Solving Satisfiability of Ground Term Algebras Using DPLL and Unification

Bahareh Badban[1], Jaco van de Pol[1], Olga Tveretina[2], Hans Zantema[2]

[1] Centrum voor Wiskunde en Informatica, Dept. of Software Engineering
P.O.-Box 94.079, 1090 GB Amsterdam, The Netherlands
Bahareh.Badban@cwi.nl, Jaco.van.de.Pol@cwi.nl
[2] Department of Computer Science, TU Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
o.tveretina@tue.nl, h.zantema@tue.nl

Abstract. Abstract datatypes can be viewed as sorted ground term algebras. Unification can be used to solve conjunctions of equations. We give a new algorithm to extend this to the full quantifier free fragment, i.e. including formulas with disjunction and negation. The algorithm is based on unification (to deal with equality) and DPLL (to deal with propositional logic). In this paper we present our algorithm as an instance of a generalized DPLL algorithm. We prove soundness and completeness of the class of generalized DPLL algorithms, in particular for the algorithm for ground term algebras.

Keywords: abstract datatypes, ground term algebra, equality, satisfiability, DPLL procedure, unification

1 Introduction

Many tools for deciding boolean combinations for certain theories have been developed over the years. Typically, such procedures decide fragments of (Presburger) arithmetic and uninterpreted functions. These theories are used in hardware [9] and software [23] verification; other applications are in interactive theorem proving, static analysis and abstract interpretation.

In many algebraic systems, function symbols are divided in constructors and defined operations. The values of the intended domains coincide with the ground terms built from constructor symbols only. This is for instance the case with the data specifications in μCRL [17], a language based on abstract data types and process algebra. We will focus on formulas with constructor symbols only.

Our main motivation is to decide boolean combinations over algebraic data types, which can be viewed as sorted ground term algebras. Although this logic has been studied quite extensively from a theoretical point of view, we are not aware of a complete tool to decide boolean combinations of equality over an arbitrary ground term algebra.

Solving satisfiability of equalities between ground terms can be considered as checking whether they are unifiable or not. Our algorithm uses standard

unification to deal with conjunctions of equalities. Our procedure solves not only conjunctions of equations but it is also extended to disjunctions and negations.

To deal with a boolean structure of a formula we use the well-known DPLL procedure since it is the basis of some of the most successful propositional satisfiability solvers. The original DPLL procedure [13, 12] was developed as a proof-procedure for first-order logic. It has been used so far almost exclusively for propositional logic because of its highly inefficient treatment of quantifiers. Robinson's contribution [24] was the discovery of the resolution rule and unification for this purpose. The idea of the DPLL procedure may be applied to other kinds of logics too.

Contribution. In Section 2 we introduce a basic framework for satisfiability problems. We focus on quantifier free fragments of first-order logic for which this yields a sound, terminating and complete decision procedure for satisfiability. Subsequently we introduce a framework for generalized DPLL procedures, called GDPLL (Section 3). It is meant for various fragments of first-order logic with equality, but without quantification. In case of equality in a ground term algebra, we will use unification as well.

This is an algorithm with four basic modules, that have to be filled in for a particular logic. These modules correspond to choosing an atom, adding it (or its negation) as a fact, reducing the intermediate formulae and a simple satisfiability criterion. We show sufficient conditions on these basic modules under which GDPLL is sound, terminating and complete. The original propositional DPLL algorithm (with or without unit resolution) can be obtained as an instance.

Finally, in Section 4 we investigate the quantifier-free logic of equality over an infinite ground term algebra (sometimes referred to as abstract datatypes, or inductive datatypes). An instance of a formula in this logic would be:

$$(x = S(y) \vee y = S(head(tail(z)))) \wedge z = cons(x, w) \wedge (x = 0 \vee z = nil)$$

We provide a concrete algorithm for this case. Our particular solution for ground term algebras depends on well known unification theory [19, 3]. We follow the almost linear implementation of [18]. Unification solves conjunctions of equations in the ground term algebra. The full first-order theory of equality in ground term algebras has been studied in [20, 11, 22]. Colmerauer [10] studied a setting with conjunctions of both equations and inequations. Using a DNF transformation, this is sufficient to solve any boolean combination. However, the DNF transformation itself may cause an exponential blow-up. For this reason we base our algorithm on DPLL, where after each case split the resulting CNFs can be reduced (also known as constraint propagation). In particular, our reduction is based on a combination of unification and unit resolution. None of the papers mentioned give concrete algorithms for use in verification, and the idea to combine unification and DPLL seems to be new.

It is an instance of GDPLL, so we show its soundness and completeness by checking the conditions mentioned above.

Related Work. ICS [25] (which is used in PVS) and CVC [6] are applicable to a logic including uninterpreted function symbols and arithmetic with + and

>. They have sound but incomplete decision procedures for equality on abstract datatypes. For instance, experiments show that CVC doesn't prove validity of the query $x \neq succ(succ(x))$.

Another sound but incomplete approach for general algebraic data types was based on equational BDDs [16]. A complete algorithm for BDDs with equations, zero and successor is treated in [4].

In the past several years various approaches based on the DPLL procedure have been proposed [1, 2, 14, 15]. MathSAT [2] combines a SAT procedure, for dealing efficiently with the propositional component of the problem and, within the DPLL architecture, of a set of mathematical deciders for theories of increasing expressive power. In [7] a full first-order theorem prover has been constructed which combines DPLL and unification. and unification is used to determine when two literals are contradictory. It differs from our approach where unification deals with conjunctions of equations and is extended to disjunctions and negations.

The recent algorithm of [21] decides the theory of uninterpreted functions with integer offsets. It is also based on an extension of DPLL, but it is interesting to note that it cannot be described as an instance of our GDPLL, where all decisions depend on the current CNF only. In the algorithm of [21], as well as in the ICS and CVC algorithms, some decisions depend on a context of previously asserted formulas.

A full version of the paper including full proofs can be found in [5]. Implementation in C is available at `http://www.cwi.nl/ ~ vdpol/gdpll.html`.

2 A Common Satisfiability Framework

In this section we define satisfiability for a general setting of which we consider four instances. Essentially we define satisfiability for instances of predicate logic. Often satisfiability of CNFs in predicate logic means that all clauses are implicitly universally quantified, and all other symbols are called Skolem constants. We work in quantifier free logics, possibly with interpreted symbols. Our variables (corresponding to the Skolem constants above) are implicitly existentially quantified at the outermost level. This corresponds to the conventions used in for instance unification theory [11, 20].

