Prove with $\mathsf{GDPLL}^{\mathsf{WD}}$ A Complete Proof Procedure for Recursive Data Structures

Technical Report soft-08-07
Chair for Software Engineering

University of Konstanz

Prove with GDPLL$^{\mathsf{WD}}$

A Complete Proof Procedure for Recursive Data Structures

Bahareh Badban

Department of Software Engineering
University of Konstanz, Germany

Abstract. In this paper we present a terminating, sound and complete algorithm for the verification of recursively defined data structures. To mention some, nat, $list$ and $tree$ data types and also $record$ are commonly used examples of such structures. Recursively defined data structures are of value for use in software verification. Many programming languages support recursive data structures. The best known example on this kind is the LISP programming language, which uses $list$. Our algorithm, GDPLL$^{\mathsf{WD}}$, which is an extension of the Davis, Putnam, Logemann and Loveland (DPLL) procedure solves satisfiability problem of recursive data types through providing witness assignments.

Introduction

DPLL [13, 14] named after Davis, Putnam, Logemann and Loveland, is mainly used to decide satisfiability of propositional formulas, represented in conjunctive normal form (CNF). Unit propagation, is the main idea of this recursive procedure. We extend this technique to the theory of recursive data structures, or Ground Term Algebra with Destructorsas we call. In application *destructors* are extensively used. They are almost a necessary element of the theories involving constructors. An example of a formula in ground term algebra with destructors could be:

$$(succ(x) = y \vee x = succ(head(tail(z)))) \wedge (z \neq cons(w, t) \vee x \neq node(y, leaf))$$

As mentioned in [6], a known reference for the quantifier-free term algebra over constructors is the approach introduced by Oppen in [22]. Oppen's algorithm gives a detailed decision procedure for a single inductive data type with a single constructor. This is where, as in [6], we are interested in any set of recursive data types, with multiple constructors, where each constructor has one or more destructors (or *selectors*) that can be used to retrieve the inner terms. In contrast to [6, 22], we use the idea of unification. Combining unification and DPLL, to our knowledge, was first introduced in [4]. Since every formula can be transformed into a conjunctive normal form (CNF), we do not only handle conjunctions of literals but also all formulas inside the theory.

We also consider destructors as partially defined functions. They are well-defined over their corresponding constructors (e.g. $pred$ over $succ$ in nat, and car over $cons$ in $list$) and are undefined elsewhere (e.g. car over nil in $list$) . Later, inside the algorithm, we deal with undefined terms in a way that will keep us from attaining false satisfiability results. For the question of how to define destructors over irrelevant constructors in general, we cite the following from [6]:

> There is no obviously correct thing to do in this case since it would correspond to an error condition in a real application. Our axiom specifies that in this case, the result is the designated ground term for that selector... It is important to notice that as a result, our procedure may give counter-intuitive results if given as input a formula whose satisfiability depends on the application of a selector to the wrong constructor... [and then in Section 6.2] it is not clear how best to interpret the application of a selector to the wrong constructor. One compelling approach is to interpret selectors as partial functions. An evaluation of a formula then has three possible outcomes: true, false, or undefined.

The current procedure is a generalization of the previous approach [3, 4] where an extension of DPLL over the theory of *ground term algebra* was introduced. There, theories are assumed to only contain constructors. Our contribution in this paper is mainly three folded:

- We deal with destructors.
- Our algorithm works over formulas of *Conjunctive Normal Form*, where disjunction is also present. Therefore, verification is a lot more complex than over a pure conjunction of literals.
- The previous work is augmented to a *proof* procedure, where an assignment is provided to witness the formula's satisfiability. This is to assure us of the verification's correctness, and besides, to have a set of values to substitute the variables with. Unsatisfiable formulas will be refuted by an empty set of assignments.

Related work. Decision procedures, for many theories already exist. Most of the approaches which treat recursive data types only deal with conjunctions of (negated) equations. Examples of these are [6, 19, 22]. Although, using a DNF transformation, it is sufficient to solve any boolean combination, but unfortunately, 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.

In [27, 30] other approaches for treating recursive data types, are introduced. As said in [6] the idea of *type completion guess* introduced in [30] unfortunately seems to be very expensive in practice.

DPLL(T) approach introduced in [16, 20, 21, 29], deals with combining decision procedures for different theories, called Satisfiability Modulo Theories (SMT). These papers, unfortunately, do not give concrete algorithms for use in verification. In [16] DPLL(T) calculus is used to provide a new approach for satisfiability in the logic of uninterpreted functions with equality (EUF), also EUF logic with successor and predecessor. SVC [8], CVC [28], CVC lite [5] and ICS [15, 25] are examples of SMT provers. CVC lite verifies a very rich sub-theory of first-order logic, yet it does not include algebraic data types. For this reason a new version of CVC called CVC3 [7], has recently been introduced, where the algorithm of [6] is incorporated.

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, 10] to transform this logic to propositional logic, so any propositional logic satisfiability checker can be used.

Our decision procedure for ground term algebras borrows ideas from the well-known unification theory, originated in [24]. Unification solves conjunctions of equations in the ground term algebra. Colmerauer [11] studied a setting with conjunctions of both equations and inequations. The full first-order theory of equality in ground term algebras is studied in [12, 18] (both focus on a complete set of rewrite rules) and [23] (which 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.

Other works on, linear and integer programming for arithmetic over integers or reals, and congruence closure algorithms which deal with uninterpreted functions can be found in [19, 26].

Road map. Section 1 explains the language, its semantics and some primary definitions. In Section 2 we extend the algorithm of previous work [3, 4]. This new algorithm, called $\mathsf{GDPLL}^{\mathsf{W}}$, is an automatic *Proof* producer for the entire theory of ground term algebra. In Section 3, we lift the method to yet more expressive logic where *destructors* are present too. The respective theory is then called ground term algebra with destructors and the algorithm is $\mathsf{GDPLL}^{\mathsf{WD}}$. We conclude the work in Section 4.

1 The Language

Our terminology here is built over that of [3, 4] with some subtle modifications.

1.1 Syntax

In this paper, a signature Σ is a pair $(\mathsf{Fun}, =)$ of a set $\mathsf{Fun} = \{f, g, h, \dots\}$ of *function symbols*, and a binary predicate $=$. With each $f \in \mathsf{Fun}$ is associated a non-negative integer n, called the *arity* of f. Functions of arity zero are called *constant symbols*, we display them by $c, c_i, \dots$. $\mathsf{Var} = \{x, y, z, \dots\}$ represents a set of *variables*. Var and Fun are disjoint sets. Terms are built from variables, constant symbols and functions over variables and constant symbols. The set $\mathsf{Term}(\Sigma, \mathsf{Var})$ of *terms* over the signature Σ is inductively defined as:

- $\mathsf{Var} \subseteq \mathsf{Term}(\Sigma, \mathsf{Var})$.
- For all $f \in \mathsf{Fun}$ and all terms $t_1, \dots, t_n$ where n is the arity of f, $f(t_1, \dots, t_n) \in \mathsf{Term}(\Sigma, \mathsf{Var})$.

$\mathsf{Term}(\Sigma)$ denotes the set of *ground terms*, which is defined as $\mathsf{Term}(\Sigma, \emptyset)$.

An *atom* a is considered to be an expression of the form $t = s$, where t and s are terms. We use notations like a, a_i, b for atoms. A *literal* l is either an atom itself or a negated atom e.g. $\neg a$. If $l \equiv \neg a$, for some atom a then $\neg l \equiv a$. A literal l is *positive* if it is an atom; it is *negative* if it is a negated atom. We use notations like l, l_i, for literals. Clearly, each atom is also a literal. A *clause* is a finite set of literals. The empty clause represents $\perp$. Here, Cl, Cl_i are notations used for clause.

In this paper, we work with *conjunctive normal form* (CNF) formulas. Each CNF is represented by a finite set of clauses. As of now, by formula we mean CNF, unless explicitly stated otherwise.

Definition 1. Here, we introduce a number of notations that we will use frequently. Below, φ stands for formula, l, l' for literal, Cl for clause, t and s for terms, S for set:

- $pos(\varphi)$ is the number of all occurrences of positive literals in φ.
- $\|S\|$ denotes cardinality of S, i.e. number of its elements.
- Cl is a unit clause if $\|Cl\|$ is equal to 1.
- Cl is a *purely positive clause* if all its literals are positive.
- $\varphi|_{t \to s}$ is obtained by replacing each occurrence of t in φ with s. Same meaning holds for $\varphi|_{l \to l'}$.
- $\mathsf{Var}(\varphi)$ is the set of all variables occurring in φ (analogously for $\mathsf{Var}(t)$ and $\mathsf{Var}(Cl)$). $\mathsf{At}(\varphi)$ (resp. $\mathsf{Term}(\varphi)$, $\mathsf{Lit}(\varphi)$ and $\mathsf{Cls}(\varphi)$) is the set of all atoms (resp. terms, literals and clauses) occurring in φ. $\mathtt{PLit}(\varphi)$ and $\mathtt{NLit}(\varphi)$ are respectively the set of all positive and negative literals in φ.
- $\varphi|_l = \{Cl - \{\neg l\} \mid Cl \in \mathsf{Cls}(\varphi),\ l \notin Cl\}$ removes all the clauses in φ containing l, and further, removes $\neg l$ from the other clauses.
- $\varphi \wedge l$ is a shortcut for $\varphi \cup \{\{l\}\}$.
- $\varphi \wedge Cl$ is a shortcut for $\varphi \cup \{Cl\}$.

1.2 Semantics

Here we present some general semantics of this paper. To prevent any notational confusion we use $\equiv$ for semantical equivalence.

