SMT Solving Modulo Tableau and Rewriting Theories

Guillaume Bury[1], Simon Cruanes[2], and David Delahaye[3]

[1] Inria/LSV/ENS Paris-Saclay, Cachan, France
Guillaume.Bury@inria.fr
[2] Aesthetic Integration, Austin (Texas), USA
simon@aestheticintegration.com
[3] LIRMM/Université de Montpellier, Montpellier, France
David.Delahaye@lirmm.fr

Abstract

We propose an automated theorem prover that combines an SMT solver with tableau calculus and rewriting. Tableau inference rules are used to unfold propositional content into clauses while atomic formulas are handled using satisfiability decision procedures as in traditional SMT solvers. To deal with quantified first order formulas, we use metavariables and perform rigid unification modulo equalities and rewriting, for which we introduce an algorithm based on superposition, but where all clauses contain a single atomic formula. Rewriting is introduced along the lines of deduction modulo theory, where axioms are turned into rewrite rules over both terms and propositions. Finally, we assess our approach over a benchmark of problems in the set theory of the B method.

1 Introduction

Over the last past few years, SMT solvers have appeared as very efficient tools to reason over some well identified theories (equality, uninterpreted functions, linear arithmetic, arrays, etc.), and have allowed us to bring SAT solving toward first order logic. Although modern SMT solvers support first order logic, most of them use heuristic quantifier instantiation for incorporating quantifier reasoning with ground decision procedures. This mechanism is relatively effective in some cases in practice, but it is not refutationally complete for first order logic. Hints (triggers) are usually required, and it is sensitive to the syntactic structure of the formula, so that it fails to prove formulas that can be easily discharged by provers based on more traditional first order proof search methods (tableaux, resolution, etc.).

In this paper, we propose to improve first order proof search by introducing rewriting as a regular SMT theory, along the lines of deduction modulo theory. Deduction modulo theory [10] focuses on the computational part of a theory, where axioms are transformed into rewrite rules, which induces a congruence over propositions, and where reasoning is performed modulo this congruence. In deduction modulo theory, this congruence is then induced by a set of rewrite rules over both terms and propositions. In our proposal, this congruence is used a first time to speed up ground reasoning by computing normal forms for terms, but this still yields an incomplete algorithm.

We thus propose to overcome the problem of completeness for first order logic by using tableau calculus as an SMT theory. The tableau calculus rules are used to encode propositional content into clauses[1] while atomic formulas are handled using satisfiability decision procedures as in regular SMT solvers. To deal with quantified first order formulas, we use metavariables and perform rigid unification modulo equalities and modulo rewriting, for which we introduce an algorithm based on superposition, but where all clauses contain a single atomic formula.

[1] Similar to a lazy CNF transformation with named formulas.

Our approach provides several advantages compared to usual SMT solving and first order proof search methods. First, we benefit from the efficiency of a SAT solver core together with a complete method of instantiation (when a propositional model is found, we try to find a conflict between two literals by unification). Second, it should be noted that our approach requires no change in the architecture of the SMT solver, since the tableau calculus and rewriting are seen as regular theories. Finally, no preliminary Skolemization and Conjunctive Normal Form (CNF) transformation is required. This transformation is performed lazily by applying the tableau rules progressively when a literal is propagated or decided. This makes the production of genuine output proofs easier, contrary to the usual approach, where the Skolemization/CNF translation is realized at the beginning and externalized with respect to the proof search.

Our proposal combining SMT solving with tableau calculus and rewriting has been implemented and the corresponding tool is called ArchSAT. This tool is able to deal with first order logic extended to polymorphic types à la ML, through a type system in the spirit of [3]. To test this tool, we propose a suite of benchmarks in the framework of the set theory of the B method [1]. This theory [7] has been expressed using first order logic extended to polymorphic types and turned into a theory that is compatible with deduction modulo theory, i.e. where a large part of axioms has been turned into rewrite rules. The benchmark itself gathers 319 lemmas coming from Chap. 2 of the B-Book [1].

The paper is organized as follows: in Sec. 2, we introduce the tableau and rewriting theories; we then describe, in Sec. 3, our mechanism of equational reasoning by means of rigid unit superposition; finally, in Sec. 4, we present some experimental results obtained by running our implementation over a benchmark of problems in the B set theory.

2 SAT Solving Modulo Tableau and Rewriting Theories

Compared to genuine tableau automated theorem provers, like Princess or Zenon for example, our approach has the benefit of being versatile since the tableau rules are actually integrated as a regular SMT theory. This way, the tableau rules can be easily combined with other theories, such as equality logic with uninterpreted functions or arithmetic. The way we integrate the tableau rules into the SMT solver (mainly by boxing/unboxing first order formulas) is close to what is done in the Satallax tool [5]. The difference resides in the fact that we are in a pure first order framework, which has significant consequences in the management of quantifiers and unification in particular (see Sec. 3).

Regarding the integration of rewriting, automated theorem provers rely on several solutions (superposition rule for first order provers, triggers for SMT solvers, etc.). But deduction modulo theory [10] is probably the most general approach, where a theory can be partly turned into a set of rewrite rules over both terms and formulas. Several proof search methods have been extended to deduction modulo theory, resulting in tools such as iProver Modulo and Zenon Modulo. This paper can be seen as a continuation of these previous experiments adapted to the framework of SMT solving.

