First-order automated reasoning with theories: when deduction modulo theory meets practice

Guillaume Burel, Guillaume Bury, Raphaël Cauderlier, David Delahaye, Pierre Halmagrand, Olivier Hermant

Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

First-Order Automated Reasoning with Theories: When Deduction Modulo Theory Meets Practice

Guillaume Burel · Guillaume Bury · Raphaël Cauderlier · David Delahaye · Pierre Halmagrand · Olivier Hermant

Received: date / Accepted: date

Abstract We discuss the practical results obtained by the first generation of automated theorem provers based on Deduction modulo theory. In particular, we demonstrate the concrete improvements such a framework can bring to first-order theorem provers with the introduction of a rewrite feature. Deduction modulo theory is an extension of predicate calculus with rewriting both on terms and propositions. It is well suited for proof search in theories because it turns many axioms into rewrite rules. We introduce two automated reasoning systems that have been built to extend other provers with Deduction modulo theory. The first one is Zenon Modulo, a tableau-based tool able to deal with polymorphic first-order logic with equality, while the second one is iProver-

A part of this work has been supported by the BWare project [56,107] (ANR-12-INSE-0010) funded by the INS programme of the French National Research Agency (ANR).

G. Burel
ENSIIE and Samovar, Télécom SudParis and CNRS, Université Paris-Saclay, Évry, France
Inria and LSV, CNRS and ENS Paris-Saclay, Université Paris-Saclay, Cachan, France
E-mail: Guillaume.Burel@ensiie.fr

G. Bury
Inria/LSV/ENS Paris-Saclay, Cachan, France
E-mail: Guillaume.Bury@inria.fr

R. Cauderlier
Irif/Université Paris-Diderot, Paris, France
E-mail: Raphael.Cauderlier@irif.fr

D. Delahaye
LIRMM, Université de Montpellier, CNRS, Montpellier, France
E-mail: David.Delahaye@lirmm.fr

P. Halmagrand
EDF R&D (OSIRIS), Palaiseau, France
E-mail: Pierre.Halmagrand@edf.fr

O. Hermant
CRI, MINES ParisTech, PSL University, Paris, France
E-mail: Olivier.Hermant@mines-paristech.fr

Modulo, a resolution-based system dealing with first-order logic with equality. We also provide some experimental results run on benchmarks that show the beneficial impact of the extension on these two tools and their underlying proof search methods. Finally, we describe the two backends of these systems to the Dedukti universal proof checker, which also relies on Deduction modulo theory, and which allows us to verify the proofs produced by these tools.

Keywords Automated Deduction · Deduction Modulo Theory · First-Order Logic · Rewriting · Automated Reasoning Systems

1 Introduction

Reasoning within theories, whether decidable or not, has become a crucial point in automated theorem proving. A theory, commonly formulated as a collection of axioms, is often necessary to specify, in a concise and understandable way, the properties of objects manipulated in software proofs, such as lists or arrays. Each theory has its own features, but a small number of them appear recurrently, including arithmetic and set theory.

Leaving the axioms and definitions of a given theory at the same level as the hypotheses is not a reasonable option: first, it induces a combinatorial explosion in the search space and second, axioms do not bear any specific meaning that an Automated Theorem Prover (ATP) can take advantage of. To avoid these drawbacks, Deduction modulo theory [60] proposes to replace axioms with rewrite rules and provides a framework combining first-order proof systems with a congruence generated by rewrite rules on terms and propositions. This last distinctive feature allows us to go beyond pure first-order reasoning. However, we must take care of preserving desirable properties for proof search, such as consistency, cut elimination, or completeness.

The goal of Deduction modulo theory is to integrate more closely the theory into the deduction kernel of the proof system. In a way, it can be seen as a natural evolution of proof systems in general, where initial systems with few deduction rules and many axioms (e.g., Frege-Hilbert's systems) have evolved into systems with more deduction rules and fewer axioms (e.g., Gentzen's natural deduction and sequent calculus). Later, and still in the same vein, some approaches proposed to transform axioms of theories into deduction rules. This is the case of Prawitz [92], who proposed to generate introduction and elimination rules from axioms. More recently, superdeduction [34] provided an extension of Prawitz's approach, where the introduction and elimination rules are compiled: steps of deduction that can be done over the rules are done once for all, which gives more compact rules. This serves a common purpose, getting rid of axioms, with the difference that the two previous approaches try to push axioms into the deduction part of the proof system, while Deduction modulo theory transforms axioms into computations. As a result, Deduction modulo theory gives computation its rightful place in proof theory.

To show the beneficial impact of Deduction modulo theory over automated deduction, we present two automated reasoning systems, that rely on different

proof search methods, developed as an extension of two pre-existing provers. The first one is Zenon Modulo, which is a tableau-based tool able to deal with polymorphic first-order logic with equality, and which is an extension of the Zenon ATP [30] to Deduction modulo theory. Compared to the regular version of Zenon, Zenon Modulo offers the ability to deal with polymorphic types. This extension may seem orthogonal with respect to the extension to Deduction modulo theory, but both extensions are actually directly connected. In fact, introducing types in logic allows us to get rid of typing predicates [105], which appeared in our benchmark as conditional hypotheses of axioms, and turning the axioms into rewrite rules is easier as Deduction modulo theory mainly relies on unconditional rewrite rules. The second ATP described in this paper is iProverModulo, which is a resolution-based system dealing with first-order logic with equality, and which is an extension of the iProver ATP [78] to Deduction modulo theory. Contrary to Zenon Modulo, rewriting is not introduced primitively in iProverModulo (at least not for rules rewriting propositions), but on top of the resolution mechanism of iProver using ordered polarized resolution modulo [35] and one-way clauses [59].

When developing automated theorem provers, the critical point is to ensure the soundness of the implementation, and the two ATPs presented in this paper provide assurance to this need of soundness by producing proof certificates, which can be checked by a third party to reach an appropriate degree of confidence. It may be desirable that these ATPs producing proof certificates satisfy the De Bruijn criterion (formulated by Barendregt in [9]), i.e. they generate proof certificates (or even directly proofs) in a format that can be independently checked by external proof tools. In addition, proof certificates must be of good quality and some principles tend to emerge in this domain. One of them is the Poincaré principle (also formulated by Barendregt in [9]), which is directly related to the size of proofs and which states that traces of computation should not be included in proof certificates. In this case, the external proof checker is expected to redo computations by itself and must be therefore adapted. Zenon Modulo and iProverModulo conform to these criteria as they are able to produce proofs to be checked by an external proof tool (De Bruijn's criterion), called Dedukti [28], which is a universal proof checker that also relies on Deduction modulo theory. The fact that Dedukti relies on Deduction modulo theory is quite useful since it allows the ATPs to produce proofs with no trace of computations and it is enough to provide the rewrite rules to Dedukti, which can perform the necessary computations when verifying the generated proofs (Poincaré's principle).

To assess the improvements that are offered by the integration of Deduction modulo theory into Zenon Modulo and iProverModulo, we provide some experimental results run on different benchmarks. If clear benefits can be observed over general-purpose benchmarks, as shown by the results of iProverModulo over the TPTP library [104], the most significant improvements can be obtained when the considered theory is known in advance and can be modeled in a tailored fashion in Deduction modulo theory. This is the case with the benchmark on which Zenon Modulo has been run, and which is made of proof

obligations in set theory (about 13,000 problems) from the formalization of industrial applications using the B method [1], and built in the framework of the BWare project [56,107]. This tends to confirm that in automated deduction, there is always a tradeoff between the power of the ATP and the way that the theory is modeled and expressed.

The paper is organized as follows: in Section 2, we introduce the principles of Deduction modulo theory; in Sections 3 and 4, we present the two ATPs Zenon Modulo and iProverModulo, with their proof search methods and some experimental results run on some benchmarks, before discussing, in Section 5, the approaches used by these two ATPs to integrate Deduction modulo theory; and in Section 6, we consider some related work regarding proof search modulo theories in particular.

2 Deduction Modulo Theory

In this section, we introduce the basic principles of Deduction modulo theory, with the description of some proof and type systems modulo theory, and the presentation of some examples as well.

2.1 Motivations

Deduction modulo theory arises from the need to introduce pure computations in proofs. If we wonder what a proof is, two trends seem to be in contradiction with each other. The first one is the so-called Babylonians' point of view, due to the amazing calculating skills developed by the Babylonians about 4000 BC, and which relies on the fact that in some cases, a proof is just a stream of computations obtained by applying a deterministic and terminating algorithm. The second trend, which could be called the Greeks' point of view due to the great achievement of the ancient Greeks in introducing demonstrative proofs in the history of logic, sees a proof as a sequence of deductions coming from the application of valid inferences to facts, which are (nonlogical) axioms, usually called "hypotheses", logical axioms defining particular syntactic constructions (connectives in the case of Hilbert's system), or previously inferred facts.

This apparent tension between computation and deduction may appear in very simple proofs. For example, let us try to prove that $\forall x.x + 2 = s(s(x))$ in Peano arithmetic, where s is the successor symbol. Using only deduction steps, the proof in sequent calculus with equality is the following:

$$\dfrac{\dfrac{\dfrac{}{x+2=s(x+1)}\,\mathcal{P}_2 \quad \dfrac{\dfrac{}{x+1=s(x+0)}\,\mathcal{P}_2}{s(x+1)=s(s(x+0))}=_{\mathrm{c}} \quad \dfrac{\dfrac{\dfrac{}{x+0=x}\,\mathcal{P}_1}{s(x+0)=s(x)}=_{\mathrm{c}}}{s(s(x+0))=s(s(x))}=_{\mathrm{c}}}{s(x+1)=s(s(x))}=_{\mathrm{t}}}{x+2=s(s(x))}=_{\mathrm{t}}}{\forall x.x+2=s(s(x))}\,\forall_{\mathrm{R}}$$

where we use the simplified notation P for "$\vdash P$", with P a proposition, and where n is a notation for $s(s(\ldots s(0)))$ with n applications of s to 0 (the usual zero of Peano arithmetic's signature). We also assume a proof system where equality is built-in. In particular, "$=_{\mathrm{t}}$" is the transitivity rule, and "$=_{\mathrm{c}}$" is the congruence rule. Otherwise, the same proof would have been much more complex, requiring a context for the equality axioms and an explicit use of them at every step. In the same way, the Peano axioms are integrated into the proof system. The (axiomatic) rules $\mathcal{P}_1$ and $\mathcal{P}_2$ respectively correspond to the axioms $\forall x.x + 0 = x$ and $\forall x, y.x + s(y) = s(x + y)$.

In this proof, Deduction modulo theory proposes to provide a genuine computational behavior to the axioms corresponding to the rules $\mathcal{P}_1$ and $\mathcal{P}_2$. This computational behavior is given by means of rewrite rules that can be applied whenever and wherever in the proof (the propositions are then seen modulo the congruence induced by the rewrite rules). The rewrite rules corresponding to $\mathcal{P}_1$ and $\mathcal{P}_2$ are therefore $x + 0 \to x$ and $x + s(y) \to s(x + y)$, and the previous proof becomes:

$$\dfrac{\dfrac{}{x + 2 = s(s(x))}\ =_{\mathrm{r}},\ x + 2 \to^* s(s(x))}{\forall x.x + 2 = s(s(x))}\ \forall_{\mathrm{R}}$$

where $=_{\mathrm{r}}$ is the reflexivity rule of equality, and "$\to^*$" the reflexive, transitive, and congruent closure of the "$\to$" relation.

As can be seen in this proof, computations are interleaved with the deduction rules, and the proof therefore appears much simpler than the one completed using pure sequent calculus. In addition to simplicity, Deduction modulo theory also allows for unbounded proof size reduction [37].

Deduction modulo theory proposes to go further by offering computation not only over terms, but also over propositions. For example, considering the inclusion axiom in set theory $\forall X, Y.X \subseteq Y \Leftrightarrow \forall x.x \in X \Rightarrow x \in Y$, the proof of $A \subseteq A$ in sequent calculus has the following form:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\ldots, x \in A \vdash A \subseteq A, x \in A}\ \mathrm{Ax}}{\ldots \vdash A \subseteq A, x \in A \Rightarrow x \in A}\ \Rightarrow\mathrm{R}}{\ldots \vdash A \subseteq A, \forall x.x \in A \Rightarrow x \in A}\ \forall\mathrm{R} \qquad \dfrac{}{\ldots, A \subseteq A \vdash A \subseteq A}\ \mathrm{Ax}}{\ldots, (\forall x.x \in A \Rightarrow x \in A) \Rightarrow A \subseteq A \vdash A \subseteq A}\ \Rightarrow\mathrm{L}}{A \subseteq A \Leftrightarrow \forall x.x \in A \Rightarrow x \in A \vdash A \subseteq A}\ \wedge\mathrm{L}}{\forall X, Y.X \subseteq Y \Leftrightarrow \forall x.x \in X \Rightarrow x \in Y \vdash A \subseteq A}\ \forall\mathrm{L} \times 2$$

In Deduction modulo theory, the inclusion axiom can be seen as a computation rule, and replaced by the rewrite rule $X \subseteq Y \to \forall x.x \in X \Rightarrow x \in Y$. The previous proof is then transformed as follows:

$$\dfrac{\dfrac{\dfrac{}{x \in A \vdash x \in A}\ \mathrm{Ax}}{\vdash x \in A \Rightarrow x \in A}\ \Rightarrow\mathrm{R}}{\vdash A \subseteq A}\ \forall\mathrm{R},\ A \subseteq A \to \forall x.x \in A \Rightarrow x \in A$$

where it can be seen, as previously, that computations are clearly separated from the deduction rules, and that the proof is much simpler than the corresponding one in pure sequent calculus.

Deduction modulo theory is one way, among others, most notably Satisfiability Modulo Theories (SMT) solving, to embed those *implicit* computation steps into deduction systems. This is what we are about to describe, first by sketching the language, and then the deduction systems that we work with.

2.2 Rewriting in First-Order Logic

The language of Deduction modulo theory is the language of many-sorted first-order logic, which we succinctly describe here. This readily adapts to untyped first-order logic by considering a single sort for terms. A more formalized extension to (ML-polymorphically) typed first-order logic will be devised in Section 2.6. Most of the material of this section is adapted from [60], with slight presentational variations.

We assume a fixed set of sorts and a signature Σ of function and predicate symbols, associated with an arity n. Function and predicate symbols are also characterized by n input sorts, in addition, function symbols also have an output sort. We also assume for each sort a denumerable set of variables, collectively denoted by V.

For instance, in Section 2.1, $+/2 : \langle \mathsf{nat}, \mathsf{nat}, \mathsf{nat} \rangle$, $s/1 : \langle \mathsf{nat}, \mathsf{nat} \rangle$, $0/0 : \langle \mathsf{nat} \rangle$ are respectively binary, unary, and nullary (constant) function symbols, while $=/2 : \langle \mathsf{nat}, \mathsf{nat} \rangle$ is a binary predicate symbol. They all take (and define, for function symbols) terms of sort nat. Given those, we form terms and predicate instances (atomic formulas) as usual: a term/formula $f(t_1, \cdots, t_n)$ is well-formed if and only if the terms t_i have sorts that comply with the input sorts of the n-ary symbol f. If f is a function symbol, the resulting term has sort the output sort of f.

The logical connectives are $\wedge, \vee, \Rightarrow$ (binary), $\neg$ (unary) and $\bot, \top$ (nullary), and the quantifiers are $\forall$ and $\exists$, they allow forming compound formulas. Later, we will also use the binary connective $\Leftrightarrow$. The set of free variables of a term t, respectively a formula A, is noted $FV(t)$, and $FV(A)$ respectively.

The substitution of u for x is a function associating terms with terms and formulas with formulas. It is noted $[u/x](t)$ and $[u/x](A)$, or for short $[u/x]t$ and $[u/x]A$. u and x have to be of the same sort. As a consequence, substitutions respect sorts on all terms and formulas. We extend this notion to parallel substitutions, which replace multiple variables in parallel. They will be noted ρ, σ, and these notations will be also used for the same concept in other contexts (e.g., type substitutions and free type variables in Section 2.6.2) throughout this paper.

Unless otherwise stated, formulas are denoted by A, B, C, and when we want to emphasize their atomicity by P, Q, R. Sets (or multisets) of formulas are denoted by Γ, Δ.

Definition 1 (Rewrite Rule, Equational Axiom, Rewrite System) A *Term Rewrite Rule* is a pair of terms t, u of the same sort, denoted by $t \to u$, such that $FV(u)$ is included in $FV(t)$.

A *Proposition Rewrite Rule* is a pair of formulas P, F, denoted by $P \to F$, such that P is an atomic formula and $FV(F)$ is included in $FV(P)$.

An *Equational Axiom* is a pair of terms t, u, denoted by $t = u$.

A *Rewrite System* is a pair $\mathcal{RE}$ composed of a set $\mathcal{R}$ of rewrite rules over terms and propositions, and a set $\mathcal{E}$ of equational axioms.

Each variable in $FV(t), FV(u)$, and $FV(P)$ has a sort. We additionally record them in a local (typing) context Γ. A fully-fledged term rewrite rule is therefore defined as $t \to_\Gamma u$. This becomes necessary in the extensions of Section 2.6, but we avoid this for the moment, as sorts are statically assigned.

Definition 2 (Equality Modulo $\mathcal{E}$) Let $\mathcal{E}$ be a set of equational axioms, and t, u two terms. t and u are $\mathcal{E}$-convertible, denoted by $t =_\mathcal{E} u$, if and only if:

- either there exists an equational axiom $e_1 = e_2 \in \mathcal{E}$, and a substitution σ, such that $t = \sigma(e_1)$ and $\sigma(e_2) = u$;
- or $t = f(t_1, \cdots, t_n)$, $u = f(u_1, \cdots, u_n)$ and $t_i =_\mathcal{E} u_i$ for every i;
- or t is u; or $u =_\mathcal{E} t$; or there exists a term v, such that $t =_\mathcal{E} v$ and $v =_\mathcal{E} u$.

In other words, $=_\mathcal{E}$ is the congruence relation generated by $\mathcal{E}$, which means that the binary relation $=_\mathcal{E}$ is the smallest relation containing $\mathcal{E}$ and stable by substitution, context relation, reflexivity, symmetry, and transitivity.

Most of the paper will consider an empty set of equational axioms $\mathcal{E}$, in which case it will be omitted.

Definition 3 (Rewriting Relation) Let $\mathcal{RE}$ be a rewrite system, and t, u two terms. t *rewrites to* u, denoted by $t \to u$, if and only if:

- either there exist a term rewrite rule $l \to r \in \mathcal{R}$ and a substitution σ, such that $t =_\mathcal{E} \sigma(l)$ and $u =_\mathcal{E} \sigma(r)$;
- or $t = f(t_1, \cdots, t_n)$, $u = f(u_1, \cdots, u_n)$, and $t_i \to u_i$ for exactly one index i, while t_i is identical to u_i for the rest of the indices.

Let A, B be two formulas. A *rewrites to* B, denoted by $A \to B$, if and only if:

- either there exist a proposition rewrite rule $l \to r \in \mathcal{R}$ and a substitution σ, such that $A =_\mathcal{E} \sigma(l)$ and $B =_\mathcal{E} \sigma(r)$;
- or $A = P(t_1, \cdots, t_n)$, $B = P(u_1, \cdots, u_n)$, and $t_i \to u_i$ for exactly one index i, while t_i is identical to u_i for the rest of the indices;
- or A and B are compound formulas with the same main connective/quantifier, and exactly one pair of the corresponding subformulas rewrites one to another, the rest of the corresponding subformulas being identical.

In other words, the relation $\to$ is the closure of $\mathcal{RE}$ by substitution and context relation. Its transitive closure is denoted by $\to^+$, its reflexive and transitive closure by $\to^*$, and its reflexive, transitive, and symmetric closure by $\equiv$, which is a congruence relation. The restriction of this congruence to the term rewrite rules of $\mathcal{R}$ and the equational axioms of $\mathcal{E}$ is denoted by $\equiv^t$.

$$\frac{}{\Gamma, P \vdash P, \Delta}\ \text{ax} \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma, A \vdash \Delta}{\Gamma \vdash \Delta}\ \text{cut}$$

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\ \wedge_L \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}\ \wedge_R$$

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}\ \vee_L \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta}\ \vee_R$$

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \Rightarrow B \vdash \Delta}\ \Rightarrow_L \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta}\ \Rightarrow_R$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, \exists x.A \vdash \Delta}\ \exists_L \qquad \frac{\Gamma \vdash [t/x]A, \exists x\ A, \Delta}{\Gamma \vdash \exists x.A, \Delta}\ \exists_R$$

$$\frac{\Gamma, \forall x.A, [t/x]A \vdash \Delta}{\Gamma, \forall x.A \vdash \Delta}\ \forall_L \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash \forall x A, \Delta}\ \forall_R$$

Fig. 1 Deduction Rules of the Classical First-Order Sequent Calculus G3

Some properties of interest of a rewriting relation are the following:

Definition 4 (Confluence, Termination) A rewrite system is said to be:

- *confluent* if for any terms, respectively formulas, such that $t \equiv u$, there exists a term, respectively a formula, v, such that $t \to^* v$ and $u \to^* v$.
- *terminating* if any term or formula rewriting sequence $t_1 \to \cdots \to t_n \to \cdots$ is finite.

Confluence and termination imply that, given two terms (or formulas) t and u, the relation $t \equiv u$ is decidable.

2.3 First-Order Logic Modulo Theory

Rewrite systems $\mathcal{RE}$ will be considered as a parameter of deduction systems that rely on Deduction modulo theory, as well as theories can be seen as a parameter when we reason within axiomatic theories. We can therefore reason modulo a rewrite system for arithmetic, for set theory, etc. Building on Definition 3 and the philosophy developed in the introduction, we aim to freely juggle convertible formulas, while reasoning on them.

