpha)) \end{aligned}$$

Fold-Unfold Modulo. We finally define *Fold-Unfold modulo*, which combines both deduction systems.

Definition 2 (Fold-Unfold modulo). Let \mathcal{R}_1 and \mathcal{R}_2 be two rewrite systems composed of proposition rewrite rules. We call Fold-Unfold \mathcal{R}_1 modulo \mathcal{R}_2 ($FUM_{\mathcal{R}_2}^{\mathcal{R}_1}$) the deduction system formed by $FU^{\mathcal{R}_1}$ where the propositions are considered modulo \mathcal{R}_2 after the addition of the folding-unfolding rules.

Notice that we fall back in the case of deduction modulo (*resp.* Fold-Unfold) when \mathcal{R}_1 (*resp.* \mathcal{R}_2) is empty. Let us now study the metaproperties of the system.

Proposition 1 (FUM soundness and completeness). Let \mathcal{R}_1 and \mathcal{R}_2 be two proposition rewrite systems. $FUM_{\mathcal{R}_2}^{\mathcal{R}_1}$ is sound and complete with respect to natural deduction within the theory formed by axioms of the form $\forall \vec{x} (P(\vec{x}) \Leftrightarrow \varphi(\vec{x}))$ for each proposition rewrite rule $P \rightarrow \varphi$ present in $\mathcal{R}_1 \cup \mathcal{R}_2$.

Proof. By combining the soundness and completeness results for deduction modulo and Fold-Unfold respectively stated in [19] and [17]. \square

Several criteria have been studied that ensure deduction modulo or Fold-Unfold strong normalization. Transferring them from one system to another has been extensively described in [17]. We adapt here one of the results that suits our needs.

Theorem 1 (Strong normalization property transfer). Let \mathcal{R}_1 and \mathcal{R}_2 be two proposition rewrite systems. Strong normalization of $DM_{\mathcal{R}_1 \cup \mathcal{R}_2}$ implies that of $FUM_{\mathcal{R}_1}^{\mathcal{R}_2}$.

Proof. The idea is to erase every instance of \mathcal{R} and \mathcal{R} in the proof terms. Each reduction step in Fold-Unfold is either a fold-unfold reduction, either a β one. The fold-unfold reductions are in finite number since they make the size of the proof decrease, and the β -reductions are simulated by the translation. The translation is well-typed thanks to the congruence of deduction modulo. See [17] for more details. \square

2.2 Semantic Tools

Let us see now a semantic criterion over computational theories that implies strong normalization in deduction modulo and, thus, in the system with the Fold-Unfold rules for the same theory. It has been introduced in [23] and used in [9] to prove strong normalization of a computational presentation of Heyting's arithmetic.

As outlined in the introduction, the basic idea of this method is to build a model of a theory in an algebra equipped with constraints met by the algebra of reducibility candidates. A trivial algebra is a set only featuring operations $\Rightarrow, \tilde{\wedge}, \dots$ for interpreting each logical connector. A theory bearing a model in every such algebras is then strongly normalizing because it bears in particular a model in the algebra of candidates. Such theories are yet very poor. In order to apply this method to more theories, one may enrich the algebra with constraints.

First we define the Pseudo Heyting algebra that is the base of our models.

Definition 3 (Pseudo Heyting algebra (PHA)). *Let \mathcal{B} be a set and \leq a relation on \mathcal{B} . A structure $\langle \mathcal{B}, \leq, \tilde{\wedge}, \tilde{\vee}, \perp, \top, \tilde{\forall}, \tilde{\exists}, \Rightarrow \rangle$ is a Pseudo Heyting algebra if for all x, y, z, c in \mathcal{B} and a in $\wp(\mathcal{B})$,*

- \leq is a preorder on \mathcal{B} ,
- \perp is a minimum of \mathcal{B} for \leq ,
- \top is a maximum of \mathcal{B} for \leq ,
- $x \tilde{\wedge} y$ is a lower bound of x and y ,
- $x \tilde{\vee} y$ is an upper bound of x and y ,
- $\tilde{\forall}$ and $\tilde{\exists}$ (infinite lower and upper bounds) are functions from $\wp(\mathcal{B})$ to \mathcal{B} such that:
 - $x \in a \Rightarrow \tilde{\forall} a \leq x$,
 - $(\forall x \in a \ c \leq x) \Rightarrow c \leq \tilde{\forall} a$,
 - $x \in a \Rightarrow x \leq \tilde{\exists} a$,
 - $(\forall x \in a \ x \leq c) \Rightarrow \tilde{\exists} a \leq c$.
- $x \leq y \Rightarrow z \Leftrightarrow x \tilde{\wedge} y \leq z$.

At this point, one could wonder why this preorder \leq which distinguishes between PHAs and Heyting Algebras is necessary. The answer has to be found in the differentiation made possible by this preorder between *equivalent* propositions and *congruent* ones. Indeed if $A \Leftrightarrow B$ in the theory, $\llbracket A \rrbracket \leq \llbracket B \rrbracket$ and $\llbracket B \rrbracket \leq \llbracket A \rrbracket$, but $\llbracket A \rrbracket$ and $\llbracket B \rrbracket$ are not necessarily equal in the PHA. On the other hand, if $A \equiv B$ (where the congruence is defined by the rewrite system) implies that $\llbracket A \rrbracket$ equals $\llbracket B \rrbracket$ in the algebra.

It is noteworthy that the algebra of reducible candidate is a PHA but not a Heyting Algebra. Indeed, $(\tilde{\top} \Rightarrow \tilde{\top}) = \tilde{\top}$ in Heyting algebras, which is not necessarily the case in the algebra of reducibility candidates. Thanks to the preorder of PHAs, we have

$(\tilde{\top} \Rightarrow \tilde{\top}) \leq \tilde{\top}$ and $\tilde{\top} \leq (\tilde{\top} \Rightarrow \tilde{\top})$ which is the case in the algebra of reducibility candidates .

Yet this definition is very general, in particular this structure doesn't respect the prerequisite to the construction of fixed point one may need while building the interpretation of a recursive theory. That's why we finally add an order relation (\sqsubseteq) to our algebra.

Definition 4 (Ordered PHA). An ordered pseudo Heyting algebra is a pseudo Heyting algebra together with a relation \sqsubseteq on \mathcal{B} such that

- \sqsubseteq is an order relation,
- $\tilde{\top} \leq b$ and $b \sqsubseteq b'$ then $\tilde{\top} \leq b'$,
- $\tilde{\top}$ is a maximal element for \sqsubseteq and $\tilde{\perp}$ is a minimal element for \sqsubseteq ,
- $\tilde{\wedge}, \tilde{\vee}, \tilde{\exists}, \tilde{\exists}$ are monotonous, \Rightarrow is left anti-monotonous and right monotonous.

Definition 5 (Complete ordered PHA). An ordered pseudo Heyting algebra is said to be complete if every subset of \mathcal{B} has a greatest lower bound for \sqsubseteq .

Complete ordered PHAs may be used to interpret propositions in a class of models of Deduction Modulo that we now introduce.

Definition 6 (Intuitionistic model). Let \mathcal{L} be the language of a first-order theory. An intuitionistic model \mathcal{M} of \mathcal{L} is :

- a set M ,
- a complete ordered pseudo Heyting algebra \mathcal{B} ,
- for each function symbol f of arity n a function \hat{f} from M^n to M ,
- for each predicate symbol P of arity n a function \hat{P} from M^n to \mathcal{B} .

