J Autom Reasoning (2016) 56:387–457
DOI 10.1007/s10817-015-9352-2

Adding Decision Procedures to SMT Solvers Using Axioms with Triggers

Claire Dross[1,2,3] · **Sylvain Conchon**[1,2] · **Johannes Kanig**[3] · **Andrei Paskevich**[1,2]

Received: 31 May 2013 / Accepted: 30 October 2015 / Published online: 17 November 2015

Abstract Satisfiability modulo theories (SMT) solvers are efficient tools to decide the satisfiability of ground formulas, including a number of built-in theories such as congruence, linear arithmetic, arrays, and bit-vectors. Adding a theory to that list requires delving into the implementation details of a given SMT solver, and is done mainly by the developers of the solver itself. For many useful theories, one can alternatively provide a first-order axiomatization. However, in the presence of quantifiers, SMT solvers are incomplete and exhibit unpredictable behavior. Consequently, this approach can not provide us with a complete and terminating treatment of the theory of interest. In this paper, we propose a framework to solve this problem, based on the notion of *instantiation patterns*, also known as *triggers*. Triggers are annotations that suggest instances which are more likely to be useful in proof search. They are implemented in all SMT solvers that handle first-order logic and are included in the SMT-LIB format. In our framework, the user provides a theory axiomatization with triggers, along with a proof of completeness and termination properties of this axiomatization, and obtains a sound, complete, and terminating solver for her theory in return. We describe and prove a corresponding extension of the traditional Abstract DPLL Modulo Theory framework. Implementing this mechanism in a given SMT solver requires a one-time development effort. We have implemented the proposed extension in the Alt-Ergo prover and we discuss some implementation details in the paper. To show that our framework can handle complex theories, we prove completeness and termination of a feature-rich axiomatization of doubly-linked lists. Our tests show that our approach results in a better performance of the solver on goals that stem from the verification of programs manipulating doubly-linked lists and sets.

Keywords Automated deduction · Satisfiability modulo theory · Decision procedures

Mathematics Subject Classification 03B10 · 03B25 · 03B35 · 68T15

✉ Claire Dross
dross@adacore.com

1 LRI, CNRS, Université Paris-Sud 11, 91405 Orsay, France

2 Toccata, INRIA Saclay-Île de France, 91893 Orsay, France

3 AdaCore, 75009 Paris, France

1 Introduction

It is often the case that satisfiability problems refer to elements to which a special meaning is associated, such as linear arithmetic, arrays, bit-vectors, etc. Satisfiability modulo theories (SMT) solvers are efficient tools for deciding satisfiability of formulas modulo *background theories* describing the meaning of those elements. In addition, they usually are decision procedures for the satisfiability of quantifier-free formulas in the theories they support. Unfortunately for the user, her SMT solver of choice may not support her theory of interest and, of course, many theories can be designed that are not supported by any solver. Adding a background theory to an SMT solver is a complex and time-consuming task that requires internal knowledge of the solver and often access to its source code.

For many useful theories, one can alternatively provide a first-order axiomatization to the SMT solver, provided it handles quantifiers. To give some examples, Simplify [15], CVC3 [19], CVC4 [4], Z3 [13], and Alt-Ergo [8] support first-order logic. Of course, any automated prover is at best semi-complete on first-order problems and even semi-completeness is unattainable when non-trivial background theories, like arithmetic, are involved. To improve the chances of finding a proof, most SMT solvers give the user some control over instantiation of quantified formulas, by allowing to annotate quantifiers with so-called *instantiation patterns* also known as *triggers*.

The basic idea behind triggers is that the solver maintains a set of "known" terms (which usually are simply the terms occurring in assumed facts) and for instantiation to take place, a known term must match the pattern. It has been demonstrated that by careful restriction of instance generation in a first-order theory—in a way that can be expressed via instantiation patterns—one can both preserve completeness and ensure termination, thus obtaining a decision procedure for the theory. The most prominent example is the decision procedure for the theory of functional arrays by Greg Nelson [30], which we will consider in greater detail below. More recently, the same work has been done for specification of more complex data-structures [10,28].

Unfortunately, the user cannot hope to prove that a given first-order SMT solver is complete and terminating on a particular set of axioms with triggers for her theory of interest. Triggers are not and were never meant to change the satisfiability of a first-order formula. Instantiation patterns are rather considered as hints to what instances are more likely to be useful, and an SMT solver can base its decisions on the triggers given by the user as well as on the triggers that it infers itself using some heuristic. In pursuit of completeness, a solver has the right to use any instantiation strategy it deems useful, and it may even ignore the triggers altogether.

And yet if we want our axiomatization to give us a decision procedure, we must be able to control instantiation of axioms in a precise and reliable manner.

Contribution In this paper, we propose a framework to add a new background theory to an SMT solver by providing a first-order axiomatization with triggers. In order to restrict instantiation in a deterministic way, we give a formal semantics to formulas with triggers, which promotes triggers to the status of guards, forbidding all instances but the ones described by the pattern.

We then consider the well-known Abstract DPLL Modulo Theory framework [6,31], a standard theoretic model of modern SMT solvers. We describe a variation of this framework that handles first-order formulas with triggers. We show that for any axiomatization that meets three conditions of *soundness*, *completeness*, and *termination*, a compliant SMT solver behaves as a decision procedure for this axiomatization.

More precisely, consider an SMT solver which effectively decides quantifier-free problems in some background theory T. In the simplest case, T can be the theory of equality and uninterpreted function symbols (EUF). It can also be the theory of linear arithmetic, bit vectors, associative arrays, or any combination of the above. A user of that prover wants to extend T with some new theory—for example, that of mutable container data structures—and obtain a decision procedure for the ground problems in this extended theory which we denote T'. To this purpose, the user writes down a set of first-order axioms with triggers and proves that this axiomatization is a sound, complete, and terminating representation of T' in T. Since the three conditions are formulated in purely logical terms, no specific knowledge of inner prover mechanisms is required to do that proof. Now, provided that the solver implements our extension of DPLL(T)—or any other method that treats axioms with triggers in accordance with our semantics—the solver is guaranteed to decide any quantifier-free problem in T' in a finite amount of time.

The method described in this paper is not intended to extend ground SMT solvers to first-order logic nor to replace usual quantifier handling heuristics in first-order SMT solvers. We do not either strive to give some ultimate semantics for triggers, on which all first-order SMT solvers should converge. Our restrictive and rigorous treatment of quantifiers and triggers should be only applied to the axioms of the theory we wish to decide, and not to first-order formulas coming with a particular problem. Indeed, while we must restrict instantiation in the former case to guarantee termination, we would gain nothing by applying the same restrictions to ordinary first-order formulas. On the contrary, we are likely to prevent the solver from finding proofs which otherwise would be discovered, and, moreover, the additional checks needed to implement the restrictions will hinder the solver's performance.

We have implemented our extension of DPLL(T) in the first-order SMT solver Alt-Ergo. In our case-study—a sound, complete, and terminating theory of imperative doubly-linked lists—our implementation, in addition to give us a decision procedure for that theory, turns out to be more efficient than the generic handling of first-order formulas in Alt-Ergo on our axiomatization. This improvement is mostly due to the fact that our procedure favors instantiation over decision, which is generally a bad strategy for potentially non-terminating axiomatizations.

This paper continues and extends our previous work on triggers [16]. The proposed formalism includes using literals instead of terms as triggers which is necessary for e.g. extensionality axioms, as is shown below. We give a formal treatment to the notion of termination and we generalize it to admit axiomatizations that can produce an infinite number of instances. Finally, our method has been implemented in a mainstream SMT solver and we report here on our experiments with this implementation.

Overview We start the technical development in Sect. 2 by introducing a formal semantics for first-order logic with a notation for triggers that restrict instantiation. Using this semantics, we define, independently from a specific solver's implementation but modulo its background theory, three properties of a set of first-order axioms with triggers—namely, *soundness*, *completeness*, and *termination*—that are required for a solver to behave as a decision procedure for this axiomatization.

In Sect. 3, we give a fairly exhaustive axiomatization for imperative doubly-linked lists as an example. We provide completeness and termination proofs of this axiomatization in our framework.

In Sect. 4, we give advice for designing axiomatizations as well as for getting through the proofs of termination and completeness. In particular, we give several techniques that can be computer assisted using a decision procedure for the background theory T.

In Sect. 5, we extend the well-known Abstract DPLL Modulo Theory framework [31] to handle such axiomatizations. We show that our version of DPLL can effectively decide the satisfiability of ground formulas in an extension of the solver's background theory, whenever this extension is defined by a sound, complete, and terminating axiomatization.

In Sect. 6, we present an implementation of our framework inside the first-order SMT solver Alt-Ergo. The Sects. 5 and 6 are independent from the rest of the article and can be skipped by readers that are not interested in solver development.

Section 7 is dedicated to an experimental validation of our implementation. We show that the overall performance of the prover for typical proof obligations has been improved for our example theory of doubly-linked lists and then we compare performances on a large set of proof obligations coming from the BWare project.

2 First-Order Logic with Triggers

In first-order SMT solvers, triggers are used to favor instantiation of universally quantified formulas with "known" terms that have a given form. Intuitively, a term is said to be known when it appears in a ground fact assumed by the solver. Here is an example of a formula with a trigger in SMT-LIB version 2 [5] notation:

```
(forall ((x Int)) (! (= (f x) c) :pattern ((g x))))
```

The bang symbol under the universal quantifier marks an annotated sub-formula and the trigger `(g x)` appears after the keyword `:pattern`. The commonly agreed meaning of the above formula is:

Deduce `(= (f t) c)` *for all terms* `t` *of type* `Int` *such that* `(g t)` *is known.*

Note that classically, triggers do not modify the semantics of the formula. They are just annotations used to specify which are the relevant instances.

The concept of triggers can be extended to literals. If an axiom can only deduce new facts when instantiated with terms having a given property P, it may be unnecessary to instantiate it with a term t without knowing *a priori* that $P(t)$ is true. In other words, we can restrict instantiation not just by the shape of known terms but also by what is known about them. For example, in the theory of extensional arrays, it is enough to apply the extensionality axiom on arrays that are known to be different [21]:

$$\forall a_1,\ a_2 : array.[a_1 \not\approx a_2]\ (\exists i : index.\ get(a_1, i) \not\approx get(a_2, i))$$

In this section, we extend the standard first-order logic with constructions for triggers. For the sake of simplicity, our formalization is unsorted although all our examples use sorts. We define what it means for a formula with triggers to be true in the context of a given set of known facts and terms. Finally, we introduce the properties of *soundness*, *completeness*, and *termination* for sets of first-order formulas with triggers.

2.1 Preliminary Notions

We work in classical untyped first-order logic and assume the standard notation for first-order formulas and terms. We denote formulas with letters φ and ψ, literals with l, terms with s

and t, and substitutions with σ and μ. Other notational conventions will be introduced in the course of the text.

To simplify our definitions, we work on formulas in negative normal form. The syntax of formulas and literals can be described as follows, A being an atom:

$$\begin{aligned} \varphi &::= l \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \forall x.\varphi \mid \exists x.\varphi \\ l &::= A \mid \neg A \end{aligned}$$

The transformation into negative normal form uses the following, usual, definitions:

$$\begin{aligned} \neg(\varphi_1 \vee \varphi_2) &\Longrightarrow \neg\varphi_1 \wedge \neg\varphi_2 \\ \neg(\varphi_1 \wedge \varphi_2) &\Longrightarrow \neg\varphi_1 \vee \neg\varphi_2 \\ \neg(\forall x.\varphi) &\Longrightarrow \exists x.\neg\varphi \\ \neg(\exists x.\varphi) &\Longrightarrow \forall x.\neg\varphi \end{aligned}$$

We say that a formula is *closed* if it has no free variables, and that a term, literal, or formula is *ground* if it has no free variables and no quantifiers. We use $\mathscr{T}(t)$, $\mathscr{T}(l)$, $\mathscr{T}(S)$ to denote the set of all terms that occur in, respectively, a term t, a literal l, or a set of terms or literals S. For the sake of simplicity, we assume that bound variables are renamed so that a quantifier over a variable x never occurs under a quantifier over the same variable x.

We reason modulo some background theory T, which we assume to be fixed for the rest of this section. Theory T must include the theory of equality and non-interpreted function symbols (EUF). In the simplest case, it is just EUF. We assume that the signature of T contains at least one constant symbol to allow constructing the Herbrand universe and can be extended at will with uninterpreted function symbols to allow Skolemization. We use the following definition for atoms:

$$A ::= \top \mid t_1 \approx t_2 \mid \ldots$$

The dots stand for other forms of predicates specific to background theories, e.g. comparison for linear arithmetic.

We use Herbrand models in our formalization, that is, we call model a set of literals L containing every valid literal l. Note that a first-order formula is satisfiable if and only if it has a Herbrand model.

We use the standard notation $L \vDash \varphi$ to state that a closed first-order formula φ is valid in a Herbrand model L. Let T be a theory, that is, a possibly infinite set of closed first-order formulas.

Definition 1 (*T-satisfiability*) We say that a first-order formula φ is valid in a model L modulo T, written $L \vDash_T \varphi$ if φ is valid in L and L is also a model of T. If a first-order formula φ has a model modulo T then we say that it is T-satisfiable or, equivalently, that is satisfiable modulo T.

We sometimes use clauses, that are disjunctive sets of literals. We say that a clause is a unit clause, if it contains only one literal. The empty clause is assumed to be equivalent to false, that is, $\neg\top$.

2.2 Logic with Triggers (Syntax and Semantics)

We introduce two new kinds of formulas. A formula φ under a trigger l is written $[l]\varphi$. It can be read as *if the literal l is true and all its sub-terms are known then assume φ*. A dual

construct for $[l]\varphi$, which we call a *witness*, is written $\langle l\rangle\varphi$. It can be read as *assume that the literal l is true and all its sub-terms are known and assume φ*. Notice that neither triggers nor witnesses are required to be tied to a quantifier. The extended syntax of formulas can be summarized as follows:

$$\varphi \ ::= \ l \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \forall x.\varphi \mid \exists x.\varphi \mid [l]\varphi \mid \langle l\rangle\varphi$$

On the new constructs, transformation into negative normal form is done using the additional equivalences $\neg\langle l\rangle\varphi \Longrightarrow [l]\neg\varphi$ and $\neg[l]\varphi \Longrightarrow \langle l\rangle\neg\varphi$.

We write $[t]\varphi$ for $[t \approx t]\varphi$, $\langle t\rangle\varphi$ for $\langle t \approx t\rangle\varphi$, $\bot$ for $\neg\top$, $t_1 \not\approx t_2$ for $\neg(t_1 \approx t_2)$, $\varphi_1 \rightarrow \varphi_2$ for $\neg\varphi_1 \vee \varphi_2$, and $\varphi_1 \leftrightarrow \varphi_2$ for $(\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$. If there are several triggers or witnesses in a row, we write $[l_1, \ldots, l_n]\varphi$ for $[l_1]\ldots[l_n]\varphi$ and $\langle l_1, \ldots, l_n\rangle\varphi$ for $\langle l_1\rangle\ldots\langle l_n\rangle\varphi$.

Example 1 Here is an axiomatization for the theory of non-extensional arrays as defined by Nelson [30]. This axiomatization uses two function symbols, one, named *get*, to model access in an array and another, named *set*, to model update of an array. It contains two axioms that describe how an array is modified by an update. The first one states that an access to the updated index returns the updated element and the second one, given with two different triggers, states that an access to any other index returns the element that was previously stored at this index.

$$W_{array} = \left\{ \begin{array}{l} \forall a,\ i,\ e.[set(a,i,e)]\ (get(set(a,i,e),i) \approx e) \\ \forall a,\ i,\ j,\ e.[get(set(a,i,e),j)]\ (i \not\approx j \rightarrow get(set(a,i,e),j) \approx get(a,j)) \\ \forall a,\ i,\ j,\ e.[set(a,i,e),\ get(a,j)]\ (i \not\approx j \rightarrow get(set(a,i,e),j) \approx get(a,j)) \end{array} \right\}$$

The trigger of the first axiom expresses that it needs only be instantiated with three terms a, i, and e if the term $set(a,i,e)$ appears in the set of assumed facts. For the second axiom, there are two different cases where it should be instantiated: if $get(set(a,i,e),j)$ appears in the set of assumed facts or if both $set(a,i,e)$ and $get(a,j)$ appear in the set of assumed facts. These two cases allow the equality $get(set(a,i,e),j) \approx get(a,j)$ to be rewritten both ways.