We will show how this can be done within sequent calculus for classical logic, whose deduction rules are summarized in Figure 1. We present the calculus G3, which does not need structural rules (contraction, weakening). The next two paragraphs sketch the features of this calculus. For more details, see [99].

A sequent is a pair of sets of formulas, the hypotheses and the conclusions respectively, separated by the turnstile symbol, denoted by $\Gamma \vdash \Delta$, and whose meaning is "from the conjunction of hypotheses Γ follows one of the conclusions of Δ (more precisely, its disjunction)". Dealing with sets forbids repetition of

$$\frac{\Gamma, B \vdash \Delta}{\Gamma, A \vdash \Delta}\ \mathrm{conv}_L, A \equiv B \qquad \frac{\Gamma \vdash B, \Delta}{\Gamma \vdash A, \Delta}\ \mathrm{conv}_R, A \equiv B$$

Fig. 2 The Conversion Rules of G3conv

formulas, this detail is not relevant in this paper, note only that Γ and Δ are allowed to be empty. A sequent rule reads bottom-up. For example, the $\wedge_R$-rule can be read as "to show the *conclusion sequent* $\Gamma \vdash A \wedge B, \Delta$ from Γ, we have to show both *premises* $\Gamma \vdash A, \Delta$ and $\Gamma \vdash B, \Delta$". Therefore, proofs are complete when no branch of the *proof tree* is open, with the sole axiom rule being able to close a branch. Finally, the variable x in the $\forall_R$ and $\exists_L$ rules obeys the *eigenvariable condition*, it must not be free in Γ, Δ ($x \notin FV(\Gamma, \Delta)$), if it is not the case, we need to rename (α-convert) x before applying the rule.

Lastly, the axiom rule involves only atomic formulas, as is the case in G3. This condition can be relaxed. Along with the axiom rule, the only other rule not dealing with a connective or a quantifier is the *cut rule*. It may be included, or not, in the sequent calculus. Its admissibility (and its elimination) is of paramount importance in logic, for practical as well as theoretical reasons [106]: to cite a few, in the presence of the cut rule, the proof search space blows up immediately and the calculus cannot be directly shown consistent, and its elimination corresponds, through the Curry-De Bruijn-Howard correspondence, to program termination.

We can integrate conversion modulo $\mathcal{RE}$ to the deduction rules by adding the left and right conversion rule of Figure 2 to the rules of Figure 1.

The rules conv_L and conv_R are the two *computation rules* of the classical sequent calculus modulo theory G3conv, and they are separated from the deduction rules. However, we usually prefer a more seamless embedding of computation *inside* deduction rules, therefore merging the conversion rules with the deduction rules, as presented in Figure 3, where the same eigenvariable proviso holds on the $\forall_R$ and $\exists_L$ rules. The calculus $\mathrm{G3}_{\equiv}$ reflects the spirit of *identification* of equivalent formulas modulo $\mathcal{RE}$: we are allowed to perform deduction on any representative of an equivalence class.

The different variants of sequent calculus, and the different ways to add rewriting modulo theory to it, are all equivalent with respect to provability, at least when considered with the cut rule. Without the cut rule, major variations, such as allowing only forward rewriting ($C \to^* A \wedge B$ in the $\wedge_L$ and $\wedge_R$ rules), may require specific properties of the rewrite system, such as confluence or cut admissibility, to guarantee system equivalence [58, 40].

Using a pattern very similar to the one just described for sequent calculus, it is possible to embed conversion by using the rewriting relation within natural deduction [61].

$$\frac{}{\Gamma, A \vdash B, \Delta}\ \text{ax, if } A \equiv B \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma \vdash \Delta}\ \text{cut, if } A \equiv B$$

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, C \vdash \Delta}\ \wedge_L\text{, if } C \equiv A \wedge B \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash C, \Delta}\ \wedge_R\text{, if } C \equiv A \wedge B$$

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, C \vdash \Delta}\ \vee_L\text{, if } C \equiv A \vee B \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash C, \Delta}\ \vee_R\text{, if } C \equiv A \vee B$$

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, C \vdash \Delta}\ \Rightarrow_L\text{, if } C \equiv A \Rightarrow B \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash C, \Delta}\ \Rightarrow_R\text{, if } C \equiv A \Rightarrow B$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, C \vdash \Delta}\ \exists_L\text{, if } C \equiv \exists x.A \qquad \frac{\Gamma \vdash [t/x]A, C, \Delta}{\Gamma \vdash C, \Delta}\ \exists_R\text{, if } C \equiv \exists x.A$$

$$\frac{\Gamma, C, [t/x]A \vdash \Delta}{\Gamma, C \vdash \Delta}\ \forall_L\text{, if } C \equiv \forall x.A \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash C, \Delta}\ \forall_R\text{, if } C \equiv \forall x.A$$

Fig. 3 Deduction Rules of the Classical First-Order Sequent Calculus Modulo Theory $\mathrm{G3}_{\equiv}$

2.4 Practical Aspects of Proof Search Modulo Theory

Proof search strategies have to mix instantiation of quantifiers with rewrite steps and take into account *narrowing*, if we want to target completeness of the proof search. However, note that completeness is not necessarily desirable when it comes to practical performances.

Definition 5 (Narrowing Relation) Let $\mathcal{R}$ be a rewrite system, and t, u two terms. t *is narrowed into* u using substitution σ, denoted by $t \rightsquigarrow^{\sigma} u$, if and only if t is not a variable and:

- either there exists a term rewrite rule $l \to r \in \mathcal{R}$ (renamed so that l and t do not share any variable), such that $\sigma(t) = \sigma(l)$ and $u = \sigma(r)$;
- or $t = f(t_1, \cdots, t_n)$, $u = f(u_1, \cdots, u_n)$, and $t_i \rightsquigarrow^{\sigma} u_i$ for exactly one index i, while u_i is identical to $\sigma(t_i)$ for the rest of the indices.

Let A, B be two formulas. A *is narrowed into* B using substitution σ, denoted by $A \rightsquigarrow^{\sigma} B$, if and only if:

- either there exists a proposition rewrite rule $l \to r \in \mathcal{R}$ (renamed so that l and A do not share any variable), such that $\sigma(A) = \sigma(l)$ and $B = \sigma(r)$;
- or $A = P(t_1, \cdots, t_n)$, $B = P(u_1, \cdots, u_n)$, and $t_i \rightsquigarrow^{\sigma} u_i$ for exactly one index i, while u_i is identical to $\sigma(t_i)$ for the rest of the indices;
- or A and B are compound formulas with the same main connective/quantifier, and exactly one pair of the corresponding subformulas is narrowed one into another, the rest of the corresponding subformulas being identical after applying σ.

In other words, $\rightsquigarrow$ is like $\to$ except that it makes use of unification instead of pattern matching on the left-hand side.

Some introductory examples have already been discussed in Section 2.1, and real-world examples will be the matter of further developments. In this section, we want to illustrate a few points that we have to care about.

A typical example of equational axioms are axioms stating commutativity or associativity of some function symbols, for instance, writing the binary symbol + in an infix fashion:

$$x + y = y + x \qquad x + (y + z) = (x + y) + z$$

Now, let us consider the rewrite rule $x+0 \to x$ and the goal $\exists x.\exists y.x+y = x$. Two ingredients are necessary to prove this formula:

- We need to instantiate y by 0, which means, from an automated deduction point of view, trying to unify the expression $x + y$ with rewrite rules that can apply to it, since the symbol + is a symbol that we can rewrite on. It becomes even more complex if we need to perform narrowing modulo the equations just described above, for instance if the goal is $\exists x.\exists y.x + y = y$.
- We also need to keep on rewriting in the middle of the proof. In particular, pre-normalizing the goal is not sufficient since some term instantiations may trigger new rewrite rules. Moreover, as we have also propositional rewriting, an atomic formula may generate a compound one.

Therefore, deciding when, where, and how far to rewrite is fully part of the proof search strategy. For instance, narrowing is costly and might be only partially implemented, e.g., on some symbols or at some places only (specific occurrences in specific predicates), at the price of losing completeness.

2.5 Polarized Rewriting in First-Order Logic Modulo Theory

A variant of Deduction modulo theory, implemented in iProverModulo and therefore considered in this paper, is *polarized* Deduction modulo theory, where rewrite rules apply only on formulas to the right-hand or left-hand side of sequents, depending on their *polarity* (we could also say, laterality).

Definition 6 (Positive Occurrence) An occurrence of a formula F in a formula G is said to be positive if it appears under negation or on the left of an implication, an even number of times (including zero). Otherwise, it is said to be negative.

For instance, in $(\neg A) \Rightarrow B$, both B and A have positive occurrences, while $\neg A$ has a negative occurrence.

We refine the rewrite relation of Definition 3 into positive and negative rewriting. This obviously applies only to formulas.

Definition 7 (Polarized Rewriting) Let $\mathcal{RE}$ be a rewrite system, such that the propositional rewrite rules of $\mathcal{R}$ are partitioned into $\mathcal{R}^+ \cup \mathcal{R}^-$, the positive and negative rewrite rules. Let A, B be two formulas, A positively rewrites to B, denoted by $A \to_+ B$ and, respectively, A negatively rewrites to B, denoted by $A \to_- B$, if and only if:

- either there exist a proposition rewrite rule $l \to r \in \mathcal{R}^+$ (respectively, $l \to r \in \mathcal{R}^-$) and a substitution σ, such that $A =_{\mathcal{E}} \sigma(l)$ and $B =_{\mathcal{E}} \sigma(r)$;

$$\frac{}{\Gamma, A \vdash B, \Delta}\ \text{ax},\quad \text{if } A \to_-^* C \text{ and } B \to_+^* C \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma \vdash \Delta}\ \text{cut},\quad \text{if } C \to_+^* A \text{ and } C \to_-^* B$$

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, C \vdash \Delta} \wedge_L, \text{ if } C \to_-^* A \wedge B \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash C, \Delta} \wedge_R, \text{ if } C \to_+^* A \wedge B$$

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, C \vdash \Delta} \vee_L, \text{ if } C \to_-^* A \vee B \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash C, \Delta} \vee_R, \text{ if } C \to_+^* A \vee B$$

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, C \vdash \Delta} \Rightarrow_L, \text{ if } C \to_-^* A \Rightarrow B \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash C, \Delta} \Rightarrow_R, \text{ if } C \to_+^* A \Rightarrow B$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, C \vdash \Delta} \neg_L, \text{ if } C \to_-^* \neg A \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash C, \Delta} \neg_R, \text{ if } C \to_+^* \neg A$$

$$\frac{\Gamma, B \vdash \Delta}{\Gamma, A \vdash \Delta} \exists_L, \text{ if } A \to_-^* \exists x B \qquad \frac{\Gamma \vdash C, A, \Delta}{\Gamma \vdash A, \Delta} \exists_R,\quad \text{if } A \to_+^* \exists x B \text{ and } [t/x]B \to_+^* C$$

$$\frac{\Gamma, A, C \vdash \Delta}{\Gamma, A \vdash \Delta} \forall_L,\quad \text{if } A \to_-^* \forall x B \text{ and } [t/x]B \to_-^* C \qquad \frac{\Gamma \vdash B, \Delta}{\Gamma \vdash A, \Delta} \forall_R, \text{ if } A \to_+^* \forall x B$$

Fig. 4 Deduction Rules of the Sequent Calculus Modulo Polarized Theory G3$\equiv_+$

- or $A = P(t_1, \cdots, t_n)$, $B = P(u_1, \cdots, u_n)$, and $t_i \to u_i$ for exactly one index i, while t_j is identical to u_j for any other j;
- or A and B are compound formulas with the same main connective/quantifier, and exactly one pair of the corresponding subformulas rewrites one to another. The said subformulas must rewrite positively (respectively, negatively), except under a negation or at the left of an implication, where they must rewrite negatively (respectively, positively). The remaining corresponding subformulas have to be identical.

$\to_+^*$ and $\to_-^*$ are the reflexive and transitive closure of $\to_+$ and $\to_-$ respectively.

The polarized sequent calculus modulo theory is presented in Figure 4. Generalizing the convertibility side condition with a $\equiv_+$ relationship is of limited interest in practice. Actually, $\equiv_+$ cannot be a congruence relation, as symmetry cannot hold. Instead, $\equiv_+$ should be defined as the reflexive and transitive closure of $\to_+ \cup {}_-\!\leftarrow$, where $A \ {}_-\!\leftarrow B$ is defined as $B \to_- A$: "backward negative rewriting" and "forward positive rewriting" are on the same side. This corresponds to the intuition that while, in Deduction modulo theory, the rewrite rule $A \to B$ stands for the axiom $\forall \overline{x}.A \Leftrightarrow B$ (the universal closure of the formula), in polarized Deduction modulo theory, $A \to_+ B$ stands for the axiom $\forall \overline{x}.B \Rightarrow A$ and $A \to_- B$ stands for the axiom $\forall \overline{x}.A \Rightarrow B$. This symmetric role of $\to_+$ and $\to_-$ can also be seen in the properties of polarized sequent calculus modulo theory, in particular the cut and axiom rules.

Although polarized Deduction modulo theory leads to more elegant presentations of theories in which there are implicational axioms, it can be related to usual Deduction modulo theory:

Theorem 1 ([40, Corollary 10]) *Given a polarized rewriting system $\mathcal{R}$, there exists an unpolarized rewriting system $\mathcal{R}^{\mp}$ such that provability (resp.*

provability without cut) in $G3_{\equiv_+}$ modulo $\mathcal{R}$ coincides with provability (resp. provability without cut) in $G3_{\equiv}$ modulo $\mathcal{R}^{\mp}$ (with forward rewriting only).

2.6 Polymorphic First-Order Logic Modulo Theory

We now switch to a more formal presentation of the syntax and type system of polymorphic first-order logic, denoted by Poly-FOL. This presentation is an adaptation of [22,23].

2.6.1 Syntax

The syntax of Poly-FOL is given in Figures 5 and 6 for types, type schemes (that bind type variables) used for polymorphic symbols, terms, formulas, and type-quantified formula. Functions and predicates may now be polymorphic and bear type arguments. In this context, formulas may be quantified over types, as long as these quantifications are universal and come at the very head of formulas.

We let Type be the type of types and assume that each type is inhabited, at least by an infinity of variables of this type.

We write f for $f()$ when f has arity 0. In addition, we may sometimes denote by $\vec{\alpha}$ and $\vec{x}$ lists of respectively type and term variables $\alpha_1 \dots \alpha_m$ and $x_1 \dots x_n$ when parameters m and n are known from the context.

Instead of defining, rather informally, a signature Σ, as in Sections 2.2 and 2.3, we follow a type-theoretic fashion and explicitly give the notion of local and global contexts in Figure 6, as pairs composed of a symbol and a type. Having contexts gives us the possibility to formalize the typing and well-formed context judgments in the subsequent Section 2.6.2. The global context Γ_{G} contains constant symbols, the equivalent of a signature, while the local context Γ_{L} contains variable symbols. In the declaration of type constructors, m is the arity of the constructor.

2.6.2 Typing

In the following, given an expression e, term or formula, we denote by $\mathrm{FV}_{\mathrm{T}}(e)$ the set of type variables occurring freely in e, either in type arguments of polymorphic symbols or in the types of variables, and $\mathrm{FV}(e)$ the set of term variables occurring freely in e.

A formula A is said to be *monomorphic* if it is not a type-quantified formula and if $\mathrm{FV}_{\mathrm{T}}(A)$ is empty. Otherwise, the formula is said to be *polymorphic*. A formula A is said to be *closed* if both $\mathrm{FV}_{\mathrm{T}}(A)$ and $\mathrm{FV}(A)$ are empty.

We define by mutual induction the well-formedness judgment $\mathsf{wf}(\Gamma)$, whose inference rules are presented in Figure 7, meaning that the context $\Gamma := \Gamma_{\mathrm{G}}; \Gamma_{\mathrm{L}}$ is well-formed, and the typing judgment $\Gamma \vdash t : \tau$, whose inference rules are presented in Figure 8, meaning that the term t is well-typed and of type τ in the well-formed context Γ. A consequence of those rules is that for any well-formed global context Γ_{G}, $\mathrm{FV}_{\mathrm{T}}(\Gamma_{\mathrm{G}})$ is empty, as well as $\mathrm{FV}(\Gamma_{\mathrm{G}})$.

Type

$$\begin{array}{rcll} \tau & ::= & \alpha & \text{(type variable)} \\ & | & T(\tau_1, \ldots, \tau_m) & \text{(type constructor application)} \end{array}$$

Type Scheme

$$\begin{array}{rcll} \sigma & ::= & \Pi\alpha_1 \ldots \alpha_m.\ \tau_1 \times \ldots \times \tau_n \to \tau & \text{(function type signature)} \\ & | & \Pi\alpha_1 \ldots \alpha_m.\ \tau_1 \times \ldots \times \tau_n \to o & \text{(predicate type signature)} \end{array}$$

Term

$$\begin{array}{rcll} t & ::= & x & \text{(variable)} \\ & | & f(\tau_1, \ldots, \tau_m; t_1, \ldots, t_n) & \text{(function application)} \end{array}$$

Formula

$$\begin{array}{rcll} A & ::= & \top \mid \bot & \text{(true, false)} \\ & | & \neg A & \text{(negation)} \\ & | & A_1 \wedge A_2 \mid A_1 \vee A_2 & \text{(conjunction, disjunction)} \\ & | & A_1 \Rightarrow A_2 \mid A_1 \Leftrightarrow A_2 & \text{(implication, equivalence)} \\ & | & P(\tau_1, \ldots, \tau_m; t_1, \ldots, t_n) & \text{(predicate application)} \\ & | & \exists x : \tau.\ A & \text{(existential quantification)} \\ & | & \forall x : \tau.\ A & \text{(universal quantification)} \end{array}$$

Type Quantified Formula

$$\begin{array}{rcll} A_T & ::= & A & \text{(formula)} \\ & | & \forall\alpha.\ A_T & \text{(type quantification)} \end{array}$$

Fig. 5 Syntactic Categories of Poly-FOL

Local Context

$$\begin{array}{rcll} \Gamma_{\mathrm{L}} & ::= & \varnothing & \text{(empty context)} \\ & | & \Gamma_{\mathrm{L}},\ \alpha : \mathsf{Type} & \text{(type variable declaration)} \\ & | & \Gamma_{\mathrm{L}},\ x : \tau & \text{(term variable declaration)} \end{array}$$

Global Context

$$\begin{array}{rcll} \Gamma_{\mathrm{G}} & ::= & \varnothing & \text{(empty context)} \\ & | & \Gamma_{\mathrm{G}},\ T :: m & \text{(type constructor declaration)} \\ & | & \Gamma_{\mathrm{G}},\ f : \sigma & \text{(function declaration)} \\ & | & \Gamma_{\mathrm{G}},\ P : \sigma & \text{(predicate declaration)} \end{array}$$

Fig. 6 Contexts of Poly-FOL

2.6.3 Adding Rewriting

In addition to the many-sorted version of Section 2.3, we now have type variables. The approach of Definition 3 does not need substantial modifications. The notion of well-formedness for contexts of Figure 7 needs only the following additional derivation rule:

$$\frac{}{\mathsf{wf}(\varnothing;\varnothing)}\ \mathrm{WF}_1 \qquad \frac{x \notin \Gamma_\mathrm{L} \qquad \Gamma_\mathrm{G};\Gamma_\mathrm{L} \vdash \tau : \mathsf{Type}}{\mathsf{wf}(\Gamma_\mathrm{G};\Gamma_\mathrm{L}, x:\tau)}\ \mathrm{WF}_2$$

$$\frac{\alpha \notin \Gamma_\mathrm{L} \qquad \mathsf{wf}(\Gamma_\mathrm{G};\Gamma_\mathrm{L})}{\mathsf{wf}(\Gamma_\mathrm{G};\Gamma_\mathrm{L}, \alpha : \mathsf{Type})}\ \mathrm{WF}_3 \qquad \frac{T \notin \Gamma_\mathrm{G} \qquad \mathsf{wf}(\Gamma_\mathrm{G};\varnothing)}{\mathsf{wf}(\Gamma_\mathrm{G}, T :: m;\varnothing)}\ \mathrm{WF}_4$$

$$\frac{\begin{array}{l}\Gamma_\mathrm{G};\alpha_1 : \mathsf{Type},\ldots,\alpha_m : \mathsf{Type} \vdash \tau_i : \mathsf{Type}, i = 1\ldots n \\ f \notin \Gamma_\mathrm{G} \qquad \Gamma_\mathrm{G};\alpha_1 : \mathsf{Type},\ldots,\alpha_m : \mathsf{Type} \vdash \tau : \mathsf{Type}\end{array}}{\mathsf{wf}(\Gamma_\mathrm{G}, f : \Pi\alpha_1 \ldots \alpha_m.\ \tau_1 \times \ldots \times \tau_n \to \tau;\varnothing)}\ \mathrm{WF}_5$$

$$\frac{P \notin \Gamma_\mathrm{G} \qquad \Gamma_\mathrm{G};\alpha_1 : \mathsf{Type},\ldots,\alpha_m : \mathsf{Type} \vdash \tau_i : \mathsf{Type}, i = 1\ldots n}{\mathsf{wf}(\Gamma_\mathrm{G}, P : \Pi\alpha_1 \ldots \alpha_m.\ \tau_1 \times \ldots \times \tau_n \to o;\varnothing)}\ \mathrm{WF}_6$$