2.1 The Tableau Theory

We introduce $\mathcal{T}$ and $\mathcal{F}$ respectively, the sets of first order terms and formulas over the signature $\mathcal{S} = (\mathcal{S}_\mathcal{F}, \mathcal{S}_\mathcal{P})$, where $\mathcal{S}_\mathcal{F}$ is the set of function symbols, and $\mathcal{S}_\mathcal{P}$, the set of predicate symbols, such that $\mathcal{S}_\mathcal{F} \cap \mathcal{S}_\mathcal{P} = \emptyset$. The set $\mathcal{T}$ is extended with two kinds of terms specific to tableau proof search. First are ϵ-terms (used instead of Skolemization) of the form $\epsilon(x).P(x)$, where $P(x)$ is a formula, and which means some x that satisfies $P(x)$, if it exists. And second, metavariables (often named free variables in the tableau-related literature) of the form X_P, where P is the formula that introduces the metavariable, and which is either $\forall x.Q(x)$ or $\neg\exists x.Q(x)$, with $Q(x)$ a formula. Metavariables are written using capitalized letters (such as X_P or Y_P), will never be substituted[2]. They will also be considered rigid in the following, meaning that in a context where we try and unify terms, by building a mapping from metavariables to terms, each metavariables may be bound at most once. This is because in tableaux calculus, if you instantiate a formula multiple times, it may create additional propositional branches, which we do not want. Instead we consider metavariables rigid, and we'll generate multiple (distinct) metavariables for the same formula if needed.

A boxed formula is of the form $\lfloor P \rfloor$, where P is a formula. A boxed formula is called an atom, and a literal is either an atom, or the negation of an atom. A literal is such that there is no negation on top of the boxed formula (which means that $\lfloor \neg P \rfloor$ is automatically translated into $\neg\lfloor P \rfloor$, and $\neg\neg\lfloor P \rfloor$ into $\lfloor P \rfloor$). A clause is a disjunction of literals. It should be noted that SMT solving usually reasons over sets of clauses composed of first order terms; here, a literal is a first order formula (possibly with quantifiers), which requires to box formulas to get a regular SAT solving problem where boxed formulas are propositional variables. The state of a SMT solver can be represented as $T \parallel S$, with T the trail of the SMT, i.e. a list of boxed formula, in chronological order left to right, and S the set of clauses to be satisfied by the solver. One very important point is that SMT theories can and will often add some clauses to the set S of clauses that the solver tries to satisfy. This is sound as long as only tautologies are added. In the following, this will often be abbreviated by only writing "adding the clause".

Tableau proof search method is integrated as a regular theory in our SMT solver. Whenever a literal l is decided or propagated by the solver, the tableaux theory generates the set of clauses $[\![l]\!]$, where the function $[\![\cdot]\!]$ is described by the rules of Fig. 1, and then add these clauses[3]. It should be noted that we use the same names for the rules as in tableau calculus (α-rules, β-rules, etc.), but there is no precedence between rules and therefore no priority in the application of the rules contrary to the tableau proof search method (where α rules are applied before β-rules, and so on). Application of this can be seen in Figure 2, on the third line, where once the solver has propagated that $A \equiv \lfloor B \to C \rfloor$ is false, the tableaux theory adds two new clauses: $A \vee B$ and $A \vee \neg C$, whose aim is to ensure that as long as A is false, B will be true and C will be false. The clauses added should be seen as an implication: $\neg A \to B$ and $\neg A \to \neg C$, where the presence of A in the clauses is important because the tableaux theory does not know whether A is always false or not: it might be that A being false was a decision of the SAT solver (or a consequence of a decision), and thus might be backtracked; in this case it is important to not have the propagations of B and $\neg C$ depend on the propagation of $\neg A$.

When the SMT solver finds a model M of the current set of clauses, we look for a conflict in M between boxed atomic formulas by unification. If there exist two literals l and $\neg l'$ in M such that $l = \lfloor Q \rfloor$ and $l' = \lfloor R \rfloor$, with Q and R two formulas, then for each binding $(X_{\forall x.P(x)} \mapsto t) \in \mathrm{mgu}(Q, R)$ (resp. $(X_{\neg\exists x.P(x)} \mapsto t) \in \mathrm{mgu}(Q, R)$) we can generate the clauses $[\![\lfloor \forall x.P(x) \rfloor]\!]$ (resp. $[\![\neg\lfloor \exists x.P(x) \rfloor]\!]$) using the rule $\gamma_{\forall\mathrm{inst}}$ (resp. $\gamma_{\neg\exists\mathrm{inst}}$) of Fig. 1, and then add these new clauses. For instance, suppose we find a model where $\lfloor P(X_{\forall x.P(x)}) \rfloor$ is true and $\lfloor P(a) \rfloor$ is false, then we'll add the following clause to the solver: $\neg\lfloor \forall x.P(x) \rfloor \vee \lfloor P(a) \rfloor$, representing the instantiation of $\forall x.P(x)$ using the term a. This clause will ensure that the model where $\lfloor P(a) \rfloor$ is false cannot happen again as long as $\lfloor \forall x.P(x) \rfloor$ is true. Note that we did not substitute any metavariables, we only added some tautologies, thus there is no need for

[2] More generally, terms are immutable and are never modified in place: application of a substitution creates new terms that may be used, but will never modify existing terms.