Definition 2. *A structure $\mathcal{D}$ over a signature $\Sigma \equiv (\mathsf{Fun}, =)$ consists of:*

- *a non-empty set* $\mathtt{DM}$ *called the* domain *of* $\mathcal{D}$,
- *for every* $f \in \mathsf{Fun}$ *of arity* n *a map* $f_{\mathcal{D}} : \mathtt{DM}^n \to \mathtt{DM}$,
- *for* $=$, *a binary predicate* $=_{\mathcal{D}}$ *representing the syntactic equality.*

A function α: $\mathsf{Term}(\Sigma, \mathsf{Var}) \to \mathtt{DM}$ is called *assignment*, if it is identical over constant symbols, and it is distributive over other terms. This means that for constant symbols c, $\alpha(c) \equiv c_{\mathcal{D}}$, and for terms of the form $f(t_1, \ldots, t_n)$, $\alpha(f(t_1, \ldots, t_n)) \equiv f_{\mathcal{D}}(\alpha(t_1), \ldots, \alpha(t_n))$. We use α, β to denote assignments.

We say α *satisfies* φ, denoted $\alpha \models \varphi$, if in each clause $Cl \in \mathsf{Cls}(\varphi)$ there exists a literal $t = u$ (or $t \neq u$) such that $\alpha(t) \equiv \alpha(u)$ (respectively $\alpha(t) \not\equiv \alpha(u)$). Obviously if $\alpha \models \varphi$ for some assignment α then φ is satisfiable (in the respective structure). Given a structure $\mathcal{D}$ and a formula φ, we say φ is *satisfiable* in $\mathcal{D}$, if there exists some assignment α such that $\alpha \models \varphi$.

Definition 3. *Let $[\,]$ be a new constant symbol which does not occur in Σ. A context F is a term in $\mathsf{Term}(\Sigma \cup \{[\,]\})$, and it can be expressed as an incomplete term or a term with holes. However a context can have zero, one or more holes [9], but for our purpose we would only consider contexts with (at most) one hole. $F[t]$ denotes the result of replacing the hole with a term t.*

1.3 Substitutions and Most General Unifiers

Here, we bring a list of the standard definitions and properties of substitutions and unifiers, taken from [2, 17].

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. $\mathsf{Dom}(\sigma) := \{x \in \mathsf{Var} \mid \sigma(x) \neq x\}$ is called it domain. If $\mathsf{Dom}(\sigma) \equiv \{x_1, \ldots, x_n\}$, then we alternatively write σ as $\sigma := \{x_1 \mapsto \sigma(x_1), \ldots, x_n \mapsto \sigma(x_n)\}$. We denote by $\mathsf{Eq}(\sigma)$ the set of equations obtained from $\mathsf{Dom}(\sigma)$, that is $\mathsf{Eq}(\sigma) \equiv \{x_1 = \sigma(x_1), \ldots, x_n = \sigma(x_n)\}$. Later we will use negation of this set, which is $\neg\mathsf{Eq}(\sigma) \equiv \{x_1 \neq \sigma(x_1), \ldots, x_n \neq \sigma(x_n)\}$.

Substitutions are extended to set of terms/literals/clauses as follows:
Over terms: $x^\sigma := \sigma(x)$, $f(t_1, \ldots, t_n)^\sigma := f(t_1^\sigma, \ldots, t_n^\sigma)$.
Over literals: $(t{=}u)^\sigma := t^\sigma{=}u^\sigma$ and $(t \neq u)^\sigma := t^\sigma{\neq}u^\sigma$.
Over clauses: $Cl^\sigma \equiv \{l_1, \ldots, l_n\}^\sigma := \{l_1^\sigma, \ldots, l_n^\sigma\}$.
Finally over formulas: $\varphi^\sigma \equiv \{C_1, \ldots, C_n\}^\sigma := \{C_1^\sigma, \ldots, C_n^\sigma\}$.

Definition 4. *The* composition *$\sigma.\rho$ of substitutions σ and ρ is defined in a way that $\sigma.\rho(x) \equiv \sigma(\rho(x))$. A substitution σ is* more general *than a substitution σ' (notation: $\sigma \lesssim \sigma'$) if there is a substitution δ such that $\sigma' \equiv \delta.\sigma$. A substitution σ is* idempotent *if $\sigma.\sigma \equiv \sigma$. We will later use σ^2 instead of $\sigma.\sigma$.*

Definition 5. *A* unifier *or* solution *of a set $S \equiv \{s_1{=}t_1, \ldots, s_n{=}t_n\}$ consisting of a finite number of atoms, is a substitution σ such that $s_i^\sigma \equiv t_i^\sigma$ for $i \equiv 1, \ldots, n$. A substitution σ is a most general unifier of S, denoted* $\mathsf{mgu}(S)$*, if:*
- *σ is a unifier of S and*
- *$\sigma \lesssim \sigma'$ for each unifier σ' of S.*

A straightforward result of Definition 5 is that $\mathsf{mgu}(\{x{=}x\}) \equiv \emptyset$.

A literal $t{=}u$ is in *solved-form* if it is of the form $x{=}u$, where x is a variable, u is a term, and x does not occur in u. Otherwise it is *non-solved*. A literal $\neg a$ is in *solved-form* if a is in solved-form. A set is in solved-form if all its members are in solved-form.

Lemma 1.
1. *If a set S of atoms has a unifier, then it has an idempotent* mgu.
2. *If $\sigma \equiv \mathsf{mgu}(S)$ and σ is idempotent, then $\mathsf{Eq}(\sigma)$ is in solved-form.*

Some notation and conventions.

– Let $\varphi \equiv \{C_1, \ldots, C_n\}$ be a set of positive unit clauses. Then, by $\sigma \equiv \mathsf{mgu}(\varphi)$ we mean $\sigma \equiv \mathsf{mgu}(\bigcup_{1 \leq i \leq n} C_i)$.
– We often simply write $\mathsf{mgu}(s{=}t)$ instead of $\mathsf{mgu}(\{s{=}t\})$.
– We use the notation $\mathsf{mgu}(S) \equiv \perp$ if S has no unifier.
– As of now we only consider *idempotent* mgus, which exist whenever there is a unifier (Lemma 1(1)).

We use the notation of imgu to stress that the most general unifier is *idempotent*. It can easily be derived that using Lemma 1(2), if $\sigma \equiv \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is an imgu, then $x_i \notin \mathsf{Var}(t_j)$ for all $1 \leq i, j \leq n$. In the previous work [3, 4] we presented GDPLL algorithm, a generalization of the well-known DPLL procedure of Davis-Putnam-Logemann-Loveland. Although this algorithm would determine the (dis)satisfaction of CNFs over the logic of Ground-Term-Algebra, yet it will not give any hint to why the formula is satisfiable in case it is. In the next section we would introduce $\mathsf{GDPLL}^{\mathsf{W}}$ which will *prove* satisfiable inputs. All the algorithms will operate over some ground-term-algebra.

2 Proof for the theory of Ground Term Algebra

Before starting with the algorithm, we need to bring forth some definitions and results from [3, 4]. A structure $\mathcal{D}$ is a *ground term algebra* over Σ if first, its domain consists only of ground terms, i.e. $\mathtt{DM} \equiv \mathsf{Term}(\Sigma)$, and second, if for all $f, g \in \mathsf{Fun}$ and all $t, t_1, \ldots, t_n, s_1, \ldots, s_m \in \mathtt{DM}$, the following properties hold:

1. if $f \neq g$, then $f_{\mathcal{D}}(t_1, \ldots, t_n) \not\equiv g_{\mathcal{D}}(s_1, \ldots, s_m)$
2. if $(t_1, \ldots, t_n) \not\equiv (s_1, \ldots, s_n)$ then $f_{\mathcal{D}}(t_1, \ldots, t_n) \not\equiv f_{\mathcal{D}}(s_1, \ldots, s_n)$
3. for all contexts $F \neq [\,]$: $t \not\equiv F[t]$

As of now, the background theory is considered to be a ground term algebra unless stated otherwise. For simplicity, we drop the subscript $_{\mathcal{D}}$ from $f_{\mathcal{D}}$ and $=_{\mathcal{D}}$. In this paper we apply the same reduction system as in [3, 4]:

Definition 6 (Reduction System). *Given a formula φ we apply the following reduction rules on it:*

1. *If $t = t \in Cl \in \mathsf{Cls}(\varphi)$ then $\varphi \longrightarrow \varphi - \{Cl\}$.*
2. *If $\perp \in \varphi$ and $\varphi \neq \{\perp\}$ then $\varphi \longrightarrow \{\perp\}$.*
3. *If $\varphi = \varphi_1 \uplus \{Cl \uplus \{t \neq u\}\}$, $t{=}u$ is non-solved, and $\sigma{=}\mathsf{imgu}(t{=}u)$ then*
   – *if $\sigma = \perp$, then $\varphi \longrightarrow \varphi_1$*
   – *otherwise $\varphi \longrightarrow \varphi_1 \cup \{Cl \cup \neg\mathsf{Eq}(\sigma)\}$*
4. *If $\varphi_1{=}\{Cl \mid Cl \in \mathsf{Cls}(\varphi)$ is a positive unit clause$\}{\neq}\emptyset$, and $\sigma{=}\mathsf{imgu}(\varphi_1)$ then*
   – *if $\sigma = \perp$, then $\varphi \longrightarrow \{\perp\}$.*
   – *otherwise let $\varphi_2 = \varphi - \varphi_1$, and $\varphi \longrightarrow {\varphi_2}^{\sigma}$.*
5. *If $\varphi = \{\{\neg a\}\} \uplus \varphi_1$ and $a \in \mathsf{At}(\varphi_1)$ then $\varphi \longrightarrow \{\{\neg a\}\} \uplus \varphi_1|_{\neg a}$.*

By $\varphi \xrightarrow{i} \varphi'$, we mean that rule i is applied over φ and that φ' is deduced. We say that rule i is *applicable* on φ if φ' and φ are syntactically different. The intention is to apply the reduction rules on a given formula only until they are applicable. When no rule is applicable, we stop the reduction process and call the entailed formula *of reduced form*. This last result is denoted $\mathsf{Reduce}_1(\varphi)$. It is worth saying that since there is no priority order for applying the reduction rules on a formula φ, we may obtain various reduced forms for φ, see [3] Chapter 4. In this paper, as in the previous work, we consider only **infinite** ground term algebras, i.e. either there exists a function symbol of arity> 0 (non-constant), or there are infinitely many constant symbols. The following lemma is proved in [4]:

Lemma 2. *For a formula φ in reduced form, the following properties hold:*

1. *φ contains no literal of the form $t = t$.*
2. *If $\perp \in \varphi$ then $\varphi \equiv \{\perp\}$.*
3. *All its negative literals are in solved-form.*
4. *φ contains no positive unit clause.*
5. *If $\varphi = \{\{\neg a\}\} \uplus \varphi_1$ then $a \notin \mathsf{At}(\varphi_1)$.*

Lemma 3. *Given a formula φ:*

1. *The Reduction System (Definition 6) is terminating over φ.*
2. *Suppose $\sigma \models l$, where l is a literal. Then, $\sigma \models \varphi$ if and only if $\sigma \models \varphi|_l$.*
3. *Given a literal l, φ is satisfiable iff either $\varphi \wedge l$ or $\varphi|_{\neg l} \wedge \neg l$ is satisfiable.*
4. *If $\varphi \overset{1,2,3,5}{\longrightarrow} \varphi'$, i.e. φ' is obtained from applying one of rules $1, 2, 3$ or 5 on φ. Then, $\alpha \models \varphi'$ implies $\alpha \models \varphi$ for any assignment α.*

Proof. See the Appendix. ■

In order to obtain a *witness* (proof) for satisfiable formulas, we need to consider each reduction rule of Definition 6 at a time. Below, the Reduce algorithm is introduced for computing a reduced form for input formulas. The output will be a reduced formula together with a substitution σ.

```
Reduce($\varphi$)
begin
    $\sigma := \emptyset$;
    while either of the rules of Definition 6 are applicable on $\varphi$ then
        choose one of the applicable rules;
        if the rule is one of the rules 1, 3 or 5 then apply that on $\varphi$ and
            call the result $\varphi$;
        if the rule is rule 2 then return $(\perp, \emptyset)$;
        if the rule is rule 4 then
            $\varphi_1 := \{C \mid C \in \varphi$ is a positive unit clause$\}$;
            $\varphi_2 := \varphi - \varphi_1$;
            if $\mathsf{imgu}(\varphi_1) \equiv \perp$ then return$(\perp, \emptyset)$;
            $\sigma := \mathsf{imgu}(\varphi_1).\sigma$ and $\varphi := {\varphi_2}^{\sigma}$;
    return $(\varphi, \sigma)$;
end
```

$f.g$ denotes functional composition of f and g, i.e. $f.g(\psi){\equiv}f(g(\psi))$. Also, $\emptyset$ is the identity element, i.e. composition of any function with $\emptyset$ is equivalent to the function itself, $f.\emptyset{\equiv}\emptyset.f \equiv f$. Composition with $\perp$ results in $\perp$. Functional composition is not commutative.

Theorem 1. *Given a formula φ and a literal $t{=}u \in \mathsf{Lit}(\varphi)$ we have (see Definition 1):*

1. *$pos(\mathsf{Reduce}_1(\varphi \wedge (t{=}u))){<}pos(\varphi)$ and $pos(\mathsf{Reduce}_1(\varphi|_{t \neq u} \wedge (t \neq u))){<}pos(\varphi)$.*
2. *A formula φ is satisfiable iff $\mathsf{Reduce}_1(\varphi)$ is satisfiable.*

Proof. Proofs are in [3] Chapter 4. There $\mathsf{Reduce}(\varphi)$ is equivalent to what we call $\mathsf{Reduce}_1(\varphi)$ here.

Corollary 1. *Given a formula φ, if $\mathsf{Reduce}(\varphi){\equiv}(\mathsf{Reduce}_1(\varphi), \sigma)$ then $\sigma{\not\equiv}\perp$ and $\sigma^2{\equiv}\sigma$. Moreover if $\beta \models \mathsf{Reduce}_1(\varphi)$ for an assignment β, then $\beta.\sigma \models \varphi$.*

Proof. See the Appendix. ■

For an example on how Reduce algorithm works see Example 3 in the Appendix. Below two essential parts of the final algorithm are uttered. These procedures will disclose an assignment for satisfiable formulas, with respect to the theory of background. Based on whether the language has any non-constant function symbols or not, the next two algorithms are used. If there exists some non-constant function symbol in the theory, then Func should be applied:

```
Func($\varphi, \sigma$)
    begin
        choose $g \in$ Fun;
        choose a constant symbol $c$;
        let $F[\,] \equiv g([\,]\underbrace{c, \ldots, c}_{\text{arity}(g)-1 \text{ times}})$;
        $M := 0$;
        $X_1 := \emptyset$;
        $X := \emptyset$;
        $i := 0$;
        while $\varphi \neq \emptyset$
            choose $Cl \in$ Cls($\varphi$);
            $\varphi := \varphi - \{Cl\}$;
            choose $x \neq u \in Cl$;
            $M := max(M, depth(u) + 1)$;
            if $x \notin X_1$ then
                $i := i + 1$;
                $X_1 := X_1 \cup \{x\}$;
                $X := X \cup \{(x, i)\}$;
            end if
        end while
        $\alpha := \{x \mapsto F^{iM}[c] \mid (x, i) \in X\} \cup \{z \mapsto c \mid z \notin X_1\}$;
        return $\alpha.\sigma$;
    end
```

Before explaining some properties of the aforementioned algorithm, let us define the $depth$ function. This function is used in the algorithm to measure the maximum depth of the nested function symbols inside a term:

Definition 7. *Given a term t, $depth(t)$ is inductively defined by:*
$depth(x) \equiv depth(c) \equiv 0$ for any variable x, and any constant symbol c, and

$$depth(f(t_1, \ldots, t_n)) \equiv 1 + \max_{1 \leq i \leq n} depth(t_i) \quad \text{if } n \geq 1$$

Corollary 2. *Assume that the background theory contains some non-constant function symbol, and* Reduce(φ)$\equiv$(ψ, σ)*, where ψ contains no purely positive clauses. Then,* Func(ψ, σ)$\equiv\alpha.\sigma$ *for some assignment α, where $\alpha.\sigma \models \varphi$, and $\alpha.\sigma \not\equiv \bot$.*

Proof. See the Appendix. ■

When all function symbols in the theory are constant, then Cons is applicable:

```
Cons($\varphi, \sigma$)
    begin
        let $n \equiv \|\varphi\|$;
        let $\mathcal{C} \equiv \{c_1, c_2, \ldots, c_{n+1}\}$ of $n + 1$ distinct constant symbols which do not occur in $\varphi$;
        $X_1 := \emptyset$;
        $X := \emptyset$;
        $i := 0$;
        while $\varphi \neq \emptyset$
            choose $C \in \varphi$ and $x \neq u \in C$;
            $\varphi := \varphi - \{C\}$;
            if $x \notin X_1$ then
                $i := i + 1$;
                $X_1 := X_1 \cup \{x\}$;
                $X := X \cup \{(x, i)\}$;
            end if
        end while
        $\alpha := \{x \mapsto c_i \mid (x,i) \in X\} \cup \{z \mapsto c_{n+1} \mid z \notin X_1\}$;
        return $\alpha.\sigma$;
    end
```

Corollary 3. *Assume that the background theory consists only of constant function symbols, and that* $\mathsf{Reduce}(\varphi) \equiv (\psi, \sigma)$*, where* ψ *contains no purely positive clauses. Then,* $\mathsf{Cons}(\psi, \sigma) \equiv \alpha.\sigma$ *for some assignment* α*, where* $\alpha.\sigma \models \varphi$*, and* $\alpha.\sigma \not\equiv \bot$*.*

Proof. Similar to Corollary 2. The only difference is that here the $j+1$ elements of $\mathcal{C}$, will entail that $\alpha \models x \neq u$. ■

$\mathsf{GDPLL}^{\mathsf{W}}$. Now, We have all the necessary components to build the proof algorithm. This algorithm will determine whether or not a formula is satisfiable and in parallel it will construct a witness (proof) for it. The input is a (CNF) formula, and the output is either (SAT, α) or $(\mathsf{UNSAT}, \emptyset)$. In case result is (SAT, α), then α will be a proof for the formula (which makes it true).

```
GDPLL^W($\varphi$)
  begin
    $(\varphi, \sigma) := \mathsf{Reduce}(\varphi)$;
    if $\bot \notin \varphi$ then
      if $\varphi$ has no purely positive clause then
        if there exists a non-constant function symbol then
          return $(\mathsf{SAT}, \mathsf{Func}(\varphi, \sigma))$;
        return $(\mathsf{SAT}, \mathsf{Cons}(\varphi, \sigma))$;
      end if
      choose $a \in \mathtt{PLit}(\varphi)$;
      if $\mathsf{GDPLL}^{\mathsf{W}}(\varphi \wedge a) \equiv (\mathsf{SAT}, \alpha)$ then return $(\mathsf{SAT}, \alpha.\sigma)$;
      if $\mathsf{GDPLL}^{\mathsf{W}}(\varphi|_{\neg a} \wedge \neg a) \equiv (\mathsf{SAT}, \alpha)$ then return $(\mathsf{SAT}, \alpha.\sigma)$;
    return $(\mathsf{UNSAT}, \emptyset)$;
  end
```

This algorithm is always terminating:

Theorem 2 (Termination). *The* $\mathsf{GDPLL}^{\mathsf{W}}(\varphi)$ *is terminating for any formula* φ *in the theory of ground term algebra, i.e. it always terminates with a final result of* (SAT, α) *or* $(\mathsf{UNSAT}, \emptyset)$*.*

Proof. See the complete proof in the Appendix. ■

This algorithm is complete, meaning that it decides only in finite steps whether the input formula is satisfiable or is unsatisfiable. Moreover, φ is satisfiable if and only if $\mathsf{GDPLL}^{\mathsf{W}}(\varphi)$ reports so, and in addition it delivers an assignment α such that $\alpha \models \varphi$.

