Centrum voor Wiskunde en Informatica

REPORTRAPPORT

SEN

Software Engineering

*Software ENgineering*

# Generalizing DPLL and Satisfiability for Equalities

B. Badban, J.C. van de Pol, O. Tveretina, H. Zantema

REPORT SEN-R0407 JUNE 2004

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

# Generalizing DPLL and Satisfiability for Equalities

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 properties 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 abstract datatypes. We provide some benchmarks.

*2000 Mathematics Subject Classification:* 68T15
*1998 ACM Computing Classification System:* I.2.3
*Keywords and Phrases:* satisfiability, DPLL procedure, equality, ground term algebra, abstract datatypes, decision procedure
*Note*: This research was carried out in the NWO-project IT-VDS

# Generalizing DPLL and Satisfiability for Equalities

Bahareh Badban[1], Jaco van de Pol[1,2], 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.** We present GDPLL, a generalization of the DPLL procedure. It solves the satisfiability problem for decidable fragments of quantifier-free first-order logic. Sufficient properties 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 abstract datatypes. We provide some benchmarks.

*2000 Mathematics Subject Classification:* 68T15 Theorem proving
*1998 ACM Computing Classification System:* I.2.3 Deduction and Theorem Proving
*Keywords and Phrases:* satisfiability, DPLL procedure, equality, ground term algebra, abstract datatypes, decision procedure.
*Note:* This research was carried out in the NWO-project IT-VDS.

## 1 Introduction

### 1.1 Contribution

In this paper we provide a generalization of the well known DPLL procedure, named after Davis-Putnam-Logemann-Loveland. DPLL [DP60,DLL62] 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. 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 abstract datatypes, or inductive datatypes). An instance of a formula in this logic would be:

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

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 formulae 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. It 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`.

### 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 [Bur94] and software [PRSS99] 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 $\mu$CRL [GR01,BFG$^+$01], a language based on abstract data types and process algebra.

At this moment, our algorithm works for constructor symbols only (such as zero, successor, nil and cons). We think that it can be extended rather easily to standard destructors (such as predecessor, head and tail) and recognizer predicates (such as nil?, succ?, cons?, zero?), because these functions can be eliminated by introducing new variables. 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 [Ack54].

### 1.3 Related Work

Our tool is comparable to ICS [SR02,FORS01] (which is used in PVS) and CVC [BDS00,SBD02], 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 abstract datatypes. In ICS abstract datatypes are specified as a combination of products and coproducts; in CVC abstract data types can be defined inductively. However, both tools are incomplete for quantifier free logic over 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 is based on equational BDDs [GP00]. A complete algorithm for BDDs with equations, zero and successor is treated in [BP04].

The recent algorithm of [NO03] decides the theory of uninterpreted function symbols. 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: in GDPLL all decisions depend on the current CNF only, while in the algorithm of [NO03] some decisions depend on the consistency of the conjunction of all choices made in the history. Also the ICS and CVC algorithms use a context of previously asserted formulas.

In the past several years various approaches based on the DPLL procedure have been proposed [GS96,ABC+02,ACG+02,GHN+04]. MathSAT [ABC+02] 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 [Bau00] 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.

Other approaches encode the satisfiability question for a particular theory to plain propositional logic. For the logic of equality and uninterpreted function symbols, one can use Ackermann's reduction [Ack54,BGV99] to eliminate the function symbols. This yields a formula with equalities between variables only. Their solution 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 [GSZA98,BGV99]. Several encodings are compared in [ZG03].

Our particular solution for ground term algebras depends on well known unification theory. Ground breaking work in this area was done by [Rob65]. We follow the almost linear implementation of [Hue76]. Unification solves conjunctions of equations in the ground term algebra. Colmerauer [Col84] 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 [LMM87] and for a textbook on unification (theory and algorithms) we recommend [BN98]. The full first-order theory of equality in ground term algebras is studied in [Mah88,CL89] (both focus on a complete set of rewrite rules) and more recently by [Pic03] (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 [CL89,Mah88].

### 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 $\mathsf{Var}$, $\mathsf{Fun}$, $\mathsf{Pr}$ are pairwise disjoint.

The set $\mathsf{Term}(\Sigma, \mathsf{Var})$ of *terms* over the signature $\Sigma$ 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,
- $f(t_1, \dots, t_n)$ is a term if $t_1, \dots, t_n$ 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 $p(t_1, \dots, t_n)$, where the $t_i$ are terms, and $p$ is a predicate symbol of arity $n$. The set of atoms over the signature $\Sigma$ is denoted by $\mathsf{At}(\Sigma, \mathsf{Var})$ or for simplicity by $\mathsf{At}$ .