[3] We use all rules except the instantiation γ-rules: $\gamma_{\forall\mathrm{inst}}$ and $\gamma_{\neg\exists\mathrm{inst}}$ which are only used once adequate terms for instantiation have been found

Analytic Rules

$$(\alpha_{\land}) \quad \llbracket \lfloor P \land Q \rfloor \rrbracket = \left\{ \begin{array}{l} \neg\lfloor P \land Q \rfloor \lor \lfloor P \rfloor \\ \neg\lfloor P \land Q \rfloor \lor \lfloor Q \rfloor \end{array} \right. \qquad (\beta_{\neg\land}) \quad \llbracket \neg\lfloor P \land Q \rfloor \rrbracket = \lfloor P \land Q \rfloor \lor \neg\lfloor P \rfloor \lor \neg\lfloor Q \rfloor$$

$$(\beta_{\lor}) \quad \llbracket \lfloor P \lor Q \rfloor \rrbracket = \neg\lfloor P \lor Q \rfloor \lor \lfloor P \rfloor \lor \lfloor Q \rfloor \qquad (\alpha_{\neg\lor}) \quad \llbracket \neg\lfloor P \lor Q \rfloor \rrbracket = \left\{ \begin{array}{l} \lfloor P \lor Q \rfloor \lor \neg\lfloor P \rfloor \\ \lfloor P \lor Q \rfloor \lor \neg\lfloor Q \rfloor \end{array} \right.$$

$$(\beta_{\Rightarrow}) \quad \llbracket \lfloor P \Rightarrow Q \rfloor \rrbracket = \neg\lfloor P \Rightarrow Q \rfloor \lor \neg\lfloor P \rfloor \lor \lfloor Q \rfloor \qquad (\alpha_{\neg\Rightarrow}) \quad \llbracket \neg\lfloor P \Rightarrow Q \rfloor \rrbracket = \left\{ \begin{array}{l} \lfloor P \Rightarrow Q \rfloor \lor \lfloor P \rfloor \\ \lfloor P \Rightarrow Q \rfloor \lor \neg\lfloor Q \rfloor \end{array} \right.$$

$$(\beta_{\Rightarrow}) \quad \llbracket \lfloor P \Leftrightarrow Q \rfloor \rrbracket = \left\{ \begin{array}{l} \neg\lfloor P \Leftrightarrow Q \rfloor \lor \lfloor P \Rightarrow Q \rfloor \\ \neg\lfloor P \Leftrightarrow Q \rfloor \lor \lfloor Q \Rightarrow P \rfloor \end{array} \right. \qquad (\beta_{\neg\Rightarrow}) \quad \llbracket \neg\lfloor P \Leftrightarrow Q \rfloor \rrbracket = \begin{array}{l} \lfloor P \Leftrightarrow Q \rfloor \\ \lor\neg\lfloor P \Rightarrow Q \rfloor \\ \lor\neg\lfloor Q \Rightarrow P \rfloor \end{array}$$

δ-Rules

$$\begin{array}{lcll} \llbracket \lfloor \exists x.P(x) \rfloor \rrbracket & = & \neg\lfloor \exists x.P(x) \rfloor \lor \lfloor P(\epsilon(x).P(x)) \rfloor & (\delta_{\exists}) \\ \llbracket \neg\lfloor \forall x.P(x) \rfloor \rrbracket & = & \lfloor \forall x.P(x) \rfloor \lor \neg\lfloor P(\epsilon(x).\neg P(x)) \rfloor & (\delta_{\neg\forall}) \end{array}$$

γ-Rules

$$\begin{array}{lcll} \llbracket \lfloor \forall x.P(x) \rfloor \rrbracket & = & \neg\lfloor \forall x.P(x) \rfloor \lor \lfloor P(X_{\forall x.P(x)}) \rfloor & (\gamma_{\forall M}) \\ \llbracket \neg\lfloor \exists x.P(x) \rfloor \rrbracket & = & \lfloor \exists x.P(x) \rfloor \lor \neg\lfloor P(X_{\neg\exists x.P(x)}) \rfloor & (\gamma_{\neg\exists M}) \\ \llbracket \lfloor \forall x.P(x) \rfloor \rrbracket & = & \neg\lfloor \forall x.P(x) \rfloor \lor \lfloor P(t) \rfloor & (\gamma_{\forall\mathrm{inst}}) \\ \llbracket \neg\lfloor \exists x.P(x) \rfloor \rrbracket & = & \lfloor \exists x.P(x) \rfloor \lor \neg\lfloor P(t) \rfloor & (\gamma_{\neg\exists\mathrm{inst}}) \end{array}$$

Figure 1: Rules of Tableau Theory

any special backtracking.

2.2 The Rewriting Theory

A rewriting theory allows us to replace instantiations by computations in the SMT solver. We aim to integrate rewriting in the broadest sense of the term as proposed by deduction modulo theory. Deduction modulo theory [10] focuses on the computational part of a theory, where axioms are transformed into rewrite rules, which induces a congruence over formulas, and where reasoning is performed modulo this congruence. In deduction modulo theory, this congruence is then induced by a set of rewrite rules over both terms and formulas.

In the following, we borrow some of the notations and definitions of [10]. We call FV the function that returns the set of free variables of a term or a formula. A term rewrite rule is a pair of terms denoted by $l \longrightarrow r$, where $\mathrm{FV}(r) \subseteq \mathrm{FV}(l)$. A formula rewrite rule is a pair of formulas denoted by $l \longrightarrow r$, where l is an atomic formula and r is an arbitrary formula, and where $\mathrm{FV}(r) \subseteq \mathrm{FV}(l)$. A class rewrite system is a pair of rewrite systems, denoted by $\mathcal{RE}$, consisting of $\mathcal{R}$, a set of formula rewrite rules, and $\mathcal{E}$, a set of term rewrite rules. Given a class rewrite system $\mathcal{RE}$, the relations $=_{\mathcal{E}}$ and $=_{\mathcal{RE}}$ are the congruences generated respectively by the sets $\mathcal{E}$ and $\mathcal{R} \cup \mathcal{E}$. In the following, we use the standard concepts of subterm and term replacement: given an occurrence ω in a formula P, we write $P_{|\omega}$ for the term or formula at ω, and $P[t]_{\omega}$ for the formula obtained by replacing $P_{|\omega}$ by t in P at ω. Given a class rewrite system $\mathcal{RE}$, the formula P $\mathcal{RE}$-rewrites to P', denoted by $P \longrightarrow_{\mathcal{RE}} P'$, if $P =_{\mathcal{E}} Q$, $Q_{|\omega} = \sigma(l)$, and $P' =_{\mathcal{E}} Q[\sigma(r)]_{\omega}$, for some rule $l \longrightarrow r \in \mathcal{R}$, some formula Q, some occurrence ω in Q, and some substitution σ. The relation $=_{\mathcal{RE}}$ is not decidable in general, but there are some cases where this relation is decidable depending on the class rewrite system $\mathcal{RE}$ and the rewrite relation $\longrightarrow_{\mathcal{RE}}$. In particular, if the rewrite relation $\longrightarrow_{\mathcal{RE}}$ is confluent and (weakly) terminating, then the relation $=_{\mathcal{RE}}$ is decidable.

