Available online at www.sciencedirect.com

Information and Computation 205 (2007) 1188–1211

Information and Computation

www.elsevier.com/locate/ic

Generalizing DPLL and satisfiability for equalities

Bahareh Badban [d], Jaco van de Pol [a,b], Olga Tveretina [c], Hans Zantema [b,*]

[a] *Centrum voor Wiskunde en Informatica, Department of Software Engineering, P.O. Box 94.079, 1090 GB Amsterdam, The Netherlands*
[b] *Department of Computer Science, TU Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands*
[c] *Faculty of Science, University of Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands*
[d] *Department für Informatik, Carl von Ossietzky Universität Oldenburg, 26111 Oldenburg, Germany*

Received 20 December 2004; revised 10 November 2006
Available online 31 March 2007

Abstract

We present GDPLL, a generalization of the DPLL procedure. It solves the satisfiability problem for decidable fragments of quantifier-free first-order logic. Sufficient conditions are identified for proving soundness, termination and completeness of GDPLL. We show how the original DPLL procedure is an instance. Subsequently the GDPLL instances for equality logic, and the logic of equality over infinite ground term algebras are presented. Based on this, we implemented a decision procedure for inductive datatypes. We provide some new benchmarks, in order to compare variants.
© 2007 Elsevier Inc. All rights reserved.

Keywords: Satisfiability; DPLL procedure; Equality; Ground term algebra; Inductive datatypes; Decision procedure

1. Introduction

1.1. Contribution

In this paper, we provide a generalization of the well-known DPLL procedure, named after Davis–Putnam–Logemann–Loveland [16,15]. This DPLL procedure has been mainly used to decide satisfiability of propositional formulas, represented in conjunctive normal form (CNF). The main idea of this recursive procedure is to choose an atom from the formula and proceed with two recursive calls: one for the formula obtained by adding this atom as a fact and one for the formula obtained by adding the negation of this atom as a fact. Intermediate formulas may be further reduced. The search terminates as soon as a satisfying assignment is found, or alternatively, a simple satisfiability criterion may be used to terminate the search.

Although the original DPLL procedure 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.

[*] Corresponding author.
E-mail addresses: Bahareh.Badban@Informatik.Uni-Oldenburg.de (B. Badban), Jaco.van.de.Pol@cwi.nl (J. van de Pol), olga@cs.ru.nl (O. Tveretina), H.Zantema@tue.nl (H. Zantema).

0890-5401/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2007.03.003

However, this idea may be applied to other kinds of logics too. We will focus on certain quantifier free fragments of first-order logic for which this yields a (terminating) sound and complete decision procedure for satisfiability.

We first introduce a basic framework for satisfiability problems. The satisfiability problem for propositional logic, logic with equalities between variables, and the logic with equality and uninterpreted function symbols naturally fit in this framework. But also logics with interpreted symbols do fit. As an example we show the (quantifier free) logic of equality over an infinite ground term algebra (sometimes referred to as algebraic datatypes, or inductive datatypes). An instance of a formula in this logic would be

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

Subsequently we introduce a framework for generalized DPLL procedures (GDPLL). 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 formulas and a satisfiability (stop) criterion. We show sufficient conditions on these basic modules under which GDPLL is sound and complete. The original propositional DPLL algorithm (with or without unit resolution) can be obtained as an instance.

Finally, we provide a concrete algorithm for the logic with equalities over the ground term algebra. 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 particular solution for ground term algebras depends on well-known unification theory [25,4]. We follow the almost linear implementation of [23]. Our algorithm not only uses standard unification to deal with conjunctions of equalities, but it is also extended to disjunctions and negations.

The algorithm is an instance of GDPLL, so we show its soundness and completeness by checking the conditions mentioned above. An implementation in C of this algorithm can be found at http://www.cwi.nl/~vdpol/gdpll.html.

This paper is an extension of [6].

1.2. Applications

Many tools for deciding boolean combinations for certain theories exist nowadays. Typically, such procedures decide fragments of (Presburger) arithmetic and uninterpreted functions. These theories are used in hardware [24] and software [29] verification; other applications are in static analysis and abstract interpretation. However, we are not aware of a complete tool to decide boolean combinations of equality over an arbitrary ground term algebra, although this logic is has been studied quite extensively from a theoretical point of view.

Our main motivation has been to decide boolean combinations over algebraic data types. 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 [21,10], a language based on abstract data types and process algebra.

Our algorithm works for constructor symbols only (such as zero, successor, nil and cons). An extension to recognizer predicates (such as nil?, succ?, cons?, zero?) has been worked out in Ref. [5], by eliminating these predicates by introducing new variables. We expect a similar approach will work out for standard destructors (such as predecessor, head and tail). Other defined operations, such as plus and append, are currently out of scope. However, a sound but incomplete algorithm could be obtained by viewing defined operations as uninterpreted function symbols, and applying Ackermann's reduction [1].

1.3. Related work

Our algorithm is comparable to ICS [31,17] (which is used in PVS) and CVC [8,32], but as opposed to these tools, our algorithm is sound and complete for the ground term algebra. ICS and CVC tools combine several decision procedures by an algorithm devised by Shostak. Among these are a congruence closure algorithm for uninterpreted functions, and a decision procedure for arithmetic, including $+$ and $>$. They also support inductive datatypes. In ICS inductive datatypes are specified as a combination of products and coproducts; in CVC algebraic data types can be defined inductively. However, both tools are incomplete for quantifier free logic over inductive datatypes. For instance, experiments show that CVC does not prove validity of the query $x \neq succ(succ(x))$.

Another sound but incomplete approach for general algebraic data types is based on equational BDDs [22]. A complete algorithm for BDDs with equations, zero and successor is treated in Ref. [7], but cannot be easily extended to arbitrary data types.

In the past several years various approaches based on the DPLL procedure have been proposed [19,3,2,27, 18]. MathSAT [3] 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. FDPLL [9] is a generalization of DPLL to first-order logic. Note that FDPLL solves a different problem. First, it deals with quantifiers. Second, it does not take into account equality, or fixed theories, such ground term algebras. The algorithm is called sound and complete, but it is not terminating, because satisfiability for first-order logic is undecidable. Our GDPLL is meant for decidable fragments, so we only dealt with quantifier free logics.

We next compare our approach technically with the closest related work [18]. As in Ref. [18] we provide a generalized DPLL procedure that can be instantiated to several background theories. Ref. [18] call their approach DPLL(*T*), indicating a modular setup, where DPLL(*X*) is a theory-independent module, and $Solver_T$ implements a solver for theory T. Hence, the conjunction of all choices made in history is kept separate from the formula to be investigated. Also the ICS and CVC algorithms use a context of previously asserted formulas.

In contrast, in our approach the only data structure is the current set of clauses. We encode all theory facts in the clause set, and do not keep a background theory. For equality logics, this typically means that we do not remove negative unit clauses. However, positive unit clauses will be removed by the reduction procedure in our algorithm, performing the corresponding substitution to all clauses.

Another difference lies in the concrete logic to which the general theory is applied. Ref. [18] considers EUF-logic, with uninterpreted function symbols, while we focus on the ground term algebra. Although syntactically the formulas appear to be the same, the semantics is completely different. For instance, in EUF the formula $g(a) = f(a)$ is satisfiable, while in ground term algebra (the Herbrand Universe) it is unsatisfiable. Our approach applied to EUF-logic has been worked out in Refs. [33,34].

Other approaches encode the satisfiability question for a particular theory into plain propositional logic. For the logic of equality and uninterpreted function symbols, one can use Ackermann's reduction [1,12] to eliminate the function symbols. This yields a formula with equalities between variables only. Solving such formulas is based on the observation that a formula with n variables is satisfiable iff it is satisfiable in a model with n elements, so each variable can be encoded by $log(n)$ boolean variables. Other encodings work via adding transitivity constraints [20,12]. Several encodings are compared in Ref. [35].

Our particular solution for ground term algebras depends on well-known unification theory. Ground breaking work in this area was done by Robinson [30]. We follow the almost linear implementation of Ref. [23]. Unification solves conjunctions of equations in the ground term algebra. Colmerauer [13] 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.

For an extensive treatment of unification, see [25] and for a textbook on unification (theory and algorithms) we recommend [4]. The full first-order theory of equality in ground term algebras is studied in Ref. [26,14] (both focus on a complete set of rewrite rules) and in Ref. [28] (who focuses on complexity results for DNFs and CNFs in case of bounded and unbounded domains). Our algorithm is consistent with Pichler's conclusion that for unbounded domains the transformation to CNF makes sense. None of these papers give concrete algorithms for use in verification, and the idea to combine unification and DPLL seems to be new.

2. Basic definitions and preliminaries

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 [14,26].