Fig. 7 Context Well-Formedness Rules for Poly-FOL

$$\frac{\alpha : \mathsf{Type} \in \Gamma}{\Gamma \vdash \alpha : \mathsf{Type}}\ \mathrm{TVar} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \mathrm{Var}$$

$$\frac{\Gamma \vdash \tau : \mathsf{Type} \qquad \Gamma, x : \tau \vdash A(x) : o}{\Gamma \vdash \varepsilon(x:\tau).\ A(x) : \tau}\ \varepsilon \qquad \frac{T :: m \in \Gamma \qquad \Gamma \vdash \tau_i : \mathsf{Type}, i = 1\ldots m}{\Gamma \vdash T(\tau_1,\ldots,\tau_m) : \mathsf{Type}}\ \mathrm{TCstr}$$

$$\frac{\begin{array}{ll}f : \Pi\alpha_1 \ldots \alpha_m.\tau_1 \times \ldots \times \tau_n \to \tau \in \Gamma & \Gamma \vdash \tau'_i : \mathsf{Type}, i = 1\ldots m \\ \rho = [\alpha_1/\tau'_1,\ldots,\alpha_m/\tau'_m] & \Gamma \vdash t_i : \rho(\tau_i), i = 1\ldots n\end{array}}{\Gamma \vdash f(\tau'_1,\ldots,\tau'_m;t_1,\ldots,t_n) : \rho(\tau)}\ \mathrm{Fun}$$

$$\frac{\begin{array}{ll}P : \Pi\alpha_1 \ldots \alpha_m.\tau_1 \times \ldots \times \tau_n \to o \in \Gamma & \Gamma \vdash \tau'_i : \mathsf{Type}, i = 1\ldots m \\ \rho = [\alpha_1/\tau'_1,\ldots,\alpha_m/\tau'_m] & \Gamma \vdash t_i : \rho(\tau_i), i = 1\ldots n\end{array}}{\Gamma \vdash P(\tau'_1,\ldots,\tau'_m;t_1,\ldots,t_n) : o}\ \mathrm{Pred}$$

$$\frac{}{\Gamma \vdash \top : o}\ \top \qquad \frac{}{\Gamma \vdash \bot : o}\ \bot \qquad \frac{\Gamma \vdash A_1 : o \qquad \Gamma \vdash A_2 : o}{\Gamma \vdash A_1 \wedge A_2 : o}\ \wedge$$

$$\frac{\Gamma \vdash A_1 : o \qquad \Gamma \vdash A_2 : o}{\Gamma \vdash A_1 \vee A_2 : o}\ \vee \qquad \frac{\Gamma \vdash A_1 : o \qquad \Gamma \vdash A_2 : o}{\Gamma \vdash A_1 \Rightarrow A_2 : o}\ \Rightarrow$$

$$\frac{\Gamma \vdash A_1 : o \qquad \Gamma \vdash A_2 : o}{\Gamma \vdash A_1 \Leftrightarrow A_2 : o}\ \Leftrightarrow \qquad \frac{\Gamma \vdash A : o}{\Gamma \vdash \neg A : o}\ \neg \qquad \frac{\Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 =_\tau t_2 : o}\ =$$

$$\frac{\Gamma, x : \tau \vdash A : o}{\Gamma \vdash \exists x : \tau.\ A : o}\ \exists \qquad \frac{\Gamma, x : \tau \vdash A : o}{\Gamma \vdash \forall x : \tau.\ A : o}\ \forall \qquad \frac{\Gamma, \alpha : \mathsf{Type} \vdash A_T : o}{\Gamma \vdash \forall \alpha.\ A_T : o}\ \forall_T$$

Fig. 8 Typing Rules for Poly-FOL

$$\frac{\Gamma_\mathrm{G};\Gamma_\mathrm{L} \vdash l : \tau \qquad \Gamma_\mathrm{G};\Gamma_\mathrm{L} \vdash r : \tau \qquad \begin{array}{l}\mathrm{FV_T}(r) \subseteq \mathrm{FV_T}(l) \\ \mathrm{FV}(r) \subseteq \mathrm{FV}(l)\end{array}}{\mathsf{wf}(\Gamma_\mathrm{G}, l \to_{\Gamma_\mathrm{L}} r;\varnothing)}\ \mathrm{WF}_7$$

In this way, rewrite rules are an integral part of the global context. This also implies the property that $\mathrm{FV_T}(r) \subseteq \mathrm{FV_T}(l) \subseteq \Gamma_\mathrm{L}$ and $\mathrm{FV}(r) \subseteq \mathrm{FV}(l) \subseteq \Gamma_\mathrm{L}$. A *rewrite system* $\mathcal{RE}$ is well-formed in a global context Γ_G if all the rewrite rules

are well-formed in Γ_{G}. In what follows, we consider only such rewrite systems. Note also that rewrite rules are not used in the typing rules of Figure 8 (see Section 2.7).

When applying a rewrite rule, we must now instantiate both term and type variables, which leads to the following modification of Definition 3. We assume given a global context Γ_{G}.

Definition 8 (Typed Rewriting Relation) Let $\mathcal{RE}$ be a rewrite system, and t, u two terms or formulas. t *rewrites to* u, denoted by $t \to u$ if and only if:

- either there exist a term or proposition rewrite rule $l \to_{\Gamma_L} r \in \mathcal{R}$, a well-typed type substitution ρ, and a well-typed term substitution σ, such that $t =_{\mathcal{E}} \sigma(\rho(t))$ and $u =_{\mathcal{E}} \sigma(\rho(r))$;
- or one of the other clauses of Definition 3 related to rewriting in subterms or subformulas is satisfied.

We do not explicitly present well-typedness rules for substitutions. They impose that, for each elementary substitution $[\tau/\alpha]$ that composes ρ, τ and α must have the same type in $\Gamma_{\mathrm{G}}; \Gamma_{\mathrm{L}}$, and for each elementary substitution t/x that composes σ, t and x must have the same type in $\Gamma_{\mathrm{G}}; \rho(\Gamma_{\mathrm{L}})$. Here, $\rho(\Gamma_{\mathrm{L}})$ denotes Γ_{L} where $\alpha : \mathsf{Type}$ is removed if α is in the domain of ρ, and substituted everywhere else. Extending Figure 8 is straightforward, but requires boilerplate technical results, for instance the derivability of $\mathsf{wf}(\Gamma_{\mathrm{G}}; \rho(\Gamma_{\mathrm{L}}))$.

The sequent calculus for Poly-FOL with rewriting is identical to the one of Figure 3 with two additional rules accounting for universal type quantification. These two new rules have no conversion side condition, because no rewrite rule acts on types.

2.7 Higher Type Systems Modulo Theory

Through the Curry-De Bruijn-Howard lens, the inference systems that we have overviewed previously are type systems. But, as type systems, they lack the precision and versatility offered by other facets of the λ cube [10]. We may also want to allow rewriting within the typing relation.

Several approaches have extended the "modulo theory" approach to typing. We can for instance cite CoqMT [103] or the Calculus of Algebraic Constructions [24]. An approach that follows the lines of Deduction modulo theory is the introduction of rewriting inside a logical framework based on typed λ-calculus with dependent types [70]. This results in the $\lambda\Pi$-calculus modulo theory [51] and has been implemented in the Dedukti tool [6,28,96].

Combining the computational power of β-reduction (function application) and rewriting results with the possibility to define *shallow embeddings* of many systems, including Coq [27], Matita [3], HOL Light [4,3,5], FoCaLiZe [46,47], etc. Dedukti is mature enough to be able to check large libraries coming from those systems. As it understands rewriting natively, it is the tool that we chose to double-check the proofs produced by our ATPs based on Deduction modulo theory and presented in Sections 3 and 4.

2.8 Soundness and Completeness

First-order Deduction modulo theory enjoys the standard Tarskian Boolean model semantics for classical logic, with the additional constraint that the rewrite system must be respected.

Definition 9 (Model of $\mathcal{RE}$) Let Σ be a signature, $\mathcal{RE}$ a rewrite system, D_s a family of sets indexed by sorts (the domain), and $\mathcal{B}$ a Boolean algebra. An interpretation of the signature Σ is given by mapping each function symbol $f/n : \langle s_1, \cdots, s_n, s_o\rangle$ to a function from $D_{s_1} \times \cdots \times D_{s_n}$ to D_{s_o} and each predicate symbol $P/n : \langle s_1, \cdots, s_n\rangle$ to a function from $D_{s_1} \times \cdots \times D_{s_n}$ to $\mathcal{B}$. It generates by induction an interpretation function, denoted by $[\![\ ]\!]$, over terms and formulas. This interpretation is parameterized by an assignment, mapping variables of sort s to elements of D_s.

An interpretation $[\![\ ]\!]$ is a *model of* $\mathcal{RE}$ if and only if, for any assignment φ, any terms t, u and formulas A, B, we have:

$$\begin{array}{rl} t \equiv u & \text{implies} \quad [\![t]\!]_\varphi = [\![u]\!]_\varphi \\ A \equiv B & \text{implies} \quad [\![A]\!]_\varphi = [\![B]\!]_\varphi \end{array}$$

With this notion of a model, we can state and prove the soundness and completeness theorems for sequent calculus. Notice that we do not only consider the Boolean algebra $\{0, 1\}$, although this algebra is initial. This enhanced flexibility is important for the constructivity of some of the proofs discussed in this section.

In particular, a Boolean algebra is equipped with an order relation $\leq$, and all algebras introduced here are complete, in the sense that arbitrary (and not only finite) lowest upper bounds and greatest lower bounds always exist. As not to further discuss semantics, which is not the core of this paper, we introduce syntactic arbitrary conjunctions and disjunctions $\bigwedge$ and $\bigvee$.

Theorem 2 (Soundness) *Let $\mathcal{RE}$ be a rewrite system and $\Gamma \vdash \Delta$ a sequent. Assume that it has a proof in sequent calculus modulo $\mathcal{RE}$.*

Then for any (complete) Boolean algebra, any domain D, any interpretation $[\![\ ]\!]$ that is a model of $\mathcal{RE}$, and any assignment φ, we have $[\![\bigwedge \Gamma]\!]_\varphi \leq [\![\bigvee \Delta]\!]_\varphi$.

This theorem is proved by a simple induction over the rules of Figure 3. This easily extends to proof systems like natural deduction or tableaux modulo theory.

Theorem 3 (Completeness) *Let $\mathcal{RE}$ be a rewrite system and $\Gamma \vdash \Delta$ a sequent. Assume that for any (complete) Boolean algebra, any domain D, any interpretation $[\![\ ]\!]$, and any assignment φ, we have $[\![\bigwedge \Gamma]\!]_\varphi \leq [\![\bigvee \Delta]\!]_\varphi$.*

Then $\Gamma \vdash \Delta$ has a proof in sequent calculus modulo $\mathcal{RE}$.

The idea behind such a theorem (in the case of sequent calculus) is to define a specific algebra, linked to provability. Instead of taking the Gödel-Henkin

path, we sketch the Lindenbaum construction, which takes advantage of the extra flexibility introduced above.

The elements of the Lindenbaum algebra are all the:

$$[A] := \{B \mid \text{both sequents } B \vdash A \text{ and } A \vdash B \text{ are provable}\}$$

for any closed formula A. This algebra is ordered by derivability ($A \leq B$ if and only if $A \vdash B$ has a proof), the least upper bound is disjunction, the greatest lower bound is conjunction, etc. The simplest choice for the domain D is the set of open terms, terms are merely interpreted by themselves[1]. The assignment φ can then be chosen to be the identity, mapping each term variable x to itself. Then, by the definition and properties of the Lindenbaum algebra, we get to show that $[\bigwedge \Gamma] = [\![\bigwedge \Gamma]\!]_\varphi \leq [\![\bigvee \Delta]\!]_\varphi = [\bigvee \Delta]$ and we can conclude that $\bigwedge \Gamma \vdash \bigvee \Delta$ is derivable, which by inversion [72] turns into a derivation of $\Gamma \vdash \Delta$.

Completeness, as it is stated in Theorem 3, holds for sequent calculus and natural deduction *for any rewrite system* $\mathcal{RE}$. However, in the general case, it does not hold for tableau calculi modulo theory, unless we allow the cut rule, neither does it hold for resolution modulo theory.

The statement of completeness can be strengthened to generate *cut-free* proofs of the sequent $\Gamma \vdash \Delta$. Such a statement entails cut admissibility by a direct combination with Theorem 2 [71]. This has a huge impact:

- It is impossible to prove such a theorem for an *arbitrary* rewrite system, even confluent and terminating [71]. Actually, this property depends on the rewrite system and is undecidable [40]. The simplest, nonterminating, rewrite system that does not enjoy cut elimination is composed of the unique rewrite rule $P \to P \Rightarrow Q$. In this system, $\vdash Q$ can be proved using a cut rule around formula P, but it cannot without cut.
- Cut-free completeness for sequent calculus and natural deduction modulo theory is equivalent to completeness of tableaux and resolution modulo theory, as those calculi translate to *cut-free* proofs [72].
- The Lindenbaum construction is too weak to show cut-free completeness: a cut is required to show transitivity of the order relation. To solve this, we can either refine the Gödel-Henkin approach, which results in tableau-like completeness proofs [31], or define the algebra differently, along the lines of Maehara [80], Okada [88], and Lipton and DeMarco [79]. Sketching such proofs, which can be found in [31,32] and depend on the rewrite system, is beyond the scope of this paper. Let us simply mention that the algebra contains *sets of contexts*, with base elements that are all the:

  $$\lceil A \rceil := \{\Gamma \mid \Gamma \vdash A \text{ is derivable without a cut}\}$$

  It is closed by arbitrary intersections and ordered by inclusion.

[1] This does not strictly conform to Definition 9, as two equivalents terms $t \equiv u$ must receive the same interpretation. This can be technically fixed by letting D be composed of *equivalence classes* of terms, or by interpreting terms by their normal forms (it requires confluence and termination), or by dropping the $[\![t]\!]_\varphi = [\![u]\!]_\varphi$ constraint *over terms* in Definition 9.

Thanks to Theorem 1, all the discussion about cut-free completeness in Deduction modulo theory applies to polarized Deduction modulo theory as well. There are systems for which it does not hold, the simplest one being:

$$P \to_- Q$$
$$P \to_+ \top$$

This system is a polarized version of the unpolarized $P \to P \Rightarrow Q$ above and, indeed, $\vdash Q$ can be proved using a cut rule around formula P, but it cannot without cut.

Given a rewriting system that does not enjoy cut-free completeness, there exist techniques to complete it to recover this property [40,39]. Those techniques are enhanced Knuth-Bendix completion procedures, that may not terminate, and geared towards cut admissibility instead of confluence. Also, any consistent first-order theory can be presented using a polarized rewriting system having this property [36].

If we are interested by a direct notion of model for polarized Deduction modulo theory, and direct completeness proofs, then we should weaken the notion of model presented in Definition 9.

Definition 10 (Model of a Polarized $\mathcal{RE}$) Let Σ be a signature, $\mathcal{RE}$ a rewrite system, such that $\mathcal{R}$ is partitioned into $\mathcal{R}^+ \cup \mathcal{R}^-$.

Consider an interpretation $[\![\ ]\!]$, defined as in Definition 9, in a Boolean algebra. Let $\leq$ be the induced ordering.

$[\![\ ]\!]$ is a model of *polarized* $\mathcal{RE}$ if and only if for any assignment φ, any terms t, u and formulas A, B, we have:

$$\begin{array}{rcll} t & \equiv & u & \text{implies} \quad [\![t]\!]_\varphi = [\![u]\!]_\varphi \\ A & \to_+ & B & \text{implies} \quad [\![A]\!]_\varphi \geq [\![B]\!]_\varphi \\ A & \to_- & B & \text{implies} \quad [\![A]\!]_\varphi \leq [\![B]\!]_\varphi \end{array}$$

Proving soundness and completeness theorems of $\mathrm{G3}_{\equiv_+}$ with respect to this notion of polarized model is a straightforward adaptation of the proofs of Theorem 2 and Theorem 3 above. Cut-free completeness proofs have not yet been investigated.

No first-order polymorphic semantics for Poly-FOL modulo theory has been introduced yet. As for polarized rewriting, one could use existing encoding of Poly-FOL in monomorphic calculi [22], as a first step towards indirect soundness and completeness results, and next, to develop a proper semantic framework.

3 The Zenon Modulo Automated Theorem Prover

This section presents the Zenon Modulo ATP, which is an extension to polymorphism and Deduction modulo theory of the Zenon tableau-based ATP [30].

Type

| Λ_{Type} (type metavariable)

Term

| X_τ (metavariable)
| $\varepsilon(x : \tau).A(x)$ (ε-term)

Formula

| $t_1 =_\tau t_2$ (term equality)

Fig. 9 Additions to the Syntactic Categories of Poly-FOL of Figure 5

$$\frac{}{\Gamma \vdash \Lambda_{\mathsf{Type}} : \mathsf{Type}}\ \text{TMeta} \qquad \frac{\Gamma \vdash \tau : \mathsf{Type}}{\Gamma \vdash X_\tau : \tau}\ \text{Meta}$$

Fig. 10 Additions to the Typing Rules of Poly-FOL of Figure 8

3.1 Specific Constructs

Zenon Modulo's features, whose behaviors are detailed later in this section, impose us to enlarge some categories and judgments of Section 2.6.

Zenon Modulo uses the expressions of Poly-FOL of Figure 5, with in addition the expressions of Figure 9. Proof search efficiency calls for metavariables (capitalized here, often named free variables in tableau-related literature), used to guess concrete instances through unification, as well as Hilbert's ε-terms at the term level ($\varepsilon(x : \tau).A(x)$ is a term meaning some x of type τ that satisfies $A(x)$, if it exists), an alternative to Skolem terms. The type level does not require ε-terms, since existential type quantification is beyond the scope of ML-polymorphism. In addition, Zenon Modulo has a specific treatment for the equality symbol, which is a polymorphic predicate symbol, noted $=$, of type $\Pi\alpha.\alpha \times \alpha \to o$. This predicate is introduced as a separate construct and we use the common infix notation $t_1 =_\tau t_2$, and $t_1 \neq_\tau t_2$ instead of $\neg(t_1 =_\tau t_2)$. These new constructs follow the additional typing rules of Figure 10.

3.2 Proof Search in Zenon Using Polymorphic Types

3.2.1 Proof Search Rules

The rules summarized in Figures 11 and 12 are an adaptation of the rules of Zenon [30] to typed formulas. We have omitted the unfolding and extension rules that were in [30]. Thanks to Deduction modulo theory, unfolding rules (which unfolds symbol definitions) are no longer relevant. The extension rule, which allows users to extend the deduction system of Zenon by defining lemmas,

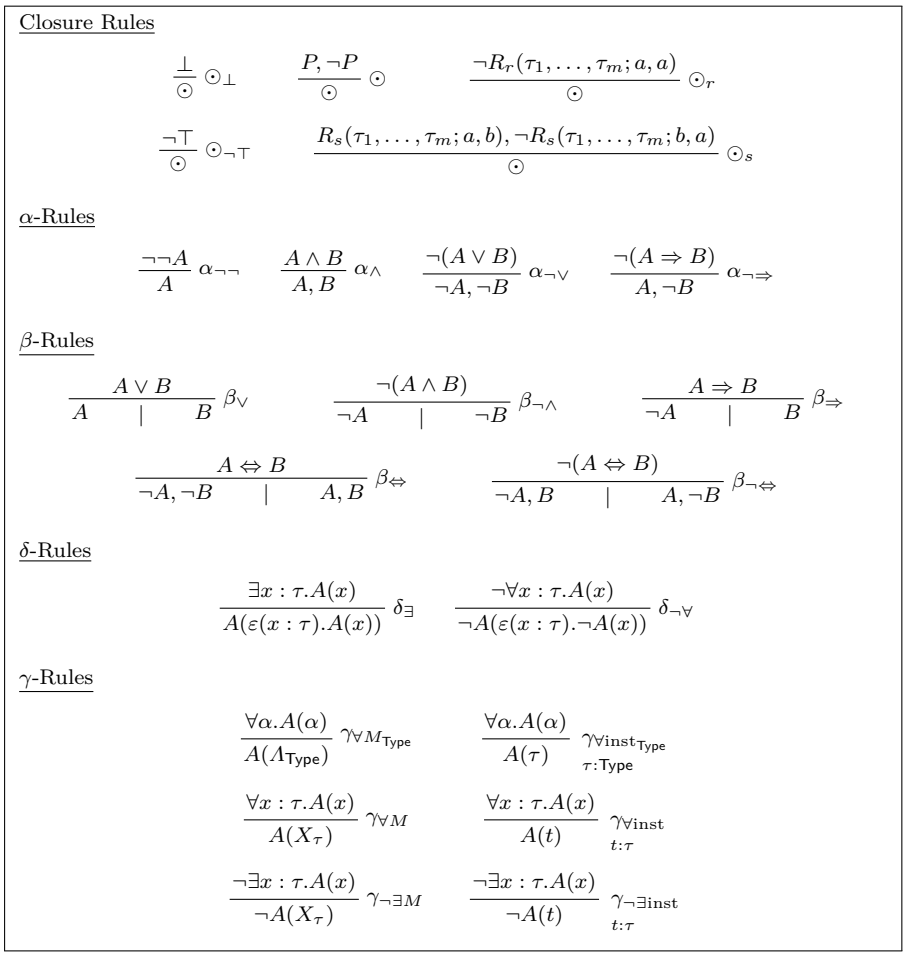

Fig. 11 Zenon Proof Search Rules (Part 1)

is unnecessary in our context. The "$\odot$" symbol is used for branch closure, while "|" separates two distinct nodes to be created, and R_r, R_s, R_t, and R_{ts} respectively denote reflexive, symmetric, transitive, and transitive-symmetric relations (predicates), including equality in particular. γ-rules now also include type quantification.