Definition 7 (Denotation). Let \mathcal{M} be a model, A be a proposition and ϕ be an assignment. We define $\llbracket A \rrbracket_\phi$ as follows.

$$\begin{array}{ll} \llbracket x \rrbracket_\phi = \phi(x) & \llbracket A \wedge B \rrbracket_\phi = \llbracket A \rrbracket_\phi \tilde{\wedge} \llbracket B \rrbracket_\phi \\ \llbracket \perp \rrbracket_\phi = \tilde{\perp} & \llbracket A \vee B \rrbracket_\phi = \llbracket A \rrbracket_\phi \tilde{\vee} \llbracket B \rrbracket_\phi \\ \llbracket \top \rrbracket_\phi = \tilde{\top} & \llbracket A \Rightarrow B \rrbracket_\phi = \llbracket A \rrbracket_\phi \Rightarrow \llbracket B \rrbracket_\phi \\ \llbracket f(t_1, \dots, t_n) \rrbracket_\phi = \hat{f}(\llbracket t_1 \rrbracket_\phi, \dots, \llbracket t_n \rrbracket_\phi) & \llbracket \forall x A \rrbracket_\phi = \tilde{\forall} \{ \llbracket A \rrbracket_{\phi, x:=v} \mid v \in M \} \\ \llbracket P(t_1, \dots, t_n) \rrbracket_\phi = \hat{P}(\llbracket t_1 \rrbracket_\phi, \dots, \llbracket t_n \rrbracket_\phi) & \llbracket \exists x A \rrbracket_\phi = \tilde{\exists} \{ \llbracket A \rrbracket_{\phi, x:=v} \mid v \in M \} \end{array}$$

Definition 8 (Models for computational theory). A model of a computational theory whose rewrite rules are $R_1 \rightarrow R'_1, \dots, R_n \rightarrow R'_n$ is such that for each assignment ϕ , $\llbracket R_i \rrbracket_\phi = \llbracket R'_i \rrbracket_\phi$ for $1 \leq i \leq n$.

Definition 9 (Super-consistency). A computational theory is super-consistent if, for each complete ordered pseudo Heyting algebra \mathcal{B} , there exists a \mathcal{B} -model of this theory.

Finally, the main property of a super-consistent theory is to bear a model valuated in the reducibility candidates algebra and thus to normalize [8].

Proposition 2 (Normalization). If a computational theory \mathcal{R} is super-consistent, then each proof in $DM_{\mathcal{R}}$ strongly normalizes.

As a consequence, by Theorem 1, if a computational theory $\mathcal{R}_1 \cup \mathcal{R}_2$ is super-consistent then $SNDM_{\mathcal{R}_1}^{\mathcal{R}_2}$ strongly normalizes.

3 Gödel System T

We now introduce the main theory of interest in this paper. We present in this section a theory $DM_{\epsilon, \text{nat}}$ and prove that its super-consistency implies the strong normalization of System T.

Definition 10. *The System T is the subset of natural deduction restricted to the \Rightarrow connective and enriched with:*

- the atomic type nat , two constants 0 and S as well as a constant rec^τ for each type τ formed with nat and \Rightarrow
- the following axioms (the third of them being a scheme)

$$\begin{aligned} 0 &: \text{nat} \\ S &: \text{nat} \Rightarrow \text{nat} \\ \text{rec}^\tau &: \text{nat} \Rightarrow \tau \Rightarrow (\text{nat} \Rightarrow \tau \Rightarrow \tau) \Rightarrow \tau \end{aligned}$$

- the additional reduction rules

$$\begin{aligned} \text{rec}^\tau 0 a f &\triangleright a \\ \text{rec}^\tau (S n) a f &\triangleright f n (\text{rec}^\tau n a f) \end{aligned}$$

Example 2 (Functions in System T).

$$\begin{aligned} + &= \lambda a : \text{nat}. \lambda b : \text{nat}. (\text{rec}^{\text{nat}} b a \lambda x : \text{nat}. \lambda y : \text{nat}. (S y)) \\ \times &= \lambda a : \text{nat}. \lambda b : \text{nat}. (\text{rec}^{\text{nat}} b 0 \lambda x : \text{nat}. \lambda y : \text{nat}. (+ y a)) \end{aligned}$$

The aim of this section is to provide an original semantic proof of System T normalization: we will first introduce a rewrite system defining a deduction modulo system $DM_{\epsilon, \text{nat}}$. Then we show the super-consistency of $DM_{\epsilon, \text{nat}}$ which implies its normalization. Finally we migrate from $DM_{\epsilon, \text{nat}}$ to a Fold-Unfold modulo deduction system $FUM_{\epsilon}^{\text{nat}}$ which faithfully mimics System T proofs and computational behavior. A final lemma reduces strong normalization for the System T to strong normalization for $FUM_{\epsilon}^{\text{nat}}$ and permits to conclude that System T is normalizing, as the process of migrating from deduction modulo to Fold-Unfold preserves the strong normalization property (Theorem 1).

3.1 The $DM_{\epsilon, \text{nat}}$ System

Definition 11 ($DM_{\epsilon, \text{nat}}$). *For any proposition P we can form with nat and \Rightarrow let \dot{P} be a first-order constant. Let ϵ be a unary predicate symbol. Let $\mathcal{R}_{\epsilon, \text{nat}}$ be the proposition rewrite system formed by the following rules.*

$$\begin{aligned} R_{\epsilon} &: \epsilon(\dot{P}) \rightarrow P \\ R_{\text{nat}} &: \text{nat} \rightarrow \forall p (\epsilon(p) \Rightarrow (\text{nat} \Rightarrow \epsilon(p) \Rightarrow \epsilon(p)) \Rightarrow \epsilon(p)) \end{aligned}$$

$DM_{\epsilon, \text{nat}}$ is the deduction modulo system for the language $\text{nat}, \epsilon, \dot{P}$ parametrized by the proposition rewrite system $\mathcal{R}_{\epsilon, \text{nat}}$.

Theorem 2. *The $DM_{\epsilon, \text{nat}}$ system is super-consistent.*

Proof. Let \mathcal{B} be any complete ordered Pseudo Heyting Algebra. We build a \mathcal{B} -model \mathcal{M} of $DM_{\epsilon, \text{nat}}$ as follows.

- The domain M of \mathcal{M} is \mathcal{B} . $\llbracket \epsilon \rrbracket = id$.
- We look for an interpretation of nat such that the following holds.

$$\llbracket \text{nat} \rrbracket = \llbracket \forall p (\epsilon(p) \Rightarrow (\text{nat} \Rightarrow \epsilon(p) \Rightarrow \epsilon(p)) \Rightarrow \epsilon(p)) \rrbracket$$

For any element f of \mathcal{B} , we build a model \mathcal{M}_f where nat is interpreted by f . Let Φ be a function from \mathcal{B} to \mathcal{B} mapping f to $\llbracket \forall p (\epsilon(p) \Rightarrow (\text{nat} \Rightarrow \epsilon(p) \Rightarrow \epsilon(p)) \Rightarrow \epsilon(p)) \rrbracket^{\mathcal{M}_f}$. \mathcal{B} is ordered and complete and Φ is monotone (because nat only appears in positive position in $\forall p (\epsilon(p) \Rightarrow (\text{nat} \Rightarrow \epsilon(p) \Rightarrow \epsilon(p)) \Rightarrow \epsilon(p))$). Thus Φ has a fixpoint F . We chose this F to interpret nat in \mathcal{M} .

- Finally we interpret each \hat{P} by its denotation in \mathcal{M} : $\hat{P} = \llbracket P \rrbracket^{\mathcal{M}}$.