2.1 Syntax

Let $\Sigma = (\mathsf{Fun}, \mathsf{Pr})$ be a signature, where $\mathsf{Fun} = \{f, g, h, \dots\}$ is a set of *function symbols*, and $\mathsf{Pr} = \{p, q, r \dots\}$ is a set of *predicate symbols*. For every function symbol and every predicate symbol its *arity* is defined, being a non-negative integer. The functions of arity zero are called *constant symbols*, the predicates of arity zero are called *propositional variables*. We assume a set $\mathsf{Var} = \{x, y, z, \dots\}$ of *variables*. The sets Var, Fun, Pr are pairwise disjoint.

The set $\mathsf{Term}(\Sigma, \mathsf{Var})$ of *terms* over the signature Σ is inductively defined in the usual way. The set of *ground terms* $\mathsf{Term}(\Sigma)$ is defined as $\mathsf{Term}(\Sigma, \emptyset)$. An *atom* a is defined to be an expression of the form $p(t_1, \dots, t_n)$, where the t_i are

terms, and p is a predicate symbol of arity n. The set of atoms over the signature Σ is denoted by $\mathsf{At}(\Sigma, \mathsf{Var})$ or for simplicity by At .

A *literal* l is either an atom a or a negated atom $\neg a$. We say that a literal l is *positive* if l coincides with an atom a, and *negative* if l coincides with a negated atom $\neg a$. In the latter case, $\neg l$ denotes the literal a. The set of all literals is denoted by Lit. We denote by Lit_p and Lit_n the sets of all positive and negative literals, respectively. A *clause* C is defined to be a finite set of literals. For the empty clause we use the notation $\bot$. A *conjunctive normal form* (CNF) is defined to be a finite set of clauses. We denote by Cnf the set of all CNFs. In the following, we write $\#S$ for the cardinality of any finite set S.

We use the following notations throughout the paper, for $\phi \in \mathsf{Cnf}$ and $l \in \mathsf{Lit}$: $\mathsf{Var}(\phi)$ is the set of all variables occurring in ϕ (similar for terms, literals and clauses); $\mathsf{Pr}(\phi)$, $\mathsf{At}(\phi)$, $\mathsf{Lit}(\phi)$, $\mathsf{Lit}_p(\phi)$ and $\mathsf{Lit}_n(\phi)$ are the sets of all predicate symbols, all atoms, literals, positive literals, and negative literals occurring in ϕ, respectively; We define $\phi|_l = \{C - \{\neg l\} \mid C \in \phi,\ l \notin C\}$; We write $\phi \wedge l$ as a shortcut for $\phi \cup \{\{l\}\}$. Finally, we say that $C \in \mathsf{Cnf}$ is a *purely positive clause* if $l \in \mathsf{Lit}_p$ for all $l \in C$; C is a *unit clause* if $\#C = 1$; $l \in \mathsf{Lit}(\phi)$ is a *pure literal* in ϕ if $\neg l \notin \mathsf{Lit}(\phi)$.

2.2 Semantics

A *structure* $\mathcal{D}$ over a signature $\Sigma = (\mathsf{Fun}, \mathsf{Pr})$ is defined to consist of

- a non-empty set D called the *domain*,
- for every $f \in \mathsf{Fun}$ of arity n a map $f_D : D^n \to D$, and
- for every $p \in \mathsf{Pr}$ of arity n a map $p_D : D^n \to \{\mathsf{true}, \mathsf{false}\}$.

Let $\mathcal{D}$ be a structure and $\sigma : \mathsf{Var} \to D$ be an *assignment*. The *interpretation* $[\![t]\!]^\sigma_\mathcal{D} : \mathsf{Term}(\Sigma, \mathsf{Var}) \to D$ of a term t is inductively defined as usual. We also define the interpretation of atoms, literals, clauses and formulas $[\![\phi]\!]^\sigma_\mathcal{D} : \mathsf{Cnf} \to \{\mathsf{false}, \mathsf{true}\}$. We omit the details, but just note that a clause is interpreted as the disjunction of its literals, and a CNF formula as the conjunction of its clauses.

In some instances of our framework for defining satisfiability all possible structures are allowed, in others we have restrictions on the structures that are allowed. Therefore in any instance we assume a notion of *admissible structure*. Depending on this notion of admissible structure we have the following definition of satisfiability.

Definition 1. *An assignment* $\sigma : \mathsf{Var} \to D$ satisfies *a CNF* ϕ *in a structure* $\mathcal{D}$, *if* $[\![\phi]\!]^\sigma_\mathcal{D} = \mathsf{true}$. *CNF* ϕ *is called* satisfiable *if it is satisfied by some assignment in some admissible structure. Otherwise* ϕ *is called* unsatisfiable.

A particular logic will consist of a signature and a set of admissible structures. By the latter, we can distinguish a completely uninterpreted setting (no restriction on structures) from a completely interpreted setting (only one structure is admissible). However, intermediate situations are possible as well.

2.3 Instances of the Satisfiability Framework

We now describe precisely various instances of the framework just described by specifying the signature and the admissible structures.

Propositional Logic. The first instance we consider is propositional logic. Here we have $\Sigma = \{\mathsf{Fun}, \mathsf{Pr}\}$, where $\mathsf{Fun} = \emptyset$ and Pr is a set of predicate symbols all having arity zero. In this way there are no terms at all occurring in atoms: an atom coincides with such a predicate symbol of arity zero. Hence a CNF in this instance coincides with a usual propositional CNF. Since there are no terms in the formula, neither variables play a role, nor the assignments. The only remaining ingredient of an interpretation is a map $p_D : D^0 \to \{\mathsf{true}, \mathsf{false}\}$ for every predicate symbol p. Since D^0 consists of one element independent of D, this interpretation is only a map from the atoms to $\{\mathsf{true}, \mathsf{false}\}$, just like intended for propositional logic. Since the domain does not play a role there is no need for defining restrictions: as admissible structures we allow all structures.

Equality Logic. The next instance we consider is equality logic. By equality logic formulas we mean formulas built from atoms of the shape $x \approx y$, where x and y are variables. We reserve the notation $\approx$ for a particular binary predicate symbol for reasoning over equality. We write $x \approx y$ instead of $\approx xy$. We will use the shortcut $x \not\approx y$ for $\neg(x \approx y)$. For ease of presentation, we will consider $x \approx y$ and $y \approx x$ as the same atom.