The rewriting theory is integrated into the SMT solver in a similar way as for the tableau theory. Whenever a literal is propagated or decided, we generate some clauses, and add them. The clauses we generate express the equivalence (resp. equality) between a formula (resp. term) and its normal form in the rewrite system. More precisely, given a literal $\lfloor P \rfloor$, where P is a formula, and a formula P' such that $P =_{\mathcal{RE}} P'$, we generate and add the following clause:

$$\left(\bigvee_{(l,r)\in\mathcal{R}} \neg\lfloor \forall \vec{x}.l \Leftrightarrow r \rfloor \right) \vee \left(\bigvee_{(l,r)\in\mathcal{E}} \neg\lfloor \forall \vec{x}.l = r \rfloor \right) \vee \lfloor P \Leftrightarrow P' \rfloor$$

where $\vec{x} = \mathrm{FV}(l) \cup \mathrm{FV}(r)$.

It should be noted that in usual SMT solvers, rewriting can be emulated by means of triggers that are actually the left-hand side members of the class rewrite system $\mathcal{RE}$ introduced above. But in our rewriting theory, we can generate the formula resulting from the rewriting steps, while triggers can just generate bindings, i.e. instances of the rewrite rules, which are used later to relate the initial and rewritten formulas. Moreover, in our case, we can perform several rewritings at once, while a trigger can only emulate one rewriting at a time.

Let us illustrate the use of the rewriting theory by means of an example in set theory. Let us prove that $(\forall s,t.s \subseteq t \Leftrightarrow \forall x.x \in s \Rightarrow x \in t) \Rightarrow a \subseteq a$, where a is a constant. The proof is given in Fig. 2. Note that, for the rewriting theory, any boxed quantified formula can be understood as a rewrite rule as long as they represent one, for instance, the formula B in the example in Figure 2.

3 Equational Reasoning with Rigid Unit Superposition

There are many ways of integrating equational reasoning in tableau methods [2,4,8,12]. Because our prover does not rely on clausal forms, but on arbitrary formulas with quantifiers occurring deep inside branches, we deal with rigid variables, i.e. variables that should be instantiated only once, since multiple instantiations would create new propositional branches. In order to find instantiations for universally quantified formulas, the procedure described in 2.1 need a unification algorithm. In order to be complete, this algorithm needs to solve rigid E-unification modulo rewrite rules: assume a set of equations E, containing rigid variables, a rewrite system $\mathcal{RE}$, and target terms s and t; we want a substitution σ such that $\bigwedge_{e\in E} e\sigma \vdash s\sigma =_{\mathcal{RE}} t\sigma$. Such a substitution is a solution to the rigid E-unification problem.

We propose here an approach based on superposition with rigid variables, as in previous work by Degtyarev and Voronkov [8] and earlier work on rigid paramodulation [13], but with significant differences. First, in order to avoid constraint solving, we do not use basic superposition nor constraints. Second, we introduce a merging rule, which factors together intermediate (dis)equations that are alpha-equivalent: with multiple instances of some of the quantified formulas (amplification), it becomes important not to duplicate work. In this aspect, our calculus is quite close to labeled unit superposition [11] when using sets as labels. Third, unlike rigid paramodulation, we use a term ordering to orient the equations.