2.1. Syntax

Let $\Sigma = (\mathsf{Fun}, \mathsf{Pr})$ be a signature, where $\mathsf{Fun} = \{f, g, h, \ldots\}$ is a set of *function symbols*, and $\mathsf{Pr} = \{p, q, r \ldots\}$ 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 $\mathsf{Var} = \{x, y, z, \ldots\}$ 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 as follows. The set of *ground terms* $\mathsf{Term}(\Sigma)$ is defined as $\mathsf{Term}(\Sigma, \emptyset)$.

- $x \in \mathsf{Var}$ is a term,
- $f(t_1, \ldots, t_n)$ is a term if $t_1, \ldots, t_n$ are terms, $f \in \mathsf{Fun}$ and n is the arity of f.

An *atom a* is defined to be an expression of the form $p(t_1, \ldots, t_n)$, 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 over the signature Σ is denoted by $\mathsf{Lit}(\Sigma, \mathsf{Var})$ or if it is not relevant by Lit. We denote by Lit_p and Lit_n, respectively, the set of all positive literals and the set of all negative literals.

A *clause C* is defined to be a finite set of literals. For the empty clause we use the notation $\perp$. 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:

Definition 1. In a CNF ϕ and literal $l \in \mathsf{Lit}$, let

- $\mathsf{Var}(\phi)$ be the set of all variables occurring in ϕ (similar for terms, literals and clauses),
- $\mathsf{Pr}(\phi)$ be the set of predicate symbols occurring in ϕ,
- $\mathsf{At}(\phi)$ be the set of all atoms occurring in ϕ,
- $\mathsf{Lit}(\phi)$, $\mathsf{Lit}_p(\phi)$, $\mathsf{Lit}_n(\phi)$ be, respectively, the set of all literals, the set of all positive literals and the set of all negative literals in ϕ,
- $\phi|_l = \{C - \{\neg l\} | C \in \phi, l \notin C\}$,
- $\phi \wedge l$ be a shortcut for $\phi \cup \{\{l\}\}$.

Finally, we say that a clause C is purely positive if all its literals are positive.

Example 2. Consider

$$\phi \equiv \{\{r, q\}, \{\neg r, p\}\}.$$

Then

$$\phi|_r \equiv \{\{p\}\}.$$

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* function $[\![-]\!]^\sigma_\mathcal{D} : \mathsf{Term}(\Sigma, \mathsf{Var}) \to D$ is inductively defined by

- $[\![x]\!]^\sigma_\mathcal{D} = \sigma(x)$ if $x \in \mathsf{Var}$,
- $[\![f(t_1, \ldots, t_r)]\!]^\sigma_\mathcal{D} = f_D([\![t_1]\!]^\sigma_\mathcal{D}, \ldots, [\![t_r]\!]^\sigma_\mathcal{D})$.

For literals, clauses and CNFs we define interpretations in $\{\mathsf{false}, \mathsf{true}\}$, also using the notation $[\![-]\!]^\sigma_\mathcal{D}$. For an atom $p(t_1, \ldots, t_n)$ we define

$$[\![p(t_1, \ldots, t_n)]\!]^\sigma_\mathcal{D} = p_D([\![t_1]\!]^\sigma_\mathcal{D}, \ldots, [\![t_r]\!]^\sigma_\mathcal{D}).$$

On the values $\mathsf{false}, \mathsf{true}$ we assume the usual boolean operations $\neg, \vee, \wedge$. For a negated atom $\neg a$ we define

$$[\![\neg a]\!]^\sigma_\mathcal{D} = \neg [\![a]\!]^\sigma_\mathcal{D}.$$

For a clause $C = \{l_1, \ldots, l_m\}$ we define

$$[\![\{l_1, \ldots, l_m\}]\!]^\sigma_\mathcal{D} = [\![l_1]\!]^\sigma_\mathcal{D} \vee \ldots \vee [\![l_m]\!]^\sigma_\mathcal{D},$$

For a CNF $\phi = \{C_1, \ldots, C_r\}$ we define

$$[\![\{C_1, \ldots, C_r\}]\!]^\sigma_\mathcal{D} = [\![C_1]\!]^\sigma_\mathcal{D} \wedge \ldots \wedge [\![C_r]\!]^\sigma_\mathcal{D}.$$

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 for every instance we introduce the notion of *admissible structure*. Depending on this notion of admissible structure we have the following definition of satisfiability.

Definition 3. An assignment $\sigma : \mathsf{Var} \to D$ *satisfies* a CNF ϕ in a structure $\mathcal{D}$, if $[\![\phi]\!]^\sigma_\mathcal{D} = \mathsf{true}$. A 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.

Lemma 4. *Suppose σ is an assignment which satisfies the literal l in some structure $\mathcal{D}$. Then for any formula ϕ it holds that σ satisfies ϕ if and only if σ satisfies $\phi|_l$.*

Proof. We prove each side separately:

- If σ satisfies ϕ then regarding Definition 1 we must prove that σ satisfies $C - \{\neg l\}$ for any $C \in \phi$, where $l \notin C$. σ does not satisfy $\neg l$, since it satisfies l, moreover σ satisfies C, since it satisfies ϕ. Hence σ satisfies $C - \{\neg l\}$. Therefore σ satisfies $\phi|_l$.
- If σ satisfies $\phi|_l$, then regarding Definition 1, we only need to show that σ satisfies every clause C of ϕ containing l. σ satisfies l therefore it will also satisfy any clause C containing that. $\square$

3. Instances

In this section, we describe precisely different instances of the framework just described by specifying the signature and the admissible structures.

3.1. 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.

3.2. Equality logic

We reserve the notation $\approx$ for a particular binary predicate symbol for reasoning over equality. For this symbol we will use infix notation, i.e., we write $x \approx y$ instead of $\approx xy$. We will use the shortcut $x \not\approx y$ for $\neg(x \approx y)$.

Since this symbol will be used for reasoning with equality, in admissible structures we will require that $\approx_D = Id_D$, where the function $Id_D : D \times D \to \{\mathsf{true}, \mathsf{false}\}$ is defined as follows:

$$Id_D(d_1, d_2) = \begin{cases} \mathsf{true} & \text{if} \, d_1 = d_2 \\ \mathsf{false} & \text{otherwise} \end{cases}$$

The first instance using $\approx$ is equality logic. By equality i logic formulas we mean formulas built from atoms of the shape $x \approx y$, where x and y are variables and usual propositional connectives. Now we define equality formulas in conjunctive normal form as an instance of the syntax described above.

For equality logic we have $s = \{\mathsf{Fun}, \mathsf{Pr}\}$, 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 $\approx_D = Id_D$.

As an example we consider

$$\phi = \{\{x \approx y\}, \{y \approx z\}, \{x \not\approx z\}\}.$$

Assume ϕ is satisfiable. Then an admissible structure $\mathcal{D}$ and an assignment $\sigma : \mathsf{Var} \to D$ exist such that $[\![\phi]\!]^{\sigma}_{\mathcal{D}} = \mathsf{true}$. Hence we have

- $[\![x \approx y]\!]^{\sigma}_{\mathcal{D}} = Id_D(\sigma(x), \sigma(y)) = \mathsf{true}$, hence $\sigma(x) = \sigma(y)$, and
- $[\![y \approx z]\!]^{\sigma}_{\mathcal{D}} = Id_D(\sigma(y), \sigma(z)) = \mathsf{true}$, hence $\sigma(y) = \sigma(z)$, and
- $[\![x \approx z]\!]^{\sigma}_{\mathcal{D}} = Id_D(\sigma(x), \sigma(z)) = \mathsf{false}$, hence $\sigma(x) \neq \sigma(z)$, contradiction.

Hence we proved that ϕ is unsatisfiable. Roughly speaking an equality logic CNF is unsatisfiable if and only if a contradiction can be derived using the CNF itself and reflexivity, symmetry and transitivity of equality, see [35].

In this basic version of equality logic there are no function symbols. In the next subsection, we discuss a way to deal with function symbols: they can be interpreted in the term algebra in which their interpretation is fixed to coincide with the term constructor. Alternatively, in EUF-logic (equality of uninterpreted functions) there is no restriction on the interpretation of the function symbols by which they are called uninterpreted.

In fact these two options are the two extremes; many combinations are possible.

3.3. Ground term algebra

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 $\approx$. The idea is that $\approx$ again represents equality and that terms are only interpreted by ground terms, i.e., in $\mathsf{Term}(\Sigma)$. Every symbol is interpreted by its term constructor. Hence we allow only one admissible structure $\mathcal{D}$, for which

- $D = \mathsf{Term}(\Sigma)$,
- $f_D(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$ for all $f \in \mathsf{Fun}$ and all $t_1, \ldots, t_n \in \mathsf{Term}(\Sigma)$, where n is the arity of f,
- $\approx_D = Id_D$.