Now we define equality formulas in conjunctive normal form as an instance of the framework described above. For equality logic we have $\Sigma = \{\mathsf{Fun}, \mathsf{Pr}\}$, where $\mathsf{Fun} = \emptyset$ and $\mathsf{Pr} = \{\approx\}$. In this way the variables are the only terms, and all atoms are of the shape $x \approx y$ for variables x, y. The admissible structures are defined to be all structures $\mathcal{D}$ for which $\approx_D = Id_D$, where the function $Id_D : D \times D \to \{\mathsf{true}, \mathsf{false}\}$ is defined such that $Id_D(d_1, d_2) = \mathsf{true}$ if $d_1 = d_2$, and false otherwise.

3 GDPLL

Most of the techniques relevant in the setting of the DPLL procedure are also applicable to GDPLL. Essentially, the DPLL procedure consists of the following three rules: the unit clause rule, the splitting rule, and the pure literal rule. Both the unit clause rule and the pure literal rule reduce the formula according some criteria. Thus, in GDPLL we may assume a function Reduce which performs all rules for formula reduction. GDPLL has a splitting rule, which carries out a case analysis with respect to an atom a. The current set of clauses ϕ splits into two sets: the one where a is true, and another where a is false.

In the following we assume a function $\mathsf{Reduce} : \mathsf{Cnf} \to \mathsf{Cnf}$. We define the set $\mathsf{Rcnf} = \{\phi \in \mathsf{Reduce}(\mathsf{Cnf}) \mid \bot \notin \phi\}$. We also assume functions

- $\mathsf{Eligible} : \mathsf{Rcnf} \to \mathsf{At}$,
- $\mathsf{SatCriterion} : \mathsf{Rcnf} \to \{\mathsf{true}, \mathsf{false}\}$,
- Filter, where $\mathsf{Filter}(\phi, a)$ is defined for $\phi \in \mathsf{Rcnf}$ and $a \in \mathsf{Eligible}(\phi)$.

We now introduce the requirements on the functions above: for all $\psi \in \mathsf{Cnf}$, for all $\phi \in \mathsf{Rcnf}$, and for all $a \in \mathsf{Eligible}(\phi)$ the functions should satisfy the following properties.

1. $\mathsf{Reduce}(\psi)$ is satisfiable iff ψ is satisfiable,
2. ϕ is satisfiable iff at least one of $\mathsf{Filter}(\phi, a)$ and $\mathsf{Filter}(\phi, \neg a)$ is satisfiable,
3. $\mathsf{Reduce}(\mathsf{Filter}(\phi, a)) \prec \phi$ and $\mathsf{Reduce}(\mathsf{Filter}(\phi, \neg a)) \prec \phi$, for some well-founded order $\prec$ on $\mathsf{Reduce}(\mathsf{Cnf})$.
4. if $\mathsf{SatCriterion}(\phi) = \mathsf{true}$ then ϕ is satisfiable,
5. if $\mathsf{SatCriterion}(\phi) = \mathsf{false}$ then $\mathsf{Eligible}(\phi) \neq \emptyset$.

We next show the pseudo-code of the skeleton of the algorithm. The procedure takes as an input $\phi \in \mathsf{Cnf}$. GDPLL proceeds until either the function SatCriterion has returned true for at least one branch, or the empty clause has been derived for all branches. Respectively, either SAT or UNSAT is returned.

```
GDPLL(φ) : {SAT, UNSAT} =
    begin
        φ := Reduce(φ);
        if (⊥ ∈ φ) then return UNSAT;
        if (SatCriterion(φ)) then return SAT;
        choose a ∈ Eligible(φ);
        if GDPLL(Filter(φ, a)) = SAT then return SAT;
        if GDPLL(Filter(φ, ¬a)) = SAT then return SAT;
        return UNSAT;
    end;
```

Theorem 2 (soundness and completeness). *Let $\phi \in \mathsf{Cnf}$. Assume properties 1-5 hold. Then we have:*

- *If ϕ is satisfiable then* $\mathsf{GDPLL}(\phi) = \mathsf{SAT}$.
- *If ϕ is unsatisfiable then* $\mathsf{GDPLL}(\phi) = \mathsf{UNSAT}$.

Proof. Let $\phi \in \mathsf{Cnf}$. Assume (induction hypothesis) that the theorem holds for all ψ such that $\mathsf{Reduce}(\psi) \prec \mathsf{Reduce}(\phi)$. By property 1, $\mathsf{Reduce}(\phi)$ is satisfiable iff ϕ is satisfiable. If $\bot \in \mathsf{Reduce}(\phi)$, then trivially ϕ is unsatisfiable, and $\mathsf{GDPLL}(\phi)$ returns UNSAT. Otherwise, if $\mathsf{SatCriterion}(\mathsf{Reduce}(\phi)) = \mathsf{true}$ then by property 4, ϕ is satisfiable, and $\mathsf{GDPLL}(\phi) = \mathsf{SAT}$. Otherwise, by property 5, $\mathsf{Eligible}(\mathsf{Reduce}(\phi)) \neq \emptyset$. By property 3, for all $\phi \in \mathsf{Cnf}$ and all $a \in \mathsf{Eligible}(\phi)$, we have $\mathsf{Reduce}(\mathsf{Filter}(\mathsf{Reduce}(\phi), a)) \prec \mathsf{Reduce}(\phi)$, and also $\mathsf{Reduce}(\mathsf{Filter}(\mathsf{Reduce}(\phi), \neg a)) \prec \mathsf{Reduce}(\phi)$, so we can apply induction, and either SAT or UNSAT is returned. Now, using property 2 and the induction hypothesis, $\mathsf{Reduce}(\phi)$ is satisfiable, if and only if one of $\mathsf{Filter}(\mathsf{Reduce}(\phi), a)$ and $\mathsf{Filter}(\mathsf{Reduce}(\phi), \neg a)$ are satisfiable; if and only if $\mathsf{GDPLL}(\mathsf{Filter}(\mathsf{Reduce}(\phi), a))$ or $\mathsf{GDPLL}(\mathsf{Filter}(\mathsf{Reduce}(\phi), \neg a))$ returns SAT; iff $\mathsf{GDPLL}(\phi)$ returns SAT. □