Theorem 3 (Soundness and Completeness).

- *φ is satisfiable if and only if* $\mathsf{GDPLL}^{\mathsf{W}}(\varphi)$ *returns* (SAT, α)*, where* $\alpha \models \varphi$ *and* $\alpha \not\equiv \bot$*.*
- *φ is unsatisfiable if and only if* $\mathsf{GDPLL}^{\mathsf{W}}(\varphi)$ *returns* $(\mathsf{UNSAT}, \emptyset)$*.*

Proof. See the detailed proof in the Appendix. ■

For an example on how $\mathsf{GDPLL}^{\mathsf{W}}$ algorithm finds a witnnes, see Example 4 in the Appendix. Next section is about adding destructors to the theory, as well. Having this extension, we will be able to express many theories of interest in our language. Destructors are those functions which decompose their associated constructors.

3 Completing with Destructors

Here, we consider signatures Σ of the form $(\mathsf{Const} \cup \mathsf{Dest}_{\mathsf{Const}}, =)$, where $\mathsf{Const} \equiv \{C_1, C_2, \ldots, C_n\}$ and $\mathsf{Dest}_{\mathsf{Const}} \equiv \{D_{11}, D_{12}, \ldots, D_{nm}\}$ are disjoint. With each $C_j \in \mathsf{Const}$ (called constructor) of arity k, k separate destructors D_{j1},... D_{jk} are associated. Each $D_{ji} \in \mathsf{Dest}_{\mathsf{Const}}$ is a unary function.

The set $\mathsf{Term}(\Sigma, \mathsf{Var})$ of *terms* over aforementioned Σ is recursively defined by:

- $\mathsf{Var} \subseteq \mathsf{Term}(\Sigma, \mathsf{Var})$.
- For all $C_i \in \mathsf{Const}$ of arity n and terms $t_1, \ldots, t_n$, $C_i(t_1, \ldots, t_n) \in \mathsf{Term}(\Sigma, \mathsf{Var})$.
- For each $D_{ij} \in \mathsf{Dest}$ and each term t, $D_{ij}(t) \in \mathsf{Term}(\Sigma, \mathsf{Var})$.

Here, as well, we work with the set of ground terms $\mathsf{Term}(\Sigma)$, and we consider only **infinite** structures (Ground Term Algebra with Destructors). Meaning that either there exists a function symbol of arity$>$ 0 (non-constant), or there are infinitely many constant symbols.

Ground Term Algebra with Destructors. A structure $\mathcal{D}$ over $\Sigma \equiv (\mathsf{Const} \cup \mathsf{Dest}_{\mathsf{Const}}, =)$, is a ground term algebra with destructors if first its domain consists only of ground terms, i.e. $\mathtt{DM} \equiv \mathsf{Term}(\Sigma)$, and second, if for all $C, C' \in \mathsf{Const}$, all $D_{ij} \in \mathsf{Dest}_{\mathsf{Const}}$ and all $t, t_1, \ldots, t_n, s_1, \ldots, s_m \in \mathtt{DM}$ the following properties hold:

1. if $C \neq C'$, then $C_{\mathcal{D}}(t_1, \ldots, t_n) \not\equiv C'_{\mathcal{D}}(s_1, \ldots, s_m)$
2. if $(t_1, \ldots, t_n) \not\equiv (s_1, \ldots, s_n)$ then $C_{\mathcal{D}}(t_1, \ldots, t_n) \not\equiv C'_{\mathcal{D}}(s_1, \ldots, s_n)$
3. for all contexts $F \neq [\,]$: $t \not\equiv F[t]$
4. $D_{ij}(s) = t_j$ if $\exists t_1 ... t_n \in \mathtt{DM}$ such that $s = C_i(t_1, ..., t_n)$, otherwise $D_{ij}(s) = undefined$.

As defined above, a destructor D_{ij}, identifies a unary partial function, which un-conceals the jth element of t, when it operates on $C_i(t)$.

Definition 8 below, is motivated by the fact that when a CNF is set to $\mathtt{true}$ by some assignment α, then each of its clauses are set to $\mathtt{true}$ by α, since, otherwise the formula will not be satisfied by α. Hence, there have to be at least one literal inside each clause which evaluates to $\mathtt{true}$ by α. Therefore, adding a new literal to this clause will not damage the value of the whole clause under application of α, even if the added literal is false.

Definition 8. $\mathsf{dest{-}free}$ *is a function which operates over clauses and simplified formulas, as below:*

- $\mathsf{dest{-}free}(Cl) := \{l \in Cl \mid l$ *contains no destructor symbol*$\}$, *i.e. it removes from Cl all the literals which contain some destructor symbol in their sub-terms.*
- $\mathsf{dest{-}free}(\varphi) := \bigcup_{Cl \in \mathsf{Cls}(\varphi)} \mathsf{dest{-}free}(Cl)$, *which does the same over formulas.*

Example 1. Consider φ to be $\{\{D_{21}(C_2(x,z)) = z\}, \{C_1(s) \neq z, C_1(D_{21}(x)) \neq y\}, \{D_{21}(C_1(x,z)) \neq z\}, \{D_{12}(C_1(x,y)) = t\}\}$. Then, $\mathsf{dest{-}free}(\varphi) \equiv \{\{\}, \{C_1(x) \neq z\}, \{\}, \{\}\}$.

Lemma 4. *Given a formula φ, if* $\mathsf{dest{-}free}(\varphi)$ *is satisfiable then so is φ. Moreover, if $\alpha \not\equiv \bot$ is an assignment which proves* $\mathsf{dest{-}free}(\varphi)$*, then, it will prove φ, as well.*

Proof. See the Appendix. ■

It is noticeable that the converse of this lemma is not correct. For example $\varphi := \{\{D_{21}(C_2(x,z)) = x\}\}$ is satisfiable under any assignment but $\mathsf{dest{-}free}(\varphi)$ is not satisfiable. In the following parts, we will sometimes need to remove only those literals which contain a specific term, e.g. t. In such a case, there will be a call to the function $\mathsf{free}(., t)$. We define this function below:

Definition 9. *Given a formula φ and a term t,* $\mathsf{free}(\varphi, t)$ *is obtained from φ by removing all positive literals which contain a sub-term t; and further replacing all negative literals containing t with a valid term, e.g. $c_0 = c_0$ (c_0 is a constant symbol). We do this using the following algorithm:*

```
free($\varphi, t$)
    begin
        while $t$ occurs in $\varphi$
            choose a positive literal $a$ containing $t$;
            $\varphi := \{Cl - \{a\} \mid Cl \in \varphi\}$;
            choose a negative literal $l$ containing $t$;
            $\varphi := \varphi|_{l \to c_0 = c_0}$;
        end while
        return $\varphi$;
    end
```

This algorithm removes all positive literals which contain t as a sub-term. It then further substitutes negative literals, of this type, with a valid literal.

Lemma 5. *Given a formula φ and a term t,* $\mathsf{free}(\varphi, t)$ *terminates.*

Proof. φ only contains a finite number of occurrences of t. ■

Corollary 4. $t \notin \mathsf{Term}(\mathsf{free}(\varphi, t))$.

Proof. This is obvious, because φ has finitely many occurrences of t, and the algorithm continues only until it removes all occurrences of t. ■

Below we use the name idle for those terms which are obtained from applying a destructor on some constructors except its associated one or on some constant symbol, e.g. $D_{jk}(C_i(t_1, ..., t_n))$, where $j \not\equiv i$. for some destructor symbol D_{jk} and some constructor symbol C_i where $j \not\equiv i$. The first step to find a satisfying assignment for a formula, is to purify it from all the redundant sub-terms. To this end, we introduce an algorithm which simplifies the formula by removing all its idle [1] sub-terms.

Definition 10. *Given a formula φ, we define:*

```
Simplify($\varphi$)
    begin
        while there is a sub-term $D_{ik}(C_i(t_1, ..., t_n))$ occurring in $\varphi$
            $\varphi := \varphi|_{D_{ik}(C_i(t_1,...,t_n)) \to t_k}$;
        while there is an idle sub-term $t$ in $\varphi$
            $\varphi :=$ free($\varphi, t$);
        return $\varphi$;
    end
```

The Simplify algorithm removes positive literals containing subterms $D_{jk}(C_i(.))$ where $j \not\equiv i$, and replaces negative ones with a valid literal. This is because inequality will always hold if some idle term occurs in either side. Furthermore, the algorithm replaces $D_{ik}(C_i(.))$ with the term it entails. In Example 1, $\mathsf{Simplify}(\varphi) \equiv \{\{x{=}z\}, \{C_1(s) \neq z, C_1(D_{21}(x)) \neq y\}, \{c_0 = c_0\}, \{y = t\}\}$.

An immediate result of φ having only finitely many occurrence of destructor symbols, is that:

Lemma 6. *For any formula φ,* $\mathsf{Simplify}(\varphi)$ *terminates.*

Corollary 5. *If $D_{ij}(t) \in$* $\mathsf{Term}(\mathsf{Simplify}(\varphi))$ *then t is neither a constant symbol nor it contains any constructor symbols.*

[1] As far as we know, these terms are always being considered *undefined* in theories which contain destructors for more than one constructors (e.g. list structure with *nil* and *cons*). For more information on this see [19].