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.

4. GDPLL

The DPLL procedure, due to Davis, Putnam, Logemann, and Loveland, is the basis of some of the most successful propositional satisfiability solvers. The original DPLL procedure 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. Therefore, we will mean this propositional version whenever we refer to DPLL. In this paper, we present a generalization GDPLL of this procedure, and we adopt it for some fragments of first-order logic. The satisfiability problem is decidable in these logics.

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 to some criteria. In GDPLL, such rules are combined in one function Reduce which performs some formula reduction. Also the notion of a splitting rule appears in GDPLL, 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 the following functions, for which we will introduce a number of requirements:

- $\mathsf{Reduce} : \mathsf{Cnf} \to \mathsf{Cnf}$.

We define the set $\mathsf{Rcnf} = \{\phi \in \mathsf{Reduce}(\mathsf{Cnf}) | \bot \notin \phi\}$. The other functions that we assume in the following are:

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

We now introduce requirements on these functions, to be referred to as Property 1–5, that are required to achieve correctness of our algorithm.

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

The skeleton of the algorithm is as follows:

```
GDPLL($\phi$) : {SAT, UNSAT} =
    begin
        $\phi$ := Reduce($\phi$);
        if ($\bot \in \phi$) then return UNSAT;
        if (SatCriterion($\phi$)) then return SAT;
        choose $a \in$ Eligible($\phi$);
        if GDPLL(Filter($\phi, a$)) = SAT then return SAT;
        if GDPLL(Filter($\phi, \neg a$)) = SAT then return SAT;
        return UNSAT;
    end;
```

The procedure takes as an input $\phi \in$ 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.

4.1. Soundness and completeness of GDPLL

Theorem 5 (soundnessandcompleteness)**.** *Let $\phi \in$ Cnf and Properties 1–5 hold. Then the following properties hold:*

- *If ϕ is satisfiable then* GDPLL(ϕ) = SAT.
- *If ϕ is unsatisfiable then* GDPLL(ϕ) = UNSAT.

Proof. Let $\phi \in$ Cnf. We apply induction on $\prec$, which is well-founded by Property 3. So assume (induction hypothesis) that for all ψ such that Reduce(ψ) $\prec$ Reduce(ϕ), we have GDPLL(ψ) returns UNSAT if ψ is unsatisfiable, and GDPLL(ψ) returns SAT if ψ is satisfiable.

By Property 1, Reduce(ϕ) is satisfiable if ϕ is satisfiable, and Reduce(ϕ) is unsatisfiable if ϕ is unsatisfiable. We now distinguish cases, according to the code of the GDPLL skeleton.

Let $\bot \in$ Reduce(ϕ). Then trivially ϕ is unsatisfiable, and GDPLL(ϕ) returns UNSAT.

Let $\bot \notin$ Reduce(ϕ). We distinguish two cases: If SatCriterion(Reduce(ϕ)) = true then by Property 4, Reduce(ϕ), and hence also ϕ, is satisfiable, and GDPLL(ϕ) = SAT.

If SatCriterion(Reduce(ϕ)) = false then by Property 5, Eligible(Reduce(ϕ)) $\neq \emptyset$. So let a be an arbitrary element in Eligible(Reduce(ϕ)). By Property 3, we obtain that

- Reduce(Filter(Reduce(ϕ), a)) $\prec$ Reduce(ϕ) and
- Reduce(Filter(Reduce(ϕ), $\neg a$)) $\prec$ Reduce(ϕ).

Again, we distinguish two cases: let Reduce(ϕ) be unsatisfiable. Then by Property 2, Filter(Reduce(ϕ), a) and Filter(Reduce(ϕ), $\neg a$) are unsatisfiable. We can apply induction hypothesis. Then both GDPLL(Filter(Reduce(ϕ), a)) and GDPLL(Filter(Reduce(ϕ), $\neg a$)) return UNSAT. By definition of GDPLL, GDPLL(ϕ) also returns UNSAT.

Let Reduce(ϕ) be satisfiable. By Property 2, at least one of Filter(Reduce(ϕ), a) and Filter(Reduce(ϕ), $\neg a$) is satisfiable. By induction hypothesis, at least one of GDPLL(Filter(Reduce(ϕ), a)) and GDPLL(Filter(Reduce(ϕ), $\neg a$)) return SAT, and by definition of GDPLL, GDPLL(ϕ) also returns SAT. □

5. Instances for GDPLL

In this section, we elaborate two instances of GDPLL as mentioned before: propositional logic and equality logic. The instance for ground term algebras is dealt with in a separate section. In particular for all of these instances we define the functions Eligible, Filter, Reduce and SatCriterion, and prove the required properties.

5.1. GDPLL *for propositional logic*

Two main operations of the standard propositional DPLL procedure are *unit propagation* and *purification*. A clause C is called a *unit clause* if $|C| = 1$. Unit clauses can only be satisfied by a specific assignment to the corresponding propositional variable. So if a unit clause $\{l\}$ occurs in ϕ, then ϕ may be replaced by $\phi|_l = \{C - \{\neg l\} | C \in \phi, l \notin C\}$. This is called *unit propagation.* It can create new unit clauses, so this process has to be repeated until no unit clauses are left.

Purification can be applied if the formula contains pure literals, i.e., literals for which the negation does not occur at all in the formula. Such literals can be eliminated by assigning true in the positive case and false in the negative case. This is called *purification.* So purification of a formula ϕ with respect to a pure literal l yields $\phi|_l$. Note that this does not introduce new unit clauses.

It can be seen that the DPLL procedure for propositional logic is a particular case of GDPLL, where unit resolution and purification are performed by Reduce. In case of propositional logic, we let an eligible atom be an arbitrary atom, i.e., to coincide with the original DPLL procedure, we choose

$$\mathsf{Eligible}(\phi) = \mathsf{At}(\phi).$$

We define SatCriterion as follows:

$$\mathsf{SatCriterion}(\phi) = \begin{cases} \mathsf{true} & \text{if } \phi = \emptyset \\ \mathsf{false} & \text{otherwise} \end{cases}$$

The function Reduce is defined as follows. Here the function UnitClause(ψ) returns a unit clause contained in ψ, and the function PureLiteral(ψ) returns a pure literal contained in ψ.

```
Reduce($\phi$);
    begin
        $\psi := \phi$;
        while (there is a unit clause in $\psi$)
        begin;
            $l :=$ UnitClause($\psi$);
            $\psi := \psi|_l$;
        end;
        while (there is a pure literal in $\psi$)
        begin;
            $l :=$ PureLiteral($\psi$);
            $\psi := \psi|_l$;
        end;
    return $\psi$;
    end;
```

We define for all $\phi \in \mathsf{Reduce}(\mathsf{Cnf})$ and all $l \in \mathsf{Lit}(\phi)$

$$\mathsf{Filter}(\phi, l) = \phi \wedge l.$$

Definition 6 (*orderingonformulas*). Given $\phi_1, \phi_2 \in \mathsf{Cnf}$, we define $\phi_1 \prec \phi_2$ if $|\mathsf{Pr}(\phi_1)| < |\mathsf{Pr}(\phi_2)|$.

The defined order is trivially well-founded.

Example 7. Consider

$$\phi_1 \equiv \{\{\neg p, q, r\}, \{\neg q, r\}, \{\neg r\}\},$$
$$\phi_2 \equiv \{\{\neg p, r\}, \{p, r\}, \{\neg r\}, \{p, \neg r\}, \{\neg p\}\}.$$

According the definition $\phi_2 \prec \phi_1$.

Theorem 8. *The functions* Reduce, Eligible, Filter, SatCriterion *satisfy the Properties 1–5.*

Proof.

(1) Property 1 holds since unit clauses can only be satisfied by a specific assignment to corresponding propositional variable, and the complementary assignment will lead to contradiction, and pure literals can be eliminated by assigning true in the positive case and false in the negative case.
(2) By definition of Filter, Property 2 trivially hold.
(3) We will prove Property 3. We have to prove that $\mathsf{Reduce}(\phi \wedge l) \prec \phi$ for all $\phi \in \mathsf{Cnf}$ and for all $l \in \mathsf{Lit}(\phi)$. We consider the case when $\phi|_l$ contains no unit clauses and pure literals. All other cases can be easily proved by induction.
Let $l \equiv p$ for some $p \in \mathrm{Pr}$. Since by the theorem conditions $l \in \mathsf{Lit}(\phi)$ then trivially $\mathsf{Pr}(\phi|_l) \subseteq \mathsf{Pr}(\phi)\backslash\{p\}$. Using Definition 6 one can see that from

$$|\,\mathrm{Pr}(\mathsf{Reduce}(\phi \wedge p))| = |\mathsf{Pr}(\phi|_p)| \leqslant |\mathsf{Pr}(\phi)\backslash\{p\}| < |\mathsf{Pr}(\phi)|$$

it follows that

$$\mathsf{Reduce}(\phi \wedge p) \prec \phi.$$

The case $l \equiv \neg p$ for some $p \in \mathsf{Pr}$ is similar.
(4) We will check Property 4. By definition, the function $\mathsf{SatCriterion}(\phi)$ returns true only if $\phi = \emptyset$, which is satisfiable by definition.
(5) Property 5 follows from the fact that if $\mathsf{SatCriterion}(\phi) = \mathsf{false}$ then by the definition of SatCriterion there is $C \in \phi$ such that $C \neq \bot$. Then $\mathsf{Lit}(\phi) \neq \emptyset$, and $\mathsf{Eligible}(\phi) \neq \emptyset$. □

We have defined the functions Eligible, Reduce, Filter, and SatCriterion. One can see that GDPLL now coincides with the DPLL procedure for propositional logic.