By construction \mathcal{M} is a \mathcal{B} -model of $DM_{\epsilon, \text{nat}}$. Thus $DM_{\epsilon, \text{nat}}$ is super-consistent. □

Corollary 1. *$DM_{\epsilon, \text{nat}}$ is strongly normalizing.*

Proof. By Proposition 2. □

3.2 From $DM_{\epsilon, \text{nat}}$ to $FUM_{\epsilon}^{\text{nat}}$

Definition 12 ($FUM_{\epsilon}^{\text{nat}}$). *$FUM_{\epsilon}^{\text{nat}}$ is the Fold-Unfold modulo system with the same language as $DM_{\epsilon, \text{nat}}$ where the congruence is defined by the rewrite rule R_{ϵ} and where the rewrite rule R_{nat} is translated into its corresponding folding and unfolding rules:*

$$\begin{aligned} (\text{FOLD}_{R_{\text{nat}}}) \quad & \frac{\Gamma \vdash \pi : \forall p (\epsilon(p) \Rightarrow (\text{nat} \Rightarrow \epsilon(p) \Rightarrow \epsilon(p)) \Rightarrow \epsilon(p))}{\Gamma \vdash \mathfrak{R}_{\text{nat}} \pi : \text{nat}} \\ (\text{UNFOLD}_{R_{\text{nat}}}) \quad & \frac{\Gamma \vdash \pi : \text{nat}}{\Gamma \vdash \mathfrak{R}_{\text{nat}} \pi : \forall p (\epsilon(p) \Rightarrow (\text{nat} \Rightarrow \epsilon(p) \Rightarrow \epsilon(p)) \Rightarrow \epsilon(p))} \end{aligned}$$

The associated instance of the fold-unfold cut reduction rule is then the following.

$$R_{\text{nat}} (\mathfrak{R}_{\text{nat}} \pi) \triangleright \pi$$

Lemma 1. *$FUM_{\epsilon}^{\text{nat}}$ is normalizing.*

Proof. By Corollary 1 and Theorem 1. □

We now show how we can simulate the computational behaviour of System T in $FUM_{\epsilon}^{\text{nat}}$. The idea is that by choosing the right representatives for 0 and S, the two reduction rules of the recursor are simulated by the only fold-unfold reduction rule associated to R_{nat} . The constructors 0 and S are encoded by the following proof-terms in $FUM_{\epsilon}^{\text{nat}}$.

$$\begin{aligned} \nu_0 &= \mathfrak{R}_{\text{nat}} \lambda p. \lambda \alpha. \lambda \beta. \alpha && : \text{nat} \\ \nu_S &= \lambda n. (\mathfrak{R}_{\text{nat}} \lambda p. \lambda \alpha. \lambda \beta. (\beta n (\lambda m. (R_{\text{nat}} m p) n \alpha \beta))) && : \text{nat} \Rightarrow \text{nat} \end{aligned}$$

The reader may point out that ν_S is not in normal form. Indeed, the presence of the redex eases the simulation proof above, since the translation of rec^τ introduces some η -expansion due to the higher-order encoding. We recognize in ν_0 and ν_S the number 0 and the successor function defined on Parigot integers [24], a recursive (compared to iterative) version of Church integers.

Definition 13 (Translation from System T to $FUM_\epsilon^{\text{nat}}$).

$$\begin{array}{ll} |t u| = |t| |u| & |0| = \nu_0 \\ |\lambda x.t| = \lambda x.|t| & |S| = \nu_S \\ |\text{rec}^\tau| = \lambda\alpha.(\mathbf{R}_{\text{nat}} \alpha \dot{\tau}) & |x| = x \text{ if } x \text{ is a variable} \end{array}$$

Lemma 2 (Soundness). *For all $\pi : T$ in System T then $|\pi| : T$ in $SNDM_\epsilon^{\text{nat}}$.*

Proof. By induction on π . Remark that $\lambda\alpha.(\mathbf{R}_{\text{nat}} \alpha \dot{\tau})$ has type $\text{nat} \Rightarrow \epsilon(\dot{\tau}) \Rightarrow (\text{nat} \Rightarrow \epsilon(\dot{\tau}) \Rightarrow \epsilon(\dot{\tau})) \Rightarrow \epsilon(\dot{\tau})$, which is congruent to $\text{nat} \Rightarrow \tau \Rightarrow (\text{nat} \Rightarrow \tau \Rightarrow \tau) \Rightarrow \tau$. \square

Lemma 3 (Simulation). *Let π and π' be two proofs in System T such that $\pi \triangleright \pi'$, then $|\pi| \triangleright_\rho^+ |\pi'|$ in SND_{nat} .*

Proof. By induction on π . The non-trivial cases are the following.

– $\text{rec}^\tau 0 u v \triangleright u$

$$\begin{aligned} |\text{rec}^\tau 0 u v| &= \lambda\alpha.(\mathbf{R}_{\text{nat}} \alpha \dot{\tau}) \nu_0 |u| |v| \\ &\triangleright \mathbf{R}_{\text{nat}} \nu_0 \dot{\tau} |u| |v| \\ &= \mathbf{R}_{\text{nat}} (\mathcal{R}_{\text{nat}} \lambda p.\lambda\alpha.\lambda\beta.\alpha) \dot{\tau} |u| |v| \\ &\triangleright (\lambda p.\lambda\alpha.\lambda\beta.\alpha) \dot{\tau} |u| |v| \\ &\triangleright^3 |u| \end{aligned}$$

– $\text{rec}^\tau (S n) u v \triangleright v n (\text{rec}_{\text{nat}}^\tau n u v)$

$$\begin{aligned} |\text{rec}^\tau (S n) u v| &= \lambda\alpha.(\mathbf{R}_{\text{nat}} \alpha \dot{\tau}) (\nu_S |n|) |u| |v| \\ &\triangleright \mathbf{R}_{\text{nat}} (\nu_S |n|) \dot{\tau} |u| |v| \\ &\triangleright \mathbf{R}_{\text{nat}} (\mathcal{R}_{\text{nat}} \lambda p.\lambda\alpha.\lambda\beta.(\beta |n| (\lambda m.(\mathbf{R}_{\text{nat}} m p) |n| \alpha \beta))) \dot{\tau} |u| |v| \\ &\triangleright (\lambda p.\lambda\alpha.\lambda\beta.(\beta |n| (\lambda m.(\mathbf{R}_{\text{nat}} m p) |n| \alpha \beta))) \dot{\tau} |u| |v| \\ &\triangleright^3 |v| |n| (\lambda m.(\mathbf{R}_{\text{nat}} m \dot{\tau}) |n| |u| |v|) \\ &= |v n (\text{rec}^\tau n u v)| \end{aligned}$$

\square

Theorem 3 (Relative normalization). *The strong normalization of $SNDM_\epsilon^{\text{nat}}$ implies that of System T.*

Proof. Consider a reduction sequence $\pi \triangleright \pi_1 \triangleright \pi_2 \triangleright \dots$ in System T. By lemma 3, $|\pi| \triangleright_\rho^+ |\pi_1| \triangleright_\rho^+ |\pi_2| \triangleright_\rho^+ \dots$ is a reduction sequence in $SNDM_\epsilon^{\text{nat}}$ and thus is finite according to lemma 1. So is the one in System T. \square

4 Inductive types

We now generalize this result to a family of mutually inductive types. This opens the door to an implementation of a proof assistant with recursors but no primitive reduction rule for recursors.

Let us first define some notations.

Definition 14. *The arity of a formula φ is a sequence of \forall and \Rightarrow symbols defined by induction on φ as follows*

- if φ is atomic $\text{arity}(\varphi) = []$,
- if $\varphi = \varphi_1 \Rightarrow \varphi_2$ then $\text{arity}(\varphi) = (\Rightarrow, \text{arity}(\varphi_2))$,
- if $\varphi = \forall x \varphi_1$ then $\text{arity}(\varphi) = (\forall, \text{arity}(\varphi_1))$.

Let φ be a formula, a sequence for φ is a sequence of distinct variables such that the n -th variable of the sequence is a proof variable if the n -th element of the arity of φ is \Rightarrow and a term variable otherwise.

We note \vec{x} variable sequences. Moreover for any variable t , $(t \ [\]) = t$ and if $\vec{x} = (x, \vec{x}')$ then $(t \ \vec{x}) = ((t \ x) \ \vec{x}')$. If C has type ϕ and \vec{x} is a sequence for ϕ , we may say that \vec{x} is a sequence for C .