Proof. See the Appendix. ■

It is important to make sure that simplifying a formula φ via this algorithm maintains its satisfiability. Because, if it does then by knowing that $\mathsf{Simplify}(\varphi)$ is (un)satisfiable one can conclude that φ is (un)satisfiable too. Below, we prove even an stronger property of the Simplify algorithm, i.e. not only φ and $\mathsf{Simplify}(\varphi)$ are equi-satisfiable for any formula φ, but also they have the same set of models (assignments which prove them):

Lemma 7. *φ is satisfiable and $\alpha \models \varphi$ if and only if $\mathsf{Simplify}(\varphi)$ is satisfiable and $\alpha \models \mathsf{Simplify}(\varphi)$.*
Proof. See the Appendix. ■

Given a formula φ we identify by $\mathsf{Dest}(\varphi)$ the set of all sub-terms in φ which are of the form $D_{ij}(y)$ for some $i, j \in \mathbb{N}$ and some $y \in \mathsf{Var}$. In Example 1, $\mathsf{Dest}(\varphi) \equiv \{D_{21}(x)\}$ and $\|\mathsf{Dest}(\varphi)\|$ is 1.

So far we have introduced all the auxiliary procedures to be used inside the main algorithm in order to derive the result that we are looking for. The main algorithm is called $\mathsf{GDPLL}^{\mathsf{WD}}$:

Definition 11 **($\mathsf{GDPLL}^{\mathsf{WD}}$).** *For a formula φ which is possibly containing destructors, the following algorithm returns (SAT, α) if φ is satisfiable; and in this case α will be a proof for φ. The algorithm returns $(\mathsf{UNSAT}, \emptyset)$ if φ is not satisfiable:*

```
$\mathsf{GDPLL}^{\mathsf{WD}}(\varphi)$
    begin
        $\varphi := \mathsf{Simplify}(\varphi)$;
        $\psi := \mathsf{dest{-}free}(\varphi)$;
        if $\mathsf{GDPLL}^{\mathsf{W}}(\psi) \equiv (\mathsf{SAT}, \alpha)$ then return $(\mathsf{SAT}, \alpha)$ else
        while $\|\mathsf{Dest}(\varphi)\| \geq 1$
            choose $D_{ij}(y) \in \mathsf{Dest}(\varphi)$;
            $t := D_{ij}(y)$;
            $n := the\ airty\ of\ C_i$;
            choose $n$ fresh variables $z_1, ..., z_n$;
            $\psi := \varphi|_{y \to C_i(z_1,...,z_n)}$;
            if $\mathsf{GDPLL}^{\mathsf{WD}}(\psi) \equiv (\mathsf{SAT}, \alpha)$ then
                $\bar{\alpha} := \{y \mapsto \alpha(C_i(z_1, ..., z_n))\} \cup \{x \mapsto \alpha(x)$ if $x \not\equiv y\}$;
                return $(\mathsf{SAT}, \bar{\alpha})$;
            else
                $\varphi := \mathsf{free}(\varphi, t)$;
                $\varphi := \mathsf{Simplify}(\varphi)$;
        end while
        return $(\mathsf{UNSAT}, \emptyset)$;
    end
```

In this algorithm we replace each occurrence of a term y inside a destructor (e.g. $D_{ij}(y)$), with a term in the image of the destructor's associated constructor (e.g. $C_i(z_1, ..., z_n)$ for $D_{ij}(y)$).

Theorem 4 **(Termination).** *$\mathsf{GDPLL}^{\mathsf{WD}}$ is terminating over any formula φ in a theory of ground term algebra with destructors.*

Proof. See the Appendix. ■

Finally, the main property of our algorithm is that:

Theorem 5 **(Soundness and Completeness).** *For any formula φ in a theory of ground term algebra with destructors:*

– *φ is satisfiable iff $\mathsf{GDPLL}^{\mathsf{WD}}(\varphi)$ returns (SAT, α), where $\alpha \models \varphi$ and $\alpha \not\equiv \perp$.*

– *φ is unsatisfiable iff* $\mathsf{GDPLL}^{\mathsf{WD}}(\varphi)$ *returns* $(\mathsf{UNSAT}, \emptyset)$.

Proof. This theorem is thoroughly proved in the Appendix. ■

Below, we exemplify our technique by a simple example from the theory of list data structure, with constructors nil and $cons$ of arity 0 and 2 respectively, where nil is also the only constant symbol. Formuals will then be like: $cons(nil, cons(nil, nil))$, etc.

Example 2. Consider $\varphi \equiv \{\{nil{=}z,\ y{\neq}head(cons(nil, x))\}, \{nil{\neq}z\}, \{y{=}tail(z)\}\}$. We investigate satisfiability of φ, together with a proof for it, by applying the $\mathsf{GDPLL}^{\mathsf{WD}}$ algorithm.

Applying $\mathsf{Simplify}$ algorithm on φ, we obtain: $\varphi := \{\{nil{=}z,\ y{\neq}nil\}, \{nil{\neq}z\}, \{y{=}tail(z)\}\}$. $\mathsf{dest{-}free}$ of this formula will then be: $\psi := \{\{nil{=}z, y{\neq}nil\}, \{nil{\neq}z\},\ \{\}\}$. For this formula, we obtain: $\mathsf{GDPLL}^{\mathsf{W}}(\psi) \equiv (\mathsf{UNSAT}, \emptyset)$. Hence, according to the $\mathsf{GDPLL}^{\mathsf{WD}}$ algorithm, we should apply the next immediate loop, since $\|\mathsf{Dest}(\varphi)\| \geq 1$.

The only sub-term with occurrence of some destructor, in φ, is $t := tail(z)$. The associated constructor for $tail$ is $cons$ which is of arity 2. Hence, $n := 2$ and let z_1, z_2 be the two fresh variables. Then, in the next step we have $\psi := \varphi|_{z \to cons(z_1,z_2)}$. This is equivalent to the following formula: $\{\{nil{=}cons(z_1, z_2),\ y{\neq}nil\}, \{nil{\neq}cons(z_1, z_2)\}, \{y{=}tail(cons(z_1, z_2))\}\}$.

Now, in the recursive step, $\mathsf{GDPLL}^{\mathsf{WD}}(\psi)$ should be computed. Here, $\psi := \mathsf{Simplify}(\psi) = \{\{nil{=}cons(z_1, z_2),\ y{\neq}nil\}, \{nil{\neq}cons(z_1, z_2)\}, \{y{=}z_2\}\}$. This formula contains no destructor symbols, hence it is equivalent to its $\mathsf{dest{-}free}$ version. Therefore, $\mathsf{GDPLL}^{\mathsf{W}}(\psi)$ should now be computed: $(\psi, \sigma) := \mathsf{Reduce}(\psi) {\equiv} (\{\{z_2{\neq}nil\}\},\ \sigma := \{y \mapsto z_2\})$, because, we apply rules 5,3 and 4 from the reduction system (Definition 6), respectively. Therefore, $\mathsf{GDPLL}^{\mathsf{W}}(\psi) := (\mathsf{SAT}, \mathsf{Func}(\psi, \sigma))$. Computing $\mathsf{Func}(\psi, \sigma))$ with $F[\,] \equiv cons([\,], nil)$, results in $\{y, z_2 \ \mapsto cons(nil, nil)\} \cup \{x \ \mapsto nil \text{ if } x \in \mathsf{var},\ x{\not\equiv}y, z_2\}$. Finally back to the main $\mathsf{GDPLL}^{\mathsf{WD}}$ algorithm:

We derive $\bar{\alpha} := \{z \ \mapsto cons(nil, cons(nil, nil))\} {\cup} \{y, z_2 \ \mapsto cons(nil, nil)\} {\cup} \{x \ \mapsto nil \text{ if } x \in \mathsf{var},\ x \notin \{z, y, z_2\}\}$. Therefore, the final result is $\mathsf{GDPLL}^{\mathsf{WD}}(\varphi) {\equiv} (\mathsf{SAT}, \bar{\alpha})$.

4 Conclusion

The idea of extending the existing theorem provers to more expressive logics has recently attracted much research. Having a powerful theorem prover which tackles larger theories, will prevent users from the obligation of converting a formula to some propositional formula; and instead would provide a technique for validating formula by means of a direct approach. Among many different methods, extensions of DPLL algorithm have been of much interest (cf. Introduction). In this paper, we introduced an extension of the DPLL procedure to the theory of Ground Term Algebra with Destructors. Our algorithm provides witness assignment for satisfiable formulas and will terminate with an empty set of assignments in case the formula is not satisfiable. In practice, destructors are extensively used; they are almost a necessary element of those theories which employ constructors. Many programming languages support recursively defined data structures, e.g. in LISP. Destructors are also widely recognized by designers of security protocols (cf. Introduction).

The technique of backtracking is a feature of DPLL-based verification algorithms. Recently, a more efficient version of this technique, which is called backjumping (or non-chronological backtracking) is introduced. For future work, we plan to apply the ideas of backjumping and (conflict) clause learning into $\mathsf{GDPLL}^{\mathsf{WD}}$, and also to implement the algorithm.

5 Acknowledgement

My special thanks goes to Jaco van de Pol and Bas Luttik for their useful comments on the earlier version of this work.