In the situation when ϕ consists of relatively few clauses relative to the number of variables in each clause splitting can be very inefficient. The following theorem allows the procedure to stop some earlier: when every clause in ϕ contains at least one negative literal. In order to do so we present a modified version of the function SatCriterion.

Theorem 9 (SAT criterion)**.** *Let* $\phi \in \mathsf{Cnf}$ *contain no purely positive clause. Then* ϕ *is satisfiable.*

Proof. For all $p \in \mathsf{Pr}$ we define $[\![p]\!]^{\sigma}_{\mathcal{D}} = \mathsf{false}$. Regarding the theorem conditions for all $C \in \phi$ there is $l \in C$ such that $l \equiv \neg p$ for some $p \in \mathsf{Pr}$. We have that $[\![l]\!]^{\sigma}_{\mathcal{D}} = \mathsf{true}$ and $[\![C]\!]^{\sigma}_{\mathcal{D}} = \mathsf{true}$ for all $C \in \phi$. By definition of a formula interpretation $[\![\phi]\!]^{\sigma}_{\mathcal{D}} = \mathsf{true}$. □

In order to improve GDPLL we redefine the function SatCriterion

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

Using the above theorem the modified function SatCriterion satisfies Properties 4 and 5, by which this optimized version of GDPLL is correct.

For forthcoming instances of GDPLL the corresponding function SatCriterion will be defined in a similar way.

5.2. 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. In the following, we consider $x \approx y$ and $y \approx x$ as the same atom.

In case of propositional logic we may choose any atom contained in a CNF to apply the split rule. The correctness of GDPLL is not immediate for other instances: in order to get termination by Property 3 recursive calls of Reduce $\circ$ Filter have to show up progress which is not obvious in instances like equality logic.

For equality logic we define an atom to be eligible if it occurs as a positive literal in the formula, i.e., $\mathsf{Eligible}(\phi) = \mathsf{Lit}_p(\phi)$.

Example 10. Let us consider the formula

$$\phi \equiv \{\{x \approx y\}, \{y \approx z\}, \{x \not\approx z\}\}.$$

One can see that $(x \approx z) \notin \mathsf{Eligible}(\phi)$ since it occurs in ϕ only as a negative literal $x \not\approx z$.

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 11. 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:

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

Definition 12 (*ordering on formulas*)**.** Given $\phi_1, \phi_2 \in \mathsf{Cnf}$, we define $\phi_1 \prec \phi_2$ if $|\mathsf{Lit}_p(\phi_1)| < |\mathsf{Lit}_p(\phi_2)|$.

The defined order is trivially well-founded.

Example 13. Consider

$$\phi_1 \equiv \{\{x \approx y, y \not\approx z\}, \{x \approx z, x \not\approx y, y \approx z\}, \{x \not\approx z\}\},$$
$$\phi_2 \equiv \{\{x \approx y, y \not\approx z\}, \{x \not\approx y, y \approx z\}, \{x \not\approx z\}, \{x \not\approx y, y \not\approx z\}\}.$$

Since $\mathsf{Lit}_p(\phi_2) = \mathsf{Lit}_p(\phi_1) \backslash \{x \approx z\}$ one can see that by the definition $\phi_2 \prec \phi_1$.

Theorem 14. *The functions* Reduce, Eligible, Filter, SatCriterion *satisfy the Properties 1–5.*

Proof.

(1) Property 1 holds since $[\![x \approx x]\!]^{\sigma}_{\mathcal{D}} = \mathsf{true}$ for all admissible $\mathcal{D}$ and all $\sigma : \mathsf{Var} \to D$, i.e., removing clauses containing $x \approx x$ from the formula and the literal $x \not\approx x$ from all clauses can be done without influence on satisfiability of the formula.

(2) We will prove Property 2. For each $a \in \mathsf{At}$, ϕ is satisfiable iff at least one of $\phi \wedge a$ and $\phi \wedge \neg a$ is satisfiable. Trivially, $\phi \wedge (x \approx y)$ is satisfiable iff $\phi|_{x\approx y}[x := y]$ is satisfiable for all $x, y \in \mathsf{Var}$. From this we can conclude that the property holds.

(3) We will prove Property 3.
At first we will prove that $\phi|_{x\approx y}[x := y] \prec \phi$ and $\phi|_{x\not\approx y} \wedge (x \not\approx y) \prec \phi$ for all $\phi \in \mathsf{Cnf}$ and all $(x \approx y) \in \mathsf{Lit}(\phi)$.

- Let $l \equiv x \approx y$.
  It follows from the definition of $\phi|_{x\approx y}$ and the fact $(x \approx y) \in \mathsf{Lit}(\phi)$ that

  $$|\mathsf{Lit}_p(\phi|_{x\approx y})| < |\mathsf{Lit}_p(\phi)|.$$

  One can easily check that for all $\psi|_{(x\approx y)} \in \mathsf{Cnf}$

  $$|\mathsf{Lit}_p(\psi|_{x\approx y}[x := y])| \leqslant |\mathsf{Lit}_p(\psi|_{x\approx y})|.$$

  We obtain that

  $$|\mathsf{Lit}_p(\phi|_{x\approx y}[x := y]| \leqslant |\mathsf{Lit}_p(\phi|_{x\approx y})| < |\mathsf{Lit}_p(\phi)|.$$

  We can conclude that

  $$\phi|_{x\approx y}[x := y] \prec \phi.$$

- Let $l \equiv x \not\approx y$.
  Since by the theorem conditions $(x \approx y) \in \mathsf{Lit}_p(\phi)$ then

  $$|\mathsf{Lit}_p(\phi|_{x\not\approx y})| < |\mathsf{Lit}_p(\phi)|.$$

  We have that

  $$|\mathsf{Lit}_p(\phi|_{x\not\approx y} \wedge (x \not\approx y))| = |\mathsf{Lit}_p(\phi|_{x\not\approx y})| < |\mathsf{Lit}_p(\phi)|.$$

  We can conclude that

  $$\phi|_{x\not\approx y} \wedge (x \not\approx y) \prec \phi.$$

Regarding the definition of the function Reduce, we obtain that for all $\phi \in \mathsf{Reduce}(\mathsf{Cnf})$ and all $a \in \mathsf{Eligible}(\phi)$

$$\mathsf{Reduce}(\mathsf{Filter}(\phi, a)) \prec \phi, \mathsf{Reduce}(\mathsf{Filter}(\phi, \neg a)) \prec \phi.$$

(4) Let $\mathsf{SatCriterion}(\phi) = \mathsf{true}$. Then either $\phi = \emptyset$ or every clause in ϕ contains at least one negative literal. If $\phi = \emptyset$ then by definition ϕ is satisfiable. Let us consider the remaining case. Let D be a domain such that $|D| \geqslant |\mathsf{Var}(\phi)|$. We choose an assignment σ such that $\sigma(x) \neq \sigma(y)$ for all $x, y \in \mathsf{Var}(\phi)$. Regarding the definition of a CNF interpretation we have that $[\![\phi]\!]^{\sigma}_{\mathcal{D}} = \mathsf{true}$.

(5) Property 5 follows from the fact that if $\mathsf{SatCriterion}(\phi) = \mathsf{false}$ then there is $C \in \phi$ such that $C \neq \bot$ and $C \cap \mathsf{Lit}_p = C$. We obtain that $\mathsf{Lit}_p(\phi) \neq \emptyset$, and $\mathsf{Eligible}(\phi) \neq \emptyset$. $\square$

Many alternative instances of GDPLL are possible. Here we chose to do the main job in Filter, while Reduce only removes trivialities $x \approx x$ from its argument. In fact in this version Reduce is only required in the first call of GDPLL as a kind of preprocessing, the other calls of Reduce may be omitted since atoms of the shape $x \approx x$ will not be created by Filter.