4.1 Simple types with recursors

Let \mathcal{RT} be a set of *type* symbols and \mathcal{C} a set of *constructor* symbols:

$$\begin{aligned} \mathcal{RT} &::= \text{nat}, \text{btree}, \text{list}, \dots \\ \mathcal{C} &::= 0, S, \text{Cons}, \dots \end{aligned}$$

Definition 15 (Constructor type for X). *The set (strictly positive) constructor types associated to X is defined by the following grammar.*

$$\begin{aligned} \mathcal{PT} &::= X \\ &| A \Rightarrow \mathcal{PT} \quad \text{where } X \text{ does not appear in } A \\ &| (\vec{A} \Rightarrow X) \Rightarrow \mathcal{PT} \quad \text{where } X \text{ does not appear in } \vec{A} \end{aligned}$$

Definition 16 (Signature). *A signature Σ is a set of pairs of typed constructors $(C_i : P_i) \in \mathcal{C} \times \mathcal{PT}$. We say that C_i is a constructor for **rt** if P_i is a constructor type for **rt**.*

Example 3 (Trees and Forests). In the following signature:

$$\begin{aligned} \text{Node} &: \text{frst} \Rightarrow \text{tree} \\ \text{Leaf} &: \text{frst} \\ \text{Cons} &: \text{tree} \Rightarrow \text{frst} \Rightarrow \text{frst} \end{aligned}$$

Node is a constructor for **tree**, Leaf and Cons are constructors for **frst**.

From here on let Σ be a signature. We call \mathcal{ST} (simple types) the set of propositions formed by \mathcal{RT} and \Rightarrow . We call ML (minimal propositional logic) the subset of natural deduction restricted to (Ax) , $(\Rightarrow I)$ and $(\Rightarrow E)$. In particular the proof-terms are restricted to variables, abstractions and applications.

For every $\mathbf{rt} \in \mathcal{RT}$, $\tau \in \mathcal{ST}$ and $\varphi \in \mathcal{PT}$ we define a proposition $\Delta_{\mathbf{rt}}^\tau(\varphi)$ by induction on the proof that φ is a constructor type for \mathbf{rt} as follows.

$$\begin{aligned}\Delta_{\mathbf{rt}}^\tau(\mathbf{rt}) &= \tau \\ \Delta_{\mathbf{rt}}^\tau(A \Rightarrow B) &= A \Rightarrow \Delta_{\mathbf{rt}}^\tau(B) \\ \Delta_{\mathbf{rt}}^\tau((\vec{A} \Rightarrow \mathbf{rt}) \Rightarrow B) &= (\vec{A} \Rightarrow \mathbf{rt}) \Rightarrow (\vec{A} \Rightarrow \tau) \Rightarrow \Delta_{\mathbf{rt}}^\tau(B)\end{aligned}$$

This intuitively corresponds to the part of a constructor in the elimination scheme we will associate to \mathbf{rt} . As an example, $\Delta_{\mathbf{nat}}^\tau(\mathbf{nat} \Rightarrow \mathbf{nat}) = \mathbf{nat} \Rightarrow \tau \Rightarrow \tau$.

Definition 17 (Elimination scheme). *To every recursive type $\mathbf{rt} \in \mathcal{RT}$ we associate an elimination scheme ε^τ parametrized by $\tau \in \mathcal{ST}$:*

$$\varepsilon^\tau(\mathbf{rt}) = \mathbf{rt} \Rightarrow \Delta_{\mathbf{rt}}^\tau(t_1) \Rightarrow \dots \Rightarrow \Delta_{\mathbf{rt}}^\tau(t_n) \Rightarrow \tau$$

where t_1, \dots, t_n are the constructor types for \mathbf{rt} in Σ .

We then enrich the deduction system with respect to Σ by adding an axiom for each constructor of Σ as well as recursor constants for each recursive type typed by the corresponding elimination scheme.

Definition 18 (Simple type system with recursors).

$$ML_\Sigma = ML \cup \Sigma \cup \{\text{rec}_{\mathbf{rt}}^\tau : \varepsilon^\tau(\mathbf{rt}) \mid \mathbf{rt} \text{ has a constructor in } \Sigma\}$$

Example 4 (Trees and Forests). Given the signature Σ of the previous example, ML_Σ is the type system ML enriched with the axioms $\text{Node} : \mathbf{fst} \Rightarrow \mathbf{tree}$, $\text{Leaf} : \mathbf{fst}$, $\text{Cons} : \mathbf{tree} \Rightarrow \mathbf{fst} \Rightarrow \mathbf{fst}$ as well as the two axiom schemes parametrized by τ :

$$\begin{aligned}\text{rec}_{\mathbf{tree}}^\tau &: \mathbf{tree} \Rightarrow (\mathbf{fst} \Rightarrow \tau) \Rightarrow \tau \\ \text{rec}_{\mathbf{fst}}^\tau &: \mathbf{fst} \Rightarrow \tau \Rightarrow (\mathbf{tree} \Rightarrow \mathbf{fst} \Rightarrow \tau \Rightarrow \tau) \Rightarrow \tau\end{aligned}$$

Reduction rules Let τ be some type. Let $\mathbf{rt} \in \mathcal{RT}$, $(C : P)$ be one of his constructors, (t, \vec{u}) a sequence for $\text{rec}_{\mathbf{rt}}^\tau$, \vec{x} a sequence for C , and f a variable. We define the term $\Theta_{\mathbf{rt}, \vec{u}}^\tau(\vec{x}, P, f)$ by induction on both \vec{x} and the proof that P is a constructor for \mathbf{rt} as follows.

$$\begin{aligned}\Theta_{\mathbf{rt}, \vec{u}}^\tau([], \mathbf{rt}, f) &= f \\ \Theta_{\mathbf{rt}, \vec{u}}^\tau((x, \vec{x}'), A \Rightarrow B, f) &= \Theta_{\mathbf{rt}, \vec{u}}^\tau(\vec{x}', B, (f x)) \\ \Theta_{\mathbf{rt}, \vec{u}}^\tau((x, \vec{x}'), (\vec{A} \Rightarrow \mathbf{rt}) \Rightarrow B, f) &= \Theta_{\mathbf{rt}, \vec{u}}^\tau(\vec{x}', B, (f x \lambda \vec{y}. (\text{rec}_{\mathbf{rt}}^\tau x \vec{y} \vec{u})))\end{aligned}$$

where \vec{y} is a sequence for \vec{A} . This corresponds to the right hand-side of the reduction rule we will associate to a constructor of \mathbf{rt} . As an example, we get the following term for the type of the constructor S .

$$\Theta_{\mathbf{nat}, (u, v)}^\tau((n), \mathbf{nat} \Rightarrow \mathbf{nat}, v) = v n (\text{rec}_{\mathbf{nat}}^\tau n u v)$$

Definition 19 (Reduction rules associated to a type). Let τ be some type. Let $\mathbf{rt} \in \mathcal{RT}$. We name $\mathcal{R}_{\mathbf{rt}}^\tau$ the set of reduction rules composed of

$$(\text{rec}_{\mathbf{rt}}^\tau \vec{u} (C_i \vec{x}_i)) \triangleright \Theta_{\mathbf{rt}, \vec{u}}^\tau(\vec{x}_i, P_i, u_i)$$

for every $(C_i : P_i)$ constructor of \mathbf{rt} , where \vec{x}_i is a sequence for C_i and \vec{u} is a sequence u_1, \dots, u_n of variables such that for all variable t , (t, \vec{u}) is a sequence for $\text{rec}_{\mathbf{rt}}^\tau$.