References

1. W. Ackermann. *Solvable Cases of the Decision Problem*. Studies in Logic and the Foundations of Mathematics. 1954.
2. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
3. B. Badban. *Verification Techniques for Extensions of Equality Logic*. PhD thesis, Amsterdam Vrij University, 2006.
4. B. Badban, J.C. van de Pol, O. Tveretina, and H. Zantema. Generalizing DPLL and satisfiability for equalities. *Journal of Information and Computation*, 205(8):1188–1211, 2007.
5. C. Barrett and S. Berezin. CVC lite: A new implementation of the cooperating validity checker. In *Proc. 16th Conference on Computer Aided Verification*, LNCS 3114, pages 515–518, 2004.
6. Clark Barrett, Igor Shikanian, and Cesare Tinelli. An abstract decision procedure for satisfiability in the theory of recursive data types. *Electr. Notes Theor. Comput. Sci.*, 174(8), 2007.
7. Clark Barrett and Cesare Tinelli. CVC3. In *CAV*, pages 298–302, 2007.
8. C.W. Barrett, D.L. Dill, and J.R. Levitt. Validity checking for combinations of theories with equality. In *1st conference on Formal Methods in Computer Aided Design*, LNCS 1166, pages 187–201, 1996.
9. M.A. Bezem, J.W. Klop, and R.C. de Vrijer, editors. *Term Rewriting Systems*. Cambridge University Press, 2003.
10. R.E. Bryant, S. German, and M.N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Conference on Computer Aided Verification*, LNCS 1633, pages 470–482, 1999.
11. A. Colmerauer. Equations and inequations on finite and infinite trees. In ICOT staff, editor, *Proc. Conference on Fifth Generation Computer Systems*, pages 85–99. North-Holland, 1984.
12. H. Comon and P. Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7(3–4):371–425, 1989.
13. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.
14. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.
15. J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated canonizer and solver. In *Conference on Computer Aided Verification*, LNCS 2102, pages 246–249, 2001.
16. H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *Conference on Computer Aided Verification*, LNCS 3114, pages 175–188, 2004.
17. 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, 1987.
18. M.J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proc. 3rd Annual Symposium on Logic in Computer Science*, pages 348–357. IEEE Computer Society Press, 1988.
19. G. Nelson and D.C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.
20. R. Nieuwenhuis and A. Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *Conference on Computer Aided Verification*, LNCS 3576, pages 321–334, 2005.
21. Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.
22. Derek C. Oppen. Reasoning about recursively defined data structures. *J. ACM*, 27(3):403–411, 1980.
23. R. Pichler. On the complexity of equational problems in CNF. *Journal of Symbolic Computation*, 36(1-2):235–269, 2003.
24. J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–49, 1965.
25. N. Shankar and H. Rueß. Combining Shostak theories. In S. Tison, editor, *Proc. 13th Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science 2378, pages 1–18. Springer-Verlag, 2002.
26. R.E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583–585, 1978.
27. Robert E. Shostak. Deciding combinations of theories. *J. ACM*, 31(1), 1984.
28. A. Stump, C.W. Barrett, and D.L. Dill. CVC: A cooperating validity checker. In J.C. Godskesen, editor, *Proc. 14th Conference on Computer Aided Verification*, Lecture Notes in Computer Science 2404, pages 500–505, 2002.
29. C. Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In *Proc. Conference on Logics in Artificial Intelligence*, LNCS 2424, pages 308–319, 2002.
30. Ting Zhang, Henny B. Sipma, and Zohar Manna. Decision procedures for recursive data structures with integer constraints. In *IJCAR*, pages 152–167, 2004.

A Proofs of Lemmas and Theorems

Proof of Lemma 3.
The first two items are entailed in [3] Chapter 4.

3. If σ satisfies $\varphi \wedge \neg l$ then it will satisfy both φ and $\neg l$. Therefore, by item 2 above, we obtain that σ satisfies $\varphi|_{\neg l} \wedge \neg l$. Analogously, if σ satisfies $\varphi|_{\neg l} \wedge \neg l$ then σ satisfies $\varphi \wedge \neg l$. So, $\varphi \wedge \neg l$ is satisfiable if and only if $\varphi|_{\neg l} \wedge \neg l$ is satisfiable. Now, we benefit from yet another lemma in the abovementioned references: Given a literal l, φ is satisfiable iff either $\varphi \wedge l$ or $\varphi \wedge \neg l$ is satisfiable. Summing up these two latter results, we deduce item 3.
4. If the applied rule is either of $1, 2, 3$ then it is trivial. If rule 5 is applied then using item 2 above, this will be straightforward too. ■

Proof of Corollary1.
According to the algorithm, the value of σ is never $\perp$. For the rest of the proof we consider a case distinction:

- If $\mathsf{Reduce}_1(\varphi) \not\equiv \perp$. Then rule 4 is applicable: It can easily be observed that none of the rules $1, 2, 3$ in Definition 6 add more positive unit clauses to a formula, and rule 5 only removes some *positive* literal. This, too will not cause any new purely positive clause. Therefore, rule 4 is applicable atmost one time. According to the algorithm, rule 4 is the only rule which effects the value of σ. So, if we consider $\varphi \equiv \varphi_1 \uplus \varphi_2$ with the same meanings as in rule 4 of the Reduction System, then $\sigma \equiv \mathsf{imgu}(\varphi_1)$, since $\sigma \not\equiv \emptyset$. Hence, $\sigma \models \varphi_1$ and $\sigma^2 \equiv \sigma$. Now,
  - If $\varphi \xrightarrow{1,2,3,5} \varphi'$ then by Lemma 3(4), if $\beta \models \varphi'$ then $\beta \models \varphi$.
  - If $\varphi \xrightarrow{4} {\varphi_2}^{\sigma}$ (it is noticeable that this φ might be different than the original one) and if $\beta \models {\varphi_2}^{\sigma}$ then $\beta.\sigma \models \varphi_2$. On the other-hand $\sigma \models \varphi_1$ (above), therefore $\beta.\sigma \models \varphi_1$. These two result in $\beta.\sigma \models \varphi_1 \uplus \varphi_2$, and hence $\beta.\sigma \models \varphi$.

  Now, since rule 4 is applicable atmost once, then the reduction sequence will be something like $\varphi \xrightarrow{1,2,3,5} \cdots \xrightarrow{1,2,3,5} \psi \xrightarrow{4} \psi' \xrightarrow{1,2,3,5} \cdots \xrightarrow{1,2,3,5} \mathsf{Reduce}_1(\varphi)$. So using the two items obtained above, we get: if $\beta \models \mathsf{Reduce}_1(\varphi)$ then $\beta \models \psi'$ and $\beta.\sigma \models \psi$ and hence $\beta.\sigma \models \varphi$. Notice that if rule 4 is not applicable then $\sigma \equiv \emptyset$ and $\beta.\sigma \equiv \beta$, and hence proof is complete according to Lemma 3(4).

- If $\mathsf{Reduce}_1(\varphi) \equiv \perp$ then there is no such assignment β. ■

Example 3. We show how to reduce $\varphi \equiv \{\{x \neq f(a,b)\}, \{x = f(y,z)\}, \{y = a, x = f(a,b)\}\}$ using the Reduce algorithm.
In the first step $\alpha := \emptyset$ and so since rules 5 and 4 are applicable on φ we should enter the **while** loop. Let us choose rule 4, then according to the algorithm, we get: $\varphi_1 := \{\{x = f(y,z)\}\}$,
$\varphi := \{\{x = f(y,z)\}\} \uplus \{\{x \neq f(a,b)\}, \{y = a, x = f(a,b)\}\}$,
$\sigma := \{x \mapsto f(y,z)\}$, $\varphi := \{\{f(y,z) \neq f(a,b)\}, \{y = a, f(y,z) = f(a,b)\}\}$ and
$\alpha := \sigma\alpha \equiv \sigma\emptyset \equiv \{x \mapsto f(y,z)\}$. Now rule 3 is the only applicable rule on φ. So by applying this rule we get $\varphi := \{\{y \neq a, z \neq b\}, \{y = a, f(y,z) = f(a,b)\}\}$. We can see that none of the rules are applicable on φ anymore, therefore the algorithm stops and returns

$$(\{\{y \neq a, z \neq b\}, y = a, f(y,z) = f(a,b)\}\}, \{x \mapsto f(y,z)\}).$$

Obviously depending on the rules we choose to rewrite φ with, we may derive different results from this algorithm.

Proof. of Corollary 2.
Clearly, $\mathsf{Func}(\psi,\sigma) \equiv \alpha.\sigma$ where $\alpha \not\equiv \bot$ is the assignment which is built inside $\mathsf{Func}(\psi,\sigma)$. Then, using Corollary 1, we immediately obtain that $\alpha.\sigma \not\equiv \bot$ (since $\sigma \not\equiv \bot$), and also in order to derive $\alpha.\sigma \models \varphi$ we only need to show that $\alpha \models \psi$ (since $\psi \equiv \mathsf{Reduce}_1(\varphi)$). Below, we prove that indeed $\alpha \models \psi$:

Assume that $(x,i) \in X$ for a literal $x \neq u$ which is picked up from a clause $Cl \in \mathsf{Cls}(\psi)$. Then, $\alpha(x) \equiv F^{iM}(c)$ where $i \geq 1$, $M \geq depth(u)+1$. To show that $\alpha \models x \neq u$ (or equivalently, $\alpha(x) \not\equiv \alpha(u)$), we first notice that $depth(\alpha(x)) \equiv i.M$ and then we consider two separate cases:

- Assume that $\mathsf{Var}(u) \cap X_1 \equiv \emptyset$, or if $y \in \mathsf{Var}(u) \cap X_1$ then $depth(y)$ does not effect the value of $depth(u)$. Meaning that, if $depth(u) \equiv 1 + depth(t)$ for some subterm $t \in u$ then $y \notin \mathsf{Var}(t)$. Therefore, we will have $depth(\alpha(u)) \equiv 1 + depth(\alpha(t)) \overset{ii}{\equiv} 1 + depth(t) \equiv depth(u)$ (ii. when $z \notin X_1$ then $\alpha(z) \equiv c$). So, $depth(\alpha(u)) \equiv depth(u) < depth(u)+1 \leq M \leq M.i \equiv depth(\alpha(x))$. Therefore $\alpha(x) \not\equiv \alpha(u)$, which means that $\alpha \models x \neq u$.

- If $\exists z \in \mathsf{Var}(u) \cap X_1$, where $depth(\alpha(u)) \equiv depth(u) + depth(\alpha(z))$. For such a variable, there exists $j \in \mathbb{N}$ such that $(z,j) \in X$, and therefore $depth(\alpha(z)) \equiv j.M$. Now, since $\psi \equiv \mathsf{Reduce}_1(\varphi)$, therefore by Lemma 2(3) $x \neq u$ is in solved-form, i.e. $x \notin u$. So $z \not\equiv x$ and hence $j \not\equiv i$. Now if $\alpha(x) \equiv \alpha(u)$, then by applying the $depth$ function on both sides we will get $i.M \equiv depth(u) + j.M$. This entails that $i \geq j$ and so $i > j$ (since $i \not\equiv j$), therefore $i \geq j+1$ since $i,j \in \mathbb{N}$. So, $depth(u) \equiv (i-j).M \geq M$ (since $i \geq j+1$) $\geq depth(u)+1$. Because, this false result is obtained from assuming that $\alpha(x) \equiv \alpha(u)$, hence it yields that $\alpha(x) \not\equiv \alpha(u)$. Therefore, $\alpha \models x \neq u$.