The proof search algorithm is an hybrid between the methods used by free-variable tableaux and plain tableaux [63]: starting from the negation of the goal, build a tree by applying the rules in a top-down fashion. When all branches end with a closure rule, the tree is closed, and it is a proof of the goal. Search is done in strict depth-first order: we close the current branch before we start working on another branch. Moreover, we work in a non-destructive way: extending a branch never changes the formulas elsewhere.

Relational Rules

$$\frac{P(\tau_1,\ldots,\tau_m;a_1,\ldots,a_n),\neg P(\tau_1,\ldots,\tau_m;b_1,\ldots,b_n)}{a_1 \neq_{\tau_1'} b_1 \quad | \quad \ldots \quad | \quad a_n \neq_{\tau_n'} b_n}\ \text{Pred}$$

$$\frac{f(\tau_1,\ldots,\tau_m;a_1,\ldots,a_n) \neq f(\tau_1,\ldots,\tau_m;b_1,\ldots,b_n)}{a_1 \neq_{\tau_1'} b_1 \quad | \quad \ldots \quad | \quad a_n \neq_{\tau_n'} b_n}\ \text{Fun}$$

$$\frac{R_s(\tau_1,\ldots,\tau_m;a,b),\neg R_s(\tau_1,\ldots,\tau_m;c,d)}{a \neq_\tau d \quad | \quad b \neq_\tau c}\ \text{Sym}$$

$$\frac{\neg R_r(\tau_1,\ldots,\tau_m;a,b)}{a \neq_\tau b}\ \neg\text{Refl}$$

$$\frac{R_t(\tau_1,\ldots,\tau_m;a,b),\neg R_t(\tau_1,\ldots,\tau_m;c,d)}{\begin{array}{c} c \neq_\tau a,\neg R_t(\tau_1,\ldots,\tau_m;c,a) \\ \mid b \neq_\tau d,\neg R_t(\tau_1,\ldots,\tau_m;b,d)\end{array}}\ \text{Trans}$$

$$\frac{R_{ts}(\tau_1,\ldots,\tau_m;a,b),\neg R_{ts}(\tau_1,\ldots,\tau_m;c,d)}{\begin{array}{c} d \neq_\tau a,\neg R_t(\tau_1,\ldots,\tau_m;d,a) \\ \mid b \neq_\tau c,\neg R_{ts}(\tau_1,\ldots,\tau_m;b,c)\end{array}}\ \text{TransSym}$$

$$\frac{a =_\tau b,\neg R_t(\tau_1,\ldots,\tau_m;c,d)}{\begin{array}{c} c \neq_\tau a,\neg R_t(\tau_1,\ldots,\tau_m;c,a) \\ \mid \neg R_t(\tau_1,\ldots,\tau_m;c,a),\neg R_t(\tau_1,\ldots,\tau_m;b,d) \\ \mid b \neq_\tau d,\neg R_t(\tau_1,\ldots,\tau_m;b,d)\end{array}}\ \text{TransEq}$$

$$\frac{a =_\tau b,\neg R_{ts}(\tau_1,\ldots,\tau_m;c,d)}{\begin{array}{c} d \neq_\tau a,\neg R_{ts}(\tau_1,\ldots,\tau_m;d,a) \\ \mid \neg R_{ts}(\tau_1,\ldots,\tau_m;a,d),\neg R_{ts}(\tau_1,\ldots,\tau_m;b,c) \\ \mid b \neq_\tau c,\neg R_{ts}(\tau_1,\ldots,\tau_m;b,c)\end{array}}\ \text{TransEqSym}$$

Fig. 12 Zenon Proof Search Rules (Part 2)

Unlike free-variable tableaux, Zenon avoids application of unifiers to the entire proof tree with an implicit unification mechanism that appeals to γ_{inst} rules. See Section 3.2.5 for more details. However, like free-variable tableaux, δ-rules introduce a form of Skolem terms with a relaxed freshness constraint, i.e. Hilbert's ε-terms.

Given an initially well-typed formula, it should be noted that the proof search rules generate only well-typed formulas. All these generated formulas can be typed in the empty local context (we use Church-style ε-terms, decorating the bound variables with their types, and the metavariables carry their types), which explains the simplified form of the typing side conditions.

3.2.2 Ordering Rule Application

The choice of the rule to apply is critical, as it has to ensure fairness and at the same time, it has to control the proof search space. As explained in [30] and depicted in Figure 11, the inference rules are divided into five classes. The proof

search algorithm of Zenon applies the rules with the following order relation $\prec$, where the relational rules of Fig. 12 are identified as β rules:

$$\odot \prec \alpha \prec \delta \prec \beta \prec \gamma$$

The rationale is simple. We start by trying to close a branch with a $\odot$ closure rule. If it is not possible, we pick a (yet unused) formula along the order $\prec$. Preferably, we apply an α rule, which deals with non-branching logical connectives. δ rules are also non branching and generate an ε-term. Otherwise, we try to apply a β rule, which deals with logical connectives but generates two branches.

If none of the previous rules was applicable, we pick one of the γ rules, which deals with quantification and generates a new metavariable. If all quantifiers have already generated a metavariable, then we try to apply another γ instantiating rule by generating a term. See Section 3.2.5 for more details regarding this step.

Fairness is ensured because formulas are eventually decomposed into literals or γ formulas, since the rules α, β, δ terminate. Any formula will therefore receive attention, except of course if a $\odot$ rule applies. Unsurprisingly, the only rules that are not terminating are the γ rules, which may generate an infinite number of new formulas. We also ensure that any such quantified formula is handled properly, first by generating *one* metavariable, which terminates, and then by generating different term instances, potentially infinitely many.

3.2.3 Pruning

Pruning [89] is a method to reduce the size of the proof tree as well as the proof search space. As explained in [30], when a branching node N has a closed subtree B as one of its branches, we determine which formulas are *useful* in B. If the formulas introduced by B are not useful in B, Zenon removes N and grafts B at its place since this subtree is valid without the application of the rule that introduced N. This locally closes the entire subtree. A formula is said to be *useful* in a subtree if it is one of the formulas closing a branch, or a formula that generated a useful formula through a rule application. An example is provided in Section 3.2.6.

3.2.4 Type Parameters in Proof Search Rules

The application of closure and relational rules dealing with more than one function or predicate symbol is conditioned by the value of the type parameters. For instance, the Pred rule of Figure 12:

$$\frac{P(\tau_1, \ldots, \tau_m; a_1, \ldots, a_n), \neg P(\tau_1, \ldots, \tau_m; b_1, \ldots, b_n)}{a_1 \neq_{\tau_1'} b_1 \quad | \quad \ldots \quad | \quad a_n \neq_{\tau_n'} b_n} \text{ Pred}$$

can be applied only if P and $\neg P$ bear the same m first type parameters $\tau_1, \ldots, \tau_m$. This rule generates n branches of the form $a_i \neq_{\tau_i'} b_i$, where τ_i' is the type of the two terms a_i and b_i. The fact that these type parameters have to be equal may be seen as a precondition to the application of this rule.

3.2.5 Dealing with Metavariables

We have introduced metavariables in Section 3.1. As explained in [30], Zenon uses them only to find candidate instances by unification.

Term metavariables are not used as variables in Zenon as they are never substituted. Instead, the following method is used. When we meet a universal formula of the form $\forall x : \tau.A$, we apply the $\gamma_{\forall M}$ rule and introduce a new metavariable X_τ, linked to this universal formula. Later, assume that we meet a potential contradiction like $X_\tau \neq_\tau a$, where $a : \tau$ is a closed term. Then, Zenon applies the $\gamma_{\forall \text{inst}}$ rule to instantiate $\forall x : \tau.A$ with a in the current branch. If this leads to a closed subtree rooted at the $\gamma_{\forall \text{inst}}$ node, pruning will then remove the nodes between the $\gamma_{\forall M}$ and $\gamma_{\forall \text{inst}}$ rules.

More generally, after an application of a Pred rule, term unification is performed explicitly in the proof tree through the relational rules of Figure 12 applied to the equality predicate and in particular the Fun rule. If unification succeeds, the branch is closed, otherwise we use the metavariables of the atomic equations $X_\tau \neq_\tau a$ as described above.

In the presence of polymorphism, type metavariables may also be introduced by the $\gamma_{\forall M_{\mathsf{Type}}}$ and $\gamma_{\neg\exists M_{\mathsf{Type}}}$ rules. Zenon's behavior for those metavariables is different since it would be irrelevant to look for possible contradictions of the form $\Lambda_{\mathsf{Type}} \neq \tau$.

Instead of a contradiction, Zenon looks for potential applications of the relational rules of Figure 12 to find type instances by unification. For example, given a type $\tau : \mathsf{Type}$ and a closed term $a : \tau$, assume that, in the global context of the current branch, there are the two formulas $P(\Lambda_{\mathsf{Type}}; X_{\Lambda_{\mathsf{Type}}})$ and $\neg P(\tau; a)$, where Λ_{Type} is a type metavariable and $X_{\Lambda_{\mathsf{Type}}}$ a term metavariable. We have a potential application of the Pred relational rule by replacing Λ_{Type} with τ. We apply the rule $\gamma_{\forall \text{inst}_{\mathsf{Type}}}$ to the type-quantified formula linked to the metavariable Λ_{Type}, and instantiate it with τ in the current branch. As with term metavariables, if this instance closes the subtree rooted at the $\gamma_{\forall \text{inst}_{\mathsf{Type}}}$ node, pruning will then remove useless nodes.

For both term and type metavariables, if the instantiations do not allow us to close the subtree, the original formulas with metavariables are still on the current branch. Hence, Zenon may generate as many instantiations as needed to find potential contradictions or application of relational rules. This allows us to avoid using iterative deepening to ensure completeness.

3.2.6 Example

As an example, assume a type $\tau : \mathsf{Type}$, two constants $a : \tau$ and $b : \tau$, and a predicate symbol P with the signature $P : \Pi\alpha.\alpha \times \alpha \to o$. Assuming that $\forall\alpha.\forall x : \alpha.\forall y : \alpha.P(\alpha; x, y)$, we aim to show that $P(\tau; a, b)$.

The proof, before the pruning of useless formulas, is given below:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall\alpha.\forall x:\alpha.\forall y:\alpha.P(\alpha;x,y),\neg P(\tau;a,b)}{\forall x:\Lambda_{\mathsf{Type}}.\forall y:\Lambda_{\mathsf{Type}}.P(\Lambda_{\mathsf{Type}};x,y)}\,\gamma_{\forall M_{\mathsf{Type}}}}{\textcircled{1}\ \forall y:\Lambda_{\mathsf{Type}}.P(\Lambda_{\mathsf{Type}};X_{\Lambda_{\mathsf{Type}}},y)}\,\gamma_{\forall M}}{\textcircled{2}\ P(\Lambda_{\mathsf{Type}};X_{\Lambda_{\mathsf{Type}}},Y_{\Lambda_{\mathsf{Type}}})}\,\gamma_{\forall \mathrm{inst}_{\mathsf{Type}}}}{\forall x:\tau.\forall y:\tau.P(\tau;x,y)}\,\gamma_{\forall M}}{\textcircled{3}\ \forall y:\tau.P(\tau;X_\tau,y)}\,\gamma_{\forall M}}{P(\tau;X_\tau,Y_\tau)}}{\textcircled{4}\ X_\tau\neq_\tau a \qquad Y_\tau\neq_\tau b}\,\mathrm{Pred}}{\forall y:\tau.P(\tau;a,y)}\,\gamma_{\forall \mathrm{inst}}}{P(\tau;a,Y'_\tau)}\,\gamma_{\forall M}}{a\neq_\tau a\ \odot}\,\odot_r \qquad \dfrac{\dfrac{Y'_\tau\neq_\tau b}{P(\tau;a,b)}\,\gamma_{\forall \mathrm{inst}}}{\odot}\,\odot}\,\mathrm{Pred}$$

We remark that, (1) when we introduce the formula $P(\Lambda_{\mathsf{Type}};X_{\Lambda_{\mathsf{Type}}},Y_{\Lambda_{\mathsf{Type}}})$, we would like to apply the Pred rule with $\neg P(\tau;a,b)$. But (2) Zenon first needs to instantiate the formula related to the type metavariable Λ_{Type} with the type τ. Then, (3) it generates some new (term) metavariables X_τ and Y_τ with the proper type. (4) We finally apply the Pred rule and identify the subterms of $P(\tau;X_\tau,Y_\tau)$ with those of $\neg P(\tau;a,b)$, which generates two branches. Then, Zenon reaches a potential contradiction with the formula $X_\tau \neq a$, thus it instantiates the formula that introduced X_τ with a. Doing the same with Y_τ, Zenon can finally close the local branch.

Thanks to the pruning of useless formulas, which includes the formulas with metavariables, the open right-hand side branch is erased, leading to the following proof tree:

$$\dfrac{\dfrac{\dfrac{\forall\alpha.\forall x:\alpha.\forall y:\alpha.P(\alpha;x,y),\neg P(\tau;a,b)}{\forall x:\tau.\forall y:\tau.P(\tau;x,y)}\,\gamma_{\forall \mathrm{inst}_{\mathsf{Type}}}}{\forall y:\tau.P(\tau;a,y)}\,\gamma_{\forall \mathrm{inst}}}{P(\tau;a,b)}\,\gamma_{\forall \mathrm{inst}}}{\odot}\,\odot$$

3.3 Extension of Zenon to Deduction Modulo Theory

3.3.1 Conversion Rule

Given a global typing context Γ_{G} and a rewrite system $\mathcal{RE}$ well-formed in Γ_{G}, extending Zenon to Deduction modulo theory then consists in adding to the proof search rules of Figures 11 and 12 the following conversion rule:

$$\frac{A}{B}\,\mathrm{conv}, A \equiv B$$

Such a rule trivially induces an infinite proof search space, but the proof search algorithm only considers confluent and terminating rewrite systems. This way, conversion is only used to produce normalized formulas during proof search. This step of normalization is performed after the application of each inference rule to all the formulas newly generated. The normalization procedure works as follows:

1. Rewrite only the literals, i.e. atomic formulas or their negation;
2. Normalize with respect to term rewrite rules;
3. Apply one step of proposition rewriting;
4. If the formula is still unchanged, quit; otherwise go back to 1.

This algorithm rewrites an atomic formula until we have either a normal form with respect to the rewrite system, or a non-atomic formula.

The metavariable instantiation mechanism of Section 3.2.5 also needs to be adapted: on a branch, we look for formulas A and B such that $A \equiv A'$, $B \equiv \neg B'$, and there exist a type substitution ρ and a term substitution σ such that $\sigma(\rho(A')) \equiv^t \sigma(\rho(B'))$, that is to say we look for convertibility modulo term rewriting and equational reasoning of the candidate instances of A' and B', while we additionally allow propositional rewriting on A and B to get A' and B'. σ and ρ are then used to perform instantiation as in Section 3.2.5.

To get a complete algorithm (see Section 3.4), we must also extend metavariable instantiation to the propositional narrowing of Section 2.4, extended to polymorphism: we look for a formula A, a type substitution ρ, and a term substitution σ, such that $A \equiv A'$, and there exist $A'_{|\omega}$ and a rule $l \to_{\Gamma_{\mathrm{L}}} r$ in $\mathcal{R}$ such that $\sigma(\rho(A'_{|\omega})) \equiv^t \sigma(\rho(l))$. Instantiation is then performed as in Section 3.2.5.

In what follows, the extension of Zenon to Deduction modulo theory will be called Zenon Modulo.

3.3.2 Example

Applying Deduction modulo theory to the example of Section 3.2.6 leads to the transformation of the axiom $\forall\alpha.\forall x : \alpha.\forall y : \alpha.P(\alpha; x, y)$ into the (proposition) rewrite rule $P(\alpha; x, y) \to_{(\alpha:\mathsf{Type},x:\alpha,y:\alpha)} \top$.

The proof produced by Zenon Modulo consists of one application of the conversion rule, followed by one application of a closure rule:

$$\dfrac{\neg P(\tau; a, b)}{\dfrac{\neg\top}{\odot}\,\neg\top}\,\mathrm{conv}, P(\tau; a, b) \equiv \top$$

It should be noted that the main benefit of Deduction modulo theory is to avoid proof search within sets of axioms. It strongly reduces the number of metavariables generated by Zenon Modulo, therefore the number of instantiations performed during proof search.

3.3.3 Generation of the Rewrite System

Turning axioms into rewrite rules is a key point in Deduction modulo theory. In Zenon Modulo, we propose two solutions to do so.

When dealing with a specific theory, users may define manually which axiom could be turned into a rewrite rule, in the sense that it is done outside Zenon Modulo. To do so, it is possible to tag axioms in input files using the special keyword "rewrite".

The second solution to turn axioms into rewrite rules is to rely on a heuristic. The main advantage of the heuristic is to be fully automatic. But it may also generate some inappropriate rewrite rules, leading to a rewrite system that does not enjoy expected properties, such as confluence and termination, and that does not allow us to have an efficient proof search.

We present, in the following, a heuristic, implemented in Zenon Modulo, which allows us to generate both term and proposition rewrite rules.

The main idea is to transform, into term rewrite rules, axioms of the form $\forall\vec{\alpha}.\forall\vec{x}.t = u$, where t is a term that is not a variable and u is any term, and to transform, into proposition rewrite rules, axioms of the form $\forall\vec{\alpha}.\forall\vec{x}.P \Leftrightarrow A$, where P is a predicate symbol and A is any formula.

However, we have to be more restrictive to avoid catching some particular kind of axioms, like those expressing commutativity properties of symbols, which would lead to immediate non-termination.

In the following, P denotes an atomic formula that is not an equality, A an arbitrary formula, t a term that is not a variable, and u an arbitrary term. In addition, we denote by $\mathrm{FV}(A)$ and $\mathrm{FV}(t)$ the union of the free term and type variables of A and t respectively.

For proposition rewrite rules, we have:

$$\begin{array}{lll} \forall\vec{\alpha}.\forall\vec{x}.P & \text{becomes} & P \to \top \\ \forall\vec{\alpha}.\forall\vec{x}.\neg P & \text{becomes} & P \to \bot \\ \forall\vec{\alpha}.\forall\vec{x}.P \Leftrightarrow A & \text{becomes} & P \to A \end{array}$$

The last rule is under the proviso that $\mathrm{FV}(A) \subseteq \mathrm{FV}(P) \subseteq \vec{\alpha} \cup \vec{x}$ and P is not unifiable with A or any subformula of A.

For term rewrite rules, we have:

$$\forall\vec{\alpha}.\forall\vec{x}.t = u \quad \text{becomes} \quad t \to u$$

provided that $\mathrm{FV}(u) \subseteq \mathrm{FV}(t) \subseteq \vec{\alpha} \cup \vec{x}$ and t is not unifiable with u or any subterm of u.

Verifying that the left-hand side of a rewrite rule is not unifiable with any subformula/subterm of the right-hand side allows us to eliminate some trivial cases of non-termination. Unfortunately, it does not guarantee that our final rewrite system is terminating since we do not test this criterion for all the rewrite rules. But this heuristic is terminating and rather efficient in practice, thus we consider it as a good compromise.

3.4 Soundness and Completeness

We do not have formally defined the notion of model for Poly-FOL modulo theory, so we cannot claim a soundness theorem. The soundness of Zenon Modulo is however clear, potential implementation bugs apart (addressed in Section 3.5).

Inspecting the rules of Figures 11 and 12, and the additional conversion rules of Zenon Modulo should be convincing enough. Actually, all of them, interpreted as non-polymorphic rules, are sound with respect to the first-order models of Definition 9, in the sense of Theorem 2. With a suitable extension of Definition 9 to Poly-FOL, we cannot see why soundness would not hold.

The question of the completeness of Zenon Modulo is more delicate. It relies at the first place on the completeness of Zenon itself, and depends on both the rewrite system and the implementation choices of Zenon Modulo.

The tableau calculus underlying non-polymorphic Zenon is straightforwardly complete. It is especially clear for the rules of Figure 11, if we forget for a moment the relational rules (including equality) of Figure 12. We have a standard tableau calculus, with unrestricted instantiation rules for quantifiers. Any off-the-shelf textbook completeness proof can be used [63,86]. Note that rules introducing metavariables are superfluous to show completeness.

We are unaware of any concrete completeness proof for the second set of relational rules of Figure 12, should it be semantic, with respect to Boolean first-order models extended to these relations, or syntactic, for instance with respect to a sequent calculus enjoying a cut rule and a cut-admissibility theorem. We however conjecture that an extension of the completeness proofs of the previous paragraph is easily achievable.

On the other hand, completeness of the *proof search algorithm* of Zenon, as presented in Section 3.2, is harder. Some of the authors made a few attempts several years ago and did not managed to come up with a complete proof. The understanding of Zenon's instantiation strategy, described in Section 3.2.5, which relies on metavariables, is a key factor, since it allows us to parsimoniously use instantiation rules. The less we instantiate, the harder it is to be complete. As far as we know, no completeness proof of this particular algorithm, even in simpler first-order cases, has been given.

Practical performances and pen-and-paper tests tend to show that Zenon's algorithm behave remarkably well and does not expose any weakness. For the rest of the section, we therefore make the assumption that this part of the algorithm is complete.

The rewrite system raises another issue, as underlined in Section 2.8. Some rewrite systems do not enjoy cut-free completeness, which easily boils down to non-completeness of the tableau calculus of Zenon Modulo : it corresponds to a sequent calculus without the cut rule. For the proof search algorithm, and even the tableau calculus, to be complete, we therefore also have to assume cut admissibility of the rewrite system, or to use the cut-admissibility results/criteria mentioned in Section 2.8.