The set $\mathcal{R}(\Sigma)$ of reduction rules scheme parametrized by τ associated to a signature Σ is then naturally defined as:

$$\mathcal{R}(\Sigma) = \bigcup_{\mathbf{rt} \in \mathcal{RT}} \mathcal{R}_{\mathbf{rt}}^\tau$$

Example 5 (Trees and Forests). For the signature Σ proposed in example 4, we get the following reduction system.

$$\begin{aligned} \text{rec}_{\mathbf{tree}}^\tau u_1 (\text{Node } x_1) &\triangleright u_1 x_1 \\ \text{rec}_{\mathbf{fst}}^\tau u_1 u_2 \text{Leaf} &\triangleright u_1 \\ \text{rec}_{\mathbf{fst}}^\tau u_1 u_2 (\text{Cons } x_1 x_2) &\triangleright u_2 x_1 x_2 (\text{rec}_{\mathbf{fst}}^\tau x_2 u_1 u_2) \end{aligned}$$

Definition 20 (ML $_\Sigma$ proof reduction). The proof-terms of ML_Σ are identified by the union of $\mathcal{R}(\Sigma)$ and β -reduction.

Proposition 3 (Subject reduction). For all signature Σ , ML_Σ enjoys the subject reduction property.

Proof. The rules of $\mathcal{R}(\Sigma)$ are type preserving. □

4.2 Translation to FUM

We will now encode both ML_Σ judgments and computational behaviour into the $FUM_{\mathcal{R}_2}^{\mathcal{R}_1}$ Fold-Unfold modulo system. The predicate symbols are those of \mathcal{RT} along with a unary predicate ϵ . To each proposition $\tau \in \mathcal{ST}$ we associate a constant symbol $\dot{\tau}$ of sort κ . We finally define \mathcal{R}_1 as the infinite proposition rewrite system which reifies them to the propositional level:

$$\mathcal{R}_1 = \{\epsilon(\dot{\tau}) \rightarrow \tau \mid \tau \in \mathcal{ST}\}$$

Remark 1. The infinite nature of \mathcal{R}_1 is not intrinsic of our encoding. We could actually have used a finite one consisting in one constant \mathbf{rt} per recursive type symbol and its a decoding rule $\epsilon(\mathbf{rt}) \rightarrow \mathbf{rt}$, along with a binary predicate symbol $\dot{\Rightarrow}$ decoded by the rule $\dot{\Rightarrow}(x, y) \rightarrow x \Rightarrow y$. As an example, the proposition $\dot{\Rightarrow}(\dot{\Rightarrow}(\mathbf{nat}, \mathbf{nat}), \mathbf{nat})$ would be congruent to $(\mathbf{nat} \Rightarrow \mathbf{nat}) \Rightarrow \mathbf{nat}$ in this setting. However, for the sake of simplicity, we stick to the infinite version in this paper.

Let us now define \mathcal{R}_2 according to the signature Σ . For every $\mathbf{rt} \in \mathcal{RT}$, $\varphi \in \mathcal{PT}$ and first-order variable p of sort κ , we define a proposition $\delta_{\mathbf{rt}}^t(\varphi)$ by induction on the structure of φ as follows.

$$\begin{aligned} \delta_{\mathbf{rt}}^p(\mathbf{rt}) &= \epsilon(p) \\ \delta_{\mathbf{rt}}^p(A \Rightarrow B) &= A \Rightarrow \delta_{\mathbf{rt}}^p(B) \\ \delta_{\mathbf{rt}}^p(\vec{A} \Rightarrow \mathbf{rt}) &= (\vec{A} \Rightarrow \mathbf{rt}) \Rightarrow (\vec{A} \Rightarrow \epsilon(p)) \Rightarrow \delta_{\mathbf{rt}}^p(B) \end{aligned}$$

Definition 21 (Proposition rewrite rules associated to \mathbf{rt}). To every recursive type $\mathbf{rt} \in \mathcal{RT}$ we associate proposition rewrite rule $R_{\mathbf{rt}}$ using the definition of δ above.

$$R_{\mathbf{rt}} : \mathbf{rt} \rightarrow \forall p \delta_{\mathbf{rt}}^p(t_1) \Rightarrow \dots \Rightarrow \delta_{\mathbf{rt}}^p(t_n) \Rightarrow \epsilon(p)$$

where t_1, \dots, t_n are the constructor types for \mathbf{rt} in Σ .

Let us state the essential property of these rules.

Lemma 4 (Positivity). For every \mathbf{rt} type in \mathcal{RT} and $R_{\mathbf{rt}} : \mathbf{rt} \rightarrow \phi$ the associated rewrite rule, the occurrences of \mathbf{rt} are in positive position in ϕ .

Proof. By induction on the type of \mathbf{rt} constructors. \square

These rules constitute $\mathcal{R}_2 = \{R_{\mathbf{rt}} \mid \mathbf{rt} \in \Sigma\}$ which means that for each $\mathbf{rt} \in \mathcal{RT}$ we get two inference rules ($\text{FOLD}_{R_{\mathbf{rt}}}$) and ($\text{UNFOLD}_{R_{\mathbf{rt}}}$).

Let us see now how we encode the constructors of ML_{Σ} with proof-terms of $FUM_{\mathcal{R}_1}^{\mathcal{R}_2}$. Let $(C : P)$ be a constructor of $\mathbf{rt} \in \mathcal{RT}$, (p, \vec{u}) a sequence for the right hand-side of $R_{\mathbf{rt}}$, \vec{x} a sequence for C , and f a proof variable. We define the term $\theta_{\mathbf{rt}, \vec{u}}^p(\vec{x}, P, f)$ by induction on both \vec{x} and the proof that P is a constructor for \mathbf{rt} as follows.

$$\begin{aligned} \theta_{\mathbf{rt}, \vec{u}}^p([], \mathbf{rt}, f) &= f \\ \theta_{\mathbf{rt}, \vec{u}}^p((x, \vec{x}'), A \Rightarrow B, f) &= \theta_{\mathbf{rt}, \vec{u}}^p(\vec{x}', B, (f x)) \\ \theta_{\mathbf{rt}, \vec{u}}^p((x, \vec{x}'), (\vec{A} \Rightarrow \mathbf{rt}) \Rightarrow B, f) &= \theta_{\mathbf{rt}, \vec{u}}^p(\vec{x}', B, (f x \wedge (p, x, \vec{u}))) \end{aligned}$$

where

- $\wedge(p, x, \vec{u}) = \lambda \vec{y}. (\lambda \alpha. (R_{\mathbf{rt}} \alpha p) x \vec{y} \vec{u})$,
- α is a proof variable of type \mathbf{rt} ,
- \vec{y} is a sequence for \vec{A} .

Definition 22 (Proof-term encoding a constructor). Let $(C_i : P_i)$ be a constructor for $\mathbf{rt} \in \mathcal{RT}$, \vec{x} a sequence for C_i and $(p, \vec{u}) = p, u_1, \dots, u_n$ a sequence for the right hand-side of $R_{\mathbf{rt}}$. We define the proof-term

$$\nu_{C_i} = \lambda \vec{x}. (\mathfrak{R}_{\mathbf{rt}} \lambda p. \lambda \vec{u}. \theta_{\mathbf{rt}, \vec{u}}^p(\vec{x}, P_i, u_i))$$

as the proof-term encoding the constructor C_i .