Now, since $\psi \equiv \mathsf{Reduce}_1(\varphi)$ hence by Lemma 2, it contains no purely positive clause or $\bot$, and all of its negative literals are of solved-form. Thus each of its clauses contains some negative literal, of solved-form. Hence, from each clause a negative literal $x \neq u$ is selected for which $\alpha \models x \neq u$, by the above results. Therefore, α satisfies at least one literal from each clause in ψ, so $\alpha \models \psi$. ■

Termination Proof, Theorem 2.
We benefit from the following two results, entailed in [3] Chapter 4, and [4]: 1) The Reduction System of Definition 6 is terminating over any formula φ. 2) $pos(\varphi \wedge (t=u)) < pos(\varphi)$ and $pos(\varphi|_{t\neq u} \wedge (t \neq u)) < pos(\varphi)$, where pos counts the number of occurrences of positive literals in the formula. So far we can deduce:

- $\mathsf{Reduce}(\varphi)$ terminates.
- $\mathsf{Func}(\varphi,\sigma)$ and $\mathsf{Cons}(\varphi,\sigma)$ terminate, since φ contains only finite number of clauses.

Now, we use induction over the number of positive literals. If $pos(\varphi) \equiv 0$ then the algorithm terminates, because of the two items above. We assume that the algorithm terminates for all formulas with $pos(\varphi) \leq n$, and we show that it will terminate also for formulas with $pos(\varphi) = n+1$. $\mathsf{GDPLL}^{\mathsf{W}}(\varphi \wedge a)$ and $\mathsf{GDPLL}^{\mathsf{W}}(\varphi|_{\neg a} \wedge \neg a)$ are terminating. We know that $pos(\varphi \wedge (t=u)) < pos(\varphi)$ and so $pos(\varphi \wedge (t=u)) \leq n$, analogously for $\varphi|_{t\neq u} \wedge (t \neq u)$. So, by the induction hypothesis, the two algorithms $\mathsf{GDPLL}^{\mathsf{W}}(\varphi \wedge (t=u))$ and $\mathsf{GDPLL}^{\mathsf{W}}(\varphi|_{t\neq u} \wedge (t \neq u))$ would terminate; Therefore, $\mathsf{GDPLL}^{\mathsf{W}}(\varphi)$ will terminate. ■

Soundness and Completeness proof of $\mathsf{GDPLL}^{\mathsf{W}}$, Theorem 3.
We start with the first item:

– We prove each side of the first item separately:
  ⇒ Suppose φ is satisfiable, and $\mathsf{Reduce}(\varphi) \equiv (\mathsf{Reduce}_1(\varphi),\sigma)$. We show that $\mathsf{GDPLL}^{\mathsf{W}}(\varphi)$ returns (SAT,α), where $\alpha \not\equiv \bot$ is a satisfying assignment for φ. Using Theorem 1(2), we derive

that $\mathsf{Reduce}_1(\varphi)$ is satisfiable and so, $\perp \notin \mathsf{Reduce}_1(\varphi)$. We continue the proof by using induction, we show that the theorem holds for formulas φ with $pos(\mathsf{Reduce}_1(\varphi)) \equiv 0$. Then we prove that if the theorem holds for all formulas with $pos(\mathsf{Reduce}_1(\varphi)) \leq n$ then it will also hold for those with $pos(\mathsf{Reduce}_1(\varphi)) \equiv n + 1$. literals in ψ. For the sake of simplicity, below we will use ψ to denote $\mathsf{Reduce}_1(\varphi)$:

1. If $pos(\psi) \equiv 0$ then according to the algorithm, either $\mathsf{GDPLL}^{\mathsf{W}}(\varphi) \equiv (\mathsf{SAT}, \mathsf{Func}(\psi, \sigma))$ or $\mathsf{GDPLL}^{\mathsf{W}}(\varphi) \equiv (\mathsf{SAT}, \mathsf{Cons}(\psi, \sigma))$, since $\perp \notin \psi$.

   a. If $\mathsf{GDPLL}^{\mathsf{W}}(\varphi) \equiv (\mathsf{SAT}, \mathsf{Func}(\psi, \sigma))$ then according to Corollary 2, $\mathsf{Func}(\psi, \sigma) \equiv \alpha.\sigma$ for some assignment α, $\mathsf{Func}(\psi, \sigma) \not\equiv \perp$ and $\alpha.\sigma$ satisfies ψ (i.e. $\mathsf{Reduce}_1(\varphi)$). Thus, using Corollary 1, $(\alpha.\sigma).\sigma$ satisfies φ, besides $\sigma^2 \equiv \sigma$. Because the functional operation $(.)$ is associative, we obtain that $\alpha.\sigma^2$ and thus $\alpha.\sigma$ satisfies φ. Summing these up, we obtain that $\mathsf{Func}(\psi, \sigma)$ is not equivalent to $\perp$ and it satisfies φ.

   b. If $\mathsf{GDPLL}^{\mathsf{W}}(\varphi) \equiv (\mathsf{SAT}, \mathsf{Cons}(\psi, \sigma))$, then similar to **a**, one can deduce that $\mathsf{Cons}(\psi, \sigma)$ is not equivalent to $\perp$ and it satisfies φ.

2. If $pos(\psi) \equiv n + 1$ and, as the induction hypothesis, for each satisfiable formula χ with $pos(\mathsf{Reduce}_1(\chi)) \leq n$ the theorem holds, i.e. $\mathsf{GDPLL}^{\mathsf{W}}(\chi) \equiv (\mathsf{SAT}, \alpha)$ where $\alpha \not\equiv \perp$ satisfies χ. Then, we show that the theorem holds also for φ:

   a. If ψ contains no purely positive clauses, then the rest of the proof will be similar to the previous case (when $pos(\psi) \equiv 0$).

   b. If ψ contains some purely positive clauses, then we chose a positive literal a from one of its purely positive clauses, and continue as it does in the $\mathsf{GDPLL}^{\mathsf{W}}$ algorithm. By Lemma 3(3), either $\psi \wedge a$ or $\psi|_{\neg a} \wedge \neg a$ is satisfiable.

   * If $\psi \wedge a$ is satisfiable: By Theorem 1(1), $pos(\mathsf{Reduce}_1(\psi \wedge a)) < pos(\psi) \equiv n + 1$ therefore using induction hypothesis we derive: $\mathsf{GDPLL}^{\mathsf{W}}(\psi \wedge a) \equiv (\mathsf{SAT}, \alpha)$ and $\alpha \not\equiv \perp$ satisfies $\psi \wedge a$, and hence satisfies ψ. Thus, $\mathsf{GDPLL}^{\mathsf{W}}(\varphi) \equiv (\mathsf{SAT}, \alpha.\sigma)$ where according to Corollary 1, $\alpha.\sigma$ satisfies φ with $\sigma \not\equiv \perp$, and therefore $\alpha.\sigma \not\equiv \perp$.
   * If $\psi|_{\neg a} \wedge \neg a$ is satisfiable: Then the rest of the proof is analogous to the previous case, except that this time $\mathsf{GDPLL}^{\mathsf{W}}(\psi|_{\neg a} \wedge \neg a) \equiv (\mathsf{SAT}, \alpha)$ where α satisfies $\psi|_{\neg a} \wedge \neg a$. Using Lemma 3 (2), we get α satisfies ψ. Therefore, similar to above, $\mathsf{GDPLL}^{\mathsf{W}}(\varphi) \equiv (\mathsf{SAT}, \alpha.\sigma)$ where $\alpha.\sigma \not\equiv \perp$ satisfies φ.

$\Leftarrow$ Suppose $\mathsf{GDPLL}^{\mathsf{W}}(\varphi) \equiv (\mathsf{SAT}, \alpha)$. Similar to the other direction above ($\Rightarrow$), it cab be proved that $\alpha \not\equiv \perp$ and $\alpha \models \varphi$.

– This is an immediate result of Theorem 2 together with the first item. ■

Example 4. Consider $\varphi \equiv \{\{f(x){=}y, g(y){=}h(z,x)\}, \{x{\neq}z\}\}$. We build an assignment α such that $\alpha \models \varphi$.

Applying $\mathsf{GDPLL}^{\mathsf{W}}$ algorithm on φ, first we get $(\varphi, \alpha) := \mathsf{Reduce}(\varphi) \equiv (\varphi, \emptyset)$, because none of the rules of the reduction system (Definition 6) are applicable.

Now we can see that $\perp \notin \varphi$ and also φ has a purely positive clause. Therefore according to the algorithm we need to choose a positive literal $a \in \mathtt{PLit}(\varphi)$. We choose $a := g(y) = h(z, x)$. Then we need to compute $\mathsf{GDPLL}^{\mathsf{W}}(\varphi \wedge (g(y) = h(z, x)))$ where $\varphi \wedge a \equiv \{\{f(x) = y, g(y) = h(z, x)\}, \{x \neq$

$z\}, \{g(y) = h(z,x)\}\}$. The first immediate step is to compute $\mathsf{Reduce}(\varphi \wedge (g(y) = h(z,x)))$. Rule 4 of the reduction system is applicable. By applying this rule on $\varphi \wedge (g(y) = h(z,x))$ we get $\sigma := \bot$, and hence it should be reduced to $\bot$. Therefore according to the algorithm $\mathsf{Reduce}(\varphi \wedge (g(y) = h(z,x))) \equiv (\bot, \emptyset)$. So $\mathsf{GDPLL}^{\mathsf{W}}(\varphi \wedge (g(y) = h(z,x))) \equiv (\mathsf{UNSAT}, \emptyset)$.