$$
\begin{array}{lll}
\emptyset \parallel \neg A & \longrightarrow & \text{(unit prop)} \\
\boldsymbol{\neg A} \parallel \mathcal{C}_1 = \neg A & \longrightarrow & \text{(Tableaux)} \\
\neg A \parallel \mathcal{C}_1, \boldsymbol{\mathcal{C}_2 = A \vee B}, \boldsymbol{\mathcal{C}_3 = A \vee \neg C} & \longrightarrow & \text{(unit prop)} \times 2 \\
\neg A, \boldsymbol{B}, \boldsymbol{\neg C} \parallel \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3 & \longrightarrow & \text{(Rewriting)} \\
\neg A, B, \neg C \parallel \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \boldsymbol{\mathcal{C}_4 = \neg B \vee D} & \longrightarrow & \text{(unit prop)} \\
\neg A, B, \neg C, \boldsymbol{D} \parallel \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4 & \longrightarrow & \text{(Tableaux)} \\
\neg A, B, \neg C, D \parallel \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \boldsymbol{\mathcal{C}_5 = \neg D \vee E}, \boldsymbol{\mathcal{C}_6 = \neg D \vee F} & \longrightarrow & \text{(unit prop)} \times 2 \\
\neg A, B, \neg C, D, \boldsymbol{E}, \boldsymbol{F} \parallel \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_5, \mathcal{C}_6 & \longrightarrow & \text{(Tableaux)} \\
\neg A, B, \neg C, D, E, F \parallel \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_5, \mathcal{C}_6, \boldsymbol{\mathcal{C}_7 = \neg F \vee \neg G \vee C} & \longrightarrow & \text{(unit prop)} \\
\neg A, B, \neg C, D, E, F, \boldsymbol{\neg G} \parallel \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_5, \mathcal{C}_6, \mathcal{C}_7 & \longrightarrow & \text{(Tableaux)} \\
\neg A, B, \neg C, D, E, F, \neg G \parallel \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_5, \mathcal{C}_6, \mathcal{C}_7, \boldsymbol{\mathcal{C}_8 = G \vee \neg H} & \longrightarrow & \text{(unit prop)} \\
\neg A, B, \neg C, D, E, F, \neg G, \boldsymbol{\neg H} \parallel \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_5, \mathcal{C}_6, \mathcal{C}_7, \mathcal{C}_8 & \longrightarrow & \text{(Tableaux)} \\
\neg A, B, \neg C, D, E, F, \neg G, \neg H \parallel \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_5, \mathcal{C}_6, \mathcal{C}_7, \mathcal{C}_8, & & \\
\quad \boldsymbol{\mathcal{C}_9 = H \vee I}, \boldsymbol{\mathcal{C}_{10} = H \vee \neg I} & \longrightarrow & \text{(unit prop)} \\
\neg A, B, \neg C, D, E, F, \neg G, \neg H, \boldsymbol{I} \parallel \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_5, \mathcal{C}_6, \mathcal{C}_7, \mathcal{C}_8, \mathcal{C}_9, \mathcal{C}_{10} & \longrightarrow & \text{(unsat)} \\
\textbf{unsat} & &
\end{array}
$$

where:

$$
\begin{array}{ll}
A \equiv \lfloor (\forall s, t. s \subseteq t \Leftrightarrow \forall x. x \in s \Rightarrow x \in t) \Rightarrow a \subseteq a \rfloor & \\
B \equiv \lfloor \forall s, t. s \subseteq t \Leftrightarrow \forall x. x \in s \Rightarrow x \in t \rfloor & C \equiv \lfloor a \subseteq a \rfloor \\
D \equiv \lfloor a \subseteq a \Leftrightarrow \forall x. x \in a \Rightarrow x \in a \rfloor & E \equiv \lfloor a \subseteq a \Rightarrow \forall x. x \in a \Rightarrow x \in a \rfloor \\
F \equiv \lfloor (\forall x. x \in a \Rightarrow x \in a) \Rightarrow a \subseteq a \rfloor & G \equiv \lfloor \forall x. x \in a \Rightarrow x \in a \rfloor \\
H \equiv \lfloor \epsilon_x \in a \Rightarrow \epsilon_x \in a \rfloor & I \equiv \lfloor \epsilon_x \in a \rfloor
\end{array}
$$

with: $\epsilon_x = \epsilon(x).\neg(x \in a \Rightarrow x \in a)$

Figure 2: Example of Proof Using the Tableaux and Rewriting Theory

3.1 Preliminary Definitions

We write $s \approx t \mid \Sigma$ (resp. $s \not\approx t \mid \Sigma$) for the unit clause that contains exactly one equation (resp. disequation) under hypothesis Σ (which is a set of substitutions). We write $\emptyset \mid \Sigma$ for the empty clause under hypothesis Σ. The meaning of $s \approx t \mid \Sigma$ is that for every $\sigma \in \Sigma$, $s \approx t$ is provable using the substitution σ for the metavariables. We also define $\mathsf{rename}(e)$, where e is a (dis)equation, as follows: let σ map every rigid variable of e to a fresh non-rigid variable, then $\mathsf{rename}(e) = e\sigma \mid \{\sigma\}$. For example, $\mathsf{rename}(p(X) \approx a)$ is $p(v_1) \approx a \mid \{X \mapsto v_1\}$.

As can be noticed, we keep a set of substitutions, rather than unit clauses paired with individual substitutions, in order to avoid duplicating the work for alpha-equivalent clauses. Indeed, because of amplification, many instances of a given (dis)equation might be present in a branch of the tableau. It would be inefficient to repeat the same inference steps with each variant of the axioms. Because we apply $\mathsf{rename}(e)$ on every input equality e, clauses do not share any variable, though they may share meta-variables in their attached sets of substitutions.

Considering a substitution as a function from variables to terms, we define the domain of a substitution σ as the set of variables that have a non-trivial binding in σ.[4] The co-domain of a substitution is the set of variables occurring in terms in the image of the domain of the substitution. In the following, we will consider idempotent substitutions, i.e. substitutions for which the domain and co-domain have an empty intersection.

The composition of two substitutions σ and σ', denoted by $\sigma \circ \sigma'$, is well-defined if and only if

[4] A trivial binding maps a variable to itself.

the domains of σ and σ' have no intersection. In this case, $\sigma \circ \sigma' \triangleq \{x \mapsto (x\sigma)\sigma' | x \in \text{domain}(\sigma)\}$. This definition extends to sets of substitutions: $\Sigma \circ \sigma' \triangleq \{\sigma \circ \sigma' | \sigma \in \Sigma\}$. We then have $\sigma \leq \sigma'$ if and only if $\exists \sigma''.\ \sigma \circ \sigma'' = \sigma'$. This notion also extends to sets of substitutions: $\Sigma \leq \Sigma'$ if and only if $\forall \sigma' \in \Sigma'.\ \exists \sigma \in \Sigma. \sigma \leq \sigma'$. The merging of two substitutions $\sigma \uparrow \sigma'$ is the supremum of $\{\sigma, \sigma'\}$ for the order $\leq$, if it exists, or $\bot$ otherwise. The merging of sets of substitutions is $\Sigma \uparrow \Sigma' \triangleq \{\sigma \uparrow \sigma' \mid \sigma \in \Sigma, \sigma' \in \Sigma'\ , \sigma \uparrow \sigma' \neq \bot\}$.