3.1 GDPLL for Propositional Logic

It can be seen that the DPLL procedure for propositional logic is a particular case of GDPLL. We have to provide Eligible, Filter, Reduce and SatCriterion. All functions except Reduce are straightforward. In order to coincide with the original DPLL procedure, we let an eligible atom be an arbitrary atom, i.e. to coincide with the original DPLL procedure. So we put:

$$\begin{aligned} \mathsf{Eligible}(\phi) &= \mathsf{At}(\phi)\ . \\ \mathsf{SatCriterion}(\phi) &= \mathsf{true}, \text{ if } \phi = \emptyset; \mathsf{false}, \text{ otherwise.} \\ \mathsf{Filter}(\phi, l) &= \phi \wedge l\ . \end{aligned}$$

Two main operations of the DPLL procedure are *unit propagation* and *purification*. The literal of a unit clause must hold, in order to make the CNF satisfiable. Unit clauses can be eliminated by replacing $\phi \wedge l$ by $\phi|_l$. Elimination of a unit clause can create a new unit clause, so this process has to be repeated until no unit clauses are left. Also pure literals can be eliminated. If l is pure in ϕ, we may replace ϕ by $\phi|_l$. Note that this cannot introduce unit clauses. In GDPLL, unit resolution and purification are performed by Reduce. So $\mathsf{Reduce}(\phi)$ is defined as the result after eliminating all unit clauses and pure literals.

Theorem 3. *The functions* Reduce, Eligible, Filter, SatCriterion *satisfy the Properties 1–5.*

Proof. For property 3, define $\phi_1 \prec \phi_2$ iff $\#\mathsf{Pr}(\phi_1) < \#\mathsf{Pr}(\phi_2)$. □

3.2 GDPLL for Equality Logic

We now define the functions Eligible, Filter, Reduce and SatCriterion for equality logic. The function Reduce removes all clauses containing a literal of the shape $x \approx x$ and literals of the shape $x \not\approx x$ from other clauses. Recall that $x \approx y$ and $y \approx x$ denote the same atom.

In case of propositional logic we chose any atom contained in a CNF to apply the split rule. The correctness of GDPLL is not immediate for other instances. For equality logic we define an atom to be eligible if it occurs as a positive literal in the formula, i.e., $\mathsf{Eligible}(\phi) = \mathsf{Lit}_p(\phi)$. We define the function SatCriterion, so that it indicates that there are no purely positive clauses left:

$$\mathsf{SatCriterion}(\phi) = \begin{cases} \mathsf{true} & \text{if } C \cap \mathsf{Lit}_n \neq \emptyset \text{ for all } C \in \phi \\ \mathsf{false} & \text{otherwise} \end{cases}$$

Example 4. Consider $\phi \equiv \{\{x \approx y, y \not\approx z\}, \{x \approx z, x \not\approx y, y \approx z\}, \{x \not\approx z\}\}$. One can easily see that the formula is satisfied by an assignment σ such that $\sigma(x') \neq \sigma(x'')$ for all $x', x'' \in \mathsf{Lit}_n(\phi)$.

We denote by $\phi[x := y]$ the formula ϕ, where all occurrences of x are replaced by y. We define the function Filter as follows

- $\mathsf{Filter}(\phi, x \approx y) = \phi|_{x \approx y}[x := y]$,
- $\mathsf{Filter}(\phi, x \not\approx y) = \phi|_{x \not\approx y} \wedge (x \not\approx y)$.

Theorem 5. *The functions* Reduce, Eligible, Filter, SatCriterion *satisfy the Properties 1–5.*

Proof. For Property 3, define $\phi_1 \prec \phi_2$ if $\#\mathsf{Lit}_p(\phi_1) < \#\mathsf{Lit}_p(\phi_2)$. □

4 Ground Term Algebra

In this section we show how to solve the satisfiability problem for CNFs over ground term algebras (sometimes referred to as inductive datatypes, or abstract datatypes). First we show how this logic fits in the framework of Section 2.

In this instance we have $\Sigma = (\mathsf{Fun}, \mathsf{Pr})$, where Fun is an arbitrary set of function symbols and Pr consists only of the binary predicate symbol $\approx$ (written infix). The idea is that $\approx$ again represents equality and that terms are interpreted by ground terms, i.e. in $\mathsf{Term}(\Sigma)$. Every symbol is interpreted by its term constructor. We assume that there exists at least one constant symbol (i.e. some $f \in \mathsf{Fun}$ has arity 0), to avoid that the set $\mathsf{Term}(\Sigma)$ of ground terms is empty. Later, we will also make the assumption that the ground term algebra is infinite (i.e. at least one symbol of arity > 0 exists, or the number of constant symbols is infinite). Hence we allow only one admissible structure $\mathcal{D}$, for which

- $D = \mathsf{Term}(\Sigma)$,
- $f_D(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$
- $\approx_D = Id_D$.

For instance, in the term algebra the CNF $\{\{f(x) = g(y)\}\}$ for $f, g \in \mathsf{Fun}, f \neq g$ is unsatisfiable since for all ground terms t, u the terms $f(t)$ and $g(u)$ are distinct. We will (tacitly) use the following properties of all ground term algebras:

Lemma 6. *In every ground term algebra $\mathcal{D}$ for Σ, the following hold:*

1. *for all $f, g \in \mathsf{Fun}$ with $f \neq g$: $\forall x, y : f_D(x) \neq g_D(y)$*
2. *for all $f \in \mathsf{Fun}$: $\forall x, y : x \neq y \Rightarrow f_D(x) \neq f_D(y)$*
3. *for all contexts $C \neq [\,]$: $\forall x : x \neq C[x]$*

After introducing some basic definitions and properties of substitutions and most general unifiers, we will define the building blocks of GDPLL, and prove the properties needed to conclude with Theorem 2 that the obtained procedure is sound and complete.