A first-order formula with triggers must be evaluated in the context of a particular set of assumed facts and known terms:

Definition 2 (*World modulo T*) We call *world* a T-satisfiable set of ground literals. A world L is *inhabited* if there is at least one term occurring in it, i.e. $\mathscr{T}(L)$ is non-empty. Throughout this article, we only consider inhabited worlds. A world L is *complete* if for any ground literal l in the signature of T, either $l \in L$ or $\neg l \in L$.

Definition 3 (*Known term modulo T*) A term t is *known* in a world L if and only if there is a term $t' \in \mathscr{T}(L)$ such that $L \vDash_T t \approx t'$.

The key intuition about our semantics for formulas with triggers is that a ground literal l can only be evaluated in a world L if every term t in $\mathscr{T}(l)$ is known in the world. If, on the contrary, some term occurring in l is unknown in L, we "refuse" to evaluate the literal, that is neither l nor $\neg l$ is true in L. To express this constraint easily, we use a unary predicate symbol *known* which we assume to be new and not to appear anywhere else in the problem. Using this symbol, the fact that a term t is known in L, can be equivalently stated as $L \cup \bigwedge_{s\in\mathscr{T}(L)} known(s) \vDash_T known(t)$. We abbreviate the conjunction $\bigwedge_{t\in S} known(t)$ as $known(S)$, where S is any set of ground terms.

Definition 4 (*Truth value modulo T*) Given a world L and a closed formula φ, we define what it means for φ to be *true* in L, written $L \rhd_T \varphi$, by induction on φ as follows:

$L \rhd_T l$	$L \models_T l$ and $L \cup \mathit{known}(\mathscr{T}(L)) \models_T \mathit{known}(\mathscr{T}(l))$
$L \rhd_T \varphi_1 \vee \varphi_2$	$L \rhd_T \varphi_1$ or $L \rhd_T \varphi_2$
$L \rhd_T \varphi_1 \wedge \varphi_2$	$L \rhd_T \varphi_1$ and $L \rhd_T \varphi_2$
$L \rhd_T \forall x.\varphi$	for every term t in $\mathscr{T}(L)$, $L \rhd_T \varphi[x \leftarrow t]$
$L \rhd_T \exists x.\varphi$	there is a term t in $\mathscr{T}(L)$ such that $L \rhd_T \varphi[x \leftarrow t]$
$L \rhd_T [l]\varphi$	if $L \rhd_T l$ then $L \rhd_T \varphi$
$L \rhd_T \langle l \rangle \varphi$	$L \rhd_T l$ and $L \rhd_T \varphi$

We say that a closed formula φ is *false* in L whenever $L \rhd_T \neg\varphi$. We call φ *feasible* if there exists a world in which φ is true.

As we have noted, a formula that contains a term unknown in a world may be neither true nor false in that world. On the other hand, it is impossible for a formula to be both true and false in the same world. In other words, there is no closed formula φ and world L such that $L \rhd_T \varphi$ and $L \rhd_T \neg\varphi$. This is easily proved by induction on the structure of φ.

According to the rules, a formula with a witness $\langle l \rangle \varphi$ is handled just as the conjunction $l \wedge \varphi$. Yet a formula with a trigger $[l]\varphi$ is not the same as the disjunction $\neg l \vee \varphi$. Indeed, consider a literal l that contains a term unknown in L, so that neither l nor $\neg l$ is true in L. Then we have $L \rhd_T [l]\bot$ but not $L \rhd_T \neg l \vee \bot$. However, if L is a complete world, then any ground literal is either true or false in L, and we can replace all triggers with implications.

Definition 5 (*Model modulo T*) A world L is said to be a *model* of a closed formula φ whenever L is complete and $L \rhd_T \varphi$. We call φ *satisfiable* if it has a model.

2.3 Relation with Traditional First-Order Logic

Let φ be a closed formula and φ' be φ where all triggers are replaced with implications and all witnesses with conjunctions. As noted above, in any complete world L, $L \rhd_T \varphi$ if and only if $L \rhd_T \varphi'$. Moreover, $L \rhd_T \varphi'$ if and only if $L \models_T \varphi'$. Indeed, since every ground term is known in a complete world, the truth value of quantified formulas and ground literals in our logic coincides with that in the usual first-order logic. Thus, L is a model of φ if and only if it is a Herbrand model of φ' in T. Consequently, φ is satisfiable in the sense of Definition 5 if and only if φ' is T-satisfiable, which justifies our reuse of the term.

For ground literals or conjunctions thereof the properties of feasibility and satisfiability are equivalent. A non-literal formula, however, can be true in some world yet have no model. For example, the formula $[a]\, a \not\approx a$ (which is an abbreviation for $[a \approx a]\, a \not\approx a$) is true in any world where a is unknown, but is false in any complete world.

Feasibility does not imply the existence of a model even in the case where the formula in question contains no triggers or witnesses. Assume T to be the theory of linear arithmetic. Then the formula $\exists y.\forall x.x \leq y$ is true in the world $\{0 \leq 0\}$. Indeed, this world "knows" only one distinct term modulo T and there is no possible instantiation to refute $\forall x.x \leq 0$. Of course, the formula $\exists y.\forall x.x \leq y$ has no model, since the only complete world for T is, by definition, the set of all ground literal facts of linear arithmetic.

It is thus all the more remarkable that the following implication holds in the background theory EUF (equality with uninterpreted functions):

Theorem 1 *Let φ be a closed first-order formula without triggers and witnesses. Let L be a world such that $L \rhd_{\mathrm{EUF}} \varphi$. Then φ is satisfiable in first-order logic with equality (and therefore has a model in the sense of Definition 5).*

Proof From now on and until the end of this section, we write $\rhd$ for $\rhd_{\mathrm{EUF}}$ and $\vDash$ for $\vDash_{\mathrm{EUF}}$. Consider a model M on domain D of the world L. Among the elements in D, some are known (i.e. a term explicitly present in L is assigned this element by the model), some aren't. Let us build a model M' on the (non-empty) domain D' which is the restriction of the elements of D that are known.

First, the known constants are assigned to their element in D, also in D'. The unknown constants (i.e. the constants that appear in φ but not in L) are assigned an arbitrary element in D'. Now it remains to define functions. For each function f, $M'[f](d_1, \dots d_n)$ (where $d_1, \dots d_n$ belong to D') is either $M[f](d_1, \dots d_n)$ if it belongs to D', or another arbitrary element in D. Notice that, by construction, all known terms have the same interpretation in M and M'.

Now, by induction we prove that M' makes φ true. Since formulas are in NNF, it is sufficient to prove that this inductive property holds: if $L \rhd \varphi$, then $M' \vDash \varphi$. This is trivially true for literals (all known terms have the same interpretation in M and M'), and direct for conjunction and disjunction. Now assume $L \rhd \forall x.\varphi$. Then, for every element $d \in D'$, there is a known term t such that $L \rhd \varphi[x \leftarrow t]$, which basically comes to say that $M'_{x/d} \vDash \varphi[x]$ holds for all d, and thus $M' \vDash \forall x.\varphi$. The existential case is similarly easy. □

2.4 Soundness and Completeness

Whenever a user wants to extend the solver's background theory T and provides for that purpose a set of axioms with triggers, she must prove that this axiomatization is an adequate representation of the extended theory T' modulo T.

Definition 6 (*Soundness modulo T*) An axiomatization W is *sound* with respect to T' if, for every T'-satisfiable set of ground literals L, $W \cup L$ is T-satisfiable.

Definition 7 (*Completeness modulo T*) An axiomatization W is *complete* with respect to T' if, for every set of ground literals L such that $W \cup L$ is feasible, L is T'-satisfiable.

Remark 1 Note that the two definitions of soundness and completeness are not symmetrical. Indeed, we want a solver to be allowed to stop whenever it has found a world L in which the axiomatization is true while returning "satisfiable". In particular, we want the solver to instantiate universal quantifiers using only terms of L and to ignore formulas protected by a trigger l if l is not true in L.

Most often, triggers are simply used to guide the instantiation and T' is the theory defined by the same set of axioms W where triggers are replaced by implications and witnesses by conjunctions. In the case of "term" triggers and witnesses $t \approx t$, this is equivalent to simply erasing them. In practice, the process is inverse: we start with a usual first-order axiomatization of the theory of interest, and then annotate axioms with triggers and witnesses in order to restrict instantiation and guarantee the termination of proof search. In this case, we do not have to prove soundness, because in complete worlds, triggers have the same semantics as implications. As for completeness, we must show that the added triggers and the restricted semantics of quantifiers do not prevent us from proving every ground statement deducible in the initial axiomatization.

Example 2 The proof that the set of axioms W_{array} shown in Example 1 is complete modulo EUF closely resembles the proof by Nelson [30]. We do not give this proof here but show that simpler or more "intuitive" variants of that axiomatization are incomplete.

- Let W^1_{array} be W_{array} where the trigger in the first axiom is replaced by $get(set(a, i, e), i)$. Consider the set of literals $L_1 = \{set(a, i, e_1) \approx set(a, i, e_2),\ e_1 \not\approx e_2\}$. It is unsatisfiable in the theory of arrays since we have both $get(set(a, i, e_1), i) \approx e_1$ and $get(set(a, i, e_2), i) \approx e_2$. Still, $W^1_{array} \cup L_1$ is true in the world L_1, since we have no term in L_1 to match the trigger.
- Let W^2_{array} be W_{array} without the second axiom. Consider the set of ground literals $L_2 = \{get(set(a, i_1, e), j) \not\approx get(set(a, i_2, e), j),\ i_1 \not\approx j,\ i_2 \not\approx j\}$. It is unsatisfiable in the theory of arrays since $get(set(a, i_1, e), j) \approx get(a, j)$ and $get(set(a, i_2, e), j) \approx get(a, j)$. Yet, $W^2_{array} \cup L_2$ is true in the world $L_2 \cup \{get(set(a, i_1, e), i_1) \approx e,\ get(set(a, i_2, e), i_2) \approx e\}$.
- Let W^3_{array} be W_{array} without the third axiom. Consider the set of ground literals $L_3 = \{set(a_1, i, e) \approx set(a_2, i, e),\ i \not\approx j,\ get(a_1, j) \not\approx get(a_2, j)\}$. It is unsatisfiable in the theory of arrays since $get(set(a_1, i, e), j) \approx get(a_1, j)$ and $get(set(a_2, i, e), j) \approx get(a_2, j)$. Still $W^3_{array} \cup L_3$ is true in the world $L_3 \cup \{get(set(a_1, i, e), i) \approx e,\ get(set(a_2, i, e), i) \approx e\}$.

Example 3 Consider the following axiomatization. We want to model conversion between two domains E and e such that every element of e can be converted to an element of E but there may be elements of E that cannot be converted to e. The axiomatization contains five function symbols. If $valid_E(x)$ (resp. $valid_e(x)$) returns $\mathtt{t}$ then x is an element of E (resp. an element of e). The conversion function $conv_{E\to e}(x)$ (resp. $conv_{e\to E}(x)$) may return either an element of e (resp. an element of E) or some unspecified "invalid" value, if x is not fit for conversion. If x is an element of E, the function $unfit_{E\to e}(x)$ returns $\mathtt{t}$ when x cannot be converted to e.

$$W_{conv} = \left\{ \begin{array}{l} \forall x.[valid_E(x) \approx \mathtt{t}]\ valid_e(conv_{E\to e}(x)) \approx \mathtt{t} \vee unfit_{E\to e}(x) \approx \mathtt{t} \\ \forall x.[valid_e(x) \approx \mathtt{t}]\ valid_E(conv_{e\to E}(x)) \approx \mathtt{t} \\ \forall x.[valid_E(x) \approx \mathtt{t},\ valid_e(conv_{E\to e}(x)) \approx \mathtt{t}]\ conv_{e\to E}(conv_{E\to e}(x)) \approx x \\ \forall x.[valid_e(x) \approx \mathtt{t}]\ conv_{E\to e}(conv_{e\to E}(x)) \approx x \end{array} \right\}$$

We want to show that W_{conv} is complete modulo EUF with respect to the same axiomatization W'_{conv} where triggers are replaced by implications. Let L be a world in which W_{conv} is true. We define $L' = L \cup \{valid_E(t) \not\approx \mathtt{t} \mid L \not\models valid_E(t) \approx \mathtt{t}\} \cup \{valid_e(t) \not\approx \mathtt{t} \mid L \not\models valid_e(t) \approx \mathtt{t}\}$ and we show that L' is a model of W'_{conv}. Since we are working only modulo EUF, L is satisfiable. What is more, by definition of EUF, for any term t, if $L \models valid_e(t) \approx \mathtt{t}$ then $L \cup known(\mathcal{T}(L)) \models known(t)$. As a consequence, for every term t, either $L \rhd valid_e(t) \approx \mathtt{t}$ or $L \not\models valid_e(t) \approx \mathtt{t}$. Since every trigger in W_{conv} are applications of $valid_e$ or $valid_E$, for every formula $\forall x.[l_1 \ldots l_n]l'_1 \vee \cdots \vee l'_m \in W_{conv}$ and every term t, either $L \rhd l_1[x \mapsto t] \wedge \cdots \wedge l_n[x \mapsto t]$ and $L \rhd l'_1[x \mapsto t] \vee \cdots \vee l'_m[x \mapsto t]$ or $L \not\models l_i$ for some i in $1..n$. In both cases, $L' \models (l_1[x \mapsto t] \wedge \cdots \wedge l_n[x \mapsto t]) \to l'_1[x \mapsto t] \vee \cdots \vee l'_m[x \mapsto t]$. As a consequence, $L' \models \forall x.(l_1 \wedge \cdots \wedge l_n) \to l'_1 \vee \cdots \vee l'_m$ for every formula $\forall x.(l_1 \wedge \cdots \wedge l_n) \to l'_1 \vee \cdots \vee l'_m \in W'_{conv}$ and L' is a model of W'_{conv}.

2.5 Termination

Once it has been established that a given set of axioms with triggers is sound and complete for our theory, we must show that the solver equipped with this axiomatization terminates on any

ground satisfiability problem. We call such axiomatizations *terminating* and the remainder of this section is dedicated to the definition of this property.

There can be no single "true" definition of a terminating axiomatization. Different variations of the solver algorithm may terminate on different classes of problems, which may be more or less difficult to describe and to reason about. We should rather strive for a "good" definition, which, on one hand, leaves room for an efficient implementation, and on the other hand, is simple enough to make it feasible to prove that a given set of axioms is terminating.

Below we present what we consider a reasonably good definition. It serves as the basis for the DPLL-based procedure described in Sect. 5. In Sect. 3, we prove that a non-trivial axiomatization of imperative doubly-linked lists is terminating according to this definition. Finally, in Sect. 6.3, we discuss possible variations of the termination property and their implications for the solver algorithm.

To bring ourselves closer to the implementation, we start by eliminating the existential quantifiers and converting axioms into a clausal form.

Skolemization The *Skolemization transformation*, denoted SKO, traverses a formula in top-down order and replaces existential quantifiers with witnesses of Skolem terms as follows:

$$\text{SKO}(\exists x.\varphi) \triangleq \langle c(\overline{y})\rangle\, \text{SKO}(\varphi[x \leftarrow c(\overline{y})]),$$

where $\overline{y}$ is the set of free variables of $\exists x.\varphi$ and c is a fresh function symbol.

Lemma 1 *Skolemization preserves feasibility and satisfiability.*

Proof It can be done by induction over φ. We construct a world for $\text{SKO}(\varphi)$ by giving the Skolem terms the same interpretation as for the corresponding ground terms in the original world for φ. In the opposite sense, if $\text{SKO}(\varphi)$ is feasible, then φ is true in the same world.