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 $\Sigma$ is denoted by $\mathsf{Lit}(\Sigma, \mathsf{Var})$ or if it is not relevant by $\mathsf{Lit}$. We denote by $\mathsf{Lit}_p$ and $\mathsf{Lit}_n$ 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. We denote by $\mathsf{Cls}$ the set of all clauses. 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 $\mathsf{Cnf}$ 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 $\phi$ and literal $l \in \mathsf{Lit}$, let*

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

*Finally, we say that $C \in \mathsf{Cnf}$ is purely positive clause if $l \in \mathsf{Lit}_p$ for all $l \in C$.*

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

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

The interpretation $[\![l]\!]^\sigma_\mathcal{D} : \mathsf{Lit} \to \{\mathsf{false}, \mathsf{true}\}$ of an atom $p(t_1, \ldots, t_n)$ is defined by:

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

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

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

The interpretation $[\![C]\!]^\sigma_\mathcal{D} : \mathsf{Cls} \to \{\mathsf{false}, \mathsf{true}\}$ of a clause $C = \{l_1, \ldots, l_m\}$ is defined by:

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

The interpretation $[\![\phi]\!]^\sigma_\mathcal{D} : \mathsf{Cnf} \to \{\mathsf{false}, \mathsf{true}\}$ of a CNF $\phi = \{C_1, \ldots, C_r\}$ is defined by:

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

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 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 3.** *An assignment $\sigma : \mathsf{Var} \to D$* satisfies *a CNF $\phi$ in a structure $\mathcal{D}$, if $[\![\phi]\!]^\sigma_\mathcal{D} = \mathsf{true}$. CNF $\phi$ is called* satisfiable *if it is satisfied by some assignment in some admissible structure. Otherwise $\phi$ 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 $\sigma$ is an assignment which satisfies the literal $l$. Then given a formula $\phi$, $\sigma$ satisfies $\phi$ if and only if $\sigma$ satisfies $\phi|_l$.*

*Proof.* We prove each side separately:

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

## 3 Instances

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

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.

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

### 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 $\mathsf{Pr}$ 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

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 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 $\Sigma = \{\mathsf{Fun}, \mathsf{Pr}\}$, 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 $\approx_D = Id_D$.

As an example we consider

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

Assume $\phi$ 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 $\phi$ 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 [ZG03].

In this basic version of equality logic there are no function symbols. In the next two subsections we discuss two ways 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, or 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 $\mathsf{Fun}$ is an arbitrary set of function symbols and $\mathsf{Pr}$ 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. In this paper, we present the general version of the procedure, and we adopt it for some fragments of first order logic. The satisfiability problem is decidable in these logics.

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 $\phi$ 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 $\mathsf{Rcnf} = \{\phi \in \mathsf{Reduce}(\mathsf{Cnf}) \mid \bot \notin \phi\}$.

In the following 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 $\psi$ is satisfiable,
2. $\phi$ 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 $\phi$ is satisfiable,
5. if $\mathsf{SatCriterion}(\phi) = \mathsf{false}$ then $\mathsf{Eligible}(\phi) \neq \emptyset$.