4.1 Substitutions and most general unifiers

We introduce here the standard definitions of substitutions and unifiers, taken from [19, 3]. A *substitution* is a function $\sigma : \mathsf{Var} \to \mathsf{Term}(\Sigma, \mathsf{Var})$ such that $\sigma(x) \neq x$ for only finitely many xs. We define the *domain*: $\mathsf{Dom}(\sigma) = \{x \in \mathsf{Var} \mid \sigma(x) \neq x\}$. If $\mathsf{Dom}(\sigma) = \{x_1, \ldots, x_n\}$, then we alternatively write σ as

$\sigma = \{x_1 \mapsto \sigma(x_1), \ldots, x_n \mapsto \sigma(x_n)\}$. The *variable range* of σ is $\mathsf{Var}(\sigma) = \bigcup_{x \in \mathsf{Dom}(\sigma)} \mathsf{Var}(\sigma(x))$. Furthermore, with $\mathsf{Eq}(\sigma)$ we denote the corresponding set of equations $\{x_1 \approx \sigma(x_1), \ldots, x_n \approx \sigma(x_n)\}$, and with $\neg\mathsf{Eq}(\sigma)$ the corresponding set of inequations. A set of equations is *in solved form* if it is of the form $\{x_1 \approx t_1, \ldots, x_n \approx t_n\}$, with all x_i different, and $x_i \notin \mathsf{Var}(t_j)$ (for all i, j).

Substitutions are homomorphically extended to terms and formulas in the usual way. We write ϕ^σ for the CNF obtained by replacing each occurrence of a variable x by $\sigma(x)$. The *composition* $\sigma\rho$ of substitutions σ and ρ is defined such that $\sigma\rho(x) = \sigma(\rho(x))$. A substitution σ is *idempotent* if $\sigma\sigma = \sigma$. A substitution σ is *more general* than σ' if $\sigma' = \rho\sigma$ for some ρ. A *unifier* or solution of a set $S = \{s_1 \approx t_1, ..., s_n \approx t_n\}$ of finite number of atoms, is a substitution σ such that $s_i^\sigma = t_i^\sigma$ for $i = 1, ..., n$. A substitution σ is a most general unifier of S or in short $\mathsf{mgu}(S)$, if σ is a unifier of S and moreover, it is more general than each unifier σ' of S. In the sequel, we will use the following well known fact on substitutions and unifiers (cf. [19, 3]):

Lemma 7. *If a set S of atoms has a unifier, then it has an idempotent mgu, say σ. Moreover, $\mathsf{Eq}(\sigma)$ is in solved form, and logically equivalent to S.*

If $n = 1$ we write $\mathsf{mgu}(s_1 \approx t_1)$. When dealing with sets of unit clauses, by $\sigma = \mathsf{mgu}(\{t_1 \approx u_1\}, ..., \{t_n \approx u_n\})$ we mean $\sigma = \mathsf{mgu}(\{t_1 \approx u_1, ..., t_n \approx u_n\})$. If S has no unifier, we write $\mathsf{mgu}(S) = \bot$. From now on by an mgu we always mean an *idempotent mgu*, which exists by the previous Lemma. As a consequence of the above lemma and the conventions, if an mgu $\sigma = \{x_1 \mapsto t_1, ..., x_n \mapsto t_n\}$ then $x_i \notin \mathsf{Var}(t_j)$ for all $1 \leq i, j \leq n$. Another consequence is that $\mathsf{mgu}(x \approx x) = \emptyset$.

4.2 The GDPLL building blocks for ground term algebras

We now come to the definition of the building blocks for the GDPLL algorithm. The functions Eligible and SatCriterion correspond to those in Section 3.2 on Equality Logic. That is, only positive literals are eligible, and we may terminate with SAT as soon as there is no purely positive clause. The function Filter corresponds to the filtering in Section 3.1 on Propositional Logic; that is we simply put the CNF in conjunction with the chosen literal. This means that all work specific for ground term algebras is done by Reduce. This function will be defined by means of a set of transformation rules, that can be applied in any order.

Definition 8. *We consider the following reduction rules, which should be applied repeatedly until ϕ cannot be modified.*

1. *if $t \approx t \in C \in \phi$ then $\phi \longrightarrow \phi - \{C\}$*
2. *if $\bot \in \phi$ and $\phi \neq \{\bot\}$ then $\phi \longrightarrow \{\bot\}$*
3. *if $\phi = \phi_1 \uplus \{C \uplus \{t \not\approx u\}\}$, and $t \approx u$ is non-solved, let $\sigma = \mathsf{mgu}(t \approx u)$ and*
   - *if $\sigma = \bot$, then $\phi \longrightarrow \phi_1$*
   - *otherwise, $\phi \longrightarrow \phi_1 \cup \{C \cup \neg\mathsf{Eq}(\sigma)\}$*
4. *if $\phi_1 = \{C \mid C \in \phi$ is a positive unit clause$\} \neq \emptyset$, take $\sigma = \mathsf{mgu}(\phi_1)$ then*
   - *if $\sigma = \bot$, then $\phi \longrightarrow \{\bot\}$*

– *otherwise let* $\phi = \phi_1 \uplus \phi_2$ *then* $\phi \longrightarrow \phi_2{}^{\sigma}$

5. *if* $\phi = \{\{\neg a\}\} \uplus \phi_1$ *and* $a \in \mathsf{At}(\phi_1)$ *then* $\phi \longrightarrow \{\{\neg a\}\} \uplus \phi_1|_{\neg a}$

We define $\mathsf{Reduce}(\phi)$ *to be any normal form of* ϕ *with respect to the rules above.*

We tacitly assume that $x \approx y$ and $y \approx x$ are treated identically. Rule 3 replaces a negative equation by its solved form. Note that solving positive equations would violate the CNF structure, so this is restricted to unit clauses, which emerge by Filtering. Rules 4 and 5 above implement unit resolution adapted to the equational case. Positive unit clauses lead to substitutions. All positive units are dealt with at once, in order to minimize the calls to mgu and to detect more inconsistencies. Negative unit clauses are put back, which is essential to prove property 1 of GDPLL.

We will show that the rules are terminating, so at least one normal form exists. We show by an example that the function Reduce is not uniquely defined. An implementation may choose any normal form.

Example 9. $\phi = \{\{f(f(y)) \not\approx f(x)\}, \{x \not\approx x\}\}$. We will subsequently apply rule 3 on $f(f(y)) \not\approx f(x)$ (with $\sigma : x \mapsto f(y)$); rule 3 on $\{x \not\approx x\}$ (with $\sigma = \bot$), and rule 2: $\phi \longrightarrow \{\{x \not\approx f(y)\}, \{x \not\approx x\}\} \longrightarrow \{\{x \not\approx f(y)\}, \{\}\} \longrightarrow \{\bot\}$.