The use of the witness is crucial here. Indeed, $\text{SKO}(\exists x.[x]\bot)$ is $\langle c\rangle[c]\bot$ which preserves infeasibility, whereas the formula $[c]\bot$ is true in any world where c is unknown. □

Skolemization preserves the soundness and completeness of a set of axioms on condition that the user problem L may not contain Skolem symbols. For example, if T' is the theory $\{\exists x.P(x)\}$ then, for the soundness of the skolemized axiom $\langle c\rangle P(c)$, we should only consider sets that do not include the constant c. For example, the ground literal $\neg P(c)$ is T'-satisfiable, but the union $\langle c\rangle P(c) \cup \neg P(c)$ has no model. This does not present a problem for us: the soundness and completeness theorems in Sect. 5 do not require skolemized axiomatizations.

Clausification

Definition 8 (*Pseudo-clause*) We say that a formula is a *pseudo-literal* if it is a literal l, a trigger $[l]C$, a witness $\langle l\rangle C$, or a universally quantified formula $\forall x.C$, where C is a disjunction of pseudo-literals, called *pseudo-clause*.

In what follows, we treat pseudo-clauses (and other kinds of clauses) as disjunctive sets, that is, we ignore the order of their elements and suppose that there are no duplicates. As for traditional logic, any skolemized formula can be transformed into a clausal form, the case of triggers and witnesses being handled using the equivalences between the formulas $[l](\varphi_1 \wedge \varphi_2)$ and $[l]\varphi_1 \wedge [l]\varphi_2$, and the formulas $\langle l\rangle(\varphi_1 \wedge \varphi_2)$ and $\langle l\rangle\varphi_1 \wedge \langle l\rangle\varphi_2$.

Before we proceed to the definition of the termination property, let us give some informal explanation of it. To reason about termination, we need an abstract representation of the

evolution of the solver's state. It is convenient to see this evolution as a game where we choose universal formulas to instantiate and our adversary decides how to interpret the result of instantiation, that is, what new facts can be assumed. Whenever we arrive at a set of facts that is inconsistent or saturated so that no new instantiations can be made, the game terminates and we win. If, on the other hand, whatever instantiations we do, the adversary can find new universal formulas for us to instantiate, the game continues indefinitely. An axiomatization is terminating if we have a winning strategy for it. In other words, no matter what partial model we explore, there is a sequence of instantiations—which our solver will eventually make due to fairness—leading either to a contradiction or to a saturated partial model.

The adversary's moves are represented by so-called *truth assignments*. Intuitively, given a current set of assumed facts, a truth assignment is any set of further facts that the solver may assume using only propositional reasoning, without instantiation. Once this completion is done, we may choose an assumed universal formula and a known term to perform instantiation, allowing for the next stage of completion and so on. A tree that inspects all possible truth assignments for certain instantiation choices (i.e. all possible adversary's responses to a particular strategy of ours) is called *instantiation tree*. An axiomatization is terminating if for any ground satisfiability problem we can construct a finite instantiation tree.

To avoid applying substitutions, we use *closures*. A closure is a pair $\varphi \cdot \sigma$ made of a pseudo-literal φ and a substitution σ mapping every free variable of φ to a ground term. In a closure $\varphi \cdot \sigma$, the substitution σ is only defined on the free variables of φ. This is done for convenience, so that we can directly compare two substitutions without making explicit the set of variables to which they are restricted. We write $\varphi\sigma$ for the application of σ to φ, and $\varnothing$ for the empty substitution. If two substitutions σ and σ' have the same domain D, we write $\sigma \dot{\approx} \sigma'$ for the formula $\bigwedge_{x \in D} x\sigma \approx x\sigma'$. If C is a pseudo-clause, we write $C \cdot \sigma$ for the disjunctive set of closures $\{\varphi \cdot \sigma' \mid \varphi \in C$ and σ' is σ restricted to the free variables of $\varphi\}$. We call disjunctive sets of closures *theory clauses*, as they come from the axiomatization of our theory of interest.

We define the facts that are readily available from a set of theory clauses V, without the need to eliminate triggers or witnesses, to instantiate a variable, or to decide which part of a disjunction to assume:

Definition 9 Given a set of theory clauses V, we define the set of literals $\lfloor V \rfloor \triangleq \{l\sigma \mid l \cdot \sigma \in V$ and l is a literal$\}$.

Definition 10 (*Truth assignment modulo T*) A *truth assignment* of a set of theory clauses V is any set of theory clauses A that can be constructed starting from V by exhaustive application of the following rules:

- if $(\varphi_1 \vee \ldots \vee \varphi_n) \cdot \sigma \in A$ then add some subset of the closures $\varphi_1 \cdot \sigma, \ldots, \varphi_n \cdot \sigma$ to A,
- if $[l]C \cdot \sigma \in A$ and $\lfloor A \rfloor \rhd_T l\sigma$ then add $C \cdot \sigma$ to A,
- if $\langle l \rangle C \cdot \sigma \in A$, then add $l \cdot \sigma$ and $C \cdot \sigma$ to A.

We say that a truth assignment A is *T-satisfiable* if the set of literals $\lfloor A \rfloor$ is T-satisfiable. A T-satisfiable truth assignment A is said to be *final* if every possible instantiation is *redundant* in A, that is for every closure $\forall x.C \cdot \sigma$ in A and every term $t \in \mathscr{T}(\lfloor A \rfloor)$, there is a ground substitution σ' such that $C \cdot \sigma' \in A$ and $\lfloor A \rfloor \vDash_T (\sigma \cup [x \mapsto t]) \dot{\approx} \sigma'$. In what follows, we write $\mathscr{T}(A)$ for $\mathscr{T}(\lfloor A \rfloor)$ and $A \vDash_T l$ for $\lfloor A \rfloor \vDash_T l$.

Remark that, in terms of solver implementation, this definition means that, while we require the solver to eliminate triggers and witnesses eagerly, it is permitted to postpone the decision over disjunctions. Such postponing corresponds to adding no closures at all in the

first case of the definition above. In this way, the solver is not urged to make choices which it will have to backtrack later, and can instead wait until subsequent instantiations reduce the choice space.

Furthermore, we do not restrain the solver to only choose a single disjunct per clause. Indeed, this restriction may be too strong for some SMT solvers, in particular those using an existing SAT solver for their propositional reasoning. Such a solver may assign more than one literal in a clause to be true. On the other hand, for an SMT solver that never assumes more than one literal in a clause, our definition of truth assignment can be refined accordingly, leading to simpler termination proofs.

Since a truth assignment only decomposes formulas and does not introduce new terms, any finite set of theory clauses has a finite number of possible truth assignments.

Definition 11 (*Instantiation tree modulo T*) An *instantiation tree* of a set of pseudo-clauses W is any tree where the root is labeled by $W \cdot \varnothing$, every node is labeled by a set of theory clauses, and every edge is labeled by a non-final truth assignment such that:

- a node labeled by V has leaving edges labeled by all T-satisfiable non-final truth assignments of V,
- an edge labeled by A leads to a node labeled by $A \cup C \cdot (\sigma \cup [x \mapsto t])$, where $\forall x.C \cdot \sigma \in A$ and $t \in \mathscr{T}(A)$.

Definition 12 (*Termination modulo T*) A set of pseudo-clauses W is *terminating* if, for every finite set of ground literals L, $W \cup L$ admits at least one finite instantiation tree.

Remark 2 In the definition of termination, we only require that W has one finite instantiation tree and not that every instantiation tree of W is finite. Indeed, we expect the solver's implementation to be fair and to do all the instances required by one particular finite instantiation tree. That it may also do other instances is not a problem as, in a finite amount of time, it will either reach an unsatisfiable state or all these unaccounted instances will be redundant.

The process of truth assignment leaves the solver a choice over what parts of a disjunction to assume. It may seem that assuming more formulas will always bring us more known terms and more universal sub-formulas to instantiate, so that it is sufficient to only consider the maximal truth assignments in an instantiation tree. However, this is not true: an assumed formula might be an equality that, instead of expanding the set of known terms, reduces it. Thus it may happen that an infinite branch in an instantiation tree passes through non-maximal truth assignments.

Example 4 The proof of termination of the theory of arrays described in Examples 1 and 2 is straightforward. It suffices to demonstrate that the axioms of W_{array} cannot create new terms. Indeed, let L be a set of ground literals and A a truth assignment of $W_{array} \cup L$. Assume that there are three terms a, i, and e in $\mathscr{T}(\lfloor A \rfloor)$ such that the term $set(a, i, e)$ is known in $\lfloor A \rfloor$, *i.e.*, $\lfloor A \rfloor \cup known(\mathscr{T}(\lfloor A \rfloor)) \rhd_T known(set(a, i, e))$. Then, for every term t in $\mathscr{T}(get(set(a, i, e), i) \approx e)$, $\lfloor A \rfloor \cup \{get(set(a, i, e), i) \approx e\} \cup known(\mathscr{T}(\lfloor A \rfloor)) \models_T known(t)$. Indeed, since $set(a, i, e)$ is known in $\lfloor A \rfloor$, this must be the case for $set(a, i, e)$ and all it subterms, and, since $get(set(a, i, e), i) \approx e$, it is also the case for $get(set(a, i, e), i)$. Thus, no instance of the first axiom can lead to the creation of new known terms. The same reasoning can be done for the second and the third axiom. Therefore, every instantiation tree of $W_{array} \cup L$ is finite.

Example 5 Let us look at a more interesting proof of termination. Consider the axiomatization W_{conv} from Example 3 and let L be a finite set of literals. We show how a finite instantiation

tree can be constructed for $W_{conv} \cup L$. For any truth assignment A, add an instance of one of the first two axioms with a term of L if there is one that is not redundant in A. If all such instances are redundant, add an instance of one of the last two axioms of W_{conv} with terms of L if there is one that is not redundant in A. The repeated application of these two steps can only construct finite trees. Indeed, the first one constructs at most two instances per term of L. The second step never adds new terms to A. Indeed, for the last axiom of W_{conv} for example, once the triggers are removed, the only new terms are $conv_{E\to e}(conv_{e\to E}(t))$, which is equal to t, and $conv_{e\to E}(t)$ which was already created by the second axiom if it was not already present. As a consequence, it constructs at most two instances per term present after the last time the first step was applied.

If neither the first nor the second step can be applied on a satisfiable truth assignment A, every non-redundant instance of an axiom in A can only be made with one of the first two axioms and a term of $\lfloor A \rfloor \backslash L$. Any such instance cannot produce new term. For example, an instance of the first axiom with a term t can only produce the term $conv_{E\to e}(t)$ since $valid_e(conv_{E\to e}(t))$ and $unfit_{E\to e}(t)$ are equated to $\mathtt{t}$. Let us show that, if neither the first nor the second step can be applied on a satisfiable truth assignment A, this term is already known in A.

For the term $conv_{E\to e}(t)$ to be deduced, the trigger of the first axiom must be true. As a consequence, $A \rhd valid_E(t) \approx \mathtt{t}$. By construction of A, t can not occur in L, otherwise, the instance has been already produced with the first step. As a consequence, $valid_E(t) \approx \mathtt{t}$ was deduced using the second axiom and there is $t' \in \mathcal{T}(L)$ such that $A \vDash valid_e(t') \approx \mathtt{t}$ and $A \vDash t \approx conv_{e\to E}(t')$. Therefore, the last axiom of W_{conv} has been instantiated with t' with the second step and $A \vDash conv_{E\to e}(conv_{e\to E}(t')) \approx t'$ and thus $A \vDash valid_e(conv_{E\to e}(t))$. Consequently, the result of an instance of the first axiom with t cannot produce any new term. We conclude the construction of the instantiation tree by making all non-redundant instances of the four axioms with terms in A, and there is only a finite number of them.

Remark 3 The axiomatization W_{conv} can be seen as the specification of a datastructure with pointers, and therefore falls in the scope of McPeak and Necula's [28] work about data structure specifications.

3 Case Study: Imperative Doubly-Linked Lists

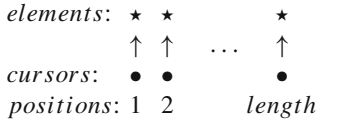
In this section, we give a rather large axiomatization as an example (more than 50 axioms). We assume that the background theory T is the combination of integer linear arithmetic and booleans or any conservative extension of it. The axiomatized theory T' contains a definition of imperative doubly-linked lists with a definition for iterators (named *cursors*), an equality function, several modification functions and so on. We prove that this axiomatization is sound, complete and terminating. It is inspired by the API of lists in the Ada standard library [17].

3.1 Presentation of the Theory

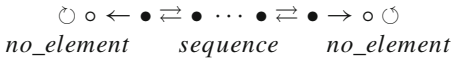
Lists are ordered containers of elements on which an equivalence is defined using the function $equal\_elements(e_1 : element\_type, e_2 : element\_type) : bool$. We represent imperative lists of elements as pairs of:

- *an iterative part:* a finite sequence of distinct cursors (used to iterate through the list),
- *a content part:* a partial mapping from cursors to elements, only defined on cursors that are in the sequence.

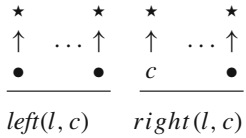
The iterative part of an imperative list l is modeled by an integer $length(l\colon list)\colon int$ representing the length of the sequence together with a total function $position(l\colon list, c\colon cursor)\colon int$ so that, for every cursor c, $position(l, c)$ returns the position of c in l if it appears in the sequence and 0 otherwise. The content part of l is modeled by a function $element(l\colon list, c\colon cursor)\colon element\_type$ so that $element(l, c)$ returns the element associated to c in l if any:



Thanks to this description, we can define several other function symbols. $has\_element(l\colon list, c\colon cursor)\colon bool$ returns true if and only if c appears in the imperative part of l and $is\_empty(l\colon list)\colon bool$ returns true if l is an empty list. The functions $last(l\colon list)\colon cursor$, $first(l\colon list)\colon cursor$, $previous(l\colon list, c\colon cursor)\colon cursor$ and $next(l\colon list, c\colon cursor)\colon cursor$ are used to iterate through the iterative part of l. If l is empty, $last(l)$ and $first(l)$ return a special cursor, named $no\_element$ that never appears in any list. This cursor is added at both ends of the iterative part of l so that $previous(l, first(l))$, $previous(l, no\_element)$, $next(l, last(l))$ and $next(l, no\_element)$ are $no\_element$:



We define two functions $left(l\colon list, c\colon cursor)\colon list$ and $right(l\colon list, c\colon cursor)\colon list$, that are used to split the list. If c appears in the imperative part of l or is $no\_element$, $left(l, c)$ (resp. $right(l, c)$) returns the prefix (resp. suffix) of l that stops (resp. starts) before c:



A special empty list $empty$ is returned by $left(l, c)$ (resp. $right(l, c)$) if the cursor c is $first(l)$ (resp. $no\_element$). On $no\_element$, $left(l, c)$ returns l.

To search the content part of l for the first occurrence of an element e modulo equivalence, we use the function $find(l\colon list, e\colon element\_type, c\colon cursor)\colon cursor$. If c appears in the iterative part of l, $find(l, e, c)$ returns the first cursor of l following c which is mapped to an element equivalent to e. If there is no such element, $no\_element$ is returned. To search the whole list l, the cursor $no\_element$ can be used instead of $first(l)$. The function $contains(l\colon list, e\colon element\_type)\colon bool$ is true if and only if l contains an element equivalent to e.

We add a notion of equality on lists: $equal\_lists(l_1\colon list, l_2\colon list)$ is true if and only if both parts of l_1 and l_2 are equal.

Finally, the last three functions describe how a list l is modified when an element is inserted, deleted or replaced in l. If $insert(l\colon list, c\colon cursor, e\colon element\_type, r\colon list)\colon bool$ is true then r can be obtained by inserting a cursor before c in the list l (or at the end if c is $no\_element$) and mapping it to e. If $delete(l\colon list, c\colon cursor, r\colon list)\colon bool$ is true then r can be obtained by deleting the cursor c from the list l. If $replace\_element(l\colon list, c\colon cursor, e\colon element\_type, r\colon list)\colon bool$ is true then r can be obtained by replacing the element associated to c in l by e.