Example 6 (Trees and Forests). The proposition rewrite rules $R_{\mathbf{tree}}$ and $R_{\mathbf{forest}}$ respectively associated to **tree** and **forest** are:

$$\begin{aligned} \text{rec}_{\mathbf{tree}}^{\tau} &\rightarrow \forall p (\mathbf{first} \Rightarrow \epsilon(p)) \Rightarrow \epsilon(p) \\ \text{rec}_{\mathbf{forest}}^{\tau} &\rightarrow \forall p \epsilon(p) \Rightarrow (\mathbf{tree} \Rightarrow \mathbf{first} \Rightarrow \epsilon(p) \Rightarrow \epsilon(p)) \Rightarrow \epsilon(p) \end{aligned}$$

The constructors **Node**, **Leaf** and **Cons** are encoded by the following proof-terms in supernatural deduction.

$$\begin{aligned} \nu_{\mathbf{Node}} &= \lambda x. (\mathfrak{R}_{\mathbf{tree}} \lambda p. \lambda \alpha. (\alpha x)) \\ \nu_{\mathbf{Leaf}} &= \mathfrak{R}_{\mathbf{forest}} \lambda p. \lambda \alpha. \lambda \beta. \alpha \\ \nu_{\mathbf{Cons}} &= \lambda x. \lambda y. (\mathfrak{R}_{\mathbf{forest}} \lambda p. \lambda \alpha. \lambda \beta. (\beta x y (\lambda \gamma. (\mathfrak{R}_{\mathbf{forest}} \gamma p) y \alpha \beta))) \end{aligned}$$

Definition 23 (Translation from ML_Σ to $FUM_{\mathcal{R}_1}^{\mathcal{R}_2}$).

$$\begin{aligned} |t u| &= |t| |u| \\ |\lambda x.t| &= \lambda x. |t| \\ |\text{rec}_{\text{rt}}^\tau| &= \lambda \alpha. (\text{R}_{\text{rt}} \alpha \dot{\tau}) \\ |\mathbf{C}| &= \nu_{\mathbf{C}} \\ |x| &= x \quad \text{if } x \text{ is a variable} \end{aligned}$$

Lemma 5 (Soundness). *For all proof-term π in ML_Σ , if $\pi : T$ then $|\pi| : T$ in $FUM_{\mathcal{R}_1}^{\mathcal{R}_2}$.*

Proof. By induction on π . □

Theorem 4 (Simulation). *Let π and π' be two proofs in ML_Σ such that $\pi \triangleright \pi'$, then $|\pi| \triangleright |\pi'|$ in $FUM_{\mathcal{R}_1}^{\mathcal{R}_2}$.*

Proof. The interesting case is that of $\text{rec}_{\text{rt}}^\tau t \vec{u}$ which is treated by induction on the constructors of rt . □

Corollary 2 (Relative normalization). *The strong-normalization of $FUM_{\mathcal{R}_1}^{\mathcal{R}_2}$ implies that of ML_Σ .*

4.3 Strong normalization

We shall now prove the super-consistency of $DM_{\mathcal{R}_1 \cup \mathcal{R}_2}$, therefore strong normalization of $FUM_{\mathcal{R}_1}^{\mathcal{R}_2}$.

Theorem 5. $\mathcal{R}_1 \cup \mathcal{R}_2$ is superconsistent.

Proof. Let B be an ordered and complete PHA. We build a B -model \mathcal{M} of $\mathcal{R}_1 \cup \mathcal{R}_2$ as follows. The domain M of \mathcal{M} is B . We therefore interpret ϵ by the identity on B .

To give an interpretation to the zero-ary predicates $\text{rt}_1, \dots, \text{rt}_n$ of \mathcal{RT} , the proof slightly differs from the one for System T since we have to handle mutually recursive types (trees and forests for instance). For every tuple (f_1, \dots, f_n) of B^n we can define a model $\mathcal{M}_{(f_1, \dots, f_n)}$ of the language $\forall, \Rightarrow, \text{rt}_1, \dots, \text{rt}_n$ where rt_1 is interpreted by f_1 , rt_2 by f_2 , etc. We call $\mathcal{E}_{\mathcal{M}}$ the set made of of these models for every (f_1, \dots, f_n) of B^n . We then define a function Φ from $\mathcal{E}_{\mathcal{M}}$ to $\mathcal{E}_{\mathcal{M}}$ which maps a model $\mathcal{M}_{(f_1, \dots, f_n)}$ to a model $\mathcal{M}_{(f'_1, \dots, f'_n)}$ where f'_i is the interpretation of the right hand-side of R_{rt_i} in $\mathcal{M}_{(f_1, \dots, f_n)}$:

$$\begin{aligned} f'_1 &= \llbracket \forall p \dots \Rightarrow \text{rt}_1 \Rightarrow \dots \Rightarrow \epsilon(p) \rrbracket^{\mathcal{M}_{(f_1, \dots, f_n)}} \\ &\vdots \\ f'_n &= \llbracket \forall p \dots \Rightarrow \text{rt}_n \Rightarrow \dots \Rightarrow \epsilon(p) \rrbracket^{\mathcal{M}_{(f_1, \dots, f_n)}} \end{aligned}$$

We extend now the order \sqsubseteq to $\mathcal{E}_{\mathcal{M}}$ the following way: $\mathcal{M}_{(f_1, \dots, f_n)} \sqsubseteq \mathcal{M}_{(f'_1, \dots, f'_n)}$ if $f_1 \sqsubseteq f'_1, \dots, f_n \sqsubseteq f'_n$. Then by lemma 4, Φ is monotone for \sqsubseteq and thus we can build a fixpoint $\mathcal{M}_{(F_1, \dots, F_n)}$ of Φ .

We complete this model $\mathcal{M}_{(F_1, \dots, F_n)}$ by interpreting every $\dot{\tau}$ by $\llbracket \tau \rrbracket$ to obtain \mathcal{M} . □

Corollary 3. *For all signature Σ , ML_Σ enjoys the strong normalization property.*

5 Extension to Heyting Arithmetic

The next natural extension of this system is to annotate types with dependent information. Extending the notion of super-consistency to dependent types in the sense of λII is an open and challenging question. Therefore we will stick to predicate logic, where the language of first-order terms is different from that of proof-terms. However, deduction modulo with a congruence on first-order terms provides a powerful framework which allows flexible typing of proof-terms as we will see.

5.1 System T for Predicate Logic : Heyting Arithmetic

A “dependent” version of System T would be some weak version of Heyting arithmetic, the language of first-order terms being the free algebra formed by the functions symbol $\mathbb{0}$ and \underline{S} . Note that they should not be confused with the proof-term constants 0 and S and we will therefore denote them by $\underline{0}$ and \underline{S} .

Definition 24 (System T for Predicate Logic). *System T for predicate logic is Deduction Modulo with $\{\underline{0}, \underline{S}\}$ as first-order language, in addition to:*

- a unary predicate Nat ,
- a proof-term constant $\mathbb{0}$ of type $\text{Nat}(\underline{0})$,
- a proof-term constant \underline{S} of type $\forall n (\text{Nat}(n) \Rightarrow \text{Nat}(\underline{S}(n)))$,
- as many proof-term constants rec^P of type

$$\forall n (\text{Nat}(n) \Rightarrow P(\underline{0}) \Rightarrow (\forall m \text{Nat}(m) \Rightarrow P(m) \Rightarrow P(\underline{S}(m))) \Rightarrow P(n))$$

as one can form unary propositions (propositions parametrized by one first-order term) with the language of predicate logic and the predicate Nat ,

- *the following reduction rule schemes,*

$$\begin{array}{c} \text{rec}^P \underline{0} \mathbb{0} u v \triangleright u \\ \text{rec}^P \underline{S}(n) (\underline{S} n \nu_n) u v \triangleright v n \nu_n (\text{rec}^P n \nu_n u v) \end{array}$$

We recognize in the decorated type of rec^P the induction scheme of Heyting arithmetic. $\text{Nat}(n)$ can therefore be read as “ n is in the smallest set closed by $\underline{0}$ and \underline{S} ”.