Hence according to the $\mathsf{GDPLL}^{\mathsf{W}}$ algorithm we have to compute $\mathsf{GDPLL}^{\mathsf{W}}(\varphi|_{g(y)\neq h(z,x)} \wedge (g(y) \neq h(z,x)))$. By Definition 1 we have $\varphi|_{g(y)\neq h(z,x)} \equiv \{\{f(x) = y\}, \{x \neq z\}\}$, and so $\varphi|_{g(y)\neq h(z,x)} \wedge (g(y) \neq h(z,x)) \equiv \{\{f(x) = y\}, \{x \neq z\}, \{g(y) \neq h(z,x)\}\}$. As the first immediate step we imply the Reduce algorithm to get a reduced form. Rule 4 is the only applicable rule; we derive $\sigma \equiv \{y \mapsto f(x)\}$ and $\{\{x \neq z\}, \{g(f(x)) \neq h(z,x)\}\}$. Now, rule 3 can be applied on $g(f(x)) \neq h(z,x)$. Here, we obtain $\mathsf{imgu}(g(f(x)) = h(z,x))$ which is equivalent to $\bot$. So, the formula should be reduced to $\{\{x \neq z\}\}$, and then, $(\{\{x \neq z\}\}, \{y \mapsto f(x)\})$ would be the outcome of this algorithm, because no other rule of the reduction system is applicable. This means that $(\varphi, \sigma) \equiv (\{\{x \neq z\}\}, \{y \mapsto f(x)\})$. Now, $\bot \notin \varphi$ so we continue with the next step. φ contains no purely positive clauses, and there exist some non-constant function symbols in the theory. So, the next step is to compute $\mathsf{Func}(\varphi, \sigma)$. Inside the algorithm, $C := \{x \neq z\}$ and $(x \neq z) \in C$ are the only possible choices. Thereafter, we obtain: $\varphi{:=}\emptyset$, $i{:=}1$, $X_1{:=}\{x\}$, $X{:=}\{(x,1)\}$ and $M{:=}max(0, depth(z)+1) \equiv 1$.

We choose $F[\,] \equiv g([\,])$ as the function symbol, and also c as the constant symbol. So, $F^1(c) \equiv g(c)$ and therefore $\alpha := \{x \mapsto g(c)\} \cup \{z \mapsto c \mid z \neq x\}$. Hence $\alpha.\sigma := \{x \mapsto g(c), y \mapsto f(g(c))\} \cup \{z \mapsto c \mid z \notin \{x,y\}\}$, for $\sigma \equiv \{y \mapsto f(x)\}$. So that $\mathsf{Func}(\varphi, \sigma) \equiv \alpha.\sigma \equiv \{x \mapsto g(c), y \mapsto f(g(c))\} \cup \{z \mapsto c \mid z \notin \{x,y\}\}$. Therefore the main $\mathsf{GDPLL}^{\mathsf{W}}$ algorithm returns

$$(\mathsf{SAT}, \{x \mapsto g(c),\ y \mapsto f(g(c))\} \cup \{z \mapsto c \mid z \notin \{x,y\}\})$$

as the outcome. Meaning that the original formula φ is satisfiable and

$$\alpha := \{x \mapsto g(c), y \mapsto f(g(c))\} \cup \{z \mapsto c \mid z \notin \{x,y\}\}$$

is a proof assignment to that (i.e. it satisfies φ).

Proof. of Lemma 4.
According to Definition 8 $\mathsf{dest{-}free}(\varphi) \equiv \bigcup_{Cl \in \mathsf{Cls}(\varphi)} \mathsf{dest{-}free}(Cl)$. Now, $\alpha \not\equiv \emptyset$ proves $\mathsf{dest{-}free}(\varphi)$ and hence α proves each clause in $\mathsf{dest{-}free}(\varphi)$. Since empty clause is unsatisfiable, this would mean that each clause $\mathsf{dest{-}free}(Cl)$ in $\mathsf{dest{-}free}(\varphi)$ contains a literal which is satisfied by α. Each clause Cl of φ is obtained by adding certain literals to $\mathsf{dest{-}free}(Cl)$. These literals, even in case they are being unsatisfiable, will not change the value of Cl, and therefore the value of φ, under α. Hence, α will satisfy φ. This shows that φ is satisfiable since $\alpha \not\equiv \bot$. ■

Proof. of Corollary5.
The $\mathsf{Simplify}$ algorithm continues as long as the formula contains some sub-term which is of $D_{lm}(C_l(...))$ or idle form. So, obviously the outcome of the algorithm cannot be simplified further, using the $\mathsf{Simplify}$ algorithm.

If t is a constant symbol then $D_{ij}(t)$ would be an idle term, and hence it will be removed from φ by the algorithm. This contradicts the fact that φ can no more be simplified. So, t cannot be a constant symbol. Now, we show that if $t \equiv f(s)$ where s is neither a constant symbol nor it contains any constructor, then so is t. If $f \equiv C_k$ for some constructor symbol C_k, then $D_{ij}(t)$ is either an idle term (when $k \not\equiv i$) or $k \equiv i$. If the first case holds then $D_{ij}(t)$ will be removed by the algorithm, and if the latter case holds then $D_{ij}(t)$ will be further simplified. These contradicts the fact that φ cannot be simplified further. ■

Proof. of Lemma7.
It is noticeable that we consider any negative literal which contains some idle subterm to be valid under any assignment; also, any positive literal containing some idle subterm to be invalid under any assignment.

Regarding this, we only need to prove that α proves a literal which contains some subterms of the form $D_{lm}(C_l(t_1, \dots, t_n))$ if and only if then it will still satisfy the literal after replacing this subterm with t_m. According to our definition of destructors, $D_{lm}(C_l(t_1, \dots, t_n))$ is equivalent to t_m and hence this replacement must be eligible under any assignment. So, doing this replacement will not effect the value of α over the literal. This completes the proof. ■

Termination Proof, Theorem 4.
According to Lemma 6 $\mathsf{Simplify}(\varphi)$ is terminating. So without loss of generality we can assume that φ is simplified, i.e. $\mathsf{Simplify}(\varphi) \equiv \varphi$. We use induction on $\|\mathsf{Dest}(\varphi)\|$. If $\|\mathsf{Dest}(\varphi)\| \equiv 0$ then the algorithm is terminating because of Theorem 3. Now, if the algorithm terminates for all formulas with $\|\mathsf{Dest}(\varphi)\| \leq n$, then we show that it terminates also for formulas with $\|\mathsf{Dest}(\varphi)\| \equiv n + 1$.

As an immediate result of Corollary 5, each subterm involving some destructor symbol should be of the form $D_{ij}(t)$ where t can either be a variable or it can contain some other destructor symbols. We distinct two cases: 1. when all such t terms in φ are variables 2. when there is such a term t which contains yet another destructor, so $t \equiv D_{lm}(s)$ for some term s and $l, m \in \mathbb{N}$. Below, we prove termination for each of these two cases separately:

1. The inner variable of these terms are replaced by an associated constructor, the resulting term would be something like $D_{lm}(C_l(z_1, ..., z_n))$. Then in the next step $\mathsf{GDPLL}^{\mathsf{WD}}$ is applied on the resulting formula ψ. Inside this $\mathsf{GDPLL}^{\mathsf{WD}}$, first ψ is being simplified by $\mathsf{Simplify}$. It is noticeable that $\|\mathsf{Dest}(\mathsf{Simplify}(\psi))\| < \|\mathsf{Dest}(\varphi)\|$, because of this recent replacement. Therefore, by induction hypothesis $\mathsf{GDPLL}^{\mathsf{WD}}(\psi)$ terminates. If $\mathsf{GDPLL}^{\mathsf{WD}}(\psi) \equiv (\mathsf{UNSAT}, \emptyset)$, then φ is replaced with $\mathsf{free}(\varphi, D_{lm}(y))$. It is obvious that $\|\mathsf{Dest}(\mathsf{free}(\varphi, D_{lm}(y)))\| < \|\mathsf{Dest}(\varphi)\|$. Therefore, by the induction hypothesis $\mathsf{GDPLL}^{\mathsf{WD}}$ terminates over this recent formula. Hence, the original algorithm will terminate.
2. We prove this case using induction on destructor-depth of the formula, which will be the maximum layers of destructors occurred in its subterms. According to the case 1. above, the algorithm terminates if the destructor-depth of the formula is not more than 1. Supposing that $\mathsf{GDPLL}^{\mathsf{WD}}(\varphi)$ terminates for each φ where destructor-depth$(\varphi) \leq k$, we prove termination for when destructor-depth$(\varphi) \equiv k+1$. Assume that φ contains a sub-term $D_{ij}(t)$ with the maximum layers of destructors which is $k + 1$. The innermost sub-term of this term is of the form $D_{lm}(y)$ for some variable y, because of Corollary 5. According to the algorithm, $D_{lm}(y)$ will be chosen in the **while** loop, and its corresponding variable y will be replaced with its associated constructor over some fresh variables. So the result will be $D_{lm}(C_l(z_1, ..., z_n))$ for some fresh variables $z_1, ..., z_n$. Obviously, inside the next immediate call to $\mathsf{GDPLL}^{\mathsf{WD}}$ this term will have a destructor-depth of less or equal to k. Similarly, each time one of these terms with $k + 1$ layers of destructors are being chosen they will be reduced to a term with less destructor symbols. There are only finite number of such sub-terms inside φ. Hence, after finite steps they all will be reduced to some term with a smaller destructor-depth. Thus, after finite number of steps, we will have destructor-depth$(\varphi) \leq k$. Therefore, by the induction hypothesis, $\mathsf{GDPLL}^{\mathsf{WD}}$ will terminate. ■