Finally, a last hurdle for the completeness of Zenon Modulo is narrowing, see Definition 5 and Section 3.3.1. It has not been integrated to the current version

of Zenon Modulo. As seen in Section 2.4, the absence of narrowing on quantified variables or, in our case, on metavariables can preclude the application of some rewrite rules, and then prevent the completion of some proofs.

Early experiments on naively adding narrowing to Zenon Modulo showed that the unceasing unification and instantiation attempts of the rewrite strategy, and the spurious addition of instantiated formulas were degrading the performance of Zenon Modulo by an order of magnitude with no clear improvement in newly proved statements (see Section 5 for further discussions). Better narrowing and instantiation heuristics would be necessary and we left this delicate point for further work.

3.5 Proof Certification Using Dedukti

The quest for an efficient ATP is error-prone. This is why, as described in [55], Zenon Modulo enjoys a backend that outputs certificates for Dedukti [28], a universal proof checker for the $\lambda\Pi$-calculus modulo theory. Since it also relies on Deduction modulo theory, Dedukti natively deals with rewriting and is well-suited to verify the proofs of Zenon Modulo. In particular, we do not record the rewriting steps in the proofs (these steps are implicitly done by Dedukti), which makes these proofs quite compact. The logic of Dedukti is constructive, and calls for a translation initially based on an optimized double-negation translation (see [55]). This translation has been replaced by a more syntactical one using excluded middle explicitly (see [48]), which is more efficient in practice.

3.6 Experimental Results

3.6.1 Motivations

The development of Zenon Modulo was driven by its application to the B method [1], as part of the industrial research project BWare [56, 107]. The BWare project aims to improve the automated verification of Proof Obligations (POs for short) coming from the modeling of applications using the B method, i.e. logical formulas expressed in a particular set theory. The main distinctive feature of the B set theory compared to usual set theories, like Zermelo-Frankel's set theory, is the addition of typing constraints to expressions. Since the type system of the B method can be interpreted as a polymorphic type system [69], an encoding of the B set theory in an ML-like language called WhyML has been proposed in BWare. WhyML is the native language of Why3 [26], a platform dedicated to program verification and used in BWare to manage several automated reasoning tools like Zenon Modulo in particular.

3.6.2 Experimental Protocol

To assess the several extensions of Zenon, we use a benchmark of POs provided by the industrial partners of the BWare project. This benchmark comes from

4 anonymized industrial projects that were selected by the industrial partners for the representativeness of their POs. The POs are not necessarily tricky mathematical properties, their difficulty comes from their size, the large context provided, or the number of quantified variables (the mean size of the statements of these POs in TFF1 format is 515 KiB, with a maximum of 2,690 KiB). To run the tests, we rely on the BWare verification platform, which we outline briefly. The POs are initially produced by Atelier B [50] (an industrial tool supporting the B method). They are then translated into Why3 files [26] using a Why3 encoding of the B set theory [82]. Next, from these files, the Why3 platform produces (through appropriate drivers) the POs for the automated deduction tools. Why3's B set theory is interpreted as a rewrite system (see [45]) for tools compliant with Deduction modulo theory, otherwise as axioms. As this theory appeals to polymorphism, the output format may be either the TFF1 format [23] of the TPTP community used for the first-order polymorphic ATPs (this translation is straightforward as there are very few differences between the polymorphic input language of Why3 and TFF1), or the regular FOF format with an encoding of the polymorphic layer [22] used for the other first-order ATPs. SMT solvers are also considered and the SMT-LIB format is used with the same encoding for polymorphism, except for Alt-Ergo, which features a native format for polymorphism, and which was also part of the BWare project (see below for more details about the tools considered in the BWare project).

3.6.3 Experimental Results

The benchmark of BWare consists of 12,876 POs[2], and the experiment was run on an Intel Xeon E5-2660 v2 2.20 GHz computer, with a timeout of 120 s and a memory limit of 1 GiB. The results are summarized in Table 1. In these results, the first table focuses on the results of five different versions of Zenon (using several extensions), mainly based on Zenon 0.8.0 and compared to the main prover (mp) of Atelier B 4.0. The second table compares the tools considered in BWare, i.e. mp, Zenon Modulo (i.e. Zenon using all the extensions introduced in the first table), iProverModulo v0.7+0.2 (an extension of iProver v0.7 to Deduction modulo theory; see Section 4 for more details), and Alt-Ergo 0.99.1, to a representative panel of first-order ATPs, such as Vampire 2.6 and E 1.8, and SMT solvers, like CVC4 1.4 and Z3 4.3.2. Among these tools, only Zenon with Deduction modulo theory (i.e. versions tagged with "M") and iProverModulo implement Deduction modulo theory, while only Zenon with types (i.e. versions tagged with "T") and Alt-Ergo rely on polymorphic types.

For both tables of Table 1, we provide the number of proved POs, the corresponding rate, and the cumulative time for the successfully proved POs (not measured for mp, since it is not possible to split the timeout by PO). The "Unique" line refers to the number of POs that are only proved by a given prover. In the second table, "Uniq. $_{(1)}$" ranges over the BWare tools (i.e. the tools considered in the BWare project), while "Uniq. $_{(2)}$" considers all the tools.

[2] This benchmark is publicly available at: `http://bware.lri.fr/`.

	All Tools (12,738/98.9%)					
#POs: 12,876	mp	Zenon	Zenon (T)	Zenon (T+A)	Zenon (T+M)	Zenon (T+M+A)
Proofs	10,995	337	6,251	7,406	10,340	12,281
Rate	85.4%	2.6%	48.5%	57.5%	80.3%	95.4%
Time (s)	-	2,316	14,452	18,514	31,665	31,689
Unique	329	0	0	0	34	946

	All Tools (12,797/99.4%)							
	BWare Tools (12,772/99.2%)				Other Tools			
#POs: 12,876	mp	Zenon Modulo	iProver Modulo	Alt-Ergo	Vampire	E	CVC4	Z3
Proofs	10,995	12,281	3,695	12,620	10,154	7,919	12,173	10,880
Rate	85.4%	95.4%	28.7%	98.0%	78.9%	61.2%	94.5%	84.5%
Time (s)	-	31,689	20,156	7,129	118,541	36,969	8,378	3,404
Uniq. (1)	109	4	0	65				
Uniq. (2)	84	0	0	13	0	0	1	12

T ≡ with types M ≡ with Deduction modulo theory A ≡ with arithmetic
Zenon Modulo ≡ Zenon (T+M+A)

Table 1 Experimental Results over the BWare Benchmark

Coverage is given on top of tables. In the second table, we distinguish the coverage for the BWare tools from the coverage for all the tools.

In the first table of Table 1, in addition to the regular version of Zenon, we present the extensions with (polymorphic) types (tagged with "T"), with types and arithmetic (tagged with "T" and "A"), with types and Deduction modulo theory (tagged with "T" and "M"), and with types, Deduction modulo theory, and arithmetic (tagged with "T", "M", and "A"), which is currently the regular version of Zenon Modulo[3]. The arithmetic extension [44] handles linear arithmetic formulas, and relies on the simplex algorithm to compute solutions for systems over rationals, as well as on the branch and bound method to deal with integer systems [49]. As can be observed, the more extensions we plug, the more POs we prove. The most significant gain is provided by the extension with types, where we get an increase of about 1755% compared to Zenon. Plugging Deduction modulo theory gives an additional increase of 65%. Finally, connecting arithmetic on top allows us to prove 20% more POs, and to improve by 10 percentage points on mp.

In the second table of Table 1, we observe that Zenon with types and Deduction modulo theory but without arithmetic (i.e. the version tagged with "T" and "M") obtains better results than the first-order ATPs Vampire and E with respect to the number of proved POs. Vampire remains close to Zenon (10,154 proofs compared to 10,340 proofs for Zenon), but Zenon appears to be about 4 times faster than Vampire over all the proved POs (with a cumulative time of 31,665 s compared to 118,541 s for Vampire). Similarly, Zenon Modulo (i.e.

[3] Available at: `http://zenon.gforge.inria.fr/`.

the version including all the extensions) proves more POs than the SMT solvers CVC4 and Z3, except Alt-Ergo. However, it should be noted that CVC4 is close to Zenon Modulo (12,173 proofs compared to 12,281 proofs for Zenon Modulo), and has a significant lower cumulative time (8,378 s compared to 31,689 s for Zenon Modulo). The low results obtained by iProverModulo (less than 30% of proved POs) can be explained by the encoding of polymorphism (as described in Section 3.6.2), which hampers the analysis of the theory and the generation of a rewrite system similar to the one found by Zenon Modulo (which offers full support for polymorphism).

The Dedukti backend mentioned in Section 3.5 deals with all the proofs produced by Zenon Modulo that do not involve arithmetic (i.e. 10,340 proofs), and all these proofs have been verified and therefore approved by Dedukti.

4 The iProverModulo Automated Theorem Prover

In this section, we present the iProverModulo ATP, which is an extension of the iProver resolution- and instantiation-based ATP [78] to include Deduction modulo theory.

4.1 Basic Definitions and Notations

As usual, literals are atomic propositions or their negations; clauses are sets of literals. A clause $C = \{L_1, \ldots, L_n\}$ can be identified with the formula $\ulcorner C \urcorner = \forall \vec{x}.\ L_1 \vee \cdots \vee L_n$, where $\vec{x} = \mathrm{FV}(L_1, \ldots, L_n)$. Such a formula is said to be clausal. It is folklore that any first-order formula A can be transformed into an equi-satisfiable formula that is a conjunction of clausal formulas. We define the clausal normal form of A, denoted by $\mathcal{C}\ell(A)$, as the set of the clauses corresponding to these clausal formulas.

4.2 Resolution Modulo Theory

One of the original motivations of Deduction modulo theory was to design automated theorem proving methods modulo theories. The introductory paper on Deduction modulo theory [60] therefore already presents a resolution method called ENAR, which stands for Extended Narrowing And Resolution. The main addition of ENAR to ordinary resolution is the extended narrowing rule, which deals with proposition rewrite rules:

$$\textsf{Ext. Narr.}\ \frac{L \vee C}{\sigma(D \vee C)}\ L \rightsquigarrow^{\sigma} A,\ D \in \mathcal{C}\ell(A)$$

where $\rightsquigarrow$ is the narrowing relation presented in Definition 5.

The drawback of this rule is that $\mathcal{C}\ell(A)$, the clausal normal form of A, has to be computed dynamically each time the rule is applied. To avoid this, Dowek [59]

refined ENAR into Polarized Resolution Modulo Theory (PRMT). In PRMT, proposition rewrite rules are polarized as explained in Section 2.5: they are tagged with a polarity + or −, and positive rules can only be applied at positive positions whereas negative rules can only be applied at negative positions. In addition, these polarized rewrite rules are assumed to be clausal: negative rules are of the form $P \to^- C$ and positive rules are of the form $P \to^+ \neg C$, where C is clausal. Consequently, in a refutational context, narrowing a clausal formula gives a formula that is already clausal, and the extended narrowing rule can be reduced to the two following instances:

$$\mathsf{Ext.\ Narr.}^- \frac{P \vee D}{\sigma(C \vee D)}\ \sigma = mgu(P, Q),\ Q \to^- C \in \mathcal{R}$$

$$\mathsf{Ext.\ Narr.}^+ \frac{\neg P \vee D}{\sigma(C \vee D)}\ \sigma = mgu(P, Q),\ Q \to^+ \neg C \in \mathcal{R}$$

where P and Q are assumed to be atomic.

Remark 1 In the original presentation of ENAR, equational axioms and *term* rewrite rules are handled using a constraint system, as in equational resolution of Plotkin [91]. However, because of the difficulty to implement them while avoiding clauses with unsatisfiable constraints, no first-order theorem prover uses such constraints as far as the authors know. We therefore preferred a presentation without equational axioms, and where term rewrite rules are handled as proposition rewrite rules. In this setting, we also have an extended narrowing rule for term rewrite rules:

$$\mathsf{Ext.\ Narr.}^{\mathsf{t}} \frac{L \vee C}{\sigma(L' \vee C)}\ L \rightsquigarrow^{\sigma} L' \text{ by a term rewrite rule}$$

To reduce the proof search space of PRMT, Burel [35] refined it with ordering constraints, as in ordered resolution. Given an ordering $\succ$, which is well-founded and stable by substitution, the rule $\mathsf{Ext.\ Narr.}^-$ is given the extra condition that P must be maximal in $P \vee C$, and $\mathsf{Ext.\ Narr.}^+$ is given the extra condition that $\neg Q$ must be maximal in $\neg Q \vee D$. Note that the rewriting relation does not need to be compatible with the ordering $\succ$. The whole calculus is presented in Figure 13. It was proved that Ordered Polarized Resolution Modulo Theory (OPRMT) is complete exactly in the same cases when ENAR and PRMT are. This depends on the rewrite system and it happens precisely when the cut rule is admissible in the sequent calculus modulo theory. See Section 2.8 for a discussion about cut-free completeness, and the end of Section 2.5 for the particular case of polarized Deduction modulo theory.

Theorem 4 ([35, Theorem 1]) *Given a set of clauses Γ, the sequent $\ulcorner \Gamma \urcorner \vdash$ has a proof without cut in $G3_{\equiv_+}$ if and only if the empty clause can be derived from Γ in OPRMT.*

Note that OPRMT can be complete even when the rewrite system is neither confluent nor terminating, since rewriting (or more precisely narrowing) is

$$\textsf{Resolution}\ \frac{P \vee C \qquad \neg Q \vee D}{\sigma(C \vee D)}\ (a),(b),(c) \qquad \textsf{Factoring}\ \frac{L \vee K \vee C}{\sigma(L \vee C)}\ (d)$$

$$\textsf{Ext. Narr.}^{-}\ \frac{P \vee C}{\sigma(D \vee C)}\ (a),(b),\ Q \to^{-} D \qquad \textsf{Ext. Narr.}^{+}\ \frac{\neg Q \vee D}{\sigma(C \vee D)}\ (a),(c),\ P \to^{+} \neg C$$

$$\textsf{Ext. Narr.}^{\mathsf{t}}\ \frac{L \vee C}{\sigma(L' \vee C)} \quad \begin{array}{l} L \text{ maximal in } L \vee C, \\ L \rightsquigarrow^{\sigma} L' \text{ by a term rewrite rule} \end{array}$$

$(a)\ \sigma = mgu(P, Q)$ $(b)\ P$ maximal in $P \vee C$ $(c)\ \neg Q$ maximal in $\neg Q \vee D$
$(d)\ L$ and K maximal in $L \vee K \vee C$, $\sigma = mgu(L, K)$

Fig. 13 Inference Rules of Ordered Polarized Resolution Modulo Theory

applied step by step, and for all rewrite rules in parallel. For instance, modulo the rewrite system consisting of two polarized rules $A \to^{+} \neg\neg A$ and $A \to^{+} \neg B$, OPRMT is complete. However, in practice, provided the *term* rewrite system is terminating and confluent, clauses can be normalized using this system, thus avoiding many applications of the extended narrowing rule (see Section 4.4).

4.3 Deviating Discount Loop to Simulate Resolution Modulo Theory

Implementing OPRMT relies on the following observation by Dowek [59]: applying extended narrowing on a clause $P \vee C$ with rewrite rule $Q \to^{-} D$ produces the same clause as resolving it with the clause $\neg Q \vee D$. Thus, the so-called *one-way clause* $\underline{\neg Q} \vee D$ can be associated with any negative rewrite rule $Q \to^{-} D$, and the one-way clause $\underline{P} \vee C$ can be associated with any positive rewrite rule $P \to^{+} \neg C$. If, instead of using the extended narrowing rule with polarized rewrite rules, we want to use resolution with the corresponding one-way clauses, we have to take the following two conditions into account:

- Since only the left-hand side of a rewrite rule can be used for narrowing, resolution can only be applied on the underlined literal of a one-way clause;
- Since it is not possible in PRMT to narrow a rewrite rule with another one, it is prohibited to apply resolution on two one-way clauses.

This is reminiscent of the set-of-support strategy for resolution [112], where the set of clauses is split into two parts : the theory and the set of support. Resolution must be applied to at least one clause of the set of support, and generated clauses are added to the set of support. Here, one-way clauses correspond to those that are in the theory part, whereas ordinary clauses are put in the set of support. The difference here is that the search space is even more restricted since resolution can only be applied on the underlined literals.

This relation with the set-of-support strategy can be exploited to embed OPRMT into a resolution-based prover using the given-clause algorithm, or one of its refinements, namely the Otter loop or the Discount loop (see [111] for an elaborate discussion of proof procedures for resolution). In Figure 14, we

```
Unprocessed := { Input clauses }
Processed := ∅
while Unprocessed ≠ ∅
  given := pick_one(Unprocessed)
  Unprocessed := Unprocessed \ { given }
  if given = □ // the empty clause
    return Unsatisfiable
  Processed := Processed ∪ { given }
  for p in Processed
    Unprocessed := Unprocessed ∪ inferences(given, p)
return Satisfiable
```

Fig. 14 Given-Clause Algorithm

recall the given clause algorithm. It proceeds by working on two separated sets of clauses: the set of unprocessed clauses and the set of processed ones. The key ingredient of the algorithm is to maintain the invariant that all non-redundant inferences between two clauses in the processed set have been applied. The resulting clause can be either in the unprocessed set or in the processed set, as long as it is not redundant. `inferences(c,d)` returns the set of clauses generated by applying all the inference rules of the calculus between the clauses `c` and `d`. Modifying this algorithm to introduce one-way clauses is relatively easy, since it only consists in placing the one-way clauses in the processed set from the beginning. That way, we ensure that no resolution will be performed on two of them. We also need to bypass the selection mechanism of the prover to make it select the underlined literal in the one-way clauses. The Otter and Discount loop differs from the given-clause algorithm by adding the possibility to simplify respectively all clauses or processed clauses. This does not affect how OPRMT can be embedded in it.

iProver is an ATP that relies on two calculi: ordered resolution on one hand, and instantiation-generation (Inst-Gen) on the other hand. Using the ideas developed above, we embedded OPRMT in the resolution part of iProver, which is based on an implementation of the Discount loop. It remained to implement an extended narrowing rule for the *term* rewrite rules. We performed this by traversing each term of the given clause to find potential positions where narrowing can occur. To reduce the number of unification attempts, the discrimination tree structure already implemented in iProver was reused to make an index of the term rewrite rules.

4.4 Efficient Normalization via Compilation of Rewrite Rules

Normalizing the clauses with regards to the rewrite system $\mathcal{R}$ is not enough to be complete. We need narrowing to instantiate clauses before rewriting them. However, provided the *term* rewrite system is confluent and terminating, it is safe to simplify a clause by replacing it with its normal form with respect to the term rewrite system, as given by the following simplification rule:

$$\mathsf{Demodulation}\ \frac{C}{\equiv \mathrm{D}}\ \text{if } C \to^* D \text{ by the term rewrite system}$$

Using this simplification rule avoids repeatedly applying Ext.Narr.^{t} to rewrite C into D, therefore producing less clauses.

We can think of several approaches to implement this normalization:

dtree Thanks to the implementation of Ext.Narr.^{t}, there is already a data structure that helps in retrieving rewrite rules whose left-hand side can be unified with some term. Since pattern matching is stronger than unification (if a term matches a pattern, then the term and the pattern can be unified), the same structure can be used to get candidates for matching. A tree traversal of the term has to be performed to search for matching candidates.

interp As iProver is written in OCaml, rewrite rules can be translated into OCaml closures (i.e. functions) performing the pattern matching. By structural induction on the left-hand side of the rule, we can build a function that matches its arguments with respect to the left-hand side and returns a substitution as follows:

```
let rec term_to_subst = function
| Fun(f, f_args, _) → (function
    Fun(g, g_args, _) when f = g →
      List.fold_left2
       (fun sub t1 t2 → merge_subst (term_to_subst t1 t2) sub)
       (Subst.create ()) f_args g_args
  | _ → raise No_match)
| Var(var,_) →
    let sub = Subst.create () in
      fun t → Subst.add var t sub
```

If the function is successful, the returned substitution is applied to the right hand side. Otherwise, we try another rewrite rule. If no rewrite rule can be applied at that position, we try the same method below in the term.

plugin The rewrite system is translated into an actual OCaml program that is compiled and linked to the iProverModulo executable using the plugin feature offered by the OCaml compiler (dynamic linking). For instance, the rules $f(X, g(X)) \to h(X)$ and $f(h(X), Y) \to Y$ are translated into the following code:

```
let match_term = function
| Fun("f",[x0; Fun("g",[x1])]) when x0 = x1 → Fun("h",[x0])
| Fun("f",[Fun("h",[x0]); y0]) → y0
| _ → raise No_match
```

This translation is fully automated. Note that there is no need to implement an efficient pattern-matching algorithm, since it is that of OCaml that will be used.

In [37], we compare these different methods. It turns out that using the plugin method adds an overhead of approximately 0.07s, corresponding to the time needed to write the OCaml file, invoke the OCaml compiler on it, and load the resulting plugin. However, this overhead is largely outweighed by the gain in normalization time that it produces, in particular on problems involving heavy computations.