Example 10. Consider $\phi = \{\{x \not\approx f(a,b)\}, \{x \approx f(y,z)\}, \{y \approx a, x \approx f(a,b)\}\}$. We show reductions to two different reduced forms:

$$
\begin{array}{ll}
\phi \longrightarrow \{\{x \not\approx f(a,b)\}, \{x \approx f(y,z)\}, \{y \approx a\}\} & \text{using 5} \\
\quad \longrightarrow \{\{f(y,z) \not\approx f(a,b)\}, \{y \approx a\}\} & \text{applying 4 on } \{x \approx f(y,z)\} \\
\quad \longrightarrow \{\{f(a,z) \not\approx f(a,b)\}\} & \text{applying 4 on } \{y \approx a\} \\
\quad \longrightarrow \{\{z \not\approx b\}\} & \text{using 3}
\end{array}
$$

$$
\begin{array}{ll}
\phi \longrightarrow \{\{f(y,z) \not\approx f(a,b)\}, \{y \approx a, f(y,z) \approx f(a,b)\}\} & \text{by 4 on } \{x \approx f(y,z)\} \\
\quad \longrightarrow \{\{y \not\approx a, z \not\approx b\}, \{y \approx a, f(y,z) \approx f(a,b)\}\} & \text{using 3}
\end{array}
$$

We will now prove termination of the reduction system. Subsequently, we will prove the correctness of the building blocks of the corresponding GDPLL procedure (i.e. property 1–5).

Definition 11. $\mathsf{pos}(\phi)$ *is the number of occurrences of positive literals in* ϕ.

Lemma 12. *The reduction system is terminating. Moreover,* $\mathsf{pos}(\phi)$ *does not increase during the reduction process on* ϕ.

Proof. Each rule application 1–5 lexicographically decreases the following pair of numbers: $\mathsf{norm}(\phi) = (\mathsf{pos}(\phi) + \#\phi, \mathsf{neg}(\phi))$, where $\mathsf{neg}(\phi)$ is the number of occurrences of negative *non-solved* literals in ϕ. It can also be checked that pos doesn't increase with each rule application. □

Lemma 13 (Termination criterion). $\mathsf{pos}(\mathsf{Reduce}(\phi \wedge l)) < \mathsf{pos}(\phi)$ *for any reduced formula* ϕ *and a literal* $l \in \{t \approx u, t \not\approx u\}$, *where* $t \approx u \in \mathsf{Lit}_p(\phi)$.

Proof. Since ϕ is reduced, the first step to reduce $\phi \wedge l$ will be by rule 4 or 5 (depending on whether l is positive or not). After applying rule 4, we have a literal of the form $t^\sigma \approx t^\sigma$. At some point in the reduction this literal (or an instance of it) must be removed (otherwise rule 1 is applicable), and here pos is decreased. The other steps don't increase pos, by the previous Lemma. In rule 5 pos is immediately decreased, and it doesn't increase by other steps. □

Lemma 14 (Reduce criterion). *Given a ground term algebra $\mathcal{D}$ and a formula ϕ in it, ϕ is satisfiable if and only if* $\mathsf{Reduce}(\phi)$ *is satisfiable.*

Proof. If $\phi \longrightarrow \phi'$ by any of the rules 1–5, ϕ is satisfiable iff ϕ' is satisfiable. □

The following lemma can be derived from [19, Thm. 10], but we provide a direct proof that reveals the satisfying assignment.

Lemma 15 (SAT criterion). *Suppose $\mathcal{D}$ is an infinite ground term algebra. Then a reduced formula ϕ is satisfiable if ϕ has no purely positive clause.*

Proof. Let $n = \#\phi$; each clause of ϕ has a negative literal of the form $x_i \not\approx t_i$, for $1 \leq i \leq n$. It suffices to provide an assignment σ which satisfies all these negative literals. If we have infinitely many constants, then a satisfying assignment can be easily provided. So we concentrate on the other case: Σ has at least one function symbol g of arity $m > 0$.

First define $S(t)$ for any term t to be the number of occurrences of non-constant function symbols in t. Let c be a constant symbol in $\mathcal{D}$, and define a context $F[\,]$ as: $F[\,] = g(\Box, c, \ldots, c)$ (with $m - 1$ occurrences of c). Finally, let $M = 1 + Max_{1 \leq i \leq n} S(t_i)$ and define the context $C[\,] = F^M[\,]$, the M-fold application of $F[\,]$. We claim that the following assignment satisfies $x_i \not\approx t_i$ for all $1 \leq i \leq n$, and hence ϕ: $\sigma(x) = C^i[c]$, if $x = x_i$ for some $1 \leq i \leq n$; $\sigma(x) = c$, otherwise. Indeed, note that $S(\sigma(x_i)) = M.i$. Moreover, if $S(t_i) = 0$, then $S(\sigma(t_i)) = M.j$ with $0 \leq j \leq n$ and $i \neq j$ ($x_i \neq t_i$ because ϕ is reduced). Otherwise, $S(\sigma(t_i)) = M.k + S(t_i)$ for some $k \geq 0$, and $0 < S(t_i) < M$. In both cases, $S(\sigma(x_i)) \neq S(\sigma(t_i))$. □

We can now combine the lemmas on the basic blocks and apply Theorem 2. First we instantiate GDPLL as follows. We take the Reduce function defined in Definition 8. We define for $\phi \in \mathsf{Reduce}(\mathsf{Cnf})$ and $l \in \mathsf{Lit}_p(\phi)$

$$\begin{aligned} \mathsf{Eligible}(\phi) &= \mathsf{Lit}_p(\phi) \\ \mathsf{Filter}(\phi, l) &= \phi \wedge l \\ \mathsf{SatCriterion}(\phi) &= \text{for all } C \in \phi,\ C \cap \mathsf{Lit}_n \neq \emptyset \end{aligned}$$

Theorem 16. *Let* $(\mathsf{Fun}; \approx)$ *be a signature with an infinite ground term algebra $\mathcal{D}$. Let ϕ be a CNF. Let* GDPLL *be instantiated as indicated above. Then*

- *If ϕ is satisfiable in $\mathcal{D}$ then* $\mathsf{GDPLL}(\phi) = \mathsf{SAT}$.
- *If ϕ is unsatisfiable in $\mathcal{D}$ then* $\mathsf{GDPLL}(\phi) = \mathsf{UNSAT}$.