In the next section, we will choose the opposite approach. There a version of GDPLL is developed for ground term algebra, which may be applied to equality logic formulas too. In that solution Filter is trivial and Reduce does the real work.

6. 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 algebraic datatypes). In Section 3.3 we showed how ground term algebras fit in the general framework. Recall that the only predicate symbol was $\approx$ (binary, written infix). Hence in the sequel, we work with an arbitrary but fixed signature of the form $\Sigma = (\mathsf{Fun}; \approx)$. 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). Recall that there is only one admissible structure $\mathcal{D}$, whose domain is $\mathsf{Term}(\Sigma)$. The interpretation f_D coincides with applying function symbol f; and $\approx$ is interpreted as syntactic identity. A context C is defined to be a term containing one occurrence of a constant $[]$, where for a term t the term obtained from C by replacing $[]$ by t is denoted by $C[t]$.

We will use the following properties of all ground term algebras:

Lemma 15. *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 5 that the obtained procedure is sound and complete.

6.1. Substitutions and most general unifiers

We introduce here the standard definitions of substitutions and unifiers, taken from [25,4].

Definition 16. 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**

$$\mathsf{Dom}(\sigma) = \{x \in \mathsf{Var} | \sigma(x) \neq x\}.$$

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.

Substitutions are extended to terms/literals/clauses/cnfs as follows:

Definition 17. We define an application of substitution $(.)^\sigma$ as below

$$x^\sigma = \sigma(x)$$
$$f(t_1, \ldots, t_n)^\sigma = f(t_1^\sigma, \ldots, t_n^\sigma)$$

$(t \approx u)^\sigma = t^\sigma \approx u^\sigma$ (likewise for its negation)
$\{l_1, \dots, l_n\}^\sigma = \{l_1^\sigma, \dots, l_n^\sigma\}$
$\{C_1, \dots, C_n\}^\sigma = \{C_1^\sigma, \dots, C_n^\sigma\}$

So, ϕ^σ is obtained from ϕ by replacing each occurrence of a variable x by $\sigma(x)$.

Definition 18. The **composition** $\rho\sigma$ of substitutions ρ and σ is defined such that $\rho\sigma(x) = (x^\rho)^\sigma$. A substitution σ is **more general** than a substitution σ', written as $\sigma \lesssim \sigma'$, if there is a substitution δ such that $\sigma' = \sigma\delta$. Furthermore, a substitution σ is **idempotent** if $\sigma\sigma = \sigma$.

Definition 19. A **unifier** or solution of a set $S = \{s_1 \approx t_1, \dots, s_n \approx t_n\}$ of finite number of atoms, is a substitution σ such that $s_i^\sigma = t_i^\sigma$ for all $1 \leqslant i \leqslant n$.
A substitution σ is a most general unifier of S or in short $\mathsf{mgu}(S)$, if

- σ is a unifier of S and
- $\sigma \lesssim \sigma'$ for each unifier σ' of S.

Definition 20. An atom $t \approx u$ is in **solved form** if it is of the form

$x \approx u$, where $x \notin \mathsf{Var}(u)$

otherwise it is non-solved. Similar for literals and sets of literals.

In the sequel, we will use the following well-known facts on substitutions and unifiers (cf. [25,4]).

Lemma 21.

(1) *A substitution σ is idempotent if and only if* $\mathsf{Dom}(\sigma) \cap \mathsf{Var}(\sigma) = \emptyset$.
(2) *If a set S of atoms has a unifier, then it has an idempotent mgu.*
(3) *If $\sigma = \mathsf{mgu}(S)$ and σ is idempotent, then $\mathsf{Eq}(\sigma)$ is in solved form, and logically equivalent to S.*

Notation and conventions.

- If $n = 1$ we simply write $\mathsf{mgu}(s_1 \approx t_1)$.
- We set $\mathsf{mgu}(S) = \bot$ if S has no unifier.
- When working on sets of unit clauses, by $\sigma = \mathsf{mgu}(\{t_1 \approx u_1\}, \dots, \{t_n \approx u_n\})$ we mean $\sigma = \mathsf{mgu}(\{t_1 \approx u_1, \dots, t_n \approx u_n\})$.
- From now on by a 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 $\sigma = \{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$ then $x_i \notin \mathsf{Var}(t_j)$ for all $1 \leqslant i, j \leqslant n$. Another consequence is that $\mathsf{mgu}(x \approx x) = \emptyset$.

6.2. *The* GDPLL *building blocks for ground term algebras*

We now come to the definition of the building blocks for GDPLL. The functions Eligible and SatCriterion correspond to those in Section 5.2 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 5.1 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. The function Reduce will be defined by means of a set of transformation rules, that can be applied in any order.

Definition 22. 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, then 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 $\phi_1 = \{C | C \in \phi \text{ is a positive unit clause}\} \neq \emptyset$, 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 equations are always oriented in a fixed order, so that $x \approx y$ and $y \approx x$ are treated identically; so a rule for symmetry is not needed. Rule 1 (reflexivity) and 2 are clear simplifications. 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.

Recall that Rcnf denotes the set of reduced formulas. We will show that the rules are terminating, so at least one normal form exists. Unfortunately, the rules are not confluent as we will show by an example, so the function Reduce is not uniquely defined. But any normal form will suffice, as we will prove. Now we give some examples of reduction, and show which shape a reduced CNF may have.

Example 23. $\phi = \{\{f(f(y)) \not\approx f(x)\}, \{x \not\approx x\}\}$. Applying rule 3 above, on $f(f(y)) \not\approx f(x)$ we will have $\sigma : x \mapsto f(y)$ therefore

$$\phi \longrightarrow \{\{x \not\approx f(y)\}, \{x \not\approx x\}\}.$$

Once more applying the same rule on $x \not\approx x$, we obtain

$$\phi \longrightarrow \{\{x \not\approx f(y)\}, \{\}\}.$$

The empty clause $\{\}$ is $\bot$, therefore regarding rule 2 we get

$$\phi \longrightarrow \{\bot\}.$$

Example 24. The formula ϕ below is reduced, since no rewrite rule of Definition 22 is applicable on it

$$\phi = \{\{x \not\approx f(y), z \approx g(x)\}, \{y \not\approx x\}\}.$$

Corollary 25. *Suppose ϕ is a reduced formula, then the following requirements will hold*

(1) *ϕ contains no literal of the form $t \approx t$.*
(2) *If $\bot \in \phi$ then $\phi \equiv \{\bot\}$.*
(3) *All its negative literals are solved.*
(4) *ϕ contains no positive unit clause.*
(5) *If $\phi = \{\{\neg a\}\} \uplus \phi_1$ then $a \notin \mathsf{At}(\phi_1)$.*

Proof. If ϕ does not satisfy one of the properties above, the corresponding rule can be applied. □

Next, we show an example where $\mathsf{Reduce}(\phi)$ is not uniquely defined.

Example 26. Consider $\phi = \{\{x \not\approx f(a,b)\}, \{x \approx f(y,z)\}, \{y \approx a, x \approx f(a,b)\}\}$. We show that using two different strategies, two distinct reduced forms for ϕ will be obtained:

(1) One approach:

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

The result is reduced because no other rule is applicable on it.

(2) Another approach:

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

which is reduced regarding the rewrite system of Definition 22.

6.3. Termination

We will now prove termination of the reduction system and of the corresponding GDPLL procedure (i.e., Property 3).

Definition 27. We define the following measures on formulas:

$$\mathsf{pos}(\phi) = \text{number of occurrences of positive literals in } \phi$$
$$\mathsf{neg}(\phi) = \text{number of occurrences of negative non} - \text{solved literals in } \phi$$

To each formula ϕ, we correspond a pair of numbers, namely $\mathsf{norm}(\phi)$ as below:

$$\mathsf{norm}(\phi) = (\mathsf{pos}(\phi) + |\phi|, \mathsf{neg}(\phi))$$

in which $|\phi|$ is the cardinality of ϕ.

Theorem 28.

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

Proof.

- We prove termination, by showing that after applying each step of the reduction system on a supposed formula, norm will decrease, with respect to the lexicographic order ($\prec_{lex}$) on pairs. So let $\phi \longrightarrow \phi'$

  (1) $\mathsf{pos}(\phi') + |\phi'| < \mathsf{pos}(\phi) + |\phi|$, obviously.

  (2) $|\phi'| = |\{\bot\}| = 1 < |\phi|$, and $\mathsf{pos}(\phi') \leqslant \mathsf{pos}(\phi)$.