To perform an inference step between two unit (dis)equations, we merge their sets of substitutions. An inference rule is said to be successful if the merging of the premises' substitution sets is non-empty. For example, the resolution step between $p(x,x)|\{X \mapsto a\}$ and $\neg p(y,b)|\{X \mapsto y\}$ is not possible, because the result would need to map X to a and b, which is impossible because X is rigid.

3.2 Inference System

In Fig. 3, we present the rules for unit superposition with rigid variables. We adopt notations and names from Schulz's paper on E [14]. A single bar denotes an inference, i.e. we add the result to the saturation set, whereas a double bar is a simplification in which the premises are replaced by the conclusion(s). The relation $\prec$ is a reduction ordering, used to orient equations and restrict inferences, thus pruning the search space. Typically, $\prec$ is one of RPO or KBO. The rules of Fig. 3 work as described below:

ER is equality resolution, where a disequation $s \not\approx t \mid \Sigma$ is solved by syntactically unifying s and t with σ, if σ is compatible with Σ.

SN/SP is superposition into positive or negative literals. A subterm of u is rewritten using $s \approx t$ after unifying it with s by σ. The rewriting is done only if $s\sigma \not\preceq t\sigma$, a sufficient (but not necessary) condition for a ground instance of $s\sigma \approx t\sigma$ to be oriented left-to-right.

TD1 deletes trivial equations that will never contribute to a proof.

TD2 deletes clauses with an empty set of substitutions. In practice, we only apply a rule if the conclusion is labeled with a non-empty set of substitutions.

ME merges two alpha-equivalent clauses into a single clause, by merging the sets of substitutions. This rule is very important in practice, to prevent the search space from exploding due to the duplicates of most formulas. Superposition deals with this explosion by removing duplicates using subsumption, but in our context subsumption is not complete because rigid variables are only proxy for ground terms: even if $C\sigma \subseteq D$, the one ground instance of C might not be compatible with the ground instance of D.

ES is a restricted form of equality subsumption. The active equation $s \approx t \mid \Sigma$ can be used to delete another clause, as in E [14]. However, ES only works if s and t are syntactically equal to the corresponding subterms in the subsumed clause C. Otherwise, there is no guarantee that further instantiations will not make $s \approx t$ incompatible with C. Moreover, C needs not be entirely removed. Only its substitutions that are compatible with Σ are subsumed.

RN/RP are rewriting of clauses, which only works for syntactical equality, not matching.

Rule **SN/SP** generates as many equations as there are in the set $(\Sigma \circ \sigma'') \uparrow (\Sigma' \circ \sigma'')$ because all substitutions may not always be merged. For instance, given two unit clauses

$f(x) = t|\{\{X_1 \mapsto x\}, \{X_2 \mapsto x\}\}$ and $f(a) = v|\{\{X_1 \mapsto a\}\}$, rule **SP** allows us to derive two distinct equations $(t = v)\{x \mapsto a\}|\{\{X_1 \mapsto a\}\}$ and $(t = v)\{x \mapsto a\}|\{\{X_1 \mapsto a; X_2 \mapsto a\}\}$, which are non-mergeable.

$$\text{SN/SP}\ \frac{s \approx t \mid \Sigma \qquad u\ R\ v \mid \Sigma'}{\sigma''(u[p \leftarrow t]\ R\ v) \mid \sigma'''}\ \text{if} \begin{cases} \sigma'' = \text{mgu}(u_{|p}, s) \quad u_{|p} \notin V \\ \sigma''(s) \not\preceq \sigma''(t) \qquad \sigma''(u) \not\preceq \sigma''(v) \\ \sigma''' \in (\Sigma \circ \sigma'') \uparrow (\Sigma' \circ \sigma'') \\ R \in \{\approx, \not\approx\} \end{cases}$$

$$\text{ER}\ \frac{s \not\approx t \mid \Sigma}{\emptyset \mid \Sigma \circ \sigma}\ \text{if}\ \sigma = \text{mgu}(s,t) \qquad \text{TD1}\ \frac{s \approx s \mid \Sigma}{\top} \qquad \text{TD2}\ \frac{s\ R\ t \mid \emptyset}{\top}\ R \in \{\approx, \not\approx\}$$

$$\text{ME}\ \frac{\rho(u) \approx \rho(v) \mid \Sigma \qquad u \approx v \mid \Sigma'}{\rho(u) \approx \rho(v) \mid \Sigma \cup (\Sigma' \circ \rho)}\ \rho \text{ is a variable renaming}$$

$$\text{ES}\ \frac{s \approx t \mid \Sigma \qquad u[p \leftarrow s] \approx u[p \leftarrow t] \mid \Sigma' \cup \Sigma''}{s \approx t \mid \Sigma \qquad u[p \leftarrow s] \approx u[p \leftarrow t] \mid \Sigma'}\ \text{if} \begin{cases} \Sigma'' \neq \emptyset \\ \Sigma \leq \Sigma'' \end{cases}$$

$$\text{RP}\ \frac{s \approx t \mid \Sigma \qquad u \approx v \mid \Sigma'}{s \approx t \mid \Sigma \qquad u[p \leftarrow t] \approx v \mid \Sigma'}\ \text{if} \begin{cases} u_{|p} = s \\ s \succ t \\ \Sigma \leq \Sigma' \\ u \not\succeq v \text{ or } p \neq \lambda \end{cases}$$

$$\text{RN}\ \frac{s \approx t \mid \Sigma \qquad u \not\approx v \mid \Sigma'}{s \approx t \mid \Sigma \qquad u[p \leftarrow t] \not\approx v \mid \Sigma'}\ \text{if} \begin{cases} u_{|p} = s \\ s \succ t \\ \Sigma \leq \Sigma' \end{cases}$$