Proof. In order to apply Theorem 2, we have to check Properties 1–5. Property 2 and 5 are obvious. Property 1 has been proved in Lemma 14. Property 3 has been proved in Lemma 13; here we set $\phi \prec \psi$ if and only if $\mathsf{pos}(\phi) < \mathsf{pos}(\psi)$, which is obviously well-founded. Property 4 has been proved in Lemma 15. □

5 Implementation and Experiments

The GDPLL algorithm instantiated for ground term algebras has been implemented in C. We implemented the almost linear unification algorithm from [18, 3], which is based on a union-find data structure on terms. Linearity essentially depends on the use of subterm sharing. The intermediate terms can even be cyclic, so a separate loop-detection is needed, which implements the "occurs-check". We used the ATerm library [8], which represents terms as directed acyclic graphs. The library provides maximal subterm sharing and automatic garbage collection. Clauses and CNFs are implemented naively as (unidirected) linked lists. We have neither implemented any form of subsumption, nor heuristics for choosing a good splitting variable.

As benchmarks, we used some purely equational formulas (`phe`, `circ`) and some formulas with function symbols (`succ`, `evod`):

$$
\begin{array}{ll}
\texttt{phe}: & (\bigwedge_{1\le i<j\le N} x_i \neq x_j) \wedge (\bigwedge_{1\le i\le N} \bigvee_{1\le j\le N, j\neq i} x_j = y) \\
\texttt{circ}: & (\bigvee_{1\le i\le N} x_i \neq x_{i+1}) \wedge (\bigwedge_{1\le i<j\le N} (x_i = x_{i+1} \vee x_j = x_{j+1})) \\
\texttt{succ}: & (\bigwedge_{1\le i<j\le N} (x_i = S(x_{i+1}) \vee x_j = S(x_{j+1}))) \wedge \bigvee_{1\le i\le N} x_i = x_{i+1} \\
\texttt{evod}: & x_1 = x_N \wedge \bigwedge_{1\le i<N} (x_i = S(x_{i+1}) \vee S(x_i) = x_{i+1})
\end{array}
$$

The first conjunct of `phe` (equational pigeon hole) expresses that all x_i's are different. The second, however, insists that at least two x_i's are equal to y. This is a clear contradiction. These formulas also occur in [26]. `circ` (a ring of equations) has variables $x_1, \ldots, x_N$, which are on a ring: we will write x_{N+1} to denote syntactically the same variable as x_1. Intuitively, the first conjunct expresses that at least one equality on the ring is false. The second conjunct makes sure that at most one equality is false. So exactly one conjunct on the ring is false, which contradicts transitivity of equality. `succ` (natural numbers with equality) also has its variables on a ring; x_{N+1} denotes the variable x_1. We also have a unary constant S. The first part expresses that for all i but some j, we have $x_i = S(x_{i+1})$. Then $x_{j+1} = S^N(x_j)$ by transitivity. This contradicts the second part, which states that for some k, $x_k = x_{k+1}$. Finally, `evod` (on even and odd natural numbers) has variables: $x_1, \ldots, x_N$ and a unary constant S. Note that the second part implies that x_i is odd for all odd i (or even). This formula is satisfiable iff N is odd.

In the table below we show the experimental results. Each row corresponds to a particular instance (N) of some formula type. For each formula instance we show its size (number of literals), the time in seconds (on a Linux AMD Athlon 2400+ processor with 2 GHz; here – means more than 600 seconds) and the number of recursive calls to the GDPLL procedure. We compared two approaches. The last two columns indicate the algorithm with full unit resolution (i.e. with rules 4 and 5 of Definition 8). In the other two columns we omitted unit resolution, reverting to a definition of Filter similar to Section 3.2.

phe		without UR		with UR	
N	# lit	#sec	#calls	#sec	#calls
40	2340	0	1639	0	77
80	9480	8	6479	0	157
120	21420	52	14519	2	237
160	38160	168	25759	4	317
200	59700	433	40199	10	397

succ		without UR		with UR	
N	# lit	#sec	#calls	#sec	#calls
50	2500	6	2741	1	2449
100	10000	92	10491	6	9899
150	22500	459	23241	20	22349
200	40000	–	–	48	39799
250	62500	–	–	103	62249

circ		without UR		with UR	
N	# lit	#sec	#calls	#sec	#calls
100	10000	3	10097	0	199
200	40000	50	40197	3	399
300	90000	258	90297	9	599
400	160000	–	–	22	799
500	250000	–	–	43	999

evod		without UR		with UR	
N	# lit	#sec	#calls	#sec	#calls
14	27	2	27487	2	12951
16	31	6	111337	9	52665
18	35	25	449927	31	213523
20	39	100	1815155	104	863819
22	43	407	7313663	505	3488871

For the instances **phe**, **circ** and **succ**, it can be concluded that without unit resolution, the number of recursive calls is quadratic in N, i.e. linear in the input size. With unit resolution, the number of recursive calls is linear in N for **phe** and **circ**, and still quadratic for **succ**. Of course, this information can also be easily obtained by an analytic argument. Still, in the latter case, the used time is much better for the variant with full unit resolution (probably due to the fact that the size of the intermediate CNFs is smaller). Finally, the **evod** formulas are the hardest for our method; every next even instance takes around 4 times more work. Here unit resolution roughly halves the number of calls to GDPLL, but overall it costs a little more time.

In [26] some experiments on the same **phe** formula type are given. Several encodings to propositional logic are tried. The best result was that **phe** with $N = 60$ took 11 seconds on a 1 GHz Pentium 4. This solution used an encoding that adds transitivity constraints and subsequently used zCHAFF to solve the resulting propositional problem. This method performed clearly better than methods based on bit-vector encoding, or the use of BDDs. We report 0.20 secs for N=60, which is about 50 times better than the best method from [26], on a machine which is at most 2.5 times faster.

6 Concluding Remarks and Further Research

In this paper we gave a framework generalizing the well-known DPLL procedure for deciding satisfiability of propositional formulas in CNF. In our generalized procedure GDPLL we kept the basic idea of choosing an atom and doing two recursive calls: one for the case where this atom holds and one for the case where this atom does not hold. All other ingredients were kept abstract: Reduce for cleaning up a formula, SatCriterion for a simple criterion to decide satisfiability, Eligible to describe which atoms are allowed to be chosen and Filter for describing the case analysis. We collected a number of sufficient conditions on these four abstract procedures for proving correctness and termination. In this way GDPLL

can be applied for any kind of logic as long as we have instantiations of the abstract procedures satisfying these conditions.