3.2 Description of the Axiomatization

We give in this section an overview of an axiomatization W_{dlli} of the theory from Sect. 3.1. We only give a few axioms. The whole axiomatization is available in "Appendix".

The functions $length$ and $position$ are constrained by the axiomatization so that they effectively give a representation of the iterative part of the list. The three following axioms express that a list contains a finite sequence of distinct cursors:

LENGTH_GTE_ZERO:

$$\forall l{:}\,list.[length(l)]length(l) \geq 0$$

POSITION_GTE_ZERO:

$$\forall l{:}\,list, c{:}\,cursor.[position(l,c)] \\ length(l) \geq position(l,c) \wedge position(l,c) \geq 0$$

POSITION_EQ:

$$\forall l{:}\,list, c_1c_2{:}\,cursor.[position(l,c_1), position(l,c_2)] \\ position(l,c_1) > 0 \rightarrow position(l,c_1) \approx position(l,c_2) \rightarrow c_1 \approx c_2$$

Functions on lists such as $right$, $previous$, $first$ or $find$ are only described on their domain of definition. We only present $find$. We use an additional intermediate function, named $find\_first(l{:}\,list, e{:}\,element\_type){:}\,cursor$, which returns the first cursor of l that is mapped to an element equivalent to e. The result $find(l,e,c)$ can then be defined to be the result of $find\_first$ on the cursors following c in l that is to say $right(l,c)$.

FIND_FIRST_RANGE:

$$\forall l{:}\,list, e{:}\,element\_type.[find\_first(l,e)] \\ find\_first(l,e) \approx no\_element \vee position(l, find\_first(l,e)) > 0$$

FIND_FIRST_NOT:

$$\forall l{:}\,list, e{:}\,element\_type, c{:}\,cursor.[find\_first(l,e), element(l,c)] \\ find\_first(l,e) \approx no\_element \rightarrow position(l,c) > 0 \rightarrow \\ equal\_elements(element(l,c), e) \not\approx \mathtt{t}$$

FIND_FIRST_FIRST:

$$\forall l{:}\,list, e{:}\,element\_type, c{:}\,cursor.[find\_first(l,e), element(l,c)] \\ 0 < position(l,c) < position(l, find\_first(l,e)) \rightarrow \\ equal\_elements(element(l,c), e) \not\approx \mathtt{t}$$

FIND_FIRST_ELEMENT:

$$\forall l{:}\,list, e{:}\,element\_type.[find\_first(l,e)]0 < position(l, find\_first(l,e)) \rightarrow \\ equal\_elements(element(l, find\_first(l,e)), e) \approx \mathtt{t}$$

FIND_FIRST:

$$\forall l{:}\,list, e{:}\,element\_type.[find(l,e,no\_element)] \\ find(l,e,no\_element) \approx find\_first(l,e)$$

FIND_OTHERS:

$$\forall l{:}\,list, e{:}\,element\_type, c{:}\,cursor.[find(l,e,c)] \\ position(l,c) > 0 \rightarrow find(l,e,c) \approx find\_first(right(l,c), e)$$

The predicates describing a modification of a list are only relevant if they are known to be true. Here are axioms describing how the result of a deletion is related to the initial state of the list. They express the relations between the two lists using functions $length$, $position$ and $element$.

DELETE_RANGE:

$$\forall l_1, l_2 : list, c : cursor.[delete(l_1, c, l_2)] \\ delete(l_1, c, l_2) \approx \mathtt{t} \rightarrow position(l_1, c) > 0$$

DELETE_LENGTH:

$$\forall l_1, l_2 : list, c : cursor.[delete(l_1, c, l_2)] \\ delete(l_1, c, l_2) \approx \mathtt{t} \rightarrow length(l_2) + 1 \approx length(l_1)$$

DELETE_POSITION_BEFORE:

$$\forall l_1, l_2 : list, c_1, c_2 : cursor.[delete(l_1, c_1, l_2), position(l_1, c_2)] \\ (delete(l_1, c_1, l_2) \approx \mathtt{t} \wedge position(l_1, c_2) < position(l_1, c_1)) \rightarrow \\ position(l_1, c_2) \approx position(l_2, c_2)$$

$$\forall l_1, l_2 : list, c_1, c_2 : cursor.[delete(l_1, c_1, l_2), position(l_2, c_2)] \\ (delete(l_1, c_1, l_2) \approx \mathtt{t} \wedge 0 < position(l_2, c_2) < position(l_1, c_1)) \rightarrow \\ position(l_1, c_2) \approx position(l_2, c_2)$$

DELETE_POSITION_AFTER:

$$\forall l_1, l_2 : list, c_1, c_2 : cursor.[delete(l_1, c_1, l_2), position(l_1, c_2)] \\ (delete(l_1, c_1, l_2) \approx \mathtt{t} \wedge position(l_1, c_2) > position(l_1, c_1)) \rightarrow \\ position(l_1, c_2) \approx position(l_2, c_2) + 1$$

$$\forall l_1, l_2 : list, c_1, c_2 : cursor.[delete(l_1, c_1, l_2), position(l_2, c_2)] \\ (delete(l_1, c_1, l_2) \approx \mathtt{t} \wedge position(l_2, c_2) \geq position(l_1, c_1)) \rightarrow \\ position(l_1, c_2) \approx position(l_2, c_2) + 1$$

DELETE_POSITION_NEXT:

$$\forall l_1, l_2 : list, c : cursor.[delete(l_1, c, l_2)] \\ delete(l_1, c, l_2) \approx \mathtt{t} \rightarrow \langle next(l_1, c) \rangle \top$$

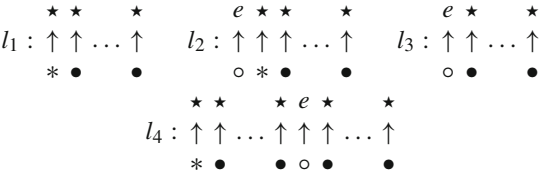
DELETE_ELEMENT:

$$\forall l_1, l_2 : list, c_1, c_2 : cursor.[delete(l_1, c_1, l_2), element(l_1, c_2)] \\ (delete(l_1, c, l_2) \approx \mathtt{t} \wedge position(l_2, c_2) > 0) \rightarrow \\ element(l_1, c_2) = element(l_2, c_2)$$

Note that the axiom DELETE_POSITION_NEXT only serves to introduce a new known term, $next(l_1, c)$. This term is needed so that the axiomatization is complete. Indeed, consider the following set of ground formulas G:

$$\left\{ \begin{array}{l} length(l_1) > 1,\ insert(l_1, first(l_1), e, l_2),\ delete(l_2, first(l_1), l_3), \\ left(l_4, first(l_2)) \approx l_1,\ right(l_4, first(l_2)) \approx l_3 \end{array} \right\}$$

It is unsatisfiable in the theory of doubly-linked lists. Indeed, here is the situation it describes:



Since the length of l_1 is strictly greater than 1, there are cursors that appear twice in l_4 (those represented by a $\bullet$) which is forbidden. Still, without this additional axiom, $G \cup W_{dlli}$ is feasible in our logic. DELETE_POSITION_NEXT introduces a known term for the first cursor of this slice, allowing the axiomatization to deduce that it appears twice in l_4.

3.3 Proofs of Completeness and Termination

In this section, we illustrate how a proof of termination and completeness can be conducted on the axiomatization W_{dlli} of doubly-linked lists. We consider termination and completeness with respect to the same axiomatization where triggers and witnesses are removed. The soundness in this case is trivial, since triggers do not allow new facts to be deduced and the only witness in W_{dlli} is a tautology.

We assume that the three types *lists*, *cursor*, and *element_type* are abstract, *i.e.*, non-interpreted in the background theory T.

Theorem 2 *The axiomatization in Sect. 3.2 is terminating, sound, and complete with respect to the same axiomatization without the triggers.*

The proof holds if *element_type* is an interpreted type, as long as it is infinite, otherwise W_{dlli} is not complete. This is discussed at the end of this section.

3.3.1 Proof of Termination

Every universal quantification ranges over lists, cursors or elements. As a consequence, if we show that only a finite number of terms of type *list*, *cursor* and *element_type* can be created, we can deduce that W_{dlli} is terminating.

Let us first look at terms of type *list*. There is only one formula containing a literal in which there is a sub-term t of type *list* that does not appear in the trigger, namely FIND_OTHERS. The trigger of this formula is $find(l, e, c)$. Such a term cannot be created by W_{dlli}. Since the symbol $find$ is not interpreted, $known(find(l, e, c))$ can only be deduced if we have $known(find(l', e', c'))$ and equalities between all arguments. These equalities are enough to ensure that the new term $right(l, c)$ is equal to the already known term $right(l', c')$. As a consequence, there can only be one new term of type *list* per term of the form $find(l, e, c)$ in the initial problem.

Then we concentrate on terms of type *cursor*. The axioms FIND_FIRST, FIND_OTHERS, CONTAINS_DEF, INSERT_NEW, INSERT_NEW_NO_ELEMENT and DELETE_POSITION_NEXT all contain a literal in which there is a sub-term t of type *cursor* that does not appear in the trigger. Also, there is an existentially quantified cursor variable in EQUAL_LISTS_INV, which amounts to a term of type *cursor* after Skolemization. All these cases can be solved with the same arguments as for the terms of type *list*. Indeed, the symbols $contains(l, e)$, $find(l, e, c)$, $insert(l_1, c, e, l_2)$, $delete(l_1, c, l_2)$, and $equal\_lists(l_1, l_2)$ are all uninterpreted and cannot be created by W_{dlli}.

Finally, let us look at terms of type *element_type*. There are a lot of such terms in W_{dlli} because the function $element$ is often used. However, most of the time, new terms of type *element_type* appear in an equality with an already known term (a sub-term of the trigger). For these terms to be deduced, the equality has to be assumed. Since the equality is with an already known term, the term is not new any more. Remain the axioms FIND_FIRST_ELEMENT and EQUAL_LISTS_INV which can both be handled with the same reasoning we did for terms of type *list* and *cursor*. Indeed, $find\_first(l, e)$ is uninterpreted and can only be created once per $contains(l, e)$ and twice per $find(l', e, c)$ which themselves cannot be created.

3.3.2 Proof of Completeness

Let G be a set of literals and L a world in which G and the axiomatization are true. To demonstrate completeness, we need to show that G is satisfiable in the theory of doubly-linked lists. We construct a model from L in the theory of doubly-linked lists. Since $L \rhd_T G$, it is also a model of G.

Since L is T-satisfiable, let I_T be a model of L. No integer constant appears in a trigger of W_{dlli}. As a consequence, the axiomatization is true in $L \cup \{i \approx i \mid i \text{ is an integer constant}\}$. For every term $t \in \mathscr{T}(L)$ of the form $length(l)$ or $position(l, c)$, we add $t \approx I_T(t)$ to L. We need to show that W_{dlli} is still true in the world L. We introduce a more general lemma that states that equalities between integers can be added to any world in which W_{dlli} is true without triggering new deductions from the axiomatization W_{dlli}:

Lemma 2 *Let L be a world in which W_{dlli} is true and t_1 and $t_2 \in \mathscr{T}(L)$ be two terms of type integer. If $L \not\models_T t_1 \not\approx t_2$ then W_{dlli} is also true in $L \cup t_1 \approx t_2$.*

Proof Triggers of W_{dlli} either have no non-variable sub-terms of type *integer* or can be written $t \approx t$ where t is of type *integer* and has no proper non-variable sub-terms of this type. In both cases, assuming an equality between two known integer terms cannot make any new sub-term of a trigger become known nor make a trigger itself become true. As a consequence, for every literal l appearing as a trigger in W_{dlli}, if $L \not\rhd_T l$, the terms $t_1, t_2 \in \mathscr{T}(L)$ have type *integer* and $L \not\models_T t_1 \not\approx t_2$ then $L \cup t_1 \approx t_2 \not\rhd_T l$. This is enough to show that, if the axiomatization is true in L, $t_1, t_2 \in \mathscr{T}(L)$ have type *integer* and $L \not\models_T t_1 \not\approx t_2$ then W_{dlli} is true in $L \cup t_1 \approx t_2$. □

We now need to give a value to lengths of lists and positions of cursors which are not known in L. For every term l of type *list* in L, if the term $length(l)$ is not in $\mathscr{T}(L)$ modulo T, we add $length(l) \approx 0$ to L and, for every term c of type *cursor*, if $position(l, c)$ is not in L, we add $position(l, c) \approx 0$. This amounts to deciding that lists that are not forced to be non-empty are empty and cursors that are not forced to be valid in a list l are not valid in l. The axiomatization is still true in L. Indeed, thanks to POSITION_GTE_ZERO, $length(l)$ is in $\mathscr{T}(L)$ whenever there is a cursor c such that $position(l, c)$ is in $\mathscr{T}(L)$.

We have to associate a cursor to every position of every non-empty list. For this, we consider *zones* of lists. We define a zone of a term l of type *list* in L to be a sublist $l[i, j]$, i and j being in $0 \ldots length(l)$ such that either $i = 0$ or there is a term c of type *cursor* in $\mathscr{T}(L)$ such that $L \models_T position(l, c) \approx i$ and, for all k such that $i < k \leq j$, there is no term c of type *cursor* in L such that $L \models_T position(l, c) \approx k$. Remark that elements that are inserted and deleted are, by construction, in a zone of size 1 only containing them (see DELETE_POSITION_NEXT). In the same way, for $right$ and *left*, cuts are always done at the junction between two different zones.

For every zone z of a list, we define the equivalence class of z, written $eq(z)$, to be the set of the zones that are bound to contain the same cursors as z by literals in L. This computation is straight-forward. For example, here is the rule for deletion: for every $l[i, j] \in eq(z)$,

$$
\begin{array}{llllll}
L \models_T delete(l, c, l') \approx \mathtt{t} & \text{and} & L \models_T j < position(l, c) & \rightarrow & l'[i, j] & \in eq(z) \\
L \models_T delete(l, c, l') \approx \mathtt{t} & \text{and} & L \models_T position(l, c) < i & \rightarrow & l'[i-1, j-1] & \in eq(z) \\
L \models_T delete(l', c, l) \approx \mathtt{t} & \text{and} & L \models_T j < position(l', c) & \rightarrow & l'[i, j] & \in eq(z) \\
L \models_T delete(l', c, l) \approx \mathtt{t} & \text{and} & L \models_T position(l', c) \leq i & \rightarrow & l'[i+1, j+1] & \in eq(z)
\end{array}
$$

The set $eq(z)$ has properties that will be useful to complete the proof:

1. Every element of $eq(z)$ is a zone.
2. Every zone in $eq(z)$ has the same length.
3. If a zone in $eq(z)$ starts with 0 then they all start with 0.
4. If we have both $L \vDash_T position(l, c) > 0$ and $l[position(l, c), \_] \in eq(z)$ then, for every zone $l'[i, \_] \in eq(z)$, $L \vDash_T position(l', c) \approx i$.

From the last two properties, we deduce that each list l appears at-most once in $eq(z)$. As a consequence, we can associate a free cursor variable to each position in the equivalent zone of a list without creating lists that may contain the same cursor twice. While there is a zone $l[i, j]$, with l known and $i < j$:

- We compute the set of zones $eq(l[i, j])$.
- To each k such that $i < k \leq j$, we associate a fresh cursor c_k.
- For each $l'[i', j']$ and each k' such that $i' < k' \leq j'$, $position(l', c_{k'-i'+i}) \approx k'$ is added to L.

Once there is no more zone $l[i, j]$, with $l \in \mathcal{T}(L)$ and $i < j$, for every term of type *list* and every term of type *cursor* of L, we can add $position(l, c) \approx 0$ to L. We can check straight-forwardly that W_{dlli} is still true in L. We now have an interpretation in the theory of doubly-linked lists of the iterative part of every list that appears in L.

Let us now consider the content part. Let e be a fresh constant of type *element_type*. For every term t of type *element_type* in $\mathcal{T}(L)$, we add $equal\_elements(t, e) \not\approx \mathtt{t}$ to L. The set L is still satisfiable since *element_type* is uninterpreted. We then map $element(l, c)$ to e for any term l of type *list* and any term c of type *cursor* in L such that $element(l, c)$ is not in L modulo T. Each axiom with $element(l, c)$ as a trigger either deduces an equality or an equivalence between new terms, or a non-equivalence between a known term and a new term. As a consequence, W_{dlli} is still true in L.