Figure 3: The Set of Rules for Unit Rigid Superposition

3.3 Rewriting

Rewrite rules can be integrated into the rigid unit superposition easily. In fact, a rewrite rule $l \longrightarrow r$ can be expressed as an equality with a hypothesis set consisting of a single trivial substitution $s \approx t \mid \{\emptyset\}$[5]. Since the trivial substitution is compatible with every substitution, it will never prevent any inference, thus allowing us to use the unit clause as many times as needed to rewrite terms without accumulating constraints, particularly using the rules RP and RN, whose side conditions are always verified by rewrite rules. Rigid unit superposition therefore provides an algorithm for rigid E-unification modulo rewrite rules, as detailed in the next paragraph.

3.4 Main Loop

Our objective with rigid E-unification is to attempt to close a branch of the tableau prover (i.e. a set of Boolean literals set to true). To do so, we first create a set of unit clauses to process

[5] The singleton set containing the trivial (or identity) substitution $\{\emptyset\}$, is not to be confused with $\emptyset$, the empty set

1	rewrite rule	$\mathrm{pair}(\mathrm{fst}(x), \mathrm{snd}(x))) \longrightarrow x$
2	axiom	$\mathrm{fst}(a) = \mathrm{fst}(b)$
3	axiom	$p(a) \neq p(\mathrm{pair}(\mathrm{fst}(b), X))$
4	rewr(1)	$\mathrm{pair}(\mathrm{fst}(x), \mathrm{snd}(x)) \approx x \mid \{\}$
5	rename(2)	$\mathrm{fst}(a) \approx \mathrm{fst}(b) \mid \{\}$
6	rename(3)	$p(a) \not\approx p(\mathrm{pair}(\mathrm{fst}(b), y)) \mid \{X \mapsto y\}$
7	RN(5,6)	$p(a) \not\approx p(\mathrm{pair}(\mathrm{fst}(a), y)) \mid \{X \mapsto y\}$
8	SN(4,7)	$p(a) \not\approx p(a) \mid \{X \mapsto \mathrm{snd}(a)\}$
9	ER(8)	$\emptyset \mid \{X \mapsto \mathrm{snd}(a)\}$

Figure 4: Proof of a Set Theory Problem

from the rewrite rules, and the renamed equational or atomic literals. Then, the given-clause algorithm is applied to try and saturate the set. Assuming a fair strategy, this will eventually find a solution (i.e. derive $\emptyset \mid \Sigma$) if there exists one. We refer the interested reader to [14] for more details.

Because the whole branch is managed by a single given-clause saturation loop, we look for all solutions susceptible to close the branch at the same time. Moreover, this technique is amenable to incrementality, i.e. every time a (dis)equation is decided by the SMT solver, we could add it to the saturation set and perform a number of steps of the given-clause algorithm.

To illustrate the calculus, we detail Fig. 4. a refutation of the following set of clauses stemming from set theory, where pair, fst, and snd are the constructor and destructors of tuples, f a function on tuples, and X a rigid variable:

$$\begin{array}{rcl} a & \preceq b & \\ \mathrm{pair}(\mathrm{fst}(x), \mathrm{snd}(x))) & \longrightarrow & x \\ \mathrm{fst}(a) & \approx & \mathrm{fst}(b) \\ p(a) & \not\approx & p(\mathrm{pair}(\mathrm{fst}(b), X)) \end{array}$$

4 Implementation and Experimental Results

In this section, we briefly describe the implementation of our approach introduced previously, and present some experimental results obtained by running this implementation over a benchmark of problems in the B set theory.

The algorithms described in this paper are implemented in the ArchSAT automated theorem prover[6]. It relies on the mSAT [6] library, derived from the Alt-Ergo Zero tool, which is a generic library for building automated deduction tools based on SAT solvers. ArchSAT (as well as mSAT) is written in OCaml. ArchSAT natively supports polymorphic types as described in [3].

4.1 Experimental Results

As a framework to test our tool, we consider the set theory of the B method [1]. This method is supported by some tool sets, such as Atelier B, which are used in industry to specify and build, by stepwise refinements, software that is correct by design. This theory is suitable as it can be easily turned into a theory that is compatible with deduction modulo theory, i.e. where

[6] Available at: https://gforge.inria.fr/projects/archsat.

319 Problems	ArchSAT	Zenon Modulo	Alt-Ergo
Proofs	272	138	232
Rate	85.3%	43.3%	72.7%
Total time (s)	268.69	2.86	8.42

Table 1: Experimental Results over the B Set Theory Benchmark

a large part of axioms can be turned into rewrite rules, and for which the rewriting theory proposed previously in Subsec. 2.2 works. Starting from the theory described in Chap. 2 of the B-Book [1], we therefore transform whenever possible the axioms and definitions into rewrite rules. The resulting theory has been introduced in [7]. As can be seen, the proposed theory is typed, using first order logic extended to polymorphic types à la ML, through a type system in the spirit of [3]. This extension to polymorphic types offers more flexibility, and in particular allows us to deal with theories that rely on elaborate type systems, like the B set theory (see Chap. 2 of the B-Book [1]).