Figure 1 shows 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;
```

**Fig. 1.** The GDPLL procedure

### 4.1 Soundness and Completeness of GDPLL

**Theorem 5.** (*soundness and completeness*) *Let* $\phi \in \mathsf{Cnf}$. *Then the following properties hold:*

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

*Proof.* Let $\phi \in \mathsf{Cnf}$. We apply induction on $\prec$, which is well-founded by property 3. So assume (induction hypothesis) that the theorem holds for all $\psi$ such that $\mathsf{Reduce}(\psi) \prec \mathsf{Reduce}(\phi)$. By property 1, $\mathsf{Reduce}(\phi)$ is satisfiable if $\phi$ is satisfiable, and $\mathsf{Reduce}(\phi)$ is unsatisfiable if $\phi$ is unsatisfiable.

Let $\perp \in \mathsf{Reduce}(\phi)$. Then trivially $\phi$ is unsatisfiable, and $\mathsf{GDPLL}(\phi)$ returns UNSAT.

Let $\perp \notin \mathsf{Reduce}(\phi)$. Assume that for all $\psi$ such that $\mathsf{Reduce}(\psi) \prec \mathsf{Reduce}(\phi)$, $\mathsf{GDPLL}(\psi)$ returns UNSAT if $\psi$ is unsatisfiable, and $\mathsf{GDPLL}(\psi)$ returns SAT if $\psi$ is satisfiable.

If $\mathsf{SatCriterion}(\mathsf{Reduce}(\phi)) = \mathsf{true}$ then by property 4, $\phi$ is satisfiable, and $\mathsf{GDPLL}(\phi) = \mathsf{SAT}$.

If $\mathsf{SatCriterion}(\mathsf{Reduce}(\phi)) = \mathsf{false}$ then by property 5, $\mathsf{Eligible}(\mathsf{Reduce}(\phi)) \neq \emptyset$.

By property 3, for all $\phi \in \mathsf{Cnf}$ and all $a \in \mathsf{Eligible}(\phi)$,

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

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

Let $\mathsf{Reduce}(\phi)$ be unsatisfiable. Then by property 2, $\mathsf{Filter}(\mathsf{Reduce}(\phi), a)$ and $\mathsf{Filter}(\mathsf{Reduce}(\phi), \neg a)$ are unsatisfiable. We can apply induction hypothesis. Then both $\mathsf{GDPLL}(\mathsf{Filter}(\mathsf{Reduce}(\phi), a))$ and $\mathsf{GDPLL}(\mathsf{Filter}(\mathsf{Reduce}(\phi), \neg a))$ return UNSAT. By definition of GDPLL, $\mathsf{GDPLL}(\phi)$ also returns UNSAT.

Let $\mathsf{Reduce}(\phi)$ be satisfiable. By property 2, at least one of $\mathsf{Filter}(\mathsf{Reduce}(\phi), a)$ and $\mathsf{Filter}(\mathsf{Reduce}(\phi), \neg a)$ is unsatisfiable. By induction hypothesis, at least one of $\mathsf{GDPLL}(\mathsf{Filter}(\mathsf{Reduce}(\phi), a))$ and $\mathsf{GDPLL}(\mathsf{Filter}(\mathsf{Reduce}(\phi), \neg a))$ return SAT, and by definition of GDPLL, $\mathsf{GDPLL}(\phi)$ also returns SAT. □

## 5 Instances for the GDPLL procedure

In this section we will define the functions Eligible, Filter, Reduce and SatCriterion for the simple instances mentioned above. The procedure for ground term algebras is dealt with in a separate section.

### 5.1 GDPLL for Propositional Logic

**Definition 6.** *We say that $l$ is a* pure literal *in $\phi$ if $\neg l \notin \mathsf{Lit}(\phi)$. $C$ in $\phi$ is a* unit clause *if $\#(C) = 1$.*

Two main operations of the DPLL procedure are *unit propagation* and *purification*. Unit clauses can only be satisfied by a specific assignment to the corresponding propositional variable, and the complementary assignment will lead to contradiction. Hence all occurrences of this variable can be eliminated. Elimination of the variable can create a new unit clause, so this process has to be repeated until no unit clauses are left. Purification can be applied if the formula contains pure literals. Such literals can be eliminated by assigning true in the positive case and false in the negative case. Note that this cannot introduce 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

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

Figure 2 shows the function Reduce. The function $\mathsf{UnitClause}(\psi)$ returns a unit clause contained in $\mathsf{Cls}(\psi)$, and the function $\mathsf{PureLiteral}(\psi)$ returns a pure literal contained in $\mathsf{Lit}(\psi)$.

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

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

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

**Fig. 2.** The Reduce function for propositional logic

**Definition 7.** (*ordering on formulas*) *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 8.* 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 9.** *The functions* Reduce*,* Eligible*,* Filter*,* SatCriterion *satisfy the Properties 1–5.*

*Proof.*
1. Properties 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 7 one can see that from

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

   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 Properties 4. By definition, the function $\mathsf{SatCriterion}(\phi)$ returns $\mathsf{true}$ 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 $\mathsf{SatCriterion}$ 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 $\mathsf{Eligible}$, $\mathsf{Reduce}$, $\mathsf{Filter}$, and $\mathsf{SatCriterion}$. One can see that $\mathsf{GDPLL}$ now coincides with the DPLL procedure for propositional logic.