Remark 4 Here we are axiomatizing lists of an abstract infinite type. This proof works for any element type with an infinite number of equivalence classes. If the element type has a finite number of equivalence classes, let us call it n, then W_{dlli} is not complete any more. For example, consider the finite set of literals L with n constants of type element $e_1 \ldots e_n$ containing $position(l, find\_first(l, e_i)) > 1$ and $e_i \not\approx e_j$ for every i and $j \in 1 \ldots n$. This set is unsatisfiable, as the first element of l must be equal to one of the elements e_i by cardinality of the type of elements. However, since $element(l, first(l))$ is not known in L, we cannot use FIND_FIRST_NOT to deduce that it cannot be equivalent to any of the elements e_i which would lead to a contradiction.

We have constructed a model for L for the axiomatization W_{dlli} described in Sect. 3.2 without the triggers. As a consequence, the axiomatization with the triggers is complete with respect to this theory.

4 Designing Terminating and Complete Axiomatizations

There is no universal recipe for designing an axiomatization nor for proving its termination and completeness. Like for designing any decision procedure, the axiomatization and the proofs strongly depend on the theory they decide. In this section, we give good practices, tips, and debugging techniques for designing terminating and complete axiomatizations. We also give several techniques that can be computer-assisted using a decision procedure for the background theory T.

4.1 Proving Termination of an Axiomatization

Here we list several techniques that can be attempted to do a proof of termination of a given axiomatization. First, we can define properties stronger than termination which can be checked in an automatable way. The idea is to over-approximate the set of literals that can be created by the axiomatization. For each such literal, we also compute a guard, that is, a set of literals that must be true for the literal to be deduced. We use fresh constant symbols to model terms that have been used to instantiate a universal quantifier:

Definition 13 For every pseudo-clause C, we define an over-approximation $\mathcal{N}(C)$ of the literals that can be deduced from C. To compute the guards associated with each literal, we use an accumulator G which is augmented whenever we go through a witness, a trigger, or a universal quantifier. We define $\mathcal{N}(C) = \mathcal{N}(C, \varnothing)$ where:

$$\begin{aligned}
\mathcal{N}(l, G) &= \{(l, G)\} \\
\mathcal{N}(\langle l \rangle C, G) &= \mathcal{N}(l, G) \cup \mathcal{N}(C, G \cup l) \\
\mathcal{N}([l]C, G) &= \mathcal{N}(C, G \cup l) \\
\mathcal{N}(\forall x.C, G) &= \mathcal{N}(C[x \mapsto a], G \cup \{a \approx a\}) \qquad a \text{ is a fresh constant symbol} \\
\mathcal{N}(C, G) &= \bigcup_{\varphi \in C} \mathcal{N}(\varphi, G)
\end{aligned}$$

For a set of pseudo-clauses W, we write $\mathcal{N}(W)$ for $\bigcup_{C \in W} \mathcal{N}(C)$.

We now show that $\mathcal{N}(W)$ is a good over-approximation of the set of literals that can be produced by W:

Lemma 3 *Let L be a set of literals. Let A be a truth assignment in an instantiation tree Z of $(W \cup L) \cdot \varnothing$. For every closure $l \cdot \sigma$ added to a set of theory clauses V during the construction of A, there is a pair $(l\tau, G)$ in $\mathcal{N}(W)$ and a substitution μ from constant symbols to terms such that $\tau\mu|_{vars(l)}$ is σ and $\lfloor V \rfloor \rhd_T G\mu$.*

Proof We call N the set of pairs (C, G) to which $\mathcal{N}$ is applied during the computation of $\bigcup_{C \in W} \mathcal{N}(C)$. We show by induction over the construction of A that, whenever we add a closure $C \cdot \sigma$ to a set of theory clauses V during the construction of A, there is a pair $(C\tau, G)$ in N and a substitution μ from constant symbols to terms such that $\tau\mu|_{vars(C)}$ is σ and $\lfloor V \rfloor \rhd_T G\mu$. Since $\mathcal{N}(W) \subseteq N$, this is enough to conclude the proof.

For simplicity, we separate the handling of a witness $\langle l \rangle C \cdot \sigma$ in the construction of a truth assignment into two parts: we first add $l \cdot \sigma$ and then $C \cdot \sigma$.

If $C \in W$ then $(C, \varnothing) \in N$. Otherwise, one of the following holds:

- $\varphi \cdot \sigma$ was added to V because $\varphi \cdot \sigma \vee C' \cdot \sigma' \in V$. By induction hypothesis, there is $((\varphi \vee C')\tau, G) \in N$ where τ maps free variables to constant symbols and a substitution μ from constant symbols to terms such that $\tau\mu|_{vars(\varphi \vee C')}$ is $\sigma \cup \sigma'$ and $\lfloor V \rfloor \rhd_T G\mu$. By definition of $\mathcal{N}$, we have $(\varphi\tau, G) \in N$.
- $l \cdot \sigma$ and $C \cdot \sigma$ were added to V because $\langle l \rangle C \cdot \sigma \in V$. By induction hypothesis, there is $(\langle l \rangle C\tau, G) \in N$ where τ maps free variables to constant symbols and a substitution μ from constant symbols to terms such that $\tau\mu|_{vars(\langle l \rangle C)}$ is σ and $\lfloor V \rfloor \rhd_T G\mu$. By definition of $\mathcal{N}$, $(l\tau, G) \in N$ and $(C\tau, G \cup l\tau) \in N$. Since $l \cdot \sigma \in V$, we have $\lfloor V \rfloor \rhd_T (G \cup l\tau)\mu$.
- $C \cdot \sigma$ was to V because $[l]C \cdot \sigma \in V$ and $\lfloor V \rfloor \rhd_T l\sigma$. By induction hypothesis, there is $([l]C\tau, G) \in N$ where τ maps free variables to constant symbols and a substitution μ from constant symbols to terms such that $\tau\mu|_{vars([l]C)}$ is σ and $\lfloor V \rfloor \rhd_T G\mu$. By definition of $\mathcal{N}$, $(C\tau, G \cup l\tau) \in N$. Since $\lfloor V \rfloor \rhd_T l\sigma$, we have $\lfloor V \rfloor \rhd_T (G \cup l\tau)\mu$.

– A new instance $C \cdot (\sigma \cup [x \mapsto t])$ was added to V because $\forall x.C \cdot \sigma \in V$ and $t \in \mathscr{T}(\lfloor V \rfloor)$. By induction hypothesis, there is $(\forall x.C\tau, G) \in N$ where τ maps free variables to constant symbols and a substitution μ from constant symbols to terms such that $\tau\mu|_{vars(\forall x.C)}$ is σ and $\lfloor V \rfloor \rhd_T G\mu$. By definition of N, there is a new constant symbol a such that $(C(\tau \cup [x \mapsto a]), G \cup a \approx a) \in N$. Since a is fresh in $C\tau$ and G, we extend μ with $[a \mapsto t]$. Since $t \in \mathscr{T}(\lfloor V \rfloor)$, we have $\lfloor V \rfloor \rhd_T (G \cup a \approx a)(\mu \cup [a \mapsto t])$. □

Using this approximation of the set of literals that can be deduced from an axiomatization W, we can show that W is terminating in several special cases. First, let us assume that W can only produce terms that either are constants or are equal to an already known term:

Definition 14 Let (l, G) be an element of $\mathscr{N}(W)$. We say that (l, G) *cannot create any new term* if, for every term $t \in \mathscr{T}(l)$, either t contains no newly introduced constant or it is equal to some term in G modulo $G \cup l$.

Theorem 3 *If no pair (l, G) in $\mathscr{N}(W)$ can create a new term then W is terminating.*

Proof Let L be a set of literals. Let S be $\mathscr{T}(L) \cup \{t \mid t \text{ is a ground term of } W\}$.

We show that, for every truth assignment A in an instantiation tree Z of $(W \cup L) \cdot \varnothing$, we have $\lfloor A \rfloor \cup known(S) \vDash_T known(\mathscr{T}(\lfloor A \rfloor))$. This is enough to show that every instantiation tree of $(W \cup L) \cdot \varnothing$ is finite as there can only be one new instance per pair of a universally quantified formula in W and a ground term in S.

Let A be a truth assignment in an instantiation tree Z of $(W \cup L) \cdot \varnothing$. By Lemma 3, whenever we add a closure $l \cdot \sigma$ to a set of theory clauses V during the construction of A, there is a pair $(l\tau, G)$ in $\mathscr{N}(W)$ and a substitution μ from constant symbols to terms such that $\tau\mu|_{vars(l)}$ is σ and $\lfloor V \rfloor \rhd_T G\mu$. By hypothesis, for every term t in l, either t contains no free variables (and thus is in S by definition of S) or $G \cup l\tau \cup known(\mathscr{T}(G)) \vDash_T known(t\tau))$. In the second case, $G\mu \cup l\sigma \cup known(\mathscr{T}(G\mu)) \vDash_T known(t\sigma)$. Since $\lfloor V \rfloor \rhd_T G\mu$, we have $\lfloor V \rfloor \cup l\sigma \cup known(\lfloor V \rfloor) \vDash_T known(t\sigma)$. By immediate induction over the construction of A, we have $\lfloor A \rfloor \cup known(S) \vDash_T known(\mathscr{T}(\lfloor A \rfloor))$. □

Example 6 Let us consider the theory W_{array} of non-extensional arrays defined in Example 1. For every pair $(l, G) \in \mathscr{N}(W_{array})$, the pair (l, G) cannot create any new term. For example, for the first axiom, we have:

$$\begin{aligned}
&\mathscr{N}(\forall a,\ i,\ e.[set(a, i, e)]\ (get(set(a, i, e), i) \approx e)) \\
&\quad = \mathscr{N}([set(a, i, e)]\ (get(set(a, i, e), i) \approx e), \{a \approx a,\ i \approx i,\ e \approx e\}) \\
&\quad = (get(set(a, i, e), i) \approx e, \{set(a, i, e) \approx set(a, i, e),\ a \approx a,\ i \approx i,\ e \approx e\})
\end{aligned}$$

If we remove the trigger from this axiom, then it can produce a new term $set(a, i, e)$ as we have $\mathscr{N}(\forall a,\ i,\ e.\ (get(set(a, i, e), i) \approx e)) = (get(set(a, i, e), i) \approx e, \{a \approx a,\ i \approx i,\ e \approx e\})$ and $get(set(a, i, e), i) \approx e \cup known(\mathscr{T}(\{a,\ i,\ e\})) \nvDash_T known(set(a, i, e))$

Another special case is what we call *well guarded* axiomatizations, that is, axiomatizations where every axiom that can create new terms is guarded by a trigger containing an uninterpreted term which cannot be created by the axiomatization:

Definition 15 Let S be the set of terms that can be created by W, that is $\bigcup_{(l,G)\in\mathscr{N}(W)} \mathscr{T}(l) \setminus \mathscr{T}(G)$. We say that W is *well guarded* if, for every pair $(l, G) \in \mathscr{N}(W)$, either (l, G) cannot create any new term or, for every newly introduced constant $c \in \mathscr{T}(l)$, there is an uninterpreted function symbol f_c that appears directly above c in an element of $\mathscr{T}(G)$ and there is no element of S starting with f_c and containing any newly introduced constant.

Theorem 4 *If W is well guarded, then it is terminating.*

Proof Let S be the set of terms that can be created by W. Let f be an uninterpreted function symbol such that there is no element of S starting with f and containing a newly introduced constant.

Let L be a set of literals and let S_f be the set of ground terms starting with f in $W \cup L$. For every truth assignment A in an instantiation tree Z of $(W \cup L) \cdot \varnothing$ and for every term $t \in \mathscr{T}(\lfloor A \rfloor)$ starting with f, we have $\lfloor A \rfloor \cup known(S_f) \vDash_T known(t)$. Indeed, assume we add a closure $l \cdot \sigma$ to a set of theory clauses V during the construction of a truth assignment A such that l contains a term t starting with f such that $t\sigma$ is not already in V modulo equality. By Lemma 3, there is a pair $(l\tau, G)$ in $\mathscr{N}(W)$ and a substitution μ from constant symbols to terms such that $\tau\mu|_{vars(l)}$ is σ and $\lfloor V \rfloor \rhd_T G\mu$. By hypothesis, either $t\tau \in \mathscr{T}(G)$ and $\lfloor V \rfloor \cup known(\mathscr{T}(\lfloor A \rfloor)) \vDash_T known(t\sigma)$ or t contains no free variable and $t \in S_f$. By an immediate induction over the construction of A, $\lfloor A \rfloor \cup known(S_f) \vDash_T known(t\sigma)$.

This is enough to show that every instantiation tree of $(W \cup L) \cdot \varnothing$ is finite. Indeed, assume we add a closure $l \cdot \sigma$ to a set of theory clauses V during the construction of a truth assignment A. By Lemma 3, there is a pair $(l\tau, G)$ in $\mathscr{N}(W)$ and a substitution μ from constant symbols to terms such that $\tau\mu|_{vars(l)}$ is σ and $\lfloor V \rfloor \rhd_T G\mu$. Like in the proof of Theorem 3, we cannot create new terms if $(l\tau, G)$ cannot create any new term. Otherwise, let $c_1 \dots c_n$ be the newly introduced constants that appear in $l\tau$ and let $f_{c_1} \dots f_{c_n}$ be uninterpreted function symbols like in the definition of a well guarded axiomatization. Then, each $c_i\mu$ must be equal to a term appearing under the function symbol f_{c_i} in a term of the finite set $S_{f_{c_i}}$ modulo $\lfloor A \rfloor$. Thus, a literal associated with the pair $(l\tau, G)$ can only create new terms a finite number of times. □

Example 7 We can extend our theory of arrays W_{array} with a Boolean function $mem(a, e)$ such that whenever $mem(a, e) \approx \mathtt{t}$ there is an index i such that $get(a, i) \approx e$. We also introduce a function symbol *choose* that is used as a Skolem function symbol to get rid of the existential quantifier:

$$W^{\star}_{array} = W_{array} \cup \{\forall a,\ e.[mem(a, e) \approx \mathtt{t}](get(a, choose(a, e)) \approx e)\}$$

Notice that this axiomatization does not give the usual full semantics of *mem* as we cannot deduce anything when $mem(a, e) \not\approx \mathtt{t}$.

We can compute:

$$\begin{aligned}&\mathscr{N}(\forall a,\ e.[mem(a, e) \approx \mathtt{t}](get(a, choose(a, e)) \approx e))\\ &\quad = \mathscr{N}([mem(a, e) \approx \mathtt{t}](get(a, choose(a, e)) \approx e), \{a \approx a,\ e \approx e\})\\ &\quad = \{(get(a, choose(a, e)) \approx e, \{mem(a, e) \approx \mathtt{t},\ a \approx a,\ e \approx e\})\}\end{aligned}$$

Thus $W^{\star}_{array}$ can create new terms. $W^{\star}_{array}$ is well guarded though, since *mem* does not appear in $\bigcup_{(l,G)\in\mathscr{N}(W^{\star}_{array})} \mathscr{T}(l) \backslash \mathscr{T}(G)$.

Finally, we can consider axiomatizations that are not well guarded as a whole but can be subdivided into a sequence of subsets that are well guarded if the preceding subsets are removed:

Definition 16 We say that an axiomatization W is *well guarded piecewise* if there is a partition $W_1 \dots W_n$ of W such that, for every $i \in 1 \dots n$ and for every pair $(l, G) \in \mathscr{N}(W_i)$, either (l, G) cannot create any new term or, for every newly introduced constant $c \in \mathscr{T}(l)$, there is an uninterpreted function symbol f_c that appears directly above c in an element

of $\mathscr{T}(G)$ and there is no element of $\bigcup_{(l,G)\in\mathscr{N}(W_i\ldots W_n)} \mathscr{T}(l)\backslash\mathscr{T}(G)$ starting with f_c and containing any newly introduced constant.