(3) · if $\sigma = \bot$ then $|\phi'|=|\phi| - 1$ and
$\mathsf{pos}(\phi') \leqslant \mathsf{pos}(\phi)$;
· otherwise, $\mathsf{pos}(\phi') = \mathsf{pos}(\phi)$ and $|\phi'| = |\phi|$ but
$\mathsf{neg}(\phi') < \mathsf{neg}(\phi)$ as we only count non-solved inequalities.

(4) Let $\phi = \phi_1 \uplus \phi_2$, where ϕ_1 is the non-empty set of the positive unit literals in ϕ

· if $\sigma = \bot$ then $|\phi'| = |\{\bot\}| = 1 \leqslant |\phi_1| \leqslant |\phi|$ and $\mathsf{pos}(\phi') = \mathsf{pos}(\bot) < 1 \leqslant \mathsf{pos}(\phi)$;
· otherwise $|{\phi_2}^{\sigma}| = |\phi_2| < |\phi_1| + |\phi_2| \leqslant |\phi|$ and
$\mathsf{pos}({\phi_2}^{\sigma}) = \mathsf{pos}(\phi_2) \leqslant \mathsf{pos}(\phi)$.

(5) Let $\phi = \{\{\neg a\}\} \uplus \phi_1$, with $a \in \mathsf{At}(\phi_1)$

· if $a \in \mathsf{Lit}_p(\phi_1)$ then using Definition 1

$$\mathsf{pos}(\phi') = \mathsf{pos}(\phi_1|_{\neg a}) \leqslant \mathsf{pos}(\phi) - 1 < \mathsf{pos}(\phi).$$

We also have $|\phi'| \leqslant |\phi|$.
· otherwise $\neg a \in \mathsf{Lit}(\phi_1)$ and hence

$$\begin{aligned} |\phi'| &= |(\phi_1|_{\neg a})| + 1 \\ &< |\phi_1| + 1 \qquad \text{Definition 1} \\ &= |\phi| \end{aligned}$$

We also have $\mathsf{pos}(\phi') \leqslant \mathsf{pos}(\phi)$.

- Following each step, it is obvious that the second part of the theorem also holds. □

Theorem 29. $\mathsf{pos}(\mathsf{Reduce}(\phi \wedge l)) < \mathsf{pos}(\phi)$ *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. If $\mathsf{Reduce}(\phi \wedge l) = \{\bot\}$ then the theorem holds obviously. Otherwise, since ϕ is reduced, the first step to reduce $\phi \wedge l$, regarding the Definition 22, will be one of the rules 4 or 5; we distinguish cases:

- If $l = t \approx u$, then

$$\begin{aligned} \phi \wedge l &= \phi \wedge t \approx u \\ &= \phi \uplus \{\{t \approx u\}\} \qquad \phi \text{ is reduced and Corollary 25(4)} \\ &\longrightarrow \phi^{\sigma} \qquad \text{Definition 22(4) and } \mathsf{Reduce}(\phi \wedge l) \neq \{\bot\} \end{aligned}$$

$t \approx u \in \mathsf{Lit}_p(\phi)$, hence $t^{\sigma} \approx u^{\sigma} \in C \in \phi^{\sigma}$, where $t^{\sigma} = u^{\sigma}$ because $\sigma = \mathsf{mgu}(t \approx u)$. For simplicity we write it as $t^{\sigma} \approx t^{\sigma}$. Assume that $\phi^{\sigma} = \phi_0 \to \phi_1 \to \cdots \to \phi_{n+1} = \mathsf{Reduce}(\phi^{\sigma})$ is the reduction sequence by which we obtain $\mathsf{Reduce}(\phi^{\sigma})$ from ϕ^{σ}.
Applying any rule of the Definition 22 on ϕ_0, $t^{\sigma} \approx t^{\sigma}$ will be either removed or replaced by a similar one $t^{\rho} \approx t^{\rho}$. Regarding the Corollary 25(1), ϕ_{n+1} does not contain any literal of the shape $w \approx w$.
Since ϕ_0 contains at least one literal of that shape($t^{\sigma} \approx t^{\sigma}$), therefore there exists a $0 \leqslant j \leqslant n+1$ such that ϕ_j has a literal of the form $w \approx w$, and ϕ_{j+1} does not have any. Now since according to the Theorem 28(2), the number of occurrences of the positive literals does not increase during the reduction process, therefore $\mathsf{pos}(\phi_j) \leqslant \mathsf{pos}(\phi_{j+1}) - 1$. Hence $\mathsf{pos}(\phi_0) < \mathsf{pos}(\phi_{n+1})$, again regarding the Theorem 28(2).

- If $l = t \not\approx u$, then

$$\begin{aligned}\phi \wedge l \ &= \{\{t \not\approx u\}\} \uplus \phi \\ &\longrightarrow \{\{t \not\approx u\}\} \uplus \phi|_{t \not\approx u} \qquad \text{Definition 22(5)}, t \approx u \in \mathsf{Lit}_p(\phi).\end{aligned}$$

According the Definition 1, $t \approx u \notin \mathsf{Lit}_p(\phi|_{t\not\approx u})$ therefore

$$\begin{aligned}\mathsf{pos}(\mathsf{Reduce}(\phi \wedge l)) &\leqslant \mathsf{pos}(\{\{t \not\approx u\}\} \uplus \phi|_{t\not\approx u}) \qquad \text{Theorem 28(2)} \\ &\leqslant \mathsf{pos}(\phi) - 1 \quad \square\end{aligned}$$

6.4. Correctness properties of the building blocks

Theorem 30 (Reducedcriteria). *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. We check in any step of the reduction that ϕ is satisfiable if and only if the result is satisfiable. So assume that $\phi \to \phi'$; we now distinguish which rule of Definition 22 is applied:

(1) It is even obvious that α satisfies ϕ if and only if α satisfies ϕ', for each assignment α.
(2) Both are unsatisfiable.
(3) (a) If α satisfies ϕ then in the first case obviously α satisfies ϕ', which is $\phi - \{C\}$. In the second case also α satisfies ϕ' because $t \not\approx u$ is replaced by the negation of its unifier, which is equivalent by Lemma 21.(3).
  (b) Let α satisfy ϕ'. If $\mathsf{mgu}(t \approx u) = \bot$, then $\phi = \phi' \cup \{C\}$ and $t \not\approx u \in C$. Note that $t \not\approx u$ is a tautology, so α satisfies ϕ. Otherwise, ϕ' is obtained from ϕ by replacing $t \not\approx u$ by $\neg\mathsf{mgu}(t \approx u)$, which is equivalent by Lemma 21.(3). In both cases α satisfies ϕ.
(4) Let ϕ_1 be the non-empty set of positive unit clauses, and $\phi = \phi_1 \uplus \phi_2$.
  (a) If $\phi' = \{\bot\}$ then ϕ' is unsatisfiable, also ϕ is unsatisfiable since ϕ_1 has no unifier.
  (b) If α satisfies ϕ then it satisfies ${\phi_2}^\sigma$ trivially. Now if α satisfies ${\phi_2}^\sigma$ then define

$$\alpha'(y) = \begin{cases} \alpha(y) & \text{if } y \in \mathsf{Var}({\phi_2}^\sigma) \\ \alpha(\sigma(y)) & \text{otherwise} \end{cases}$$

  α' satisfies ϕ.

(5) Is obvious regarding Lemma 4. $\square$

Definition 31. Given a term t, we define $S(t)$ to be the number of occurrences of non-constant function symbols in t:

$$\begin{aligned} S(x) &= 0 \\ S(c) &= 0 && \text{if } c \text{ is a constant symbol} \\ S(f(t_1,\ldots,t_n)) &= 1 + \sum_{i=1}^{n} S(t_i) && \text{if } n \geqslant 1.\end{aligned}$$

Theorem 32 *(*SAT criteria*). Suppose $\mathcal{D}$ is a ground term algebra with infinitely many closed terms, then a reduced formula ϕ is satisfiable if ϕ has no purely positive clause.*

Proof. Suppose ϕ is a CNF formula which has the properties of the theorem, i.e., ϕ is reduced and ϕ has no purely positive clause (in particular, $\bot \notin \phi$). Let $n = |\phi|$. Then each clause of this formula has a negative literal of the form $x_i \not\approx t_i$, for $1 \leqslant i \leqslant n$, which is also solved regarding Corollary 25. It suffices to provide an assignment σ

which satisfies all these negative literals, because then each clause is satisfiable with that σ, which implies that ϕ is satisfiable. We distinguish two cases:

- $\mathcal{D}$ has at least one function symbol g, of arity m, bigger than zero.
  Suppose c is a constant symbol in $\mathcal{D}$. We identify a new function f as: $f[] = g([], \underbrace{c, \ldots, c}_{m-1 \text{ times}})$. Now define a number $M = 1 + \mathsf{Max}_{1 \leqslant i \leqslant n} S(t_i)$.
  Then define a context $C = f^M[]$, the M-fold application of f. Consider the following assignment:

  $$\sigma(x) = \begin{cases} C^i(c) & \text{if } x = x_i, \text{ for some } 1 \leqslant i \leqslant n \\ c & \text{otherwise} \end{cases}$$