In the situation when $\phi$ consists of relatively few clauses comparing to the number of variables in each clause splitting can be very inefficient. The following theorem allows the procedure to stop when every clause in $\phi$ contains at least one negative literal.

**Theorem 10.** (SAT criterion) *Let $\phi \in \mathsf{Cnf}$ contain no purely positive clause. Then $\phi$ is satisfiable.*

*Proof.* For propositional logic the domain can be an arbitrary set. Let some set $D$ be a domain. Since no term contains variables any assignment plays no role, and we can assume an arbitrary assignment $\sigma$. 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 $[\![\phi]\!]^{\sigma}_{\mathcal{D}} = \mathsf{true}$. □

*Example 11.* Consider

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

One can see easily that the formula is satisfiable.

Using the above theorem we can define the function $\mathsf{SatCriterion}$.

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

One can easily check that the function $\mathsf{SatCriterion}$ satisfies Properties 4 and 5. In following sections we will define the function $\mathsf{SatCriterion}$ in a similar way.

### 5.2 GDPLL for Equality Logic

We now define the functions $\mathsf{Eligible}$, $\mathsf{Filter}$, $\mathsf{Reduce}$ and $\mathsf{SatCriterion}$ for equality logic. The function $\mathsf{Reduce}$ 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 $\mathsf{GDPLL}$ 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. $\mathsf{Eligible}(\phi) = \mathsf{Lit}_p(\phi)$.

*Example 12.* 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 $\phi$ 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 13.* 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 $\sigma$ such that $\sigma(x') \neq \sigma(x'')$ for all $x', x'' \in \mathsf{Lit}_n(\phi)$.

We denote by $\phi[x := y]$ the formula $\phi$, 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 14.** (*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 15.* 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 16.** *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 $a \in \mathsf{At}$, $\phi$ 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 $\psi|_{(x\approx y)} \in \mathsf{Cnf}$
   $$\#\mathsf{Lit}_p(\psi|_{x\approx y}[x := y]) \leq \#\mathsf{Lit}_p(\psi|_{x\approx y}).$$
   We obtain that
   $$\#\mathsf{Lit}_p(\phi|_{x\approx y}[x := y] \leq \#\mathsf{Lit}_p(\phi|_{x\approx y}) < \#\mathsf{Lit}_p(\phi).$$
   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 $\mathsf{Reduce}$, 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 $\phi$ contains at least one negative literal. If $\phi = \emptyset$ then by definition $\phi$ is satisfiable. Let us consider the remaining case. Let $D$ be a domain such that $\#D \geq |\mathsf{Var}(\phi)|$. We choose an assignment $\sigma$ 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 $\mathsf{Eligible}(\phi) \neq \emptyset$.

□

Many variations of this instance for 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 abstract 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.

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

**Lemma 17.** *In every ground term algebra $\mathcal{D}$ for $\Sigma$, 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 [LMM87,BN98].

**Definition 18.** *A **substitution** is a function $\sigma : \mathsf{Var} \to \mathsf{Term}(\Sigma, \mathsf{Var})$ such that $\sigma(x) \neq x$ for only finitely many $x$s. 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 $\sigma$ as*

$$\sigma = \{x_1 \mapsto \sigma(x_1), \ldots, x_n \mapsto \sigma(x_n)\}.$$

*The variable range of $\sigma$ 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 as follows:

**Definition 19.** *We define an application of substitution* $(.)^\sigma$ *as below:*

$$\begin{aligned} x^\sigma &= \sigma(x) \\ f(t_1, ...t_n)^\sigma &= f(t_1^\sigma, ...t_n^\sigma) \\ (t \approx u)^\sigma &= t^\sigma \approx u^\sigma \qquad\qquad (\textit{likewise for its negation}) \\ \{l_1, ...l_n\}^\sigma &= \{l_1^\sigma, ...l_n^\sigma\} \\ \{C_1, ...C_n\}^\sigma &= \{C_1^\sigma, ...C_n^\sigma\} \end{aligned}$$

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

**Definition 20.** *The* **composition** $\sigma\rho$ *of substitutions* $\sigma$ *and* $\rho$ *is defined such that* $\sigma\rho(x) = \sigma(\rho(x))$. *A substitution* $\sigma$ *is* ***more general*** *than a substitution* $\sigma'$ *if there is a substitution* $\delta$ *such that* $\sigma' = \delta\sigma$. *We write* $\sigma \lesssim \sigma'$. *Furthermore, a substitution* $\sigma$ *is* ***idempotent*** *if* $\sigma\sigma = \sigma$.

**Definition 21.** *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* $\sigma$ *such that* $s_i^\sigma = t_i^\sigma$ *for* $i = 1, ..., n$.
*A substitution* $\sigma$ *is a most general unifier of* $S$ *or in short* $\mathsf{mgu}(S)$, *if*