Theorem 5 *If W is well guarded piecewise, then it is terminating.*

Proof Let $W_1 \ldots W_n$ be the aforementioned partition of W. Let L be a set of ground literals and let Z be an instantiation tree of $(W \cup L) \cdot \varnothing$. We show that Z is finite.

By Lemma 3, whenever we add a closure $l \cdot \sigma$ to a set of theory clauses V during the construction of a truth assignment A in Z, there is a pair $(l\tau, G)$ in $\mathscr{N}(W)$ and a substitution μ from constant symbols to terms such that $\tau\mu|_{vars(l)}$ is σ and $\lfloor V \rfloor \rhd_T G\mu$.

We consider sequences of sets of theory clauses $V_0 \ldots V_k \ldots$ such that V_0 is $(W \cup L) \cdot \varnothing$ and $V_1 \ldots$ are the intermediate steps of construction of truth assignments on some branch in Z: for any $k > 0$, V_k is either a truth assignment in Z or a subset thereof. In the first case, if V_k is not final, V_{k+1} is $V_k \cup C \cdot (\sigma \cup \{x \mapsto t\})$ where $(\forall x.C, t)$ is the new instance added to V_k in Z. In the second case, V_{k+1} is $V_k \cup \{C \cdot \sigma\}$, where $C \cdot \sigma$ is the new theory clause added to V_k during construction of a truth assignment in Z.

Let $V_0 \ldots V_{k_0} \ldots$ be one such sequence and let us show that if there exists $i \in 1 \ldots n$ such that every closure $l \cdot \sigma$ added to the sequence after V_{k_0} has a counterpart $(l\tau, G)$ in $\mathscr{N}(W_i \cup \cdots \cup W_n)$, then the sequence is necessarily finite. We proceed by induction over i.

Let S be the set of terms that can be created by $W_i \cup \cdots \cup W_n$. Let f be an uninterpreted function symbol such that there is no element of S starting with f and containing a newly introduced constant. Let S_f be the set of ground terms starting with f in $\{C\sigma \mid C \cdot \sigma \in V_{k_0}\}$. Like in the proof of Theorem 4, for every set of theory clauses V_k in the sequence $V_{k_0} \ldots$ and for every term $t \in \mathscr{T}(\lfloor V_k \rfloor)$ starting with f, we have $\lfloor V_k \rfloor \cup known(S_f) \vDash_T known(t)$.

We can show that there can only be a finite number of literals $l \cdot \sigma$ added in the sequence $V_{k_0} \ldots$ such that $(l\tau, G)$ is in $\mathscr{N}(W_i)$ using the same reasoning as in the proof of Theorem 4. Indeed, if the pair $(l\tau, G)$ can create a new term, then, for every newly introduced constant c_i that appears in $l\tau$, there is an uninterpreted function symbol f_{c_i} protecting it like in the definition of a well guarded axiomatization. Then, $c_i\mu$ must be equal to a term appearing under the function symbol f_{c_i} in a term of the finite set $S_{f_{c_i}}$ modulo $\lfloor V_k \rfloor$. Thus, there can only be a finite number of literals $l \cdot \sigma$ added in the sequence $V_{k_0} \ldots$ such that $(l\tau, G)$ is in $\mathscr{N}(W_i)$.

By induction hypothesis, there can only be finite sequences between them. Thus, the sequence $V_0 \ldots V_{k_0} \ldots$ is finite. □

Example 8 We may extend $W^{\star}_{array}$ with some additional information about *mem*, like, for example, the fact that an array a updated with a value e always contains e:

$$W^{\star\star}_{array} = W^{\star}_{array} \cup \{\forall a,\ i,\ e.[set(a, i, e)](mem(set(a, i, e), e) \approx \mathtt{t})\}$$

Here we see that $W^{\star\star}_{array}$ is not well guarded anymore, as terms starting with *mem* can now be created by the axiomatization. It is well guarded piecewise though, using the partition

$$W_1 = W_{array} \cup \{\forall a,\ i,\ e.[set(a, i, e)](mem(set(a, i, e), e) \approx \mathtt{t})\}$$
$$W_2 = \{\forall a,\ e.[mem(a, e) \approx \mathtt{t}](get(a, choose(a, e)) \approx e)\}$$

Indeed, W_1 cannot create any new term and the symbol *mem* does not appear in the terms created by W_2.

Remark 5 If the axiomatization W is written in a multi-sorted language and the quantification is done on variables of sorts $s_1 \ldots s_n$, then it is enough to consider the terms created by the axiomatization that are of sorts $s_1 \ldots s_n$. For example, the axiomatization W_{dlli} defined in Sect. 3.2 is well guarded piecewise, as can be seen from the proof of termination in Sect. 3.3.

When an axiomatization does not have any of the above-stated properties, a different approach must be used. One such case is axiomatization W_{conv} presented in Example 3. Indeed, since W_{conv} admits infinite instantiation trees, the set of literals it creates cannot be bounded, and Theorems 3–5 do not apply. Instead, we consider the set of new terms created by the axiomatization using a particular tree construction strategy, and show that there can be only a finite number of them.

Such a proof can be complex, in particular if quantification is done on interpreted sorts. Using the same idea as in the definition of well guarded axiomatizations, it can be simplified further if the triggers of the axiomatization contain uninterpreted function symbols. Indeed, in this case, it is enough to show that there can only be a finite number of new terms starting with the function symbols that appear in triggers.

Like in the definition of well guarded piecewise axiomatization, such a proof can sometimes be done modularly. The idea is to find a partition $S_0 \dots S_n$ of the set of function symbols that appear in triggers in an axiomatization W such that, if W_i is the subset of formulas of W using symbols of S_i in triggers, W_i can only create new terms starting with symbols from $S_0 \cup \dots \cup S_i$. We can then show separately that, for each set S_i, formulas of W_i cannot create an infinite number of new terms starting with function symbols in S_i.

Note that this reasoning can be strengthened to only check for new terms that actually match triggers in the axiomatization. Still, in this case, reasoning must be done modulo equality. This is even more difficult if triggers contain terms of an interpreted sort, since equality between such terms can be deduced by theory reasoning.

4.2 Designing a Complete Axiomatization

The natural way to prove the completeness of an axiomatization W is to give a general method for completing a world L in which W is true into a model of the theory. This implies that terms of L that are interpreted must be given a value in this world which may create new equalities between them. The reasoning is much easier if these equalities cannot unlock new triggers in W. In multi-sorted logic, it is enough to restrict triggers so that, for every trigger l in the axiomatization and every subterm t of l of an interpreted sort, either t is a variable or t is top-level in l. Indeed, the only triggers that can be unlocked are then those where l becomes true because of the new equalities, which are generally much easier to reason about. For example, we use this technique in the proof of completeness of the theory of doubly-linked lists to show that adding an equality between known integer terms cannot unlock new deductions. Constants of interpreted types can be allowed in triggers without losing this property if they are known in every world in which the axiomatization is true.

Another important point is that, when triggers guard a disjunction, they should not prevent us from deducing an element of the disjunction when the others are false. In the same way, if an element of the disjunction is an equality, then the rewriting should be possible both ways. For example, in the theory of non-extensional arrays, we duplicated the second axiom so that there is a trigger coming from each side of the equality:

$$\forall a,\ i,\ j,\ e.[get(set(a,i,e),\ j)]\ (i \not\approx j \rightarrow get(set(a,i,e),\ j) \approx get(a,\ j))$$
$$\forall a,\ i,\ j,\ e.[set(a,i,e),\ get(a,\ j)]\ (i \not\approx j \rightarrow get(set(a,i,e),\ j) \approx get(a,\ j))$$

In some cases, there are good reasons not to apply this rule. It allows to orient deduction and rewriting. Still, if this choice is made, then there can be literals that cannot be deduced to be true and terms that cannot be deduced to be known even if they are entailed by the theory. Such literals and terms should not appear in triggers.

Example 9 Consider triggers containing the function symbol $equal\_lists$ in the theory of doubly-linked lists. We cannot deduce that $equal\_lists(l_1, l_2) \approx \mathtt{t}$ is true for two lists l_1 and l_2 if the term $equal\_lists(l_1, l_2)$ does not appear in the context, since every axiom involving a term starting with $equal\_lists$ has this term as a trigger.

In general, $equal\_lists$ should not be used as a trigger if we want a complete axiomatization. Adding the following axiom for prefix of a list would be at the cost of the completeness of our theory:

IS_PREFIX_INV:

$$\begin{array}{l}\forall l_1, l_2 : list.[is\_prefix(l_1, l_2)]\ is\_prefix(l_1, l_2) \not\approx \mathtt{t} \rightarrow \\ \quad (\forall c : cursor.[equal\_lists(l_1, left(l_2, c))] \\ \qquad (has\_element(l_2, c) \approx \mathtt{t} \vee c \approx no\_element) \rightarrow \\ \qquad\quad equal\_lists(l_1, left(l_2, c)) \not\approx \mathtt{t})\end{array}$$

For example, consider the unsatisfiable set of literals $L = \{is\_prefix(empty, l) \not\approx \mathtt{t}\}$. We cannot deduce that L is unsatisfiable in our theory using our axiom as we do not generate the term $equal\_lists(empty, left(l, first(l)))$.

A notable exception to this principle are defining axioms. Axioms used to provide an uninterpreted symbol f with its meaning can generally use f in their triggers even though we cannot deduce the value of a term starting with f in the axiomatization every time we can deduce it in the theory. For example, the axiom EQUAL_LISTS_LENGTH has $equal\_lists(l_1, l_2)$ as a trigger and yet the axiomatization is complete. Indeed, in the theory of doubly-linked lists, we cannot deduce that two lists are equal if we do not know *a priori* that they have the same length. Thus, every world in which the axiomatization is true can be completed by assuming $equal\_lists(l_1, l_2) \not\approx \mathtt{t}$ whenever we do not have $length(l_1) \approx length(l_2)$.

EQUAL_LISTS_LENGTH:

$$\begin{array}{l}\forall l_1, l_2 : list.[equal\_lists(l_1, l_2)] \\ \quad equal\_lists(l_1, l_2) \approx \mathtt{t} \rightarrow length(l_1) \approx length(l_2)\end{array}$$

4.3 An Automatable Debugger for Completeness

When designing an axiomatization W, it may be useful to have an automatable way to search for counter-examples to the completeness of W with respect to the first-order axiomatization W' which is W with triggers replaced with implications. More precisely, we look for sets of literals L such that $L \cup W'$ is unsatisfiable in first-order logic but there is a world in which $L \cup W$ is true. The algorithm below searches systematically for sets of literals L such that $L \cup W'$ is unsatisfiable in first-order logic. An implementation of the solver can then be used to decide if there is a world in which $L \cup W$ is true (see Definition 7).

The idea is to apply paramodulation to the axioms in W, seen as clauses with free variables, to deduce new clauses. We use lazy paramodulation: for $C_1 \vee f(t_1, \ldots, t_n) \approx s$ and $C_2 \vee A[f(t'_1, \ldots, t'_n)] \in G$, we produce $C_1 \vee C_2 \vee t_1 \not\approx t'_1 \vee \cdots \vee t_n \not\approx t'_n \vee A[s]$. Every newly deduced clause can be negated to produce a potential counter-example.

Example 10 The two first-order axioms of the theory of arrays are converted into the two clauses $get(set(a, i, e), i) \approx e$ and $i \approx j \vee get(a, j) \approx get(set(a, i, e), j)$ where a, i, j, and e are free variables. We can then use the above algorithm to infer the counter-examples that we produced in Example 2. The first axiom coupled with itself gives:

$$set(a_1, i_1, e_1) \not\approx set(a_2, i_2, e_2) \vee i_1 \not\approx i_2 \vee e_1 \approx e_2$$

So we deduce a potential counter-example $\{set(a_1, i, e_1) \approx set(a_2, i, e_2),\ e_1 \not\approx e_2\}$ which is indeed a counter-example for the axiom with a bad trigger in Example 2:

$$\forall a, i, e.[get(set(a, i, e), i)]\ get(set(a, i, e), i) \approx e$$

The second axiom contains two occurrences of *get*. Rewriting the equality both ways in each occurrence of *get* gives the four following clauses. The first and the fourth give the counter-examples for the other two cases in Example 2:

$$i_1 \approx j \vee i_2 \approx j \vee set(a_1, i_1, e_1) \not\approx set(a_2, i_2, e_2) \vee get(a_1, j) \approx get(a_2, j)$$
$$i_1 \approx j \vee i_2 \approx j \vee get(a, j) \approx get(set(set(a, i_1, e_1), i_2, e_2), j)$$
$$i_1 \approx j \vee i_2 \approx j \vee get(set(set(a, i_2, e_2), i_1, e_1), j) \approx get(a, j)$$
$$i_1 \approx j \vee i_2 \approx j \vee get(set(a_1, i_1, e_1), j) \approx get(set(a_2, i_2, e_2), j)$$

We see that this automatable approach manages to find all the counter-examples we used in Example 2.

5 Extension of DPLL(T) to the Logic with Triggers

In this section, we introduce an extension of abstract DPLL modulo theories [31] that handles formulas with triggers and witnesses. We show that if a set of axioms is sound and complete with respect to a theory T' that extends the solver's background theory T, then our procedure is sound and complete on any ground satisfiability problem in T'. Moreover, we show that under certain fairness restrictions on derivations, our procedure terminates on any ground satisfiability problem if the axiomatization is terminating. This section and the following one are independent from the rest of this article and can be skipped by readers not interested in SMT solver development.

5.1 Preliminaries

We describe a solver that takes a set of first-order axioms with triggers and witnesses, denoted Ax, and a set of ground clauses, denoted G. Before starting the DPLL procedure, we skolemize and clausify the axioms in Ax, producing a set of pseudo-clauses W, as described in Sect. 2.4. Then we convert W into a set of theory clauses (disjunctions of closures) by coupling it with the empty substitution: $W \cdot \varnothing$. We run the procedure on $W \cdot \varnothing$ and G, with one of the three possible outcomes:

- the solver returns *Unsat*, meaning that the union $Ax \cup G$ is unsatisfiable—therefore, if Ax is sound with respect to T', the set G is T'-unsatisfiable;
- the solver returns *Sat*, meaning that there exists a ground formula G' such that $G' \vDash_T G$ and the union $Ax \cup G'$ is feasible—therefore, if Ax is complete with respect to T', then G' is T'-satisfiable, and consequently, G is T'-satisfiable;
- the solver runs indefinitely—if W is terminating, this cannot happen in a solver satisfying the conditions defined in our framework.

When we don't have the soundness and completeness properties for Ax, the union $Ax \cup G$ may be both feasible (true in some world) and unsatisfiable (false in every complete world). In this case, the solver is nondeterministic. For example, let Ax be the single axiom $[a] \bot$ and G the single clause $a \approx a \vee \top$. Then the solver may drop $\top$ from G, learn constant a, remove the trigger and let the contradiction out, producing *Unsat*. Alternatively, the solver

may discard the whole clause G as redundant and return *Sat*: the union $Ax \cup G$ is true in the empty world.

Note the slightly complicated explanation of the *Sat* case: instead of finding a world directly for $Ax \cup G$, the solver only ensures the feasibility of Ax joined with some ground antecedent of G modulo T, which is not at all guaranteed to contain the same terms and to behave the same as G with respect to the $\rhd_T$ relation. This is an important feature of our approach: the input problem G is considered modulo theory T and the solver is free to make simplifications as long as they are permitted by T, such as learning or forgetting redundant clauses modulo T, without regard to known and unknown terms. In that way, we stay consistent with the traditional semantics of DPLL. On the other hand, axiomatization Ax is treated according to the semantics in Sect. 2.2.

To maintain this distinction, the solver works with two distinct kinds of clauses. The clauses coming from Ax are theory clauses: disjunctions of closures that accumulate ground substitutions into free variables. The clauses coming from G are the usual disjunctions of ground literals; we call them *user clauses* to distinguish them from the clauses of the first kind. The empty clause $\bot$ is considered to be a user clause. A *super-clause* is either a theory clause or a user clause.