Our procedure GDPLL was worked out for three such fragments of increasing generality: propositional logic, equality logic and ground term algebra. For the last one we succeeded in giving a powerful instance of the procedure Reduce based on unification. In this way the other three abstract procedures could be kept trivial yielding a powerful implementation for satisfiability of CNFs in which the atoms are equations between open terms to be interpreted in ground term algebra. Note that the resulting algorithm can be easily extended to compute a satisfying assignment (if any).

Another interpretation of equations between terms is allowing an arbitrary domain. This is usually called the logic of uninterpreted functions. How to find suitable instances for the four abstract procedures in GDPLL for this logic is one of the topics of ongoing research. Also the addition of other interpreted functions (such as + or append) or predicates (like >) is subject to future research.

References

1. ARMANDO, A., CASTELLINI, C., GIUNCHIGLIA, E., GIUNCHIGLIA, F., AND TACCHELLA, A. Sat-based decision procedures for automated reasoning: a unifying perspective. IRST Technical Report 0202-05, Istituto Trentino di Cultura, February 2002.
2. AUDEMARD, G., BERTOLI, P., CIMATTI, A., KORNILOWICZ, A., AND SEBASTIANI, R. A sat based approach for solving formulas over boolean and linear mathematical propositions. In *Automated Deduction - CADE-18:18th International Conference on Automated Deduction* (Copenhagen, Denmark, July 27-30 2002), A. Voronkov, Ed., Springer-Verlag Heidelberg, pp. 195–210.
3. BAADER, F., AND NIPKOW, T. *Term Rewriting and All That.* Cambridge University Press, New York, 1998.
4. BADBAN, B., AND POL, J. V. D. Zero, successor and equality in BDDs. *Annals of Pure and Applied Logic* (2004). see http://www.cwi.nl/ftp/CWIreports/SEN/SEN-R0231.pdf.
5. BADBAN, B., POL, J. V. D., TVERETINA, O., AND ZANTEMA, H. Generalizing DPLL and satisfiability for equalities. CSR Technical Report 04/14, Technical University of Eindhoven, 2004.
6. BARRETT, C. W., DILL, D. L., AND STUMP, A. A framework for cooperating decision procedures. In *Proceedings of the 17th International Conference on Automated Deduction (Pittsburgh, PA)* (2000), D. A. McAllester, Ed., vol. 1831 of *Lecture Notes in Artificial Intelligence*, Springer, pp. 79–98.
7. BAUMGARTNER, P. FDPLL – A First-Order Davis-Putnam-Logeman-Loveland Procedure. In *CADE-17* (2000), D. McAllester, Ed., vol. 1831 of *LNAI*, Springer, pp. 200–219.
8. BRAND, M., JONG, H. D., KLINT, P., AND OLIVIER, P. Efficient Annotated Terms. *Software – Practice & Experience 30* (2000), 259–291.
9. BURCH, J.R. AND D.L. DILL. Automatic verification of pipelined microprocessors control. In *Proceedings of Computer Aided Verification, CAV'94* (June 1994), D. L. Dill, Ed., LNCS 818, Springer-Verlag, pp. 68–80.

10. COLMERAUER, A. Equations and Inequations on Finite and Infinite Trees. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS-84)* (Tokyo, Japan, Nov. 1984), ICOT, pp. 85–99.
11. COMON, H., AND LESCANNE, P. Equational problems and disunification. *Journal of Symbolic Computation 7*, 3–4 (1989), 371–425.
12. DAVIS, M., LOGEMANN, G., AND LOVELAND, D. A machine program for theorem proving. *Communications of the ACM 5*, 7 (1962), 394–397.
13. DAVIS, M., AND PUTNAM, H. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery 7*, 3 (1960), 201–215.
14. GANZINGER, H., HAGEN, G., NIEUWENHUIS, R., OLIVERA, A., AND TINELLI, C. Dpll(t): Fast decision procedures. In *16th International Conference on Computer Aided Verification (CAV)* (2004), July.
15. GIUNCHIGLIA, F., AND SEBASTIANI, R. Building decision procedures for modal logics from propositional decision procedures: the case study of modal K. pp. 583–597.
16. GROOTE, J., AND POL, J. V. D. Equational binary decision diagrams. In *Proc. of LPAR 2000* (2000), M. Parigot and A. Voronkov, Eds., LNAI 1955, Springer, pp. 161–178.
17. GROOTE, J., AND RENIERS, M. Algebraic process verification. In *Handbook of Process Algebra*, J. Bergstra, A. Ponse, and S. Smolka, Eds. Elsevier, 2001, ch. 17.
18. HUET, G. *Résolution d'équations dans les langages d'ordre 1, 2, ..., ω.* PhD thesis, Université Paris, 7, 1976.
19. LASSEZ, J.-L., MAHER, M. J., AND MARRIOTT, K. Unification Revisited. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan Kaufmann Publishers, Los Altos, California, 1987, pp. 587–625.
20. MAHER, M. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proceedings, Third Annual Symposium on Logic in Computer Science* (1988), IEEE Computer Society, pp. 348–357.
21. NIEUWENHUIS, R., AND OLIVERAS, A. Congruence closure with integer offsets. In *10th Int. Conf. on Logics for Programming, AI and Reasoning (LPAR)* (Sept. 2003).
22. PICHLER. On the complexity of equational problems in CNF. *Journal of Symbolic Computation 36* (2003).
23. PNUELI, A., RODEH, Y., SHTRICHMAN, O., AND SIEGEL, M. Deciding equality formulas by small domains instantiations. In *Proceedings of Computer Aided Verification, CAV'99* (1999), N. Halbwachs and D. Peled, Eds., LNCS 1633, Springer-Verlag.
24. ROBINSON, J. A machine-oriented logic based on the resolution principle. *Journal of the ACM 12*, 1 (1965), 23–49.
25. SHANKAR, N., AND RUESS, H. Combining Shostak theories. In *Proceedings of the 13th International Conference on Rewriting Techniques and Applications (Copenhagen, Denmark)* (July 2002), S. Tison, Ed., vol. 2378 of *Lecture Notes in Computer Science*, Springer, pp. 1–18.
26. ZANTEMA, H., AND GROOTE, J. F. Transforming equality logic to propositional logic. In *Proceedings of 4th International Workshop on First-Order Theorem Proving (FTP'03)* (2003), vol. 86(1) of *Electronic Notes in Theoretical Computer Science*.