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

**Definition 22.** *An atom* $t \approx u$ *is in* ***solved form*** *if it is of the form:*

$$x \approx u, \textit{ where } x \notin u$$

*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. [LMM87,BN98]):

**Lemma 23.**

1. *A substitution* $\sigma$ *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* $\sigma$ *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 $\mathsf{mgu}(S) = \perp$ if $S$ has no unifier.
- When working on sets of unit clauses, by $\sigma = \mathsf{mgu}(\{t_1 \approx u_1\}, ..., \{t_n \approx u_n\})$ we mean $\sigma = \mathsf{mgu}(\{t_1 \approx u_1, ..., t_n \approx u_n\})$.
- 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 $\sigma = \{x_1 \mapsto t_1, ..., x_n \mapsto t_n\}$ then $x_i \notin \mathsf{Var}(t_j)$ for all $1 \leq i, j \leq n$. Another consequence is that $\mathsf{mgu}(x \approx x) = \emptyset$.

### 6.2 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 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 24.** *We consider the following reduction rules, which should be applied repeatedly until $\phi$ 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 $\phi_1 = \{C \mid C \in \phi \text{ is a positive unit clause}\} \neq \emptyset$, 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* Reduce$(\phi)$ *to be any normal form of $\phi$ 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 25.* $\phi = \{\{f(f(y)) \not\approx f(x)\}, \{x \not\approx x\}\}$. 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 $\perp$, therefore regarding the rule 2 we get:

$$\phi \longrightarrow \{\perp\}.$$

*Example 26.* The formula $\phi$ below is reduced, since no rewrite rule of Definition 24 is applicable on it:

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

**Corollary 27.** *Suppose $\phi$ is a reduced formula, then the following requirements will hold:*

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

*Proof.* If $\phi$ doesn't 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 28.* 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 $\phi$ will be obtained:

1. One approach:

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

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

2. Another approach:

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

   Which is reduced regarding the rewrite system of Definition 24.

### 6.3 Termination

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

**Definition 29.** *We define the following measures on formulas:*

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

*To each formula $\phi$, 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 $\phi$.*

**Theorem 30.**

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

*Proof.* – We prove termination, by showing that after applying each step of the reduction system on a supposed formula, $\mathsf{norm}$ 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 $\mathsf{pos}(\phi') \leq \mathsf{pos}(\phi)$.

3. 
   - if $\sigma = \bot$ then $\#\phi' {=} \#\phi - 1$ and $\mathsf{pos}(\phi') \leq \mathsf{pos}(\phi)$.
   - 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 $\phi_1$ is the non-empty set of the positive unit literals in $\phi$.
   - if $\sigma = \bot$ then $\#\phi' = \#\{\bot\} = 1 \leq \#\phi_1 \leq \#\phi$ and $\mathsf{pos}(\phi') = \mathsf{pos}(\bot) < 1 \leq \mathsf{pos}(\phi)$.
   - otherwise $\#{\phi_2}^{\sigma} = \#\phi_2 < \#\phi_1 + \#\phi_2 \leq \#\phi$ and $\mathsf{pos}({\phi_2}^{\sigma}) = \mathsf{pos}(\phi_2) \leq \mathsf{pos}(\phi)$.
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

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

     We also have $\#\phi' \leq \#\phi$.
   - otherwise $\neg a \in \mathsf{Lit}(\phi_1)$ and hence

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

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

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

□

**Theorem 31.** $\mathsf{pos}(\mathsf{Reduce}(\phi \wedge l)) < \mathsf{pos}(\phi)$ *for any reduced formula* $\phi$ *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 $\phi$ is reduced, the first step to reduce $\phi \wedge l$, regarding the Definition 24, will be one of the rules 4 or 5, as the followings:

– If $l = t \approx u$, then

$$\begin{aligned} \phi \wedge l &= \phi \wedge t \approx u \\ &= \phi \uplus \{\{t \approx u\}\} && \phi \text{ is reduced and Corollary 27(4)} \\ &\longrightarrow \phi^\sigma && \text{Definition 24(4) and } \mathsf{Reduce}(\phi \wedge l) \neq \{\bot\} \end{aligned}$$