  We claim that σ satisfies $x_i \not\approx t_i$ for each $1 \leqslant i \leqslant n$.
  Indeed, note that $S(\sigma(x_i)) = M * i$. Moreover, if $S(t_i) = 0$, then $S(\sigma(t_i)) = M * j$ with $0 \leqslant j \leqslant n$ 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 $k \geqslant 0$, and $0 < S(t_i) < M$. In both cases, $S(\sigma(x_i)) \neq S(\sigma(t_i))$.
- $\mathcal{D}$ has no non-constant function symbols. Therefore each of its negative literals are of the shape $x \not\approx t$, in which $x \neq t$ and t is a variable or a constant symbol, since $x \not\approx t$ is a solved atom. Define

  $$V_\phi = \{x | x \text{ is a variable occurring in } \phi\}$$
  $$C_\phi = \{c | c \text{ is a constant symbol occurring in } \phi\}$$

  We know that the two given sets, are of finite cardinality. Without loss of generality suppose that $V_\phi = \{x_1, x_2, \ldots, x_n\}$, for some $n \in \mathbb{N}$. Since $\mathcal{D}$ has infinitely many constant symbols, there exists a set $\mathcal{C} = \{c_1, c_2, \ldots, c_{n+1}\}$, of $n+1$ distinct constant symbols of $\mathcal{D}$, such that $C_\phi \cap \mathcal{C} = \emptyset$. Define

  $$\sigma(x) = \begin{cases} c_i & \text{if } x = x_i, \text{ for some } x_i \in V_\phi \\ c_{n+1} & \text{otherwise} \end{cases}$$

  Now $x_i \not\approx t$ has one of the following shapes:

  · $x_i \not\approx x_j$. Then σ satisfies it since $\sigma(x_i) \neq \sigma(x_j)$.
  · $x_i \not\approx c$. Then σ satisfies it since $\sigma(x_i) = c_i \neq c = \sigma(c)$, because $C_\phi \cap \mathcal{C} = \emptyset$.
  · $x_i \not\approx y$, where $y \notin V_\phi$. Then σ satisfies it since $\sigma(x_i) = c_i \neq c_{n+1} = \sigma(y)$. □

6.5. Correctness of GDPLL *for ground term algebras*

We can now combine the lemmas on the basic blocks, and apply Theorem 5 in order to conclude correctness of GDPLL for ground term algebras. First we instantiate GDPLL as follows. We take the Reduce function defined in Definition 22. We define for $\phi \in \mathsf{Reduce}(\mathsf{Cnf})$ and $l \in \mathsf{Lit}(\phi)$

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

Theorem 33. *Let* (Fun; $\approx$) *be a signature with infinitely many ground terms. Let $\mathcal{D}$ be its ground term algebra. Let ϕ be a CNF. Let* GDPLL *be instantiated as indicated above. Then*

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

Proof. In order to apply Theorem 5, we have to check Properties 1–5. Properties 2 and 5 are obvious. Property 1 has been proved in Theorem 30. Property 3 has been proved in Theorem 29; 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 Theorem 32. □

7. Implementation and experiments

7.1. Implementation

The GDPLL algorithm instantiated for ground term algebras has been implemented in C. As term representation we used the ATerm library [11]. This library provides a data structure for terms as directed acyclic graphs. Every subterm is stored at most once, implementing a maximal sharing discipline. The ATerm library also provides automatic garbage collection, and ATermTables, which represent a finite function from ATerm $\to$ ATerm by means of a hash table.

We implemented the almost linear unification algorithm from Ref. [4], which is based on Ref. [23]. It is based on 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". Intermediate cyclic terms are represented as a combination of an ATerm and an ATermTable. For instance, the ATerm $f(a, g(x))$ in combination with the ATermTable $[x \mapsto f(a, g(x))]$ represents a cyclic term.

Clauses and CNFs are implemented naively as (unidirected) linked lists. We did no attempt to implement any form of subsumption. Also, we have not yet implemented heuristics for choosing a good splitting variable (actually we choose the last literal of the first purely positive clause encountered). Note that unit resolution is built-in in the reduction rules. We use the following strategy for reduction: rules 1, 2 and 3 are always immediately applied. Furthermore, rule 5 has priority over rule 4, as we believe that this order enables longer sequences of unit resolution, possibly cutting down the size of the search tree.

An implementation in C of this algorithm can be found at http://www.cwi.nl/~ vdpol/gdpll.html.

7.2. Description of formulas

As benchmarks, we used some purely equational formulas (`phe`, `circ`) and some formulas with function symbols (`succ`, `evod`). First these formulas will be described.

7.2.1. `phe` *– equational pigeon hole*

Variables: $x_1, \ldots, x_N, y$.

$$\left(\bigwedge_{1 \leqslant i < j \leqslant N} x_i \neq x_j \right) \wedge \left(\bigwedge_{1 \leqslant i \leqslant N} \bigvee_{1 \leqslant j \leqslant N, j \neq i} x_j = y \right).$$

Intuitively, the first conjunct 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 Ref. [35].

7.2.2. `circ` *– a ring of equations*

Variables: $x_1, \ldots, x_N$. Imagine they are on a ring; we will write x_{N+1} to denote syntactically the same variable as x_1.

$$\left(\bigvee_{1 \leqslant i \leqslant N} x_i \neq x_{i+1} \right) \wedge \left(\bigwedge_{1 \leqslant i < j \leqslant N} \left(x_i = x_{i+1} \vee x_j = x_{j+1} \right) \right).$$

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.

7.2.3. `succ` *– natural numbers with equality*

Variables: $x_1, \dots, x_N$; unary constant S. Imagine they are on a ring; we will write x_{N+1} to denote syntactically the same variable as x_1.

$$\left(\bigwedge_{1 \leqslant i < j \leqslant N} (x_i = S(x_{i+1}) \vee x_j = S(x_{j+1})) \right) \wedge \bigvee_{1 \leqslant i \leqslant N} x_i = x_{i+1}.$$

Here the first part expresses 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, $x_k = x_{k+1}$.

7.2.4. `evod` *– even and odd natural numbers*

Variables: $x_1, \dots, x_N$; unary constant S.

$$x_1 = x_N \wedge \bigwedge_{1 \leqslant i < N} (x_i = S(x_{i+1}) \vee S(x_i) = x_{i+1}).$$

Note that this formula implies that either x_i is odd iff i is odd, or x_i is even iff i is even. This formula is satisfiable when N is odd, unsatisfiable when N is even.

7.3. Performance results

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;—means more than 600 s), and the number of recursive calls to the GDPLL procedure. We compared two approaches. The last columns indicate the algorithm with full unit resolution (i.e., with rules 4 and 5 of Definition 22). In the other two columns we omitted unit resolution, reverting to a definition of Filter similar to Section 5.2.

For the instances `phe`, `circ` and `succ`, it appears 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`. Instead of by observing the table this information can be obtained by an analytic argument. Still, 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 four times more work. Here unit resolution roughly halves the number of calls to GDPLL, but overall it costs a little more time.

In Ref. [35] some experiments on the same `phe` formula type are given. Several encodings to propositional logic are tried. The best result was that `phe` with $N = 60$ takes 11 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 s for $N = 60$, which is about 50 times better than the best method from [35], on a machine which is at most 2.5 times faster.

Type	*N*	nr of literals	Without UR		With UR	
			Time (s)	nr of calls	Time (s)	nr of calls
`phe`	40	2340	0	1639	0	77
	80	9480	8	6479	0	157
	120	21,420	52	14,519	2	237
	160	38,160	168	25,759	4	317
	200	59,700	433	40,199	10	397
`circ`	100	10,000	3	10,097	0	199
	200	40,000	50	40,197	3	399
	300	90,000	258	90,297	9	599
	400	160,000	–	–	22	799
	500	250,000	–	–	43	999
`succ`	50	2500	6	2741	1	2449
	100	10,000	92	10,491	6	9899
	150	22,500	459	23,241	20	22,349
	200	40,000	–	–	48	39,799
	250	62,500	–	–	103	62,249
`evod`	12	23	0	6763	0	3171
	14	27	2	27,487	2	12,951
	16	31	6	111,337	9	52,665
	18	35	25	449,927	31	213,523
	20	39	100	1,815,155	104	863,819
	22	43	407	7,313,663	505	3,488,871

8. 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 conditions on these four abstract procedures for which we proved 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. In fact even the notion of CNF is not essential for our framework. However, since all applications we have in mind are settings of CNFs, we started by presenting a general framework for CNFs in fragments of first-order logic.