Axioms	give rewrite rules	
$\forall \vec{x}.\ t = u$	$t \to u$	provided $FV(u) \subseteq FV(t)$
$\forall \vec{x}.\ P \Rightarrow A$	$P \to^- \forall \vec{y}.\ C$	for all $C \in \mathcal{C}\ell(A)$, with $\vec{y} = FV(C) \setminus FV(P)$
$\forall \vec{x}.\ A \Rightarrow P$	$P \to^+ \neg\forall \vec{y}.\ C$	for all $C \in \mathcal{C}\ell(\neg A)$, with $\vec{y} = FV(C) \setminus FV(P)$
$\forall \vec{x}.\ P \Leftrightarrow A$ $\forall \vec{x}.\ A \Leftrightarrow P$	the same as those of $\forall \vec{x}.\ P \Rightarrow A$ and $\forall \vec{x}.\ A \Rightarrow P$	
$\forall \vec{x}.\ \neg P \Rightarrow A$	$P \to^+ \neg\forall \vec{y}.\ C$	for all $C \in \mathcal{C}\ell(A)$, with $\vec{y} = FV(C) \setminus FV(P)$
$\forall \vec{x}.\ A \Rightarrow \neg P$	$P \to^- \forall \vec{y}.\ C$	for all $C \in \mathcal{C}\ell(\neg A)$, with $\vec{y} = FV(C) \setminus FV(P)$
$\forall \vec{x}.\ \neg P \Leftrightarrow A$ $\forall \vec{x}.\ A \Leftrightarrow \neg P$	the same as those of $\forall \vec{x}.\ \neg P \Rightarrow A$ and $\forall \vec{x}.\ A \Rightarrow \neg P$	
$\forall \vec{x}.\ P$	$P \to^+ \neg\bot$	
$\forall \vec{x}.\ \neg P$	$P \to^- \bot$	
$\forall \vec{x}.\ A \wedge B$	the same as those of $\forall \vec{x}.\ A$ and $\forall \vec{x}.\ B$	

Table 2 Transformation Rules of the Equiv Heuristic

4.5 Orienting Axioms into Rewrite Rules

As in Section 3.3.3 for Zenon Modulo, we are confronted with the problem of turning a theory presented by axioms into a rewrite system, with the extra condition that we need a polarized and clausal rewrite system. We can use the same kind of heuristics that we used for Zenon Modulo, but we can also design techniques that are proved to be complete, in the sense that the resulting rewrite system enjoys the cut-admissibility property. We devised three strategies to orient axioms into rewrite rules, which are named Equiv, ClausalAll, and Sat, and which are described below.

Equiv: This is a heuristic based on the shape of the formula, as in Section 3.3.3. The transformations are summarized in Table 2. Using *polarized* Deduction modulo theory allows us to cover a few more cases than Zenon Modulo, in particular axioms involving implications.

Example 1 If we apply the Equiv heuristic on the inclusion axiom in set theory (see Section 2.1)

$$\forall A, B.\ A \subseteq B \Leftrightarrow \forall x.\ x \in A \Rightarrow x \in B$$

according to the fourth line of Table 2, we have to compute the clausal normal form of $\forall x.\ x \in A \Rightarrow x \in B$, which contains only the clause $\neg(X \in A) \vee X \in B$, and the clausal normal form of $\neg(\forall x.\ x \in A \Rightarrow x \in B)$, which contains two singleton clauses $\mathrm{diff}(A, B) \in A$ and $\neg(\mathrm{diff}(A, B) \in B)$, where diff is a Skolem function symbol introduced to eliminate the negated universal quantifier. $\mathrm{diff}(A, B)$ can be seen as a counter-example of $A \subseteq B$ if it exists.

We thus obtain the following rewrite system:

$$A \subseteq B \to^- \forall x.\ \neg x \in A \vee x \in B$$
$$A \subseteq B \to^+ \neg\mathrm{diff}(A, B) \in A$$
$$A \subseteq B \to^+ \neg\neg\mathrm{diff}(A, B) \in B$$

Note that the double negation in the last rewrite rule has been kept to let the rewrite rules correspond to the associated one-way clauses (see Section 4.3):

$$\underline{\neg(A \subseteq B)} \vee \neg(X \in A) \vee X \in B$$
$$\underline{A \subseteq B} \vee \mathrm{diff}(A, B) \in A$$
$$\underline{A \subseteq B} \vee \neg(\mathrm{diff}(A, B) \in B)$$

ClausalAll: We rely on the completeness of the set-of-support strategy to define a complete heuristic to orient axioms. We assume that the theory is presented by means of a set of clauses. Otherwise, it has to be transformed into clausal normal form using standard techniques.

Definition 11 Given a set of clauses Γ, we define the polarized rewrite system $\mathcal{R}_\Gamma$ consisting of, for each clause C in Γ, for each literal L in C:

- if $L = P$ is positive, a positive rewrite rule $P \to^+ \neg\forall\vec{x}.\ L_1 \vee \cdots \vee L_m$, where $\vec{x}$ are the free variables of C that are not free in P, and $L_1, \ldots, L_m$ are the literals of C different from P;
- if $L = \neg P$ is negative, a negative rewrite rule $P \to^- \forall\vec{x}.\ L_1 \vee \cdots \vee L_m$, where $\vec{x}$ are the free variables of C that are not free in P, and $L_1, \ldots, L_m$ are the literals of C different from $\neg P$.

Example 2 Let Γ be the set of clauses corresponding to the above inclusion axiom in set theory. It is (unsurprisingly) identical to the three one-way clauses above, besides the lack of literal selection.

$$\neg(A \subseteq B) \vee \neg(X \in A) \vee X \in B$$
$$A \subseteq B \vee \mathrm{diff}(A, B) \in A$$
$$A \subseteq B \vee \neg(\mathrm{diff}(A, B) \in B)$$

Then $\mathcal{R}_\Gamma$ is:

$$A \subseteq B \to^- \forall x.\ \neg x \in A \vee x \in B$$
$$X \in A \to^- \forall b.\ \neg A \subseteq b \vee X \in b$$
$$X \in B \to^+ \neg\forall a.\ \neg a \subseteq B \vee X \in a$$
$$A \subseteq B \to^+ \neg\mathrm{diff}(A, B) \in A$$
$$\mathrm{diff}(A, B) \in A \to^+ \neg A \subseteq B$$
$$A \subseteq B \to^+ \neg\neg\mathrm{diff}(A, B) \in B$$
$$\mathrm{diff}(A, B) \in B \to^- A \subseteq B$$

Remark 2 The number of rewrite rules in $\mathcal{R}_\Gamma$ is equal to the number of literal occurrences in Γ.

We can prove that the rewrite systems obtained from axioms as shown above enjoy cut admissibility.

Theorem 5 ([36, Theorem 14]) *The consistency of a finite set of clauses Γ implies the admissibility of the cut rule in the polarized sequent calculus modulo the rewrite system $\mathcal{R}_\Gamma$.*

Sat: To reduce the number of rules generated by the ClausalAll technique, it is possible to associate a polarized rewrite system with a set of clauses for ordered resolution with selection [7] by considering as left-hand sides only the literals that are selected in a clause. Thus, we would not produce a rule for each literal but only for those that are in the selected literals of the clause or maximal for the chosen ordering if no literal is selected.

Example 3 If we still consider the example of inclusion, with an ordering such that literals with $\subseteq$ are greater than literals with $\in$. The resulting rewrite system is reduced to:

$$A \subseteq B \rightarrow^{-} \forall x.\ \neg x \in A \vee x \in B$$
$$A \subseteq B \rightarrow^{+} \neg\mathrm{diff}(A, B) \in A$$
$$A \subseteq B \rightarrow^{+} \neg\neg\mathrm{diff}(A, B) \in B$$

However, ordered resolution with selection appears to be not compatible with the set-of-support strategy, in the sense that their combination jeopardizes completeness. In fact, the rewrite system corresponding to the clauses may not admit cut (although it is the case in the example above). Nevertheless, a sufficient condition to ensure completeness is the saturation of the set of clauses used as a complement of the set of support (i.e. the theory): the clauses that can be obtained by applying the resolution inference rules within it must either already belong to the theory or be redundant (i.e. they must be semantically implied by smaller clauses).

Theorem 6 ([39, Theorem 7]) *If a finite set of clauses Γ is saturated with respect to ordered resolution with selection, then the cut rule is admissible in the polarized sequent calculus modulo the rewrite system $\mathcal{R}_\Gamma$ restricted to the rules whose left-hand side is selected (or maximal if none is selected).*

Note that saturation is a sufficient, but not necessary condition, to ensure cut admissibility. For instance, the example above is not saturated if we choose an ordering that makes the left-hand sides of the clauses maximal or selected, although cut admissibility holds.

Since saturation implies satisfiability of the set of clauses (if it does not contain the empty clause), which is undecidable, saturating a set of clauses may not terminate. However, it can be semi-automated. Using the SPASS ATP

on the example above (with precedence $\subseteq > \in >$ diff, and $\subseteq$ and $\in$ dominant predicates), saturation generates two new clauses:

$$\underline{\neg X \in A} \vee \mathrm{diff}(A, B) \in A \vee X \in B$$
$$\underline{\neg \mathrm{diff}(A, B) \in B} \vee \neg X \in A \vee X \in B$$

The following rewrite system therefore admits cut:

$$A \subseteq B \to^{-} \forall x.\ \neg x \in A \vee x \in B$$
$$A \subseteq B \to^{+} \neg \mathrm{diff}(A, B) \in A$$
$$A \subseteq B \to^{+} \neg\neg \mathrm{diff}(A, B) \in B$$
$$X \in A \to^{-} \forall x.\ \mathrm{diff}(A, B) \in A \vee x \in B$$
$$\mathrm{diff}(A, B) \in B \to^{-} \forall x.\ \neg x \in A \vee x \in B$$

These different heuristics have been implemented in a tool called autotheo[4]. This tool proposes the following strategies to orient a set of axioms:

ClausalAll: The set of axioms is put in clausal normal form, and is transformed into a rewrite system as described above. The clausal normal form transformation is performed using E [98].

Sat: The set of axioms is saturated using E, and is transformed into a rewrite system restricted to the selected literals. The saturation may not terminate, and this technique therefore does not always succeed.

Equiv(X): In this technique, we use the heuristic based on the form of the axiom. If the axiom does not have an appropriate form, we apply the strategy X (in parameter) on it.

Presat(X): Since saturation may not terminate, we can decide to stop it after a certain amount of clauses has been generated. In this technique, we saturate the set of axioms using E until N clauses have been processed. These processed clauses are transformed into a rewrite system restricted to the selected literals as in Sat. The unprocessed clauses generated during the saturation are transformed using the X strategy (in parameter).

Id: The axiom is not transformed into a rewrite rule and it is kept as it is.

Nil: The axiom is dropped. This strategy is mainly useful when used in combination with Presat.

4.6 Dedukti Output

As with Zenon Modulo, iProverModulo is able to output formal proof terms that can be checked by Dedukti, a proof-checker based on the $\lambda\Pi$-calculus modulo theory. Only the resolution proofs can be output, which means that the transformation to clausal normal form and the use of the Inst-Gen calculus of iProver are currently not taken into account. The general setting to express

[4] Available at:
`http://www.ensiie.fr/~guillaume.burel/blackandwhite_autotheo.html.en`.

resolution and superposition proofs in Dedukti is given in [38], and its application to iProverModulo is sketched in [6] (Section 4.4).

iProverModulo can be used to produce Dedukti proofs for 3,383 problems of the TPTP problem library v6.3.0[5] [104].

4.7 Experimental Results

4.7.1 Manually Orienting Theories

We tested iProverModulo on 5 theories, corresponding to axiom files of the TPTP library. We manually transformed these axiom files into rewrite systems, by reasoning about how to ensure cut admissibility[6]. More precisely, we considered ANA001, axioms defining the analysis (limits) for continuous functions, BOO001, axioms defining a ternary Boolean algebra (a Boolean algebra with a ternary multiplication function), FLD001, axioms defining ordered fields, and SET001 and SET002, axioms defining a weak set theory using respectively predicates or function symbols to define unions, intersections, differences, and complements. We tested all the problems of the TPTP library v4.0.0 that use these axiom sets.

As we need to switch off some simplifications in order for OPRMT to be complete, we compare it to the following calculi:

Restricted resolution: in this case, the same options are given to iProver as when OPRMT is tried. The only difference is that the `--modulo` flag is switched off, the axioms being therefore considered as normal clauses instead of rewrite rules.

Default resolution: in this case, the default options of iProver are used. Only the Inst-Gen prover is turned off.

Full iProver: iProver is launched with its default options. In particular, the Inst-Gen prover is combined with the resolution prover.

We ran each problem with a timeout of 60 s for each of the 4 calculi mentioned above. Tests were performed on an Intel Core i3-330M 2.13 GHz computer. The results are summarized in Table 3 and represented graphically in Figure 15. The time taken for a given problem by OPRMT is compared to the time taken by the other calculi. Since the scale is logarithmic, for all points above the dashed line, OPRMT is 10 times faster than the other calculi, and for all points above the dotted line, 100 times faster. As we can see, OPRMT is always at least as efficient as restricted or default resolution, and in most of the cases at least 10 times better. This was expected because having proved cut admissibility for the considered rewrite system implies that the theory is

[5] The corresponding generated Dedukti files are available at:
`https://cloud.lsv.ens-cachan.fr/public.php?service=files&t=59f1cdee894ea25967a51bcadc76052a`.

[6] The rewrite systems that we designed to present these theories are given at:
`http://www.ensiie.fr/~guillaume.burel/empty_iProverModulo.html.en`.

	ANA001		BOO001		FLD001	
	# (%)	$\bar{t}$	# (%)	$\bar{t}$	# (%)	$\bar{t}$
OPRMT	3 (75)	11.41	3 (100)	0.01	40 (29)	0.95
restricted resolution	0 (0)	NA	0 (0)	NA	23 (17)	2.85
default resolution	1 (25)	25.34	1 (33)	25.46	40 (29)	13.55
full iProver	1 (25)	0.18	1 (33)	0.42	42 (31)	4.69
	SET001		SET002		Total	
	# (%)	$\bar{t}$	# (%)	$\bar{t}$	# (%)	$\bar{t}$
OPRMT	15 (100)	0.01	8 (100)	0.01	69 (42)	1.05
restricted resolution	15 (100)	4.05	5 (63)	8.06	43 (26)	3.88
default resolution	15 (100)	0.96	7 (88)	22.99	64 (39)	12.00
full iProver	15 (100)	0.17	7 (88)	7.11	66 (40)	3.79

Table 3 Comparison of Different Calculi on Problems Extracted from the TPTP Library. #: number of solved problems; %: percentage in the problem set corresponding to the theory; $\bar{t}$: average time to find a proof for the solved problems.

consistent, and the prover does not try to find a contradiction in the theory. A more surprising result is that using iProver in its whole is only rarely much better than using OPRMT. This means that the gain of using OPRMT relative to using ordered resolution is comparable to the gain obtained by combining it with the Inst-Gen method (including the use of an efficient SAT solver).

4.7.2 Automated Transformation of Theories

To test iProverModulo in a fully automated way, we tested it on the 12,720 problems in FOF or CNF format of the TPTP v7.1.0 library whose declared status is either Theorem or Unsatisfiable. We used autotheo to orient axioms. We compared the following provers:

autotheo+iProverModulo : we have a schedule that first orients the axioms using the Equiv(ClausalAll) strategy and launches iProverModulo with the resulting rewrite system for half of the given time. If the set of clauses is saturated (which means that the resulting rewrite system probably does not enjoy cut admissibility), or if the time limit is exceeded, we restart by orienting the axioms using the ClausalAll strategy and launching iProverModulo with the resulting rewrite system for the remaining of the time. In both cases, the Inst-Gen calculus is turned off since we want to compare the resolution calculi.

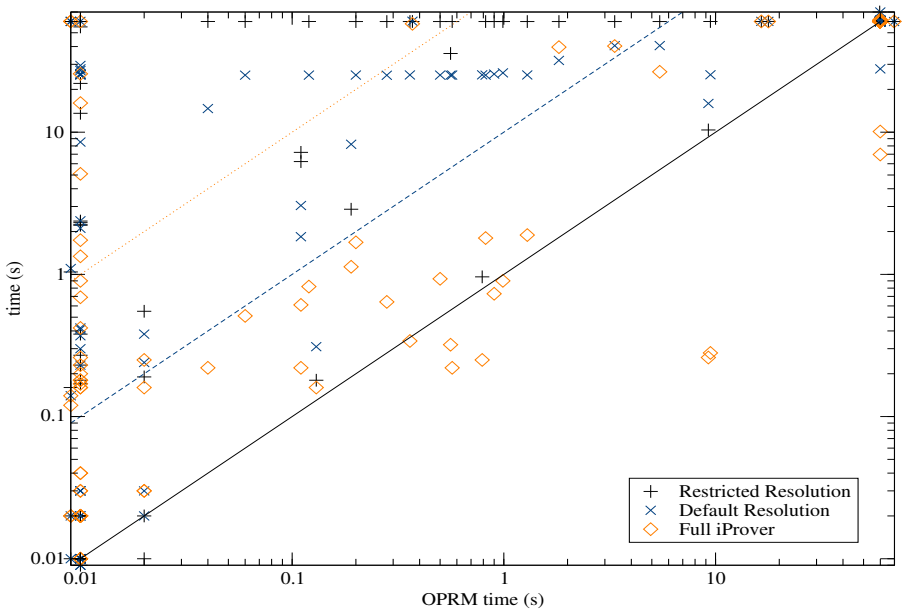

Fig. 15 Comparison of Different Calculi on Problems Extracted from the TPTP Library. The x-axis gives the time taken by OPRMT, the y-axis by the other calculi.

iProver : we launch the same version (0.7) of iProver as the one used in iProverModulo, with the same options as those given to iProverModulo. The Inst-Gen calculus is still turned off.

Zenon Modulo : we launch Zenon Modulo with the heuristic presented in Section 3.3.3.

E : as a reference of a state-of-the-art prover, we launch E version 1.8 with automatic options (`-auto`).

We ran each problem with a timeout of 120 s for each prover (which means that each orientation strategy got a timeout of 60 s for autotheo+iProverModulo), and a memory limit of 1 GiB. Tests were performed on virtualized Intel Core Processor (Haswell, no TSX) 2.30 GHz CPUs.

The results are summarized in Table 4. The "Gave up" line indicates when the prover exhausted the proof search space without finding a proof. This should not happen if the prover is complete. The "Unique proof" line corresponds to the number of problems that were proved by the given tool but not by the three others.

If we compare autotheo+iProverModulo to iProver, these results show that reasoning modulo a theory expressed as a rewrite system improves proof search, even though the translation of the theory into the rewrite system was performed automatically. Compared to E, autotheo+iProverModulo is able to prove almost half of the problems solved by a state-of-the-art theorem prover.

iProverModulo and Zenon Modulo give up for a relative high number of problems, which reveals some incompleteness. In the case of iProverModulo,

	iProverModulo	iProver	Zenon Modulo	E
Proof found	4,399 (34.6%)	3,682 (28.9%)	2,569 (20.2%)	8,909 (70.0%)
Gave up	664 (5.2%)	196 (1.5%)	1,252 (9.8%)	6 (0.0%)
Resource out	7,657 (60.2%)	8,842 (69.5%)	8,899 (70.0%)	3,805 (29.9%)
Unique proof	38	35	61	3,886

Proved only by iProverModulo and/or Zenon Modulo: 109

Table 4 Comparison of autotheo+iProverModulo to iProver, Zenon Modulo and E on 12,720 TPTP Problems.

the ClausalAll strategy should theoretically provide rewrite systems for which iProverModulo should be complete. However, this is only the case when the *term* rewrite system is confluent and terminating. This is one of the explanations why iProverModulo gives up finding a proof for some problems. Another explanation is the interaction of resolution modulo theory with the axioms for equality, which may be incomplete. In the case of Zenon Modulo, in addition to the fact that Zenon itself might not be complete, the absence of narrowing often explains why Zenon Modulo gives up on some problems.

Although iProverModulo and Zenon Modulo do not prove as many theorems as E, there are 109 problems where they can find a proof but not E. For Zenon Modulo, this is in particular the case in the SET and SEU category, which is not surprising since Zenon Modulo was designed to prove problems in a set theory. A few other problems proved only by iProverModulo and/or Zenon Modulo comes from the SWV (software verification) category. This may probably be explained by the fact that these problems contains a relatively big theory that can be well oriented as a rewriting system.

5 Deduction Modulo Theory in Zenon Modulo and iProverModulo

In this section, we aim to discuss and compare the approaches used by Zenon Modulo and iProverModulo to integrate Deduction modulo theory. The approaches used by the two tools are not only different because they rely on different proof search methods, but also because they have been designed in different contexts for different purposes. This is why the presentations of the two ATPs in Sections 3 and 4 come with two different benchmarks (and not the same one), for which they obtain best results.

The integration of Deduction modulo theory in Zenon Modulo is purely native. A genuine rewriting is performed once an atomic formula can be matched by the left-hand side of a rewrite rule. As explained in Section 3.3.1, the normalization process is actually engaged only when the formula is atomic. This approach allows us not to uselessly normalize all the formulas all the time. The rewritings that have been performed by the normalization of a given formula are not tracked by the prover, which means that we really reason over

propositions modulo a congruence, which is induced by the rewrite system, i.e. we do not distinguish congruent formulas.

As a consequence and as shown in Section 2.1, the proofs obtained by Zenon Modulo are simpler and shorter compared to the equivalent proofs that could be found without Deduction modulo theory using the regular version of Zenon. However, this imposes that the backend verifier, which verifies that the produced proofs are sound, must be compliant with Deduction modulo theory, i.e. this verifier must be able to reason modulo a congruence over propositions. This is the case of Dedukti, which can verify the proofs generated by Zenon Modulo [48]. But Zenon has also a Coq output [30], which cannot be used by Zenon Modulo as it produces no trace of rewriting and Coq requires to explicitly mention every single rewriting. A solution could be to produce traces of rewriting in all the cases, removing them when verifying proofs with Dedukti and providing them to Coq when using Coq verification, but we would lose the advantage of having compact proofs, which is not satisfactory. We must therefore realize that when we aim to integrate Deduction modulo theory, it must be used at all stages, proof search and proof verification.