To test ArchSAT in this theory, we consider a set of 319 lemmas coming from Chap. 2 of the B-Book [1][7]. These lemmas are properties of various difficulty regarding the set constructs introduced by the B method. It should be noted that these constructs and notations are, for a large part of them, specific to the B method, as they are used for the modeling of industrial projects, and are not necessarily standard in set theory.

As tools, we consider ArchSAT (development version[8]). We also include other automated theorem provers, able to deal with first order logic with polymorphic types and rewriting natively. In particular, we consider Zenon Modulo (version 0.4.2), a tableau-based prover that is an extension of Zenon to deduction modulo theory. To show the impact of rewriting on the results, we also include the Alt-Ergo SMT solver (version 1.01). It would have been possible to also consider provers dealing with pure first order logic and encode the polymorphic layer. But preliminary tests have been conducted and very low results have been obtained even for the best state-of-the-art provers (we have considered E and CVC4 in particular), which indicates that polymorphism encoding adds a lot of noise in proof search and is not effective in practice.

The experiment was run on an Intel Xeon E5-1650 v3 3.50 GHz computer, with a timeout of 90s (beyond this timeout, results do not change) and a memory limit of 1 GiB. The results are summarized in Tab. 1. In these results, we observe that ArchSAT obtains better results, in terms of proved problems, than Zenon Modulo and Alt-Ergo, which tends to show the effectiveness of our approach in practice. Looking at the cumulative times, Alt-Ergo is not really faster than ArchSAT, which take more time to find few more difficult problems: with a timeout of 3s, ArchSAT finds 260 proofs in 16.61 s, while Alt-Ergo obtains the same results.

5 Conclusion

We have described the architecture of ArchSAT, an automated theorem prover that combines a SMT solver with tableau calculus and rewriting. Compared to several other tools, ArchSAT appears quite effective in practice, as shown by some experimental results obtained by running our implementation over a benchmark of problems in the B set theory.

[7] Available at: https://github.com/delahayd/bset

[8] Git branch smt18.

As perspectives, we plan to realize more tests of ArchSAT over other theories where a large part of these theories can be turned into rewrite rules. In particular, a regular trigger mechanism has been also implemented in ArchSAT and can be used to deal with conditional rewriting (the instantiation is delayed and performed once the condition has been evaluated to true). This feature should open up a range of new perspectives on the theories that our approach could handle. We also aim to apply our tool to the benchmark of the BWare project [9], which consists of a large collection of proof obligations coming from the development of industrial applications using the B method. This collection gathers about 13,000 problems, and should allow us to understand to what extent our tool scales up, though it requires to extend ArchSAT to handle arithmetic, which is why it has not been tested yet.

References

[1] J.-R. Abrial. *The B-Book, Assigning Programs to Meanings.* Cambridge University Press, Cambridge (UK), 1996. ISBN 0521496195.

[2] P. Backeman and P. Rümmer. Theorem Proving with Bounded Rigid E-Unification. In *Conference on Automated Deduction (CADE)*, volume 9195 of *LNCS*, pages 572–587, Berlin (Germany), Aug. 2015. Springer.

[3] J. C. Blanchette and A. Paskevich. TFF1: The TPTP Typed First-Order Form with Rank-1 Polymorphism. In *Conference on Automated Deduction (CADE)*, volume 7898 of *LNCS*, pages 414–420, Lake Placid (NY, USA), June 2013. Springer.

[4] D. Brand. Proving Theorems with the Modification Method. *SIAM Journal on Computing*, 4(4):412–430, Dec. 1975.

[5] C. E. Brown. Satallax: An Automatic Higher-Order Prover. In *International Joint Conference on Automated Reasoning (IJCAR)*, volume 7364 of *LNCS*, pages 111–117, Manchester (UK), June 2012. Springer.

[6] G. Bury. mSAT: An OCaml SAT Solver. In *OCaml Users and Developers Workshop (OCaml)*, Sept. 2017. https://gbury.eu/public/papers/icfp2017_msat.pdf.

[7] G. Bury, D. Delahaye, D. Doligez, P. Halmagrand, and O. Hermant. Automated Deduction in the B Set Theory using Typed Proof Search and Deduction Modulo. In *Logic for Programming, Artificial Intelligence and Reasoning (LPAR) – Short Presentations*, volume 35, pages 42–58, Suva (Fiji), Nov. 2015. EasyChair.

[8] A. Degtyarev and A. Voronkov. What You Always Wanted to Know About Rigid E-Unification. In *Logics in Artificial Intelligence (JELIA)*, volume 1126 of *LNCS*, pages 50–69, Évora (Portugal), Sept. 1996. Springer.

[9] D. Delahaye, C. Dubois, C. Marché, and D. Mentré. 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)*, volume 8477 of *LNCS*, pages 126–127, Toulouse (France), June 2014. Springer.

[10] G. Dowek, T. Hardin, and C. Kirchner. Theorem Proving Modulo. *Journal of Automated Reasoning (JAR)*, 31(1):33–72, Sept. 2003.

[11] K. Korovin and C. Sticksel. Labelled Unit Superposition Calculi for Instantiation-Based Reasoning. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 6397 of *LNCS*, pages 459–473, Yogyakarta (Indonesia), Oct. 2010. Springer.

[12] R. Letz and G. Stenz. Integration of Equality Reasoning into the Disconnection Calculus. In *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*, volume 2381 of *LNCS*, pages 176–190, Copenhagen (Denmark), July 2002. Springer.

[13] D. A. Plaisted. Special Cases and Substitutes for Rigid E-Unification. *Applicable Algebra in Engineering, Communication and Computing (AAECC)*, 10(2):97–152, Jan. 2000.

[14] S. Schulz. E – A Brainiac Theorem Prover. *AI Communications*, 15(2-3):111–126, Aug. 2002.