$t \approx u \in \mathsf{Lit}_p(\phi)$, 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 $\phi^\sigma = \phi_0 \rightarrow \phi_1 \rightarrow \ldots \rightarrow \phi_{n+1} = \mathsf{Reduce}(\phi^\sigma)$ is the reduction sequence by which we obtain $\mathsf{Reduce}(\phi^\sigma)$ from $\phi^\sigma$.
Applying any rule of the Definition 24 on $\phi_0$, $t^\sigma \approx t^\sigma$ will be either removed or replaced by a similar one $t^\rho \approx t^\rho$. Regarding the Corollary 27(1), $\phi_{n+1}$ does not contain any literal of the shape $w \approx w$.
Since $\phi_0$ contains at least one literal of that shape($t^\sigma \approx t^\sigma$), therefore there exists a $0 \leq j \leq n+1$ such that $\phi_j$ has a literal of the form $w \approx w$, and $\phi_{j+1}$ does not have any. Now since according to the Theorem 30(2), the number of occurrences of the positive literals does not increase during the reduction process, therefore $\mathsf{pos}(\phi_j) \leq \mathsf{pos}(\phi_{j+1}) - 1$. Hence $\mathsf{pos}(\phi_0) < \mathsf{pos}(\phi_{n+1})$, again regarding the Theorem 30(2).

– If $l = t \not\approx u$, then

$$\begin{aligned} \phi \wedge l &= \{\{t \not\approx u\}\} \uplus \phi \\ &\longrightarrow \{\{t \not\approx u\}\} \uplus \phi|_{t \not\approx u} && \text{Definition 24(5), } t \approx u \in \mathsf{Lit}_p(\phi) \end{aligned}$$

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

$$\begin{aligned} \mathsf{pos}(\mathsf{Reduce}(\phi \wedge l)) &\leq \mathsf{pos}(\{\{t \not\approx u\}\} \uplus \phi|_{t \not\approx u}) && \text{Theorem 30(2)} \\ &\leq \mathsf{pos}(\phi) - 1 \end{aligned}$$

□

### 6.4 Correctness properties of the building blocks

**Theorem 32.** *(Reduced Criteria) Given a ground term algebra* $\mathcal{D}$ *and a formula* $\phi$ *in it,* $\phi$ *is satisfiable if and only if* $\mathsf{Reduce}(\phi)$ *is satisfiable.*

*Proof.* We check in any step of the reduction that $\phi$ is satisfiable if and only if the result is satisfiable. So assume that $\phi \rightarrow \phi'$; we now distinguish which rule of Definition 24 is applied:

1. It is even obvious that $\alpha$ satisfies $\phi$ if and only if $\alpha$ satisfies $\phi'$, for each assignment $\alpha$.
2. Both are unsatisfiable.
3. (a) If $\alpha$ satisfies $\phi$ then in the first case obviously $\alpha$ satisfies $\phi'$, which is $\phi - \{C\}$. In the second case also $\alpha$ satisfies $\phi'$ because $t \not\approx u$ is replaced by the negation of its unifier, which is equivalent by Lemma 23.(3).
   (b) If $\alpha$ satisfies $\phi'$, in which either $\phi' = \phi - \{C\}$ and $t \not\approx u \in C \in \phi$ is a tautology, or $\phi'$ is obtained from $\phi$ by replacing $t \not\approx u$ with $\neg\mathsf{mgu}(t \approx u)$, see Lemma 23.(3). In any case $\alpha$ satisfies $\phi$ obviously.
4. Let $\phi_1$ be the non-empty set of positive unit clauses, and $\phi = \phi_1 \uplus \phi_2$.
   (a) If $\phi' = \{\bot\}$ then $\phi'$ is unsatisfiable, also $\phi$ is unsatisfiable since $\phi_1$ has no unifier.
   (b) If $\alpha$ satisfies $\phi$ then it satisfies ${\phi_2}^\sigma$ trivially. Now if $\alpha$ satisfies ${\phi_2}^\sigma$ then define:
$$\alpha'(y) = \begin{cases} \alpha(y) & \text{if } y \in \mathsf{Var}({\phi_2}^\sigma) \\ \alpha(\sigma(y)) & \text{otherwise} \end{cases}$$
   $\alpha'$ satisfies $\phi$.
5. Is obvious regarding Lemma 4.

□

**Definition 33.** *Given a term $t$ we define $S(t)$ to be the number of occurrences of non-constant function symbols in $t$:*

$$\begin{aligned} S(x) &= 0 \\ S(c) &= 0 && \textit{if } c \textit{ is a constant symbol} \\ S(f(t_1, ...t_n)) &= 1 + \sum_{i=1}^{n} S(t_i) && \textit{if } n \geq 1 \\ S(t \not\approx u) &= 0 \\ S(t \approx u) &= S(t) + S(u) \end{aligned}$$

**Theorem 34.** *(SAT criteria) Suppose $\mathcal{D}$ is a ground term algebra with infinitely many closed terms, then a reduced formula $\phi$ is satisfiable if $\phi$ has no purely positive clause.*

*Proof.* Suppose $\phi$ is a CNF formula which has the properties of the theorem, i.e. $\phi$ is reduced and $\phi$ 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 $1 \leq i \leq n$, which is also solved regarding Corollary 27. It suffices to provide an assignment $\sigma$ which satisfies all these negative literals, because then each clause is satisfiable with that $\sigma$, which implies that $\phi$ 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: $f[\,] = g([\,], \underbrace{c, ..., c}_{m-1 \text{ times}})$. Now define a number $M = 1 + Max_{1 \leq i \leq n} S(t_i)$.

Then define a context $C = f^M[\ ]$, $M$-fold application of $f$. Consider the following assignment:

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

We claim that $\sigma$ satisfies $x_i \not\approx t_i$ for each $1 \leq i \leq n$:
Indeed, note that $S(\sigma(x_i)) = M.i$. Moreover, if $S(t_i) = 0$, then $S(\sigma(t_i)) = M.j$ with $0 \leq j \leq n$ and $i \neq j$ ($x_i \neq t_i$ because $\phi$ 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, $S(\sigma(x_i)) \neq S(\sigma(t_i))$.

– $\mathcal{D}$ 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:

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

We know that the two given sets, are of finite cardinality. without loss of generality suppose that $V_\phi = \{x_1, x_2, ...x_n\}$, for some $n \in \mathbb{N}$. Since $\mathcal{D}$ has infinitely many constant symbols, there exists a set $\mathcal{C} = \{c_1, c_2, ...c_{n+1}\}$, 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 $\sigma$ satisfies it since $\sigma(x_i) \neq \sigma(x_j)$.
- $x_i \not\approx c$.Then $\sigma$ 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 $\sigma$ 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 24. 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 35.** *Let* $(\mathsf{Fun}; \approx)$ *be a signature with infinitely many ground terms. Let* $\mathcal{D}$ *be its ground term algebra. Let* $\phi$ *be a CNF. Let* GDPLL *be instantiated as indicated above. Then*

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

*Proof.* In order to apply Theorem 5, we have to check Properties 1–5. Property 2 and 5 are obvious. Property 1 has been proved in Theorem 32. Property 3 has been proved in Theorem 31; 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 34. □

## 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 [BJKO00]. 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 $\rightarrow$ ATerm by means of a hash table.

We implemented the almost linear unification algorithm from [BN98], which is based on [Hue76]. 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: Rule 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.

`phe` - *equational pigeon hole.* Variables: $x_1, \ldots, x_N, y$.

$$\Big(\bigwedge_{1 \le i < j \le N} x_i \ne x_j\Big) \wedge \Big(\bigwedge_{1 \le i \le N} \bigvee_{1 \le j \le N, j \ne i} x_j = y\Big)$$

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 [ZG03].

`circc` - *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 $x_1$.

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

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

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

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 $k$, $x_k = x_{k+1}$.

`evod` - *even and odd natural numbers.* Variables: $x_1, \ldots, x_N$; unary constant $S$.

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

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 Table 3 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 seconds), 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 24). 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 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 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 [ZG03] 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 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 [ZG03], on a machine which is at most 2.5 times faster.

| type | N | # lit | without UR | | with UR | |
|---|---|---|---|---|---|---|
| | | | #secs | #GDPLL | time (secs) | # GDPLL |
| phe | 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 |
| circ | 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 |
| succ | 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 |
| evod | 12 | 23 | 0 | 6763 | 0 | 3171 |
| | 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 |

**Fig. 3.** Experimental results with and without Unit Resolution

## 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. 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 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


[ABC+02] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In A. Voronkov, editor, *Automated Deduction - CADE-18:18th International Conference on Automated Deduction*, pages 195–210, Copenhagen, Denmark, July 27-30 2002. Springer-Verlag Heidelberg.

[ACG+02] A. Armando, C. Castellini, E. Giunchiglia, F. Giunchiglia, and A. Tacchella. SAT-based decision procedures for automated reasoning: a unifying perspective. IRST Technical Report 0202-05, Istituto Trentino di Cultura, February 2002.

[Ack54] W. Ackermann. *Solvable Cases of the Decision Problem.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.

[Bau00] P. Baumgartner. FDPLL – A First-Order Davis-Putnam-Logeman-Loveland Procedure. In David McAllester, editor, *CADE-17*, volume 1831 of *LNAI*, pages 200–219. Springer, 2000.