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. The resulting algorithm can be extended easily 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 has been worked out in Ref. [33,34]. The addition of other interpreted functions (such as $+$ or append) or predicates (like $>$) is subject to future research.

References

[1] W. Ackermann, Solvable Cases of the Decision Problem. Studies in Logic and the Foundations of Mathematics, North Holland, Amsterdam, 1954.

[2] A. Armando, C. Castellini, E. Giunchiglia, F. Giunchiglia, A. Tacchella, SAT-based decision procedures for automated reasoning: a unifying perspective, IRST Technical Report 0202-05, Istituto Trentino di Cultura, February 2002.

[3] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, R.A. Sebastiani, SAT based approach for solving formulas over boolean and linear mathematical propositions, in: A. Voronkov (Ed.), Automated Deduction—CADE-18: 18th International Conference on Automated Deduction, Lecture Notes in Computer Science, vol. 2393, Springer-Verlag, 2002, pp. 195–210.

[4] F. Baader, T. Nipkow, Term Rewriting and All That, Cambridge University Press, New York, 1998.

[5] B. Badban, Verification Techniques for Extensions of Equality Logic, Ph.D. Thesis, Free University, Amsterdam, 2006.

[6] B. Badban, J. van de Pol, O. Tveretina, H. Zantema, Solving satisfiability of ground term algebras using DPLL and unification, in: Workshop on Unification, Cork, Ireland, July 2004.

[7] B. Badban, J. van de Pol, Zero, successor and equality in binary decision diagrams, Annals of Pure and Applied Logic 133 (1–3) (2005) 101–123.

[8] C. Barrett, D. Dill, A. Stump, A framework for cooperating decision procedures, in: D.A. McAllester (Ed.), Proceedings of the 17th International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence, vol. 1831, Springer, Pittsburgh, PA, 2000, pp. 79–98.

[9] P. Baumgartner, FDPLL—a First-Order Davis–Putnam–Logeman–Loveland Procedure, in: D. McAllester (Ed.), Proceedings of the 17th Conference on Automated Deduction, Lecture Notes in Artificial Intelligence, vol. 1831, Springer-Verlag, 2000, pp. 200–219.

[10] S. Blom, W. Fokkink, J. Groote, I. van Langevelde, B. Lisser, J. van de Pol, μCRL: a toolset for analysing algebraic specifications, in: G. Berry, H. Comon, A. Finkel (Eds.), Proceedings of the 13th Conference on Computer Aided Verification, Lecture Notes in Computer Science, vol. 2102, Springer-Verlag, 2001, pp. 250–254.

[11] M. Brand, H.d. Jong, P. Klint, P. Olivier, Efficient annotated terms, Software—Practice & Experience 30 (2000) 259–291.

[12] R. Bryant, S. German, M. Velev, Exploiting positive equality in a logic of equality with uninterpreted functions, in: N. Halbwachs, D. Peled (Eds.), Proceedings of the 11th Conference on Computer Aided Verification, Lecture Notes in Computer Science, vol. 1633, Springer-Verlag, 1999, pp. 470–482.

[13] A. Colmerauer, Equations and inequations on finite and infinite trees, in: I. Staff (Ed.), Proceedings of the Conference on Fifth Generation Computer Systems, North Holland, 1984, pp. 85–99.

[14] H. Comon, P. Lescanne, Equational problems and disunification, Journal of Symbolic Computation 7 (3–4) (1989) 371–425.

[15] M. Davis, G. Logemann, D. Loveland, A machine program for theorem proving, Communications of the ACM 5 (7) (1962) 394–397.

[16] M. Davis, H. Putnam, A computing procedure for quantification theory, Journal of the Association for Computing Machinery 7 (3) (1960) 201–215.

[17] J.-C. Filliâtre, S. Owre, H. Ruess, N. Shankar, ICS: integrated canonizer and solver, in: G. Berry, H. Comon, A. Finkel (Eds.), Proceedings of the 13th Conference on Computer Aided Verification, Lecture Notes in Computer Science, vol. 2102, Springer-Verlag, 2001, pp. 246–249.

[18] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, C. Tinelli, DPLL(T): fast decision procedures, in: R. Alur, D. Peled (Eds.), Proceedings of the 16th Conference on Computer Aided Verification, Lecture Notes in Computer Science, vol. 3114, 2004, pp. 175–188.

[19] F. Giunchiglia, R. Sebastiani, Building decision procedures for modal logics from propositional decision procedures: the case study of modal K, in: M.A. McRobbie J.K. Slaney (Eds.), Proceedings of the 13th International Conference on Automated Deduction (CADE-96), Lecture Notes in Artificial Intelligence, vol. 1104, Springer-Verlag, 1996, pp. 583–597.

[20] A. Goel, K. Sajid, H. Zhou, A. Aziz, BDD based procedures for a theory of equality with uninterpreted functions, in: A.J. Hu, M. Vardi (Eds.), Proceedings of the 10th Conference on Computer Aided Verification, Lecture Notes in Computer Science, vol. 1427, Springer-Verlag, 1998, pp. 244–255.

[21] J. Groote, M. Reniers, Algebraic process verification, in: J. Bergstra, A. Ponse, S. Smolka (Eds.), Handbook of Process Algebra, Elsevier, NewYork, 2001, ch. 17.

[22] J. Groote, J. van de Pol, Equational binary decision diagrams, in: M. Parigot, A. Voronkov (Eds.), Proceedings of the 7th Conference on Logic for Programming and Automated Reasoning, Lecture Notes in Artificial Intelligence, vol. 1955, Springer-Verlag, 2000 , pp. 161–178.

[23] G. Huet, Résolution d'équations dans les languages d'ordre 1, 2, ... , ω, Ph.D. Thesis, Université Paris 7, 1976.

[24] J.R. Burch, D.L. Dill, Automatic verification of pipelined microprocessors control, in: D.L. Dill (Ed.), Proceedings of the 6th Conference on Computer Aided Verification, Lecture Notes in Computer Science, vol. 818, Springer-Verlag, 1994, pp. 68–80.

[25] J.-L. Lassez, M.J. Maher, K. Marriott, Unification revisited, in: J. Minker (Ed.), Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann Publishers, Los Altos, CA, 1987, pp. 587–625.

[26] M. Maher, Complete axiomatizations of the algebras of finite, rational and infinite trees, in: Proceedings of the 3rd Annual Symposium on Logic in Computer Science, IEEE Computer Society, 1988, pp. 348–357.

[27] R. Nieuwenhuis, A. Oliveras, Congruence closure with integer offsets, in: M.Y. Vardi, A. Voronkov (Eds.), Proceedings of the 10th International Conference on Logics for Programming AI and Reasoning (LPAR), Lecture Notes in Artificial Intelligence, vol. 2850, Springer-Verlag, 2003, pp. 78–90.

[28] R. Pichler, On the complexity of equational problems in CNF, Journal of Symbolic Computation 36 (2003) 235–269.

[29] A. Pnueli, Y. Rodeh, O. Shtrichman, M. Siegel, Deciding equality formulas by small domains instantiations, in: N. Halbwachs, D. Peled (Eds.), Proceedings of the 11th Conference on Computer Aided Verification, Lecture Notes in Computer Science, vol. 1633, Springer-Verlag, 1999, pp. 455–469.

[30] J. Robinson, A machine-oriented logic based on the resolution principle, Journal of the ACM 12 (1) (1965) 23–49.

[31] N. Shankar, H. Ruess, Combining Shostak theories, in: S. Tison (Ed.), Proceedings of the 13th Conference on Rewriting Techniques and Applications, Lecture Notes in Computer Science, vol. 2378, Springer-Verlag, 2002, pp. 1–18.

[32] A. Stump, C. Barrett, D. Dill, CVC: a cooperating validity checker, in: J. Godskesen (Ed.), Proceedings of the 14th Conference on Computer Aided Verification, Lecture Notes in Computer Science, vol. 2404, Springer-Verlag, 2002, pp. 500–505.

[33] O. Tveretina, A decision procedure for equality logic with uninterpreted functions, in: B. Buchberger, J.A. Campbell (Eds.), Artificial Intelligence and Symbolic Mathematical Computation, Lecture Notes in Artificial Intelligence, vol. 3249, Springer-Verlag, 2004, pp. 63–76.

[34] O. Tveretina, Decision Procedures for Equality Logic with Uninterpreted Functions, Ph.D. Thesis, Technical University of Eindhoven, 2005.

[35] H. Zantema, J.F. Groote, Transforming equality logic to propositional logic, in Proceedings of the 4th International Workshop on First-Order Theorem Proving (FTP'03), Electronic Notes in Theoretical Computer Science, 86(1) (2003) 1–12.