Besides the current set of clauses (which can be modified by learning and forgetting), DPLL-based procedures maintain a set of currently assumed facts. In our procedure, these facts, which we collectively call *super-literals*, may be of three different kinds:

- a literal l;
- a closure $\varphi \cdot \sigma$;
- an *anti-closure* $\neg(\varphi \cdot \sigma)$.

The latter kind appears when we backtrack a decision step over a closure. We extend the $\mathscr{T}$ operation (set of subterms) to substitutions, closures, and anti-closures as follows:

$$\begin{aligned}
\mathscr{T}(\sigma) &\triangleq \bigwedge_{x \in dom(\sigma)} \mathscr{T}(x\sigma) \\
\mathscr{T}(l \cdot \sigma) &\triangleq \mathscr{T}(l\sigma) \\
\mathscr{T}(\varphi \cdot \sigma) &\triangleq \mathscr{T}(\sigma) \qquad \text{if } \varphi \text{ is not a literal} \\
\mathscr{T}(\neg(\varphi \cdot \sigma)) &\triangleq \varnothing
\end{aligned}$$

Non-literal closures $\varphi \cdot \sigma$, where φ is a formula under a trigger, a witness, or a universal quantifier, are treated as opaque boxes so that the only terms we can learn from them are the ones brought by substitution σ. An anti-closure $\neg(\varphi \cdot \sigma)$ does not give us any new terms at all (and thus should not be confused with $(\neg\varphi) \cdot \sigma$). Indeed, if the solver at some moment decides to assume a given closure and later reverts this decision, it should not retain the terms learned from that closure.

Given a set of super-literals M, we define $\textsc{Lit}(M)$ to be the set of literals in M, and $\textsc{Clo}(M)$ to be the set of closures in M. Given a set of super-clauses F, we define $\textsc{Lit}(F)$ to be the set of unit user clauses in F, and $\textsc{Clo}(F)$ to be the set of unit theory clauses in F.

To model the trigger mechanism, we need a way to protect a super-clause so that its elements are not available until a certain condition is fulfilled. We define a *guarded clause* as a pair $H \to C$, where the *guard* H is a conjunctive set of closures and C is a super-clause. If M is a set of super-literals and F a set of guarded clauses, we define the set of *available super-clauses* to be the set of super-clauses of F whose guard is directly in M:

$$\textsc{Avb}(F, M) \triangleq \textsc{Lit}(M) \cup \textsc{Clo}(M) \cup \{C \mid H \to C \in F \text{ and } H \subseteq M\}$$

Any more complex reasoning on guards is left to DPLL, which will be in charge of propagating literals from guards whenever they are entailed by the current partial model. We also use the set of guards of F, defined as $\text{GRD}(F) \triangleq \{H \mid H \to C \in F\}$.

We now extend Definitions 4 and 5 onto super-literals and guarded clauses.

Definition 17 (*Truth value*) Given a world L, we define what it means for a super-literal, a super-clause, a guard, or a guarded clause to be *true* in L, written $L \blacktriangleright_T F$, as follows:

$L \blacktriangleright_T l$	$L \vDash_T l$
$L \blacktriangleright_T \varphi \cdot \sigma$	$L \cup \mathit{known}(\mathscr{T}(L)) \vDash_T \mathit{known}(\mathscr{T}(\sigma))$ and $L \rhd_T \varphi\sigma$
$L \blacktriangleright_T \neg(\varphi \cdot \sigma)$	if $L \cup \mathit{known}(\mathscr{T}(L)) \vDash_T \mathit{known}(\mathscr{T}(\sigma))$ then $L \not\rhd_T \varphi\sigma$
$L \blacktriangleright_T C$	C is a user clause and $L \vDash_T C$
$L \blacktriangleright_T C$	C is a theory clause and for some $\varphi \cdot \sigma \in C$, $L \blacktriangleright_T \varphi \cdot \sigma$
$L \blacktriangleright_T H$	H is a guard and for each $\varphi \cdot \sigma \in H$, $L \blacktriangleright_T \varphi \cdot \sigma$
$L \blacktriangleright_T H \to C$	if $L \blacktriangleright_T H$ then $L \blacktriangleright_T C$

We say that a super-literal is *false* in L when its negation is true in L. We call a super-literal, a super-clause, a guard, or a guarded clause *feasible* if there exists a world in which it is true. We call a super-literal, a super-clause, a guard, or a guarded clause *satisfiable* if there exists a complete world— which we then call its *model*—in which it is true.

On normal literals (not closures) and user clauses, $\blacktriangleright_T$ coincides with $\vDash_T$: a user clause C is true in a world L if and only if it is true in every model of L. On closures and theory clauses, $\blacktriangleright_T$ refers to $\rhd_T$: a theory clause is true in L if and only if one of its closures is true in L. By a slight abuse of terminology, we reuse the terms of Definitions 4 and 5, even though they have different meanings for ordinary literals; in this section, we follow Definition 17.

We define a version of entailment that treats closures as opaque "atoms" whose arguments are given by the accumulated substitution. This is the entailment used in the DPLL solver, the semantics of closures being taken care of by specific additional rules.

Definition 18 We define an encoding $[\![\ ]\!]$ of super-literals and guarded clauses into literals and clauses. In the rules below, P_φ is a fresh predicate symbol that we associate to every pseudo-literal φ which is not a literal. The arity of P_φ is the number of free variables in φ.

$$
\begin{aligned}
[\![l]\!] &\triangleq l \\
[\![l \cdot \sigma]\!] &\triangleq l\sigma \\
[\![\varphi \cdot \sigma]\!] &\triangleq P_\varphi(\mathit{vars}(\varphi))\sigma \qquad \text{if } \varphi \text{ is not a literal} \\
[\![\neg(\varphi \cdot \sigma)]\!] &\triangleq \neg[\![\varphi \cdot \sigma]\!] \\
[\![e_1 \vee \cdots \vee e_m]\!] &\triangleq [\![e_1]\!] \vee \cdots \vee [\![e_m]\!] \\
[\![(g_1 \wedge \cdots \wedge g_n) \to (e_1 \vee \cdots \vee e_m)]\!] &\triangleq \neg[\![g_1]\!] \vee \cdots \vee \neg[\![g_n]\!] \vee [\![e_1]\!] \vee \cdots \vee [\![e_m]\!]
\end{aligned}
$$

Let S be a conjunctive set of super-literals and/or guarded clauses. Let E be a super-literal, a super-clause, or a guarded clause. We define $S \vDash^\star_T E$ to be $[\![S]\!] \vDash_T [\![E]\!]$.

It is easy to see that $\vDash^\star_T$ is a conservative extension of the usual first-order entailment $\vDash_T$ onto super-literals and guarded clauses. We also show that $S \vDash^\star_T E$ properly approximates the fact that models of S are models of E.

Lemma 4 *Let S be a conjunctive set of super-literals and/or guarded clauses and let E be a super-literal, a super-clause, or a guarded clause such that $S \vDash^\star_T E$. Then every model of S is a model of E.*

Proof Let L be a model of S. We define $L' = L \cup \{[\![e]\!] \mid e$ is a super-literal such that $L \blacktriangleright_T e\}$. The set L' is satisfiable and complete. Indeed, for every closure $\varphi \cdot \sigma$ and every substitution σ' such that $L \vDash_T \sigma \doteq \sigma'$, $L \blacktriangleright_T \varphi \cdot \sigma$ if and only if $L \not\blacktriangleright_T \neg(\varphi \cdot \sigma')$ and $L \blacktriangleright_T \neg\varphi \cdot \sigma$ if and only if $L \not\blacktriangleright_T \varphi \cdot \sigma'$.

We show that $L' \vDash_T [\![S]\!]$. Since $L \blacktriangleright_T S$, for every super-literal e in S, $[\![e]\!] \in L'$. Let $H \to C$ be a guarded clause of S. If $L \not\blacktriangleright_T H$ then there is $e \in H$ such that $L \blacktriangleright_T \neg e$. By construction, $[\![\neg e]\!] \in L'$ and $L' \vDash_T [\![H \to C]\!]$. Otherwise, there is $e \in C$ such that $L \blacktriangleright_T e$, $[\![e]\!] \in L'$ and $L' \vDash_T [\![H \to C]\!]$.

Since $S \vDash_T^\star E$, L' is a model of $[\![E]\!]$. Thus, if E is a super-literal then $[\![E]\!] \in L'$ and, by construction of L', $L \blacktriangleright_T E$. If E is a guarded clause $(g_1 \wedge \cdots \wedge g_n) \to (e_1 \vee \cdots \vee e_m)$ then there is $e \in \{\neg g_1 \ldots \neg g_n, e_1 \ldots e_m\}$ such that $L \blacktriangleright_T e$ and therefore either $L \not\blacktriangleright_T g_1 \wedge \cdots \wedge g_n$ or $L \blacktriangleright_T e_1 \vee \cdots \vee e_m$. The case where E is a super-clause is handled in the same way. □

5.2 Informal Introduction

We start by a quick overview of the abstract DPLL modulo theories framework. For a more detailed description, interested readers can either look at the Handbook of Satisfiability [6] or at the implementation oriented survey by Nieuwenhuis et al. [31]. Here, we only present the abstract DPLL(T) framework. How an actual implementation of DPLL(T) may combine these rules in practice is explained in Sect. 6.1.

A set of rules for classical DPLL(T) is given in Fig. 1. A DPLL(T) procedure applies these rules to construct a model of a set of ground clauses F. The partial model is represented as a set of literals M that are assumed to be true. We call *state* of the procedure the pair $M \parallel F$ and we say that a literal l is *defined* in M if either l or $\neg l$ is in M.

`UnitPropagate`:

$$M \parallel F, C \vee l \implies M l \parallel F, C \vee l \quad \textbf{if} \begin{cases} \neg C \subseteq M \\ l \text{ is undefined in } M \end{cases}$$

`Decide`:

$$M \parallel F \implies M l^{\mathrm{d}} \parallel F \quad \textbf{if} \begin{cases} l \text{ or } \neg l \text{ occurs in a clause of } F \\ l \text{ is undefined in } M \end{cases}$$

`Fail`:

$$M \parallel F, C \implies \mathit{fail} \quad \textbf{if} \begin{cases} \neg C \subseteq M \\ M \text{ contains no decision literals} \end{cases}$$

`Restart`:

$$M \parallel F \implies \varnothing \parallel F$$

T`-Propagate`:

$$M \parallel F \implies M l \parallel F \quad \textbf{if} \begin{cases} l \notin M \\ M \vDash_T l \\ l \text{ or } \neg l \text{ occurs in } F \end{cases}$$

T`-Learn`:

$$M \parallel F \implies M \parallel F, C \quad \textbf{if} \begin{cases} \text{every atom of } C \text{ occurs in } F \cup M \\ F \vDash_T C \end{cases}$$

T`-Forget`:

$$M \parallel F, C \implies M \parallel F \quad \textbf{if} \begin{cases} F \vDash_T C \end{cases}$$

T`-Backjump`:

$$M l^{\mathrm{d}} N \parallel F \implies M l' \parallel F \quad \textbf{if} \begin{cases} \text{there is } C \in F \text{ such that } \neg C \subseteq M l^{\mathrm{d}} N \\ F, M \vDash_T l', \\ l' \text{ is undefined in } M, \text{ and} \\ l' \text{ or } \neg l' \text{ occurs in } F \cup M l^{\mathrm{d}} N \end{cases}$$

Fig. 1 Transition rules of Abstract DPLL Modulo Theories

Literals of clauses in F can be given an arbitrary truth value using the rule `Decide`. Literals of M whose truth value was chosen arbitrarily are labeled with a letter `d` and called decision literals. If every element of a clause is false but one, the remaining element has to be true for the clause to be verified. It can be propagated into the partial model using `UnitPropagate`. At any moment, the solver may abandon the current partial model using `Restart`. If every element of a clause is false in the partial model M then it is called a *conflict clause*. If there is a conflict clause in F and there is no arbitrary choice in M, then a special state, named *fail*, can be reached through `Fail`. It means that no model can be found for F, and thus F is unsatisfiable.

The remaining rules introduce theory reasoning into the framework. If there is a literal l that appears in a clause of F which is entailed by M modulo T, then it can be propagated into M using T`-Propagate`. The set of ground clauses F can also be modified during the search using T`-Learn` and T`-Forget` to add or remove clauses that are redundant with respect to the other clauses in F.

Finally, if every element of a ground clause of F is false and there is at least a decision literal in M, the rule T`-Backjump` can be applied. It allows to remove one or several decisions of M as long as there is a new literal that can be added to M. A literal can be added to M if it is entailed by M and F. Usually, in practice, the unsatisfiable part of $M \cup F$ is analyzed to find the first decision literal responsible for the conflict.

We finally define when a solver implementing DPLL is allowed to deduce the satisfiability or unsatisfiability of a set of ground clauses F:

Property 1 The solver can return *Unsat* on F if $\varnothing \parallel F \Longrightarrow^{\star} \mathit{fail}$.

Property 2 The solver can return *Sat* on F if $\varnothing \parallel F \Longrightarrow^{\star} M \parallel F'$ where:

(i) M contains at least an element per clause of F' and
(ii) M is satisfiable modulo T.

Now that we have presented classical DPLL(T), let us see on an example what should be added to accommodate clauses with quantifiers, triggers, and witnesses. We use the theory of arrays W_{array} defined in Example 1 as the axiomatization of an extension T' of the background theory T and the set of literals L_3 defined in Example 2 as a the user's problem whose unsatisfiability modulo T' we want to prove.

The solver is thus launched on the set:

$$
\begin{aligned}
F_0 &= L_3 \cup W_{array} \cdot \varnothing \\
&= \left\{ \begin{array}{l}
i \not\approx j, \\
get(a_1, j) \not\approx get(a_2, j), \\
set(a_1, i, e) \approx set(a_2, i, e), \\
\forall a,\ i,\ e.[set(a, i, e)]\ (get(set(a, i, e), i) \approx e) \cdot \varnothing, \\
\forall a,\ i,\ j,\ e.[get(set(a, i, e), j)]\ (i \not\approx j \rightarrow get(set(a, i, e), j) \approx get(a, j)) \cdot \varnothing, \\
\forall a,\ i,\ j,\ e.[set(a, i, e),\ get(a, j)]\ (i \not\approx j \rightarrow get(set(a, i, e), j) \approx get(a, j)) \cdot \varnothing
\end{array} \right\}
\end{aligned}
$$

Using our encoding into literals, we can apply classical DPLL(T) rules on F_0. We start from the empty partial model $M_0 = \varnothing$. The set $L_3 \cup W_{array} \cdot \varnothing$ only contains unit clauses. We can therefore use unit propagation to add them to M_0 so that $M_1 = L_3 \cup W_{array} \cdot \varnothing$. Without dedicated handling of closures, we can do nothing more.

We first demonstrate how guarded clauses are handled in our framework and leave the actual generation of guarded clauses from closures for later. Intuitively, a guard prevents DPLL(T) from looking into a clause until every element of the guard is satisfied by the

current partial model. DPLL(T) should be able to propagate guards that are implied by the current partial model modulo theory. However, to comply with our semantics, theory entailment on guards should use our semantics for formulas with triggers rather than the usual one, so that we need the presence of every sub-term of a trigger t to access a formula protected by t.

More precisely, a guarded clause $g_1 \wedge \cdots \wedge g_n \rightarrow e_1 \vee \cdots \vee e_m$ is handled by our extension of DPLL(T) in the following way:

- we cannot decide upon the truth value of g_i in a guard,
- we can only decide or propagate the truth value of e_i if $\{g_1 \ldots g_n\} \subseteq M$,
- we can only propagate the truth value of g_i in a guard using theory reasoning, and so if and only if $[\![g_i]\!]$ is true in the world $[\![M]\!]$.

Let us go back to our example. To show that L_3 is unsatisfiable in W_{array}, we need to instantiate twice the third axiom of the theory of arrays:

$$\forall a,\ i,\ j,\ e.[set(a, i, e),\ get(a, j)]\ (i \not\approx j \rightarrow get(set(a, i, e), j) \approx get(a, j))$$

once with a_1, i, j, and v and once with a_2, i, j, and v.