5.2 Simulation

We now show how we can simulate this later system in Fold-Unfold Modulo the way we did for System T. We turn the type of rec into a proposition rewrite rule:

$$\mathbb{R}_{\text{Nat}} : \text{Nat}(n) \rightarrow \forall P (P(\underline{0}) \Rightarrow (\forall m \text{Nat}(m) \Rightarrow P(m) \Rightarrow P(\underline{S}(m))) \Rightarrow P(n))$$

which gives the following typing rules when turned into a Fold-Unfold fashion:

$$\begin{array}{c} (\text{UNFOLD}_{\mathbb{R}_{\text{Nat}}}) \frac{\pi : \text{Nat}(n)}{\mathbb{R}_{\text{Nat}} \pi : \forall P (P(\underline{0}) \Rightarrow (\forall m \text{Nat}(m) \Rightarrow P(m) \Rightarrow P(\underline{S}(m))) \Rightarrow P(n))} \\ (\text{FOLD}_{\mathbb{R}_{\text{Nat}}}) \frac{\pi : \forall P (P(\underline{0}) \Rightarrow (\forall m \text{Nat}(m) \Rightarrow P(m) \Rightarrow P(\underline{S}(m))) \Rightarrow P(n))}{\mathbb{R}_{\text{Nat}} \pi : \text{Nat}(n)} \end{array}$$

This definition however makes use of second-order quantification, which is not handled by super-consistency. We can avoid this situation by using the comprehension scheme of the theory of classes, the same way we associated a first-order constant $\dot{\tau}$ to every proposition τ of system T:

$$\mathbb{R}_{\text{Nat}} : \text{Nat}(n) \rightarrow \forall P (\underline{0} \in P \Rightarrow (\forall m \text{ Nat}(m) \Rightarrow m \in P \Rightarrow \underline{S}(m) \in P) \Rightarrow n \in P)$$

The details of the decoding rules associated to \in being rather complex [25], we do not detail them *in extenso*. See [9] for a completely formalized version of this technique applied to arithmetic.

Simulating System T for Predicate Logic behaviour is then straightforward as shown by the translation map below :

$$\begin{array}{ll} |t u| = |t| |u| & |0| = \nu_0 \\ |\lambda x.t| = \lambda x.|t| & |S| = \nu_S \\ |\text{rec}^P| = \lambda n.\lambda \nu_n.(\mathbb{R}_{\text{Nat}} \nu_n \dot{P}) & |x| = x \quad \text{if } x \text{ is a variable} \end{array}$$

where

$$\begin{array}{l} \nu_0 = \mathbb{R}_{\text{Nat}} \lambda p.\lambda \alpha.\lambda \beta.\alpha \\ \nu_S = \lambda n.\lambda \nu_n.(\mathbb{R}_{\text{Nat}} \lambda p.\lambda \alpha.\lambda \beta.(\beta n \nu_n (\lambda m.\lambda \nu_m.(\mathbb{R}_{\text{Nat}} \nu_m p) n \nu_n \alpha \beta))) \end{array}$$

As for System T before, this encoding faithfully mimics System T for Predicate Logic computational behaviour.

$$\begin{array}{l} |\text{rec}^P \underline{0} 0 u v| \triangleright^+ |u| \\ |\text{rec}^P \underline{S}(n) (S n \nu_n) u v| \triangleright^+ |v n \nu_n (\text{rec}^P n \nu_n u v)| \end{array}$$

Finally, the above rewrite rule defining Nat as well as the second-order encoding rules form a super-consistent theory.

Theorem 6. *The proposition rewrite system made of \mathbb{R}_{Nat} and the rewrite rules defining \in is super-consistent.*

Proof. By erasing every information concerning $+$, \times and $=$ in the super-consistency proof of [9]. The main difference w.r.t. the proof of Theorem 2 is to split the first-order terms into two sorts: ι for those representing naturals and κ for those encoding propositions. Then we chose \mathbb{N} as the interpretation domain of ι , and functions from \mathbb{N} to \mathcal{B} as the interpretation domain of κ . The remaining of the proof is similar to the one above. \square

From which we immediately derive the following.

Corollary 4. *System T for Predicate Logic enjoys strong normalization.*

Proof. The proposition rewrite system made of \mathbb{R}_{Nat} and the rewrite rules defining \in is super-consistent. Thus, by Theorem 1, it is terminating. We can conclude that the system T for predicate logic is terminating by Proposition 2. \square

5.3 Rewriting First-Order Terms

Our definition of System T for Predicate Logic is however somewhat poor : it lacks arithmetic function symbols along with their computational behaviour in the first-order terms side and can't type proof-terms like the definition of $+$ or \times .

We can handle this issue two ways. Either by adding missing definitions of arithmetic under the form of axioms.

$$\begin{aligned} & \forall x (\underline{0} + x = x) \\ & \forall x \forall y (\underline{S}(x) + y = \underline{S}(x + y)) \\ & \forall x (\underline{S} \times x) = \underline{0} \\ & \forall x \forall y (\underline{S}(x) \times y) = y + (x \times y) \end{aligned}$$

Either by adding them under the form of first-order terms rewrite rules.

$$\begin{aligned} & \underline{0} + x \rightarrow x \\ & \underline{S}(x) + y \rightarrow \underline{S}(x + y) \\ & \underline{0} \times x \rightarrow \underline{0} \\ & \underline{S}(x) \times y \rightarrow y + (x \times y) \end{aligned}$$

The later solution has the greatest advantage to leave the context empty, thus to preserve the witness property of the system. Moreover, the super-consistency proof of the system smoothly adapts: one has only to check that $\llbracket l \rrbracket = \llbracket r \rrbracket$ for every rewrite rule $\llbracket l \rrbracket \rightarrow \llbracket r \rrbracket$, which is trivial when interpreting the theory on the domain \mathbb{N} .

A nice consequence of defining function symbols by rewrite rules over first-order terms is a great flexibility in proof-term typing. As an example, the enriched (*i.e.* adapted to System T for Predicate Logic) of the function $+$ defined in the proof-term side may be typed by either

$$\begin{aligned} & + : \forall x \forall y \text{Nat}(x) \Rightarrow \text{Nat}(y) \Rightarrow \text{Nat}(x + y) \\ \text{or} \quad & + : \forall x \forall y \text{Nat}(y) \Rightarrow \text{Nat}(x) \Rightarrow \text{Nat}(y + x) \end{aligned}$$

depending on whether $+$ is defined by respectively

$$\begin{aligned} & \underline{0} + x \rightarrow x & \text{or} & & x + \underline{0} \rightarrow x \\ & \underline{S}(x) + y \rightarrow \underline{S}(x + y) & & & x + \underline{S}(y) \rightarrow \underline{S}(x + y) \end{aligned}$$

We can even allow both definitions (both rewrite systems) since this does not break the construction of the model, so the strong normalization of the system still holds.

6 Related work

The work on termination of higher-order rewriting started by Blanqui, Jouannaud and Okada and further extended by Blanqui [26] features rewrite rules at the proof-term level in the Calculus of Constructions. The authors present a criterion, called *general schema* which ensures the termination of the resulting system for a large class of rewrite

systems. In particular, [27] shows how to encode the whole Calculus of Inductive Constructions in this framework. The first difference with our work is that we abstract over the structure of reducibility candidates whereas this later work makes explicitly use of them. The second difference is that our approach simulates recursors by identifying propositions, whereas it is obtained with rewrite rules on proof-terms (basically the reduction rules associated to the recursor) in the cited work.

Identifying types the way we do is however very close to the approach of David and Nour [28] who prove that propositional logic with equations on types strongly normalizes *if and only if* the equations do not feature negative occurrences of the defined type. Moreover, they show that the proof can be formalized in the only Peano arithmetic. The main difference with our approach, apart from the fact that we do not deal directly with reducibility candidates, is that their proof require as many fixpoints as there are uses of the induction scheme whereas ours builds one unique fixpoint by quantifying over types thanks to second-order encoding.

7 Conclusion

We have introduced a new semantic method to prove System T termination by exhibiting an intermediate system between deduction modulo and System T based on Fold-Unfold. We then have extended the result to a family of inductive types and have finally shown how the method could be applied to Heyting arithmetic as a dependent version of System T with rewrite rules over proposition. An interesting research topic would be the encoding of parametrized inductive types by means of proposition rewrite rules. Finally, it would be interesting to port the result to sequent calculus, thus enabling recursors in their usually associated proof-term languages [29,30].