As for iProverModulo, the integration of Deduction modulo theory is actually simulated for proposition rewrite rules (there is no rewriting mechanism introduced for them in the architecture of the prover) contrary to Zenon Modulo. As explained in Section 4.3, this is possible by introducing the notion of one-way clause [59], which imposes some additional constraints on the resolution rule. These constraints can be easily implemented by using a strategy similar to the set-of-support strategy for resolution [112] and by patching the selection mechanism. The advantage of such integration is that it is quite generic and can be ported to any resolution-like ATP. The drawback is that formulas are not normalized, but they are narrowed step by step; the confluence of the rewriting system, if assumed true, is not taken into account and all rewriting paths are potentially considered. For term rewrite rules, however, iProverModulo has a dedicated rewriting mechanism, which is quite efficient thanks to the compilation of the rewrite rules as explained in Section 4.4.

Regarding the verification of proofs and as said in Section 4.6, iProverModulo is only able to partially verify them contrary to Zenon Modulo, which can verify proofs in their entirety. In fact, iProverModulo can verify pure resolution proofs according to what is described in [38], but cannot deal with clausification and proofs using the instantiation-generation (Inst-Gen) calculus [65]. These two parts are seen as black boxes in iProverModulo and rely on external tools, so that it is difficult to verify proofs as a whole. A very current trend is to build new ATPs aggregating several automated proof techniques and tools, and it therefore becomes complicated to verify the produced proofs. However, this verification has never been so important since this combination of tools increases the risk of building an overall unsound tool.

Zenon Modulo and iProverModulo have not been designed in the same frameworks nor to deal with the same kind of problems. This therefore had significant consequences on the choices that have been made regarding their respective implementations.

Zenon Modulo has been built in the framework of the industrial research project BWare [56,107], whose aim was to improve the automated verification of Proof Obligations (POs for short) coming from the modeling of industrial applications using the B method [1]. As the B method relies on a set theory, the goal has been to develop (express) this theory manually using Deduction modulo theory together with the corresponding tool (Zenon Modulo), which had to be fully compliant with Deduction modulo theory. Besides Deduction modulo theory, we had to add arithmetic [44] (as a significant part of the POs involves arithmetic) and polymorphic types [41]. Polymorphic types allowed us to express the B set theory in a compact way, and especially as a pure rewrite system (where typing constraints have been removed). Thus, Zenon Modulo has been designed in this very specific context with this very specific purpose, and as can be seen in Section 3.6, it obtains very good results, in particular compared to some of the best state-of-the-art ATPs.

As the goal was to only focus on the BWare benchmark, Deduction modulo theory has not been fully implemented in Zenon Modulo. In particular, this is the case of narrowing for at least two reasons. First, we do not need to build sets in proofs of POs and we therefore do not need to narrow metavariables representing sets. Second, in the B set theory, there are about 40 set constructs introduced by defining axioms (rewrite rules in our case). To narrow a set metavariable, we would therefore need to choose among these 40 possibilities, which implies a large proof search space due to this combinatorial choice and may be ineffective in most of the cases.

Even though it was developed for a specific purpose, the very first versions of Zenon Modulo have been tested over a more general-purpose benchmark, i.e. the TPTP library [104]. Over all the problems, we slightly improved the results of Zenon (2.5% in the best case), and for some categories, we even obtained significant improvements (about 50% in the SET category, for example). These saw-tooth results can be explained since Deduction modulo theory provides best results when the theory can be built manually. Here, in this experiment, the rewrite systems are built automatically on the fly by means of several heuristics, which may produce inappropriate rewrite systems. The full results of this experiment can be found in [55].

Compared to Zenon Modulo, iProverModulo has a more generic approach and has not been designed to deal with a specific form of problems in a specific theory. This is evidenced in particular by the benchmark that has been chosen to test the prover, i.e. problems from the TPTP library (see the results in Section 4.7). As Zenon Modulo, iProverModulo is able to deal not only with given rewrite systems, but can also build on the fly rewrite systems in a fully automated way. The difference with Zenon Modulo is that it proposes full support for Deduction modulo theory, with complete strategies in some cases (for example, the ClausalAll strategy for consistent theories, see Section 4.5).

To conclude this comparison between the approaches adopted by Zenon Modulo and iProverModulo to integrate Deduction modulo theory, it should be noted that the goal of these experiments is to assess the improvement that can be provided by Deduction modulo theory to ATPs in practice. The aim is therefore

to show this improvement for a given prover (either Zenon or iProver, in our case), and not necessarily to compare the extended versions to other ATPs. This comparison with other ATPs remains important but appears to be more marginal since even extended with Deduction modulo theory, the new prover is mainly dependent on the proof engine of the initial prover, even though, in some cases, we obtain very positive results (for example, Zenon Modulo outperforms some of the best state-of-the-art ATPs when tested over the BWare benchmark; see the results in Section 3.6). In this context, we plan to investigate other integrations of Deduction modulo theory to other proof techniques and tools (see Section 7.2 for more details about our perspectives in this matter).

6 Related Work

Historically, automated reasoning consists of uniform proof search procedures for first-order logic. If these procedures, which are mainly divided into two schools, namely Beth and Hintikka's tableaux [21, 20, 74, 73] and Robinson's resolution [94], rely on a quite expressive formalism, they fail to be efficient when reasoning in some specific theories. The reason is that they are purely syntactical approaches, which use the axioms of the theory mechanically without any insight that could be provided by the semantics of the theory. As they are, these methods therefore have a limited success since we must find some appropriate mechanisms to gain efficiency when reasoning modulo theories. This is exactly the topic of this paper, which proposes a method based on Deduction modulo theory [60].

Reasoning modulo theories is a current trend in automated reasoning and a very active field of research in this community. However, it drew attention at the earliest age of automated reasoning. For example, the set-of-support strategy, that was introduced by Wos et al. [112] the same year as Robinson's resolution (1965), can be seen as a way to take into account that a theory is consistent. In 1972, Plotkin [91] also showed how to deal with equational theories in resolution-based ATPs. One of the most influencing attempt to tackle reasoning modulo theories dates back in the middle of 80's with the work of Stickel [102] in the context of resolution. Following this idea of theory resolution, several first-order calculi have been given sound and complete theory extensions that rely on the computation of complete sets of theory unifiers. However, these initiatives failed to produce efficient implementations, mostly due to the practical difficulty, or the theoretical impossibility, of computing theory unifiers for concrete background theories of interest.

As pointed out by [100], to gain efficiency when reasoning modulo theories, the idea is to address only (expressive enough) decidable or semi-decidable fragments of a certain logic, and incorporate specific reasoning over certain domains, such as equality, arithmetic, or data structures (arrays, lists, stacks, etc.). In this vein, SMT (Satisfiability Modulo Theories) solving [12, 13] has appeared as a very concrete and effective solution over the last two decades. The advent of SMT solving is mainly due to the concomitance of some significant

advances in several domains. One of them concerns the increasing efficiency of SAT solvers (using Davis-Putnam-Logemann-Loveland's algorithm [53,52] or the conflict-driven clause learning approach [81]), which SMT solvers rely on. But SMT solving has also been inspired by other results regarding decision procedures and combination of these procedures (e.g., Nelson-Oppen's congruence closure procedure [85], Nelson-Oppen's combination method [84], and Shostak's combination method [101]), as well as other systems, in particular, Boyer and Moore's Nqthm prover [33] and Detlefs, Nelson, and Saxe's Simplify prover [57].

Even though SMT solvers only deal with ground clauses, some of them are able to propose support for quantifier reasoning. However, quantifier reasoning is a long-standing challenge for SMT solving, for which there is no general decision procedure for quantifiers. Most of state-of-the-art SMT solvers with support for quantifiers therefore use heuristic quantifier instantiation [67] for incorporating quantifier reasoning with ground decision procedures. One of these heuristic instantiation-based approaches, which is often used in the implementation of SMT solvers, is the E-matching algorithm [54], which was first introduced by the Simplify theorem prover, previously mentioned. If heuristic instantiation appears to be quite effective in some domains (e.g., software verification applications [11, 64]), it is not complete for first-order logic. This is a significant difference with what we propose in this paper. Deduction modulo theory allows us to extend TPTP-like first-order ATPs, which are originally complete for first-order logic as they rely on unification rather than matching, without compromising this completeness. All the difficulty therefore depends on the design of the rewrite system modeling the theory, which must have appropriate properties ensuring that cut-free completeness is preserved.

In practice, there is also another significant difference between SMT solving and Deduction modulo theory. Adding a new theory in a SMT solver requires to check beforehand how this theory can be integrated and combined with the other theories, as well as to develop the corresponding decision procedure by complying with the interface proposed by the solver to integrate new theories. In Deduction modulo theory, the integration is smoother and more generic, as once the rewrite system is designed, we just have to feed the compatible ATPs with this rewrite system without having to develop any additional material.

Besides SMT solving, there is also a plethora of work regarding reasoning modulo theories in the context of first-order calculi. In particular, many papers are directly in line with the ideas of Stickel [102] regarding theory resolution, previously mentioned. Stickel's work was ported to many calculi, such as path resolution [83], the connection method [90], connection graphs [87], or model elimination [14]. We can also mention ordered theory resolution [15], imposing ordering restrictions on theory resolution, or theory instantiation [66], relying on instantiation calculus [65] (implemented in the iProver ATP [78]). Still in the domain of resolution, we have hierarchical superposition calculus [8,17] as well, introduced as a generalization of superposition calculus for black-box style theory reasoning, and implemented in the Beagle tool [16] in particular. There is also some work in the context of sequent calculus and tableaux. For example,

there are some generic approaches, such as [19], where the authors propose the use of incremental theory reasoning in free-variable semantic tableaux, or [108], which describes a multi-theory version of the semantic tableau calculus. Some other approaches focus on specific theories. For instance, [18] studies various ways of handling equality in tableaux, while [95] introduces a sequent calculus integrating linear integer arithmetic, which combines free-variable tableaux with incremental closure [68] and the Omega quantifier elimination procedure (this work has been implemented in the Princess tableau-based ATP).

All the work described above shares the same line of research, which consists in adding black-box reasoners for specific background theories to first-order automated reasoning methods. All the difficulty therefore resides in getting the regular general-purpose reasoner and the background reasoner(s) to cooperate in a sound and complete way. The approach of Deduction modulo theory is quite different in the sense that we remain in the same formalism, i.e. first-order logic where we reason modulo a congruence over terms and propositions. As a consequence, there is no notion of foreground and background reasoners, and there is no need to worry about the interface between the general-purpose reasoning and the theory reasoning. However, the considered theory must contain a computable part that can be exhibited and exploited to prove theorems in this theory. If this works quite well for some (parts of) theories, e.g. set theory (see the results obtained by Zenon Modulo over the BWare benchmark in Section 3), it cannot be applied for some other theories, which either do not contain any computable part or contain computable parts but which cannot be exploited. For example, although arithmetic can be expressed using Deduction modulo theory [62], it is not of great help to automatically prove lemmas in the (decidable) linear fragment of arithmetic. In this case, we need a specialized procedure, i.e. a background reasoner, which cannot be deduced from the expression of the theory.

Still in the vein of reasoning modulo theories in the context of first-order calculi, we have to mention the work that has been made around superdeduction [34], which is in direct correspondence with the ideas of Deduction modulo theory. While Deduction modulo theory aims to integrate axioms of the theory as computation rules, superdeduction proposes to integrate them as deduction rules. The two approaches are therefore very similar and play on the duality between computation and deduction. But superdeduction goes further and adds to this transformation the decomposition of the connectives occurring in the formula introduced by the axiom (the formula corresponding to the right-hand side of the rewrite rule). This corresponds to an extension of Prawitz's folding (resp. unfolding) rules [92], where as most connectives of the formula as possible are introduced (resp. eliminated). Superdeduction may be seen as the alliance of Deduction modulo theory with focusing, which is a technique initially introduced in the framework of linear logic [2]. In practice, a tool, called Super Zenon [75], has been implemented and integrates superdeduction into the Zenon ATP. Several experiments have been conducted with Super Zenon using in particular a benchmark of proof rules of Atelier B [50], maintained by the Siemens company, and another one coming from the TPTP

library [104]. These experiments have shown significant improvements (in terms of number of proved problems) and speed-ups (both in terms of proof time and proof size) compared to the use of the regular version of Zenon in particular. The detailed results can be found in [75,76].

When reasoning modulo theories, it may be necessary to rely on types in first-order logic, at least to distinguish the parts of the formula requiring theory reasoning from the other parts of the formula. However, typed first-order logic, aka many-sorted first-order logic, plays a greater role than only distinguishing parts of a formula as the use of types can provide useful guidance for proof search in automated deduction. In this vein, a number of papers like [97,110, 77,25,109,93] have emerged, where simple types, polymorphic types, and even dependent types have been introduced in the context of first-order logic. Beyond proof search guidance and depending on the considered typing system, it also increases the expressiveness of first-order logic. This feature is quite important in the framework of Deduction modulo theory as it allows us to get rid of the typing conditions in first-order formulas [105], which can be turned into pure rewrite rules, avoiding conditional rewrite rules. In practice, although there are some initiatives, such as the introduction of the TFF1 language [23] to deal with polymorphic first-order formulas in the TPTP library [104] for example, it appears that very few ATPs are able to deal with elaborate types, i.e. beyond simple types (which are mainly used for theory reasoning and arithmetic in particular). For instance, currently and to our knowledge, there are only four ATPs able to deal with polymorphic first-order logic, namely Zenon Modulo [55], Zipperposition[7] (developed by S. Cruanes), ArchSAT[8] [42] (developed by G. Bury), and Alt-Ergo[9] (developed by the OCamlPro company). One explanation is that although polymorphism is not difficult to introduce from a theoretical point of view, it remains a little complicated to implement and involves rather significant changes in the architecture of the prover. An alternative is to encode polymorphic types into first-order logic [22], which has been implemented in tools like Why3 [26] for example, but this option may be less effective in practice (see the low results obtained by iProverModulo over the BWare benchmark in Section 3 for instance).

7 Conclusion

7.1 Achievements

Deduction modulo theory can be seen as an evolution of proof systems, where theories are integrated as computations (rewrite rules) more closely to the deduction kernel of these proof systems. The initial goal of Deduction modulo theory was to improve automated deduction by allowing us to perform pure

[7] Available at: `https://github.com/c-cube/zipperposition/`.
[8] Available at: `https://github.com/Gbury/archsat/`.
[9] Available at : `https://alt-ergo.ocamlpro.com/`.

computations during proof search. That is why the seminal paper about Deduction modulo theory [60] introduces not only the theory itself, but also an integration of Deduction modulo theory into a resolution-based proof search method (ENAR). In the wake of this work, another paper followed that described a tableau method for Deduction modulo theory [29]. These papers introduced new generation of proof search methods based on Deduction modulo theory, but only theoretically and no experiment was conducted before the development of Zenon Modulo and iProverModulo, which respectively extend Zenon and iProver to Deduction modulo theory, and which are presented in this paper. These tools satisfied a real need of experimentation because even if a proof search method is sound and complete, it may also unfortunately turn out to be ineffective in practice.

As shown by the experimental results in Secs. 3 and 4, Zenon Modulo and iProverModulo provide significant improvements in terms of proved problems, compared to the tools from which they have been developed but also compared to other state-of-the-art ATPs. These improvements are observed over specific benchmarks (like the BWare benchmark, which gathers problems from software verification using the B method in particular), but also more general-purpose benchmarks (like the TPTP library). When the theory is known in advance (for example, this is the case of the B set theory used by Zenon Modulo over the BWare benchmark), the results are even better as the theory can be modeled in a suitable way using Deduction modulo theory, ensuring properties like confluence or cut-free completeness. But even when the theory (or a part of the theory) is transformed on the fly into a rewrite system by means of heuristics (some of which may be quite efficient, preserving completeness for example), the results are very satisfactory as well (see the results of iProverModulo over the TPTP library, and also those of Zenon Modulo over the same library in [55]).

Beyond the improvements obtained by Zenon Modulo and iProverModulo, it should be noted that both tools have adopted a certifying approach, since they are able to produce proofs that can be checked by an independent and external tool (in that sense, they satisfy the De Bruijn criterion as formulated by Barendregt in [9]). This feature is quite important, as many bugs may occur in the implementation of ATPs (some parts of code are highly non-trivial, such as the clausification process in resolution-based provers for instance), and we have no choice but to note that very few of them adopt such an approach. The tool used to check the proofs produced by Zenon Modulo and iProverModulo is Dedukti, a universal proof checker that also relies on Deduction modulo theory. The fact that Dedukti relies on Deduction modulo theory allows our ATPs to produce proofs with no trace of computations and it is enough to provide the rewrite rules to Dedukti, which can perform the necessary computations when verifying the generated proofs. In that sense, our ATPs satisfy the Poincaré principle as also formulated by Barendregt in [9].

7.2 Future Work

Building on the quite satisfactory results obtained by Zenon Modulo and iProverModulo, we plan to apply Deduction modulo theory to other proof search methods. In particular, we aim to study Deduction modulo theory in the framework of superposition. From a theoretical point of view, this study raises several problems, including the fact that there are actually two rewrite systems to be considered, one coming from the set of equations handled by the superposition calculus and the other coming from the Deduction modulo theory. The two rewrite systems have to be merged, and we have to wonder if the resulting system enjoys the same properties (confluence, termination, etc.). Currently, we are testing an experimental prototype of a superposition-based ATP where rewriting over terms and propositions has been integrated. More precisely, this prototype is built on top of the Zipperposition tool and is tested over a small benchmark of B set problems (about 300 problems), for which it obtains very promising results [43].

We also plan to study Deduction modulo theory in the scope of SMT solving. Most of the modern SMT solvers use heuristic quantifier instantiation for incorporating quantifier reasoning with ground decision procedures. This mechanism is not refutationally complete for first-order logic, and generally requires hints (triggers), which are sensitive to the syntactic structure of the formula. Inserting triggers at some appropriate places in axioms allows us to emulate rewriting in SMT solvers. A rewrite system is then a set of axioms with triggers (inserted manually or automatically), and no change in the architecture of the SMT solver is actually required. An experiment is currently being carried out using the ArchSAT SMT solver [42] over the same benchmark of B set problems introduced previously for Zipperposition, and ArchSAT obtains quite promising results as well [43].

Deduction modulo theory has another line of work, which is the interoperability between proof systems, and which will be subject to intensive research in the next few years. We aim to ensure this interoperability by means of the Dedukti tool, which offers a logical framework based on the $\lambda\Pi$-calculus modulo theory, and the first step consists in showing that many theories can be expressed using Dedukti. As can be seen in [6], many systems can be embedded into Dedukti (constructive and classical predicate logic, simple type theory, programming languages, pure type systems, the calculus of inductive constructions with universes, etc.), and large libraries coming from various systems can be translated and checked by Dedukti (HOL Light, Matita, FoCaLiZe, etc.). This work around Dedukti is essential as it is the backend of our ATPs based on Deduction modulo theory. As mentioned earlier, this backend is very appropriate since our ATPs can produce proofs without computations, which are redone by Dedukti (Poincaré's principle). Conversely, the Dedukti outputs of our ATPs bring new encodings to Dedukti therefore showing its versatile feature and its ability of scaling-up (the considered benchmarks, BWare or TPTP, consist of large libraries, and in the case of BWare, each problem involves large formulas). A second step toward interoperability can be to use our ATPs to

perform proof reconstruction within Dedukti, when the latter receives only partial proof objects. Finally, we intend to use Dedukti to realize concrete interoperability between two proof systems. An example of application could arise from automated deduction, where we can envision that two ATPs use Dedukti as a communication language (not only as a backend to certify their proofs) in order to collaborate in their proof search.

Acknowledgements We thank the anonymous reviewers for their careful reading of our manuscript and their many insightful comments and suggestions, which helped us improve and clarify this manuscript.