[BDS00] C.W. Barrett, D.L. Dill, and A. Stump. A framework for cooperating decision procedures. In D. A. McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction (Pittsburgh, PA)*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 79–98. Springer, 2000.

[BFG+01] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I.A. van Langevelde, B. Lisser, and J.C. van de Pol. $\mu$CRL: A toolset for analysing algebraic specifications. In G. Berry etal, editor, *Proc. CAV'01*, LNCS 2102, pages 250–254. Springer, 2001.

[BGV99] R.E. Bryant, S. German, and M.N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In N. Halbwachs and D Peled, editors, *Proc. of Computer Aided Verification, CAV'99*, LNCS 1633. Springer-Verlag, 1999.

[BJKO00] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient Annotated Terms. *Software – Practice & Experience*, 30:259–291, 2000.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, New York, 1998.

[BP04] B. Badban and J.C. van de Pol. Zero, successor and equality in BDDs. *Annals of Pure and Applied Logic (to appear)*, 2004. Also appeared as Technical Report, CWI, Amsterdam, The Netherlands, SEN-R0231, see `http://www.cwi.nl/ftp/CWIreports/SEN/SEN-R0231.pdf`.

[Bur94] Burch, J.R. and D.L. Dill. Automatic verification of pipelined microprocessors control. In D. L. Dill, editor, *Proceedings of Computer Aided Verification, CAV'94*, LNCS 818, pages 68–80. Springer-Verlag, June 1994.

[CL89] H. Comon and P. Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7(3–4):371–425, 1989.

[Col84] A. Colmerauer. Equations and Inequations on Finite and Infinite Trees. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS-84)*, pages 85–99, Tokyo, Japan, November 1984. ICOT.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7(3):201–215, 1960.

[FORS01] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated canonizer and solver. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (Paris, France)*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249. Springer-Verlag, July 2001.

[GHN+04] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Olivera, and C. Tinelli. DPLL(T): Fast decision procedures. In *16th International Conference on Computer Aided Verification (CAV)*. July, 2004.

[GP00] J.F. Groote and J.C. van de Pol. Equational binary decision diagrams. In M. Parigot and A. Voronkov, editors, *Proc. of LPAR 2000*, LNAI 1955, pages 161–178. Springer, 2000.

[GR01] J.F. Groote and M.A. Reniers. Algebraic process verification. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 17. Elsevier, 2001.

[GS96] F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures: the case study of modal K. In M. A. McRobbie and J. K. Slaney, editors, *Proceedings of the Thirteenth International Conference on Automated Deduction (CADE-96)*, volume 1104 of *LNAI*, pages 583–597. Springer, 1996.

[GSZA98] A. Goel, K. Sajid, H. Zhou, and A. Aziz. BDD based procedures for a theory of equality with uninterpreted functions. In *Proceedings of Computer Aided Verification, CAV'98*, LNCS 1427, pages 244–255. Springer-Verlag, 1998.

[Hue76] G. Huet. *Résolution d'équations dans les langages d'ordre 1, 2, . . . , ω*. PhD thesis, Université Paris, 7, 1976.

[LMM87] J-L. Lassez, M. J. Maher, and K. Marriott. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann Publishers, Los Altos, California, 1987.

[Mah88] M.J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proceedings, Third Annual Symposium on Logic in Computer Science*, pages 348–357. IEEE Computer Society, 1988.

[NO03] R. Nieuwenhuis and A. Oliveras. Congruence closure with integer offsets. In *10th Int. Conf. on Logics for Programming, AI and Reasoning (LPAR)*, Sept. 2003.

[Pic03] R. Pichler. On the complexity of equational problems in CNF. *Journal of Symbolic Computation*, 36, 2003.

[PRSS99] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small domains instantiations. In N. Halbwachs and D Peled, editors, *Proceedings of Computer Aided Verification, CAV'99*, LNCS 1633. Springer-Verlag, 1999.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–49, 1965.

[SBD02] A. Stump, C.W. Barrett, and D.L. Dill. CVC: a cooperating validity checker. In J. C. Godskesen, editor, *Proceedings of the International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science, 2002.

[SR02] N. Shankar and H. Rueß. Combining Shostak theories. In S. Tison, editor, *Proceedings of the 13th International Conference on Rewriting Techniques and Applications (Copenhagen, Denmark)*, volume 2378 of *Lecture Notes in Computer Science*, pages 1–18. Springer, July 2002.

[ZG03] H. Zantema and J. F. Groote. Transforming equality logic to propositional logic. In *Proceedings of 4th International Workshop on First-Order Theorem Proving (FTP'03)*, volume 86(1) of *Electronic Notes in Theoretical Computer Science*, 2003.