References

1. Gentzen, G.: Untersuchungen über das logische schließen. In Berka, K., Kreiser, L., eds.: Logik-Texte: Kommentierte Auswahl zur Geschichte der Modernen Logik (vierte Auflage). Akademie-Verlag, Berlin (1986) 206–262
2. Prawitz, D.: Hauptsatz for higher order logic. *J. Symb. Log.* **33**(3) (1968) 452–457
3. Takahashi, M.: A proof of cut-elimination theorem in simple type theory. *Journal of the Mathematical Society of Japan* **19** (1967) 399–410
4. Peter B. Andrews: Resolution in type theory. **36** (1971) 414–432
5. Hermant, O.: Semantic cut elimination in the intuitionistic sequent calculus. In Urzyczyn, P., ed.: Typed Lambda-Calculi and Applications. Volume 3461 of Lecture Notes in Computer Science., Nara, Japan, Springer-Verlag (2005) 221–233
6. DeMarco, M., Lipton, J.: Completeness and cut-elimination in the intuitionistic theory of types. *J. Log. Comput.* **15**(6) (2005) 821–854
7. Okada, M.: A uniform semantic proof for cut-elimination and completeness of various first and higher order logics. *Theor. Comput. Sci.* **281**(1-2) (2002) 471–498
8. Dowek, G., Werner, B.: Proof normalization modulo. *Journal of Symbolic Logic* **68**(4) (2003) 1289–1316
9. Allali, L.: Algorithmic equality in Heyting arithmetic modulo. In: TYPES. (2007) To appear.

10. Dowek, G., Werner, B.: Arithmetic as a theory modulo. In Giesl, J., ed.: Proceedings of RTA'05. Volume 3467 of LNCS., Springer (2005) 423–437
11. Tait, W.W.: Intensional interpretation of functionals of finite type I. *jsl* **32** (1967) 198–212
12. Werner, B.: Une Théorie des Constructions Inductives. PhD thesis, Université Paris 7 (1994)
13. Paulin-Mohring, C.: Inductive definitions in the system coq - rules and properties. In: TLCA '93: Proceedings of the International Conference on Typed Lambda Calculi and Applications, London, UK, Springer-Verlag (1993) 328–345
14. Gimenez, E.: A tutorial on recursive types in coq (1998)
15. Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (September 2007)
16. Prawitz, D.: Natural Deduction. A Proof-Theoretical Study. Almqvist and Wiksell (1965)
17. Dowek, G.: About folding-unfolding cuts and cuts modulo. *J. Log. Comput.* **11**(3) (2001) 419–429
18. Girard, J.Y.: Proofs and Types. Volume 7 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge (1989)
19. Dowek, G., Hardin, T., Kirchner, C.: Theorem proving modulo. *Journal of Automated Reasoning* **31**(1) (Nov 2003) 33–72
20. Dowek, G., Miquel, A.: Cut elimination for zermelo's set theory. Available on author's web page
21. Dowek, G.: Proof normalization for a first-order formulation of higher-order logic. In Gunter, E., Felty, A., eds.: TPHOL. Volume 1275 of LNCS., Springer (1997) 105–119
22. Prawitz, D.: Natural Deduction, a Proof-theoretical Study. (1965)
23. Dowek, G.: Truth values algebras and proof normalization. In: TYPES. (2006) 110–124
24. Parigot, M.: Programming with proofs: A second order type theory. In: ESOP '88: Proceedings of the 2nd European Symposium on Programming, London, UK, Springer-Verlag (1988) 145–159
25. Kirchner, F.: A finite first-order theory of classes. In: Proc. 2006 Int. Workshop on Proofs and Programs. Lecture notes in Computer Science, Springer-Verlag (2006)
26. Blanqui, F.: Definitions by rewriting in the calculus of constructions. In: Logic in Computer Science. (2001) 9–18
27. Blanqui, F.: Inductive types in the calculus of algebraic constructions. TLCA'03 (2003) 61–86
28. David, R., Nour, K.: An arithmetical proof of the strong normalization results for the lambda-calculus with recursive equations on types. TLCA'07 (2007) 84–101
29. Urban, C.: Classical Logic and Computation. PhD thesis, University of Cambridge (October 2000)
30. Herbelin, H.: Séquents qu'on calcule. PhD thesis, Université Paris 7 (January 1995)

A Deduction modulo typing rules

$$\begin{array}{c}
(Ax) \frac{}{\Gamma \vdash_{\equiv} \alpha : B} \alpha : A \in \Gamma \text{ and } A \equiv B \\
(\Rightarrow I) \frac{\Gamma \alpha : A \vdash_{\equiv} \pi : B}{\Gamma \vdash_{\equiv} \lambda \alpha. \pi : C} C \equiv (A \Rightarrow B) \\
(\Rightarrow E) \frac{\Gamma \vdash_{\equiv} \pi : C \quad \Gamma \vdash_{\equiv} \pi' : A}{\Gamma \vdash_{\equiv} (\pi \pi') : B} C \equiv (A \Rightarrow B) \\
(\wedge I) \frac{\Gamma \vdash_{\equiv} \pi : A \quad \Gamma \vdash_{\equiv} \pi' : B}{\Gamma \vdash_{\equiv} \langle \pi, \pi' \rangle : C} C \equiv (A \wedge B) \\
(\wedge E_2) \frac{\Gamma \vdash_{\equiv} \pi : C}{\Gamma \vdash_{\equiv} \text{fst}(\pi) : A} C \equiv (A \wedge B) \\
(\wedge E_1) \frac{\Gamma \vdash_{\equiv} \pi : C}{\Gamma \vdash_{\equiv} \text{snd}(\pi) : B} C \equiv (A \wedge B) \\
(\vee I_1) \frac{\Gamma \vdash_{\equiv} \pi : A}{\Gamma \vdash_{\equiv} i(\pi) : C} C \equiv (A \vee B) \\
(\vee I_2) \frac{\Gamma \vdash_{\equiv} \pi : B}{\Gamma \vdash_{\equiv} i(\pi) : C} C \equiv (A \vee B) \\
(\vee E) \frac{\Gamma \vdash_{\equiv} \pi_1 : D \quad \Gamma \alpha : A \vdash_{\equiv} \pi_2 : C \quad \Gamma \beta : B \vdash_{\equiv} \pi_3 : C}{\Gamma \vdash_{\equiv} (\delta \pi_1 \alpha. \pi_2 \beta. \pi_3) : C} D \equiv (A \vee B) \\
(\top I) \frac{}{\vdash_{\equiv} I : A} A \equiv \top \\
(\perp E) \frac{\Gamma \vdash_{\equiv} \pi : B}{\Gamma \vdash_{\equiv} (\delta_{\perp} \pi) : A} B \equiv \perp \\
(\forall I) \frac{\Gamma \vdash_{\equiv} \pi : A}{\Gamma \vdash_{\equiv} \lambda x \pi : B} B \equiv (\forall x A), x \notin FV(\Gamma) \\
(\forall E) \frac{\Gamma \vdash_{\equiv} \pi : B}{\Gamma \vdash_{\equiv} (\pi t) : A\{x := t\}} B \equiv (\forall x A) \\
(\exists I) \frac{\Gamma \vdash_{\equiv} \pi : A\{x := t\}}{\Gamma \vdash_{\equiv} \langle t, \pi \rangle : B} B \equiv (\exists x A) \\
(\exists E) \frac{\Gamma \vdash_{\equiv} \pi : C \quad \Gamma \alpha : A \vdash_{\equiv} \pi' : B}{\Gamma \vdash_{\equiv} (\delta_{\exists} \pi x. \alpha. \pi') : B} C \equiv (\exists x A) \text{ and } x \notin FV(\Gamma, B)
\end{array}$$

Fig. 1. Typing rules for deduction modulo with a congruence \equiv