References

1. Abrial, J.R.: The B-Book, Assigning Programs to Meanings. Cambridge University Press, Cambridge (UK) (1996). ISBN 0521496195
2. Andreoli, J.M.: Logic Programming with Focusing Proofs in Linear Logic. Journal of Logic and Computation (JLC) **2**(3), 297–347 (1992)
3. Assaf, A.: A Framework for Defining Computational Higher-Order Logics. Ph.D. thesis, École polytechnique (2015)
4. Assaf, A.: Conservativity of Embeddings in the $\lambda\Pi$ Calculus Modulo Rewriting. In: Typed Lambda Calculi and Applications (TLCA), *LIPIcs*, vol. 38, pp. 31–44. Schloss Dagstuhl, Leibniz-Zentrum fuer Informatik, Warsaw (Poland) (2015)
5. Assaf, A., Burel, G.: Translating HOL to Dedukti. In: Proof eXchange for Theorem Proving (PxTP), *EPTCS*, vol. 186, pp. 74–88. Open Publishing Association, Berlin (Germany) (2015)
6. Assaf, A., Burel, G., Cauderlier, R., Delahaye, D., Dowek, G., Dubois, C., Gilbert, F., Halmagrand, P., Hermant, O., Saillard, R.: Dedukti: a Logical Framework based on the $\lambda\Pi$-Calculus Modulo Theory (2016). Submitted, available at: `http://www.lsv.ens-cachan.fr/~dowek/Publi/expressing.pdf`
7. Bachmair, L., Ganzinger, H.: Resolution Theorem Proving. In: Handbook of Automated Reasoning, vol. 1, pp. 19–99. Elsevier and MIT Press (2001)
8. Bachmair, L., Ganzinger, H., Waldmann, U.: Refutational Theorem Proving for Hierarchic First-Order Theories. Applicable Algebra in Engineering, Communication and Computing **5**, 193–212 (1994)
9. Barendregt, H., Barendsen, E.: Autarkic Computations in Formal Proofs. Journal of Automated Reasoning (JAR) **28**(3), 321–336 (2002)
10. Barendregt, H., Dekkers, W., Statman, R.: Lambda Calculus with Types. Cambridge University Press (2013). ISBN 9780521766142
11. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: Formal Methods for Components and Objects (FMCO), *LNCS*, vol. 4111, pp. 364–387. Springer, Amsterdam (The Netherlands) (2005)
12. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability Modulo Theories. In: Handbook of Satisfiability, *Frontiers in Artificial Intelligence and Applications*, vol. 185, pp. 825–885. IOS Press (2009)
13. Barrett, C.W., Tinelli, C.: Satisfiability Modulo Theories. In: Handbook of Model Checking, pp. 305–343. Springer (2018)
14. Baumgartner, P.: A Model Elimination Calculus with Built-in Theories. In: German Conference on Artificial Intelligence (GWAI), *LNCS*, vol. 671, pp. 30–42. Springer, Bonn (Germany) (1992)
15. Baumgartner, P.: An Order Theory Resolution Calculus. In: Logic Programming and Automated Reasoning (LPAR), *LNCS*, vol. 624, pp. 119–130. Springer, St. Petersburg (Russia) (1992)

16. Baumgartner, P., Bax, J., Waldmann, U.: Beagle – A Hierarchic Superposition Theorem Prover. In: Conference on Automated Deduction (CADE), *LNCS*, vol. 9195, pp. 367–377. Springer, Berlin (Germany) (2015)
17. Baumgartner, P., Waldmann, U.: Hierarchic Superposition with Weak Abstraction. In: Conference on Automated Deduction (CADE), *LNCS*, vol. 7898, pp. 39–57. Springer, Lake Placid (NY, USA) (2013)
18. Beckert, B.: Semantic Tableaux with Equality. Journal of Logic and Computation (JLC) **7**(1), 39–58 (1997)
19. Beckert, B., Pape, C.: Incremental Theory Reasoning Methods for Semantic Tableaux. In: Theorem Proving with Analytic Tableaux and Related Methods (TABLEAUX), *LNCS*, vol. 1071, pp. 93–109. Springer, Terrasini (Palermo, Italy) (1996)
20. Beth, E.W.: The Foundations of Mathematics: A Study in the Philosophy of Science. Studies in Logic and the Foundations of Mathematics. North-Holland Pub. Co. (1959)
21. Beth, E.W.: Formal Methods: An Introduction to Symbolic Logic and to the Study of Effective Operations in Arithmetic and Logic, *Synthese Library*, vol. 4. D. Reidel Pub. Co. (1962)
22. Blanchette, J.C., Böhme, S., Popescu, A., Smallbone, N.: Encoding Monomorphic and Polymorphic Types. Logical Methods in Computer Science (LMCS) **12**(4) (2016)
23. Blanchette, J.C., Paskevich, A.: TFF1: The TPTP Typed First-Order Form with Rank-1 Polymorphism. In: Conference on Automated Deduction (CADE), *LNCS*, vol. 7898. Springer (2013)
24. Blanqui, F., Jouannaud, J.P., Okada, M.: The Calculus of Algebraic Constructions. In: Rewriting Techniques and Applications (RTA), *LNCS*, vol. 1631. Springer, Trento (Italy) (1999)
25. Bläsius, K.H., Hedtstück, U., Rollinger, C.R. (eds.): Sorts and Types in Artificial Intelligence, Workshop, Eringerfeld, FRG, April 24-26, 1989, Proceedings, *LNCS*, vol. 418. Springer (1989)
26. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd Your Herd of Provers. In: International Workshop on Intermediate Verification Languages (Boogie), pp. 53–64. Wrocław (Poland) (2011)
27. Boespflug, M., Burel, G.: CoqInE: Translating the Calculus of Inductive Constructions into the $\lambda\Pi$-calculus Modulo. In: Proof eXchange for Theorem Proving (PxTP), *CEUR Workshop Proceedings*, vol. 878, pp. 44–50. David Pichardie and Tjark Weber, Manchester (UK) (2012)
28. Boespflug, M., Carbonneaux, Q., Hermant, O.: The $\lambda\Pi$-Calculus Modulo as a Universal Proof Language. In: Proof Exchange for Theorem Proving (PxTP), pp. 28–43. Manchester (UK) (2012)
29. Bonichon, R.: TaMeD: A Tableau Method for Deduction Modulo. In: International Joint Conference on Automated Reasoning (IJCAR), *LNCS*, vol. 3097, pp. 445–459. Springer, Cork (Ireland) (2004)
30. Bonichon, R., Delahaye, D., Doligez, D.: Zenon: An Extensible Automated Theorem Prover Producing Checkable Proofs. In: Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), *LNCS/LNAI*, vol. 4790, pp. 151–165. Springer, Yerevan (Armenia) (2007)
31. Bonichon, R., Hermant, O.: A Semantic Completeness Proof for TaMeD. In: Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), *LNCS*, vol. 4246, pp. 167–181. Springer, Phnom Penh (Cambodia) (2006)
32. Bonichon, R., Hermant, O.: On Constructive Cut Admissibility in Deduction Modulo. In: Types for Proofs and Programs (TYPES), *LNCS*, vol. 4502, pp. 33–47. Springer, Nottingham (UK) (2006)
33. Boyer, R.S., Moore, J.S.: A Theorem Prover for a Computational Logic. In: Conference on Automated Deduction (CADE), vol. 449, pp. 1–15. Springer, Kaiserslautern (Germany) (1990)
34. Brauner, P., Houtmann, C., Kirchner, C.: Principles of Superdeduction. In: Logic in Computer Science (LICS), pp. 41–50. IEEE Computer Society Press, Wrocław (Poland) (2007)
35. Burel, G.: Embedding Deduction Modulo into a Prover. In: Computer Science Logic (CSL), *LNCS*, vol. 6247, pp. 155–169. Springer, Brno (Czech Republic) (2010)

36. Burel, G.: Consistency Implies Cut Admissibility. In: Proof-Search in Axiomatic Theories and Type Theories (PSATTT). Wrocław (Poland) (2011)
37. Burel, G.: Efficiently Simulating Higher-Order Arithmetic by a First-Order Theory Modulo. Logical Methods in Computer Science (LMCS) **7**(1), 1–31 (2011)
38. Burel, G.: A Shallow Embedding of Resolution and Superposition Proofs into the $\lambda\Pi$-Calculus Modulo. In: Proof eXchange for Theorem Proving (PxTP), *EPiC Series*, vol. 14, pp. 43–57. EasyChair (2013)
39. Burel, G.: Cut Admissibility by Saturation. In: Rewriting Techniques and Applications (RTA) and Typed Lambda Calculi and Applications (TLCA), *LNCS*, vol. 8560, pp. 124–138. Springer, Vienna (Austria) (2014)
40. Burel, G., Kirchner, C.: Regaining Cut Admissibility in Deduction Modulo using Abstract Completion. Information and Computation **208**(2), 140–164 (2010)
41. Bury, G., Cauderlier, R., Halmagrand, P.: Implementing Polymorphism in Zenon. In: International Workshop on the Implementation of Logics (IWIL), *EPiC Series in Computing*, vol. 40, pp. 15–20. EasyChair, Suva (Fiji) (2015)
42. Bury, G., Cruanes, S., Delahaye, D.: SMT Solving Modulo Tableau and Rewriting Theories. In: Satisfiability Modulo Theories (SMT). Oxford (UK) (2018)
43. Bury, G., Cruanes, S., Delahaye, D., Euvrard, P.L.: An Automation-Friendly Set Theory for the B Method. In: Abstract State Machines, Alloy, B, VDM, and Z (ABZ), *LNCS*, vol. 10817, pp. 409–414. Springer, Southampton (UK) (2018)
44. Bury, G., Delahaye, D.: Integrating Simplex with Tableaux. In: Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX), *LNCS*, vol. 9323, pp. 86–101. Springer, Wrocław (Poland) (2015)
45. Bury, G., Delahaye, D., Doligez, D., Halmagrand, P., Hermant, O.: Automated Deduction in the B Set Theory Using Typed Proof Search and Deduction Modulo. In: Logic for Programming, Artificial Intelligence and Reasoning (LPAR), Short Papers, *EPiC Series in Computing*, vol. 35, pp. 42–58. EasyChair, Suva (Fiji) (2015)
46. Cauderlier, R.: Object-Oriented Mechanisms for Interoperability between Proof Systems. Ph.D. thesis, Conservatoire National des Arts et Métiers (Cnam) (2016)
47. Cauderlier, R., Dubois, C.: ML Pattern-Matching, Recursion, and Rewriting: From FoCaLiZe to Dedukti. In: International Colloquium on Theoretical Aspects of Computing (ICTAC), *LNCS*, vol. 9965, pp. 459–468. Springer, Taipei (Taiwan, ROC) (2016)
48. Cauderlier, R., Halmagrand, P.: Checking Zenon Modulo Proofs in Dedukti. In: Proof eXchange for Theorem Proving (PxTP), *EPTCS*, vol. 186, pp. 57–73. Open Publishing Association, Berlin (Germany) (2015)
49. Chvátal, V.: Linear Programming. Series of Books in the Mathematical Sciences. W. H. Freeman and Company, New York (USA) (1983). ISBN 0716715872
50. ClearSy: Atelier B 4.2.1 (2015). `http://www.atelierb.eu/`
51. Cousineau, D., Dowek, G.: Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo. In: Typed Lambda Calculi and Applications (TLCA), *LNCS*, vol. 4583, pp. 102–117. Springer, Paris (France) (2007)
52. Davis, M., Logemann, G., Loveland, D.W.: A Machine Program for Theorem-Proving. Communications of the ACM (CACM) **5**(7), 394–397 (1962)
53. Davis, M., Putnam, H.: A Computing Procedure for Quantification Theory. Journal of the ACM **7**(3), 201–215 (1960)
54. De Moura, L.M., Bjørner, N.: Efficient E-Matching for SMT Solvers. In: Conference on Automated Deduction (CADE), *LNCS*, vol. 4603, pp. 183–198. Springer, Bremen (Germany) (2007)
55. Delahaye, D., Doligez, D., Gilbert, F., Halmagrand, P., Hermant, O.: Zenon Modulo: When Achilles Outruns the Tortoise using Deduction Modulo. In: Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), *LNCS/ARCoSS*, vol. 8312, pp. 274–290. Springer, Stellenbosch (South Africa) (2013)
56. Delahaye, D., Dubois, C., Marché, C., Mentré, D.: The BWare Project: Building a Proof Platform for the Automated Verification of B Proof Obligations. In: Abstract State Machines, Alloy, B, VDM, and Z (ABZ), *LNCS*, vol. 8477, pp. 126–127. Springer, Toulouse (France) (2014)
57. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A Theorem Prover for Program Checking. Journal of the ACM **52**(3), 365–473 (2005)

58. Dowek, G.: Confluence as a Cut Elimination Property. In: Rewriting Techniques and Applications (RTA), *LNCS*, vol. 2706, pp. 2–13. Springer (2003)
59. Dowek, G.: Polarized Resolution Modulo. In: Theoretical Computer Science (TCS), *IFIP Advances in Information and Communication Technology*, vol. 323, pp. 182–196. Springer, Brisbane (Australia) (2010)
60. Dowek, G., Hardin, T., Kirchner, C.: Theorem Proving Modulo. Journal of Automated Reasoning (JAR) **31**(1), 33–72 (2003)
61. Dowek, G., Werner, B.: Proof Normalization Modulo. Journal of Symbolic Logic (JSL) **68**(4), 1289–1316 (2003)
62. Dowek, G., Werner, B.: Arithmetic as a Theory Modulo. In: Rewriting Techniques and Applications (RTA), *LNCS*, vol. 3467, pp. 423–437. Springer, Nara (Japan) (2005)
63. Fitting, M.: First-Order Logic and Automated Theorem Proving, 2nd edn. Springer (1996). ISBN 9781461223603
64. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java. In: Programming Language Design and Implementation (PLDI), pp. 234–245. ACM, Berlin (Germany) (2002)
65. Gaanzinger, H., Korovin, K.: New Directions in Instantiation-Based Theorem Proving. In: Logic in Computer Science (LICS), pp. 55–64. IEEE Computer Society, Ottawa (Canada) (2003)
66. Ganzinger, H., Korovin, K.: Theory Instantiation. In: Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), *LNCS*, vol. 4246, pp. 497–511. Springer, Phnom Penh (Cambodia) (2006)
67. Ge, Y., Barrett, C., Tinelli, C.: Solving Quantified Verification Conditions Using Satisfiability Modulo Theories. In: Conference on Automated Deduction (CADE), *LNCS*, vol. 4603, pp. 167–182. Springer, Bremen (Germany) (2007)
68. Giese, M.: Incremental Closure of Free Variable Tableaux. In: International Joint Conference on Automated Reasoning (IJCAR), *LNCS*, vol. 2083, pp. 545–560. Springer, Siena (Italy) (2001)
69. Halmagrand, P.: Soundly Proving B Method Formulae Using Typed Sequent Calculus. In: International Colloquium on Theoretical Aspects of Computing (ICTAC), *LNCS*, vol. 9965, pp. 196–213. Springer, Taipei (Taiwan, ROC) (2016)
70. Harper, R., Honsell, F., Plotkin, G.D.: A Framework for Defining Logics. Journal of the ACM **40**(1), 143–184 (1993)
71. Hermant, O.: Semantic Cut Elimination in the Intuitionistic Sequent Calculus. In: Typed Lambda-Calculi and Applications (TLCA), *LNCS*, vol. 3461, pp. 221–233. Springer, Nara (Japan) (2005)
72. Hermant, O.: Resolution is Cut-Free. Journal of Automated Reasoning (JAR) **44**(3), 245–276 (2010)
73. Hintikka, J.: Notes on the Quantification Theory. Societas Scientiarum Fennica, Commentationes Physico-Mathematicae **17**(12), 1–13 (1955)
74. Hintikka, J.: Two Papers on Symbolic Logic: Form and Content in Quantification Theory and Reductions in the Theory of Types. Societas Philosophica, Acta philosophica Fennica **8**, 7–55 (1955)
75. Jacquel, M., Berkani, K., Delahaye, D., Dubois, C.: Tableaux Modulo Theories using Superdeduction: An Application to the Verification of B Proof Rules with the Zenon Automated Theorem Prover. In: International Joint Conference on Automated Reasoning (IJCAR), *LNCS*, vol. 7364, pp. 332–338. Springer, Manchester (UK) (2012)
76. Jacquel, M., Berkani, K., Delahaye, D., Dubois, C.: Tableaux Modulo Theories using Superdeduction. Global Journal of Advanced Software Engineering (GJASE) **1**, 1–13 (2014)
77. Kifer, M., Wu, J.: A First-Order Theory of Types and Polymorphism in Logic Programming. In: Logic in Computer Science (LICS), pp. 310–321. IEEE Computer Society, Amsterdam (The Netherlands) (1991)
78. Korovin, K.: iProver – An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In: International Joint Conference on Automated Reasoning (IJCAR), *LNCS*, vol. 5195, pp. 292–298. Springer, Sydney (Australia) (2008)
79. Lipton, J., DeMarco, M.: Completeness and Cut-elimination in the Intuitionistic Theory of Types. Journal of Logic and Computation **15**, 821–854 (2005)

80. Maehara, S.: Lattice-valued Representation of the Cut-elimination Theorem. Tsukuba Journal of Mathematics **15**(2), 509–521 (1991)
81. Marques Silva, J.P., Lynce, I., Malik, S.: Conflict-Driven Clause Learning SAT Solvers. In: Handbook of Satisfiability, *Frontiers in Artificial Intelligence and Applications*, vol. 185, pp. 131–153. IOS Press (2009). ISBN 9781586039295
82. Mentré, D., Marché, C., Filliâtre, J.C., Asuka, M.: Discharging Proof Obligations from Atelier B using Multiple Automated Provers. In: Abstract State Machines, Alloy, B, VDM, and Z (ABZ), *LNCS*, vol. 7316, pp. 238–251. Springer, Pisa (Italy) (2012)
83. Murray, N.V., Rosenthal, E.: Theory Links: Applications to Automated Theorem Proving. Journal of Symbolic Computation (JSC) **4**(2), 173–190 (1987)
84. Nelson, G., Oppen, D.C.: Simplification by Cooperating Decision Procedures. ACM Transactions on Programming Languages and Systems (TOPLAS) **1**(2), 245–257 (1979)
85. Nelson, G., Oppen, D.C.: Fast Decision Procedures Based on Congruence Closure. Journal of the ACM **27**(2), 356–364 (1980)
86. Nerode, A., Shore, R.A.: Logic for Applications. Texts and Monographs in Computer Science. Springer (1993). ISBN 9780387941295
87. Ohlbach, H.J., Siekmann, J.H.: The Markgraf Karl Refutation Procedure. In: Computational Logic, Essays in Honor of Alan Robinson, pp. 41–112. The MIT Press (1991)
88. Okada, M.: Phase Semantic Cut-Elimination and Normalization Proofs of First- and Higher-Order Linear Logic. Theoretical Computer Science (TCS) **227**, 333–396 (1999)
89. Oppacher, F., Suen, E.: HARP: A Tableau-Based Theorem Prover. Journal of Automated Reasoning (JAR) **4**(1), 69–100 (1988)
90. Petermann, U.: Towards a Connection Procedure with Built in Theories. In: Logics in AI, European Workshop JELIA, *LNCS*, vol. 478, pp. 444–543. Springer, Amsterdam (The Netherlands) (1990)
91. Plotkin, G.D.: Building-in Equational Theories. Machine Intelligence **7**, 73–90 (1972)
92. Prawitz, D.: Natural Deduction. A Proof-Theoretical Study. Stockholm Studies in Philosophy **3** (1965)
93. Rabe, F.: First-Order Logic with Dependent Types. In: International Joint Conference on Automated Reasoning (IJCAR), *LNCS*, vol. 4130, pp. 377–391. Springer, Seattle (WA, USA) (2006)
94. Robinson, J.A.: A Machine-Oriented Logic Based on the Resolution Principle. Journal of the ACM **12**(1), 23–41 (1965)
95. Rümmer, P.: A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In: Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), *LNCS*, vol. 5330, pp. 274–289. Springer, Doha (Qatar) (2008)
96. Saillard, R.: Typechecking in the $\lambda\Pi$-Calculus Modulo: Theory and Practice. Ph.D. thesis, École Nationale Supérieure des Mines de Paris (2015)
97. Schmitt, P.H., Wernecke, W.: Tableau Calculus for Order Sorted Logic. In: Sorts and Types in Artificial Intelligence, pp. 49–60 (1989)
98. Schultz, S.: System Description: E 0.81. In: International Joint Conference on Automated Reasoning (IJCAR), *LNCS*, vol. 3097, pp. 223–228. Springer, Cork (Ireland) (2004)
99. Schwichtenberg, H., Troelstra, A.S.: Basic Proof Theory, 2nd edn. Cambridge University Press (2000). ISBN 9780521779111
100. Shankar, N.: Little Engines of Proof. In: Formal Methods Europe (FME), *LNCS*, vol. 2391, pp. 1–20. Springer, Copenhagen (Denmark) (2002)
101. Shostak, R.E.: Deciding Combinations of Theories. Journal of the ACM **31**(1), 1–12 (1984)
102. Stickel, M.E.: Automated Deduction by Theory Resolution. Journal of Automated Reasoning (JAR) **1**(4), 333–355 (1985)
103. Strub, P.-Y.: Coq Modulo Theory. In: Computer Science Logic (CSL), *LNCS*, vol. 6247, pp. 529–543. Springer, Brno (Czech Republic) (2010)
104. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. Journal of Automated Reasoning (JAR) **43**(4), 337–362 (2009)
105. Sutcliffe, G., Schulz, S., Claessen, K., Baumgartner, P.: The TPTP Typed First-Order Form with Arithmetic. In: Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), *LNCS*, vol. 7180, pp. 406–419. Springer, Mérida (Venezuela) (2012)

106. Szabo, M.E. (ed.): Collected Papers of Gerhard Gentzen. Studies in Logic and the Foundation of Mathematics. North-Holland Publishing Company (1969). ISBN 9780720422542
107. The BWare Project: (2012). http://bware.lri.fr/
108. Tinelli, C.: Cooperation of Background Reasoners in Theory Reasoning by Residue Sharing. Journal of Automated Reasoning (JAR) **30**(1), 1–31 (2003)
109. Walther, C.: Many-Sorted Inferences in Automated Theorem Proving. In: Sorts and Types in Artificial Intelligence, *LNCS*, vol. 418, pp. 18–48. Springer, Eringerfeld (Germany) (1989)
110. Weidenbach, C.: First-Order Tableaux with Sorts. Logic Journal of the IGPL **3**(6), 887–906 (1995)
111. Weidenbach, C.: Combining Superposition, Sorts and Splitting. In: Handbook of Automated Reasoning, vol. 2, pp. 1965–2013. Elsevier and MIT Press (2001)
112. Wos, L., Robinson, G.A., Carson, D.F.: Efficiency and Completeness of the Set of Support Strategy in Theorem Proving. Journal of the ACM **12**(4), 536–541 (1965)