Since we focus on handling of guarded clauses for now, we do not use F_0 directly but a manually computed set F' containing both L_3 and two guarded clauses, one for each instance, where guards are used to model triggers:

$$F' = \left\{ \begin{array}{l} i \not\approx j, \\ get(a_1, j) \not\approx get(a_2, j), \\ set(a_1, i, e) \approx set(a_2, i, e), \\ set(a, i, v) \approx set(a, i, v) \cdot [a \mapsto a_1,\ i \mapsto i,\ v \mapsto e] \wedge \\ \quad get(a, j) \approx get(a, j) \cdot [a \mapsto a_1,\ j \mapsto j] \rightarrow \\ \quad\quad i \approx j \cdot [i \mapsto i,\ j \mapsto j] \vee \\ \quad\quad get(set(a, i, v), j) \approx get(a, j) \cdot [a \mapsto a_1,\ i \mapsto i,\ j \mapsto j,\ v \mapsto e], \\ set(a, i, v) \approx set(a, i, v) \cdot [a \mapsto a_2,\ i \mapsto i,\ v \mapsto e] \wedge \\ \quad get(a, j) \approx get(a, j) \cdot [a \mapsto a_2,\ j \mapsto j] \rightarrow \\ \quad\quad i \approx j \cdot [i \mapsto i,\ j \mapsto j] \vee \\ \quad\quad get(set(a, i, v), j) \approx get(a, j) \cdot [a \mapsto a_2,\ i \mapsto i,\ j \mapsto j,\ v \mapsto e] \end{array} \right\}$$

Let us demonstrate how guards can be used to refute F'. We start from the empty partial model $M'_0 = \varnothing$.

1. We firt use `UnitPropagate` to propagate literals from unit clauses.

$$M'_1 = \left\{ \begin{array}{l} i \not\approx j, \\ get(a_1, j) \not\approx get(a_2, j), \\ set(a_1, i, e) \approx set(a_2, i, e) \end{array} \right\}$$

2. For every closure $l \cdot \sigma$ occurring in a guard in F', we have $[\![M'_1]\!] \rhd [\![l \cdot \sigma]\!]$, because $\mathscr{T}(M'_1)$ contains all the needed terms. Thus, we can propagate the guards using `T-Propagate`:

$$M'_2 = M'_1 \cup \left\{ \begin{array}{l} set(a, i, v) \approx set(a, i, v) \cdot [a \mapsto a_1,\ i \mapsto i,\ v \mapsto e] \\ get(a, j) \approx get(a, j) \cdot [a \mapsto a_1,\ j \mapsto j] \\ set(a, i, v) \approx set(a, i, v) \cdot [a \mapsto a_2,\ i \mapsto i,\ v \mapsto e], \\ get(a, j) \approx get(a, j) \cdot [a \mapsto a_2,\ j \mapsto j] \end{array} \right\}$$

3. Since $[\![M_2']\!] \vDash_T [\![i \not\approx j \cdot [i \mapsto i,\ j \mapsto j]]\!]$, we can propagate this closure using `T-Propagate`. Note that, since this closure is not part of a guard, we do not require its terms to appear in the partial model M_2':

$$M_3' = M_2' \cup \{i \not\approx j \cdot [i \mapsto i,\ j \mapsto j]\}$$

4. Since, in the last two guarded clauses of F', every literal but one is invalidated by M_3', we can use `Unit-Propagate` to extend M_3' further:

$$M_4' = M_3' \cup \left\{ \begin{array}{l} get(set(a, i, v), j) \approx get(a, j) \cdot [a \mapsto a_1,\ i \mapsto i,\ j \mapsto j,\ v \mapsto e], \\ get(set(a, i, v), j) \approx get(a, j) \cdot [a \mapsto a_2,\ i \mapsto i,\ j \mapsto j,\ v \mapsto e] \end{array} \right\}$$

5. $[\![M_4']\!]$ is unsatisfiable. Since we did not make any decision while constructing M_4', F' is unsatisfiable.

To complete our running example, we need to extend our framework so that the two instances from F' can be deduced directly from the third axiom of W_{array}. For this, we introduce specific rules to convert triggers and quantifiers into guarded clauses of closures:

`Trigger-Unfold`:

$$M \parallel F \implies M \parallel F, [l]C \cdot \sigma \wedge l \cdot \sigma \to C \cdot \sigma \qquad \textbf{if } [l]C \cdot \sigma \text{ is in } M$$

`Instantiate`:

$$\begin{array}{ll} M \parallel F \implies & M \parallel F, \forall x.C \cdot \sigma \wedge x \approx x \cdot [x \mapsto t] \to C \cdot (\sigma \cup [x \mapsto t]) \\ & \textbf{if } \forall x.C \cdot \sigma \text{ is in } M, t \in \mathscr{T}(M), \text{and } C \cdot (\sigma \cup [x \mapsto t]) \text{ is new modulo } T \end{array}$$

Just as above, guards represent triggers, and we also use them to prevent the added clauses from lingering after the closure that allowed us to deduce them is removed from M by backtracking. To ensure termination, we also make sure we only produce new instances of universally quantified formulas, that is, instances that are not already in F modulo T. Note that the guards of formulas introduced by `Instantiate` can always be propagated immediately.

Thanks to these new rules, we should now be able to deduce the unsatisfiability of F_0. Let us go back to where we stopped, that is, to the partial model:

$$M_1 = \left\{ \begin{array}{l} i \not\approx j, \\ get(a_1, j) \not\approx get(a_2, j), \\ set(a_1, i, v) \approx set(a_2, i, v), \\ \forall a, i, v.\ [set(a, i, v)]\ get(set(a, i, v), i) \approx v \cdot \varnothing, \\ \forall a,\ i,\ j,\ v.[get(set(a, i, v), j)]\ (i \not\approx j \to get(set(a, i, v), j) \approx get(a, j)) \cdot \varnothing, \\ \forall a,\ i,\ j,\ v.[set(a, i, v),\ get(a, j)]\ (i \not\approx j \to get(set(a, i, v), j) \approx get(a, j)) \cdot \varnothing \end{array} \right\}$$

1. We have three closures containing universally quantified formulas in M_1. We instantiate the last one with a_1, i, j, and e using the rule `Instantiate`.

$$F_1 = F_0 \cup \left\{ \begin{array}{l} ((\forall a, i, j, v.\ [set(a, i, v),\ get(a, j)]\ \ldots) \cdot \varnothing \wedge \\ \quad a \approx a \cdot [a \mapsto a_1] \wedge i \approx i \cdot [i \mapsto i] \wedge j \approx j \cdot [j \mapsto j] \wedge v \approx v \cdot [v \mapsto e]) \to \\ \qquad [set(a, i, v),\ get(a, j)]\ (i \approx j \vee get(set(a, i, v), j) \approx get(a, j)) \cdot \\ \qquad\quad [a \mapsto a_1,\ i \mapsto i,\ j \mapsto j,\ v \mapsto e] \end{array} \right\}$$

2. As a_1, i, j, and e are known in M_1, we propagate the guards using theory reasoning.

$$M_2 = M_1 \cup \left\{ \begin{array}{l} a \approx a \cdot [a \mapsto a_1], \\ i \approx i \cdot [i \mapsto i], \\ j \approx j \cdot [j \mapsto j], \\ v \approx v \cdot [v \mapsto e] \end{array} \right\}$$

3. Then we note that our instance is a unit clause. We add the closure containing the formula with triggers to M_2 using unit propagation.

$$M_3 = M_2 \cup \left\{ \begin{array}{l} [set(a,i,v),\ get(a,j)]\ (i \approx j \vee get(set(a,i,v),j) \approx get(a,j)) \cdot \\ \qquad [a \mapsto a_1,\ i \mapsto i,\ j \mapsto j,\ v \mapsto e] \end{array} \right\}$$

4. We can now unfold this new closure using the rule `Trigger-Unfold`.

$$F_2 = F_1 \cup \left\{ \begin{array}{l} (([set(a,i,e),\ get(a,j)]\ \ldots) \cdot [a \mapsto a_1,\ i \mapsto i,\ j \mapsto j,\ v \mapsto e] \wedge \\ \quad set(a,i,v) \approx set(a,i,v) \cdot [a \mapsto a_1,\ i \mapsto i,\ v \mapsto e] \wedge \\ \quad get(a,j) \approx get(a,j) \cdot [a \mapsto a_1,\ j \mapsto j]) \rightarrow \\ \qquad i \approx j \cdot [i \mapsto i,\ j \mapsto j] \vee \\ \qquad get(set(a,i,v),j) \approx get(a,j) \cdot [a \mapsto a_1,\ i \mapsto i,\ j \mapsto j,\ v \mapsto e] \end{array} \right\}$$

5. We can do the same reasoning with the term a_2 instead of a_1 so that F_3 now also contains

$$\begin{array}{l} (([set(a,i,e),\ get(a,j)]\ \ldots) \cdot [a \mapsto a_2,\ i \mapsto i,\ j \mapsto j,\ v \mapsto e] \wedge \\ \quad set(a,i,v) \approx set(a,i,v) \cdot [a \mapsto a_2,\ i \mapsto i,\ v \mapsto e] \wedge \\ \quad get(a,j) \approx get(a,j) \cdot [a \mapsto a_2,\ j \mapsto j]) \rightarrow \\ \qquad i \approx j \cdot [i \mapsto i,\ j \mapsto j] \vee \\ \qquad get(set(a,i,v),j) \approx get(a,j) \cdot [a \mapsto a_2,\ i \mapsto i,\ j \mapsto j,\ v \mapsto e] \end{array}$$

6. We recognize the two guarded clauses that we added manually to F' to demonstrate the usage of guarded clauses. We can therefore apply the same reasoning to deduce that F_3 is unsatisfiable.

Let us now proceed to a rigorous definition of DPLL(T) for first-order axiomatizations with triggers. The rest of this section is mostly independent from the rest of the article and can be skipped by readers who are not interested in the theoretical justification of our framework.

5.3 Weak Entailment

To define our extension of DPLL(T), we need one last tool. We define a weak version of entailment that preserves feasibility as well as satisfiability while remaining easily computable by a working solver.

Definition 19 Let F be a set of super-clauses and C a super-clause. We write $F \vdash^\star_T C$ if and only if one of the following conditions holds:

- C is a unit user clause and $\textsc{Lit}(F) \cup \lfloor \textsc{Clo}(F) \rfloor \models_T C$;
- C is a non-unit user clause and $\{C' \mid C' \text{ is a user clause of } F\} \cup \lfloor \textsc{Clo}(F) \rfloor \models_T C$;
- C is a theory clause $D \cdot \sigma$ and there is $l \in D$ such that $F \cup known(\mathscr{T}(\textsc{Clo}(F))) \vdash^\star_T known(\mathscr{T}(l\sigma))$ and $F \vdash^\star_T l\sigma$;
- C is a theory clause $D \cdot \sigma$ and there is a theory clause $C' \cdot \sigma' \in F$ such that $C' \subseteq D$, $F \vdash^\star_T \sigma|_{Dom(\sigma')} \dot{\approx} \sigma'$, and $F \cup known(\mathscr{T}(\textsc{Clo}(F))) \vdash^\star_T known(\mathscr{T}(\sigma|_{Dom(\sigma')}))$.

Recall that $\textsc{Clo}(F)$ gives us all unit theory clauses in F, and $\lfloor \textsc{Clo}(F) \rfloor$ selects in $\textsc{Clo}(F)$ all closures over literals $l \cdot \sigma$ and applies the substitution, producing $l\sigma$ (Definition 9).

On unit user clauses, $\vdash^\star_T$ is stronger than the usual entailment modulo T. The reason is that $\vdash^\star_T$ is used in particular to decide that a clause is unnecessary for the proof and therefore can be forgotten or not generated. In the definition of truth assignment, we state that the solver should assume unit clauses eagerly while it is allowed to postpone deciding on the literals of non-unit clauses. Thus, even if a set of non-unit clauses entails a unit clause C, the solver cannot be allowed to forget C without compromising termination. For example, the set of axioms $F = \{c \approx c, f(c) \approx f(c), f(c) \approx c, \forall x[f(c) \approx c].f(x) \approx x, \forall x.f(x) \approx f(x)\}$ is terminating (every term introduced by the last axiom can be equated to an already known term by the previous one). Still, consider the set $G = \{f(c) \approx c, f(c) \approx c \vee c \not\approx c\}$. We have $F\backslash\{f(c) \approx c\} \cdot \varnothing \cup G \vdash^\star_T f(c) \approx c \cdot \varnothing$, and thus $f(c) \approx c$ can be removed from F. We have $f(c) \approx c \vee c \not\approx c \vDash_T f(c) \approx c$. Assume we can remove $f(c) \approx c$ from G. Then, the solver can produce an infinite number of terms from $F\backslash\{f(c) \approx c\}$. It may never choose to deduce $f(c) \approx c$ from $f(c) \approx c \vee c \not\approx c$ which would allow all these terms to collapse.

In the last two cases of Definition 19, known terms are only provided by the unit theory clauses of F and not by the user clauses. Indeed, as we said earlier, we treat user clauses according to the usual first-order semantics, where a literal may be replaced by an equivalent one regardless of its subterms.

We now check that we have provided a reasonable definition for $\vdash^\star_T$, that is, it approximates entailment in our logic (formulas are valid in the same worlds), it is stronger than $\vDash^\star_T$, and it is transitive. These properties will only be used in proofs later in this section but we think they give a useful insight on the meaning of $\vdash^\star_T$.

Lemma 5 *Let C be a super-clause and F be a set of super-clauses such that $F \vdash^\star_T C$. For every world L such that $L \blacktriangleright F$, $L \blacktriangleright C$.*

Proof We have four cases to consider. Assume that C is a unit user clause and $\text{LIT}(F) \cup \lfloor \text{CLO}(F) \rfloor \vDash_T C$. Since $L \blacktriangleright F$, $L \vDash_T \text{LIT}(F)$ and $L \vDash_T \lfloor \text{CLO}(F) \rfloor$. As a consequence, $L \vDash_T C$. The case where C is a non-unit user clause is handled in the same way.

Otherwise, C is a theory clause $D \cdot \sigma$. Assume that there is $l \in D$ such that $F \vdash^\star_T l\sigma$ and $F \cup \mathit{known}(\mathscr{T}(\text{CLO}(F))) \vdash^\star_T \mathit{known}(\mathscr{T}(l\sigma))$. Since $L \blacktriangleright F$, $L \vDash_T \text{LIT}(F) \cup \lfloor \text{CLO}(F) \rfloor$ and $L \cup \mathit{known}(\mathscr{T}(L)) \vDash_T \mathit{known}(\mathscr{T}(\text{CLO}(F)))$. As a consequence, $L \blacktriangleright l \cdot \sigma|_{vars(l)}$ and $L \blacktriangleright C$.

Otherwise, there is a theory clause $C' \cdot \sigma' \in F$ such that $C' \subseteq D$, $F \vdash^\star_T \sigma|_{Dom(\sigma')} \dot{\approx} \sigma'$, and $F \cup \mathit{known}(\mathscr{T}(\text{CLO}(F))) \vdash^\star_T \mathit{known}(\mathscr{T}(\sigma|_{Dom(\sigma')}))$. Since $L \blacktriangleright F$, $L \vDash_T \sigma|_{Dom(\sigma')} \dot{\approx} \sigma'$ and $L \cup \mathit{known}(\mathscr{T}(L)) \vDash_T \mathit{known}(\mathscr{T}(\sigma|_{Dom(\sigma')}))$, as per the previous case. Furthermore, there is an element $\varphi \cdot \sigma'|_{vars(\varphi)}$ of $C' \cdot \sigma'$ such that $L \blacktriangleright \varphi \cdot \sigma'|_{vars(\varphi)}$ and thus $L \rhd \varphi\sigma'$. Since every term substituted by σ into a free variable of φ is known from L, and since σ and σ' substitute the same terms modulo L and T into every free variable of φ, we have $L \rhd \varphi\sigma$. As a consequence, $L \blacktriangleright \varphi \cdot \sigma|_{vars(\varphi)}$ and $L \blacktriangleright C$. □

Lemma 6 *Let C be a super-clause and F be a set of super-clauses such that $F \vdash^\star_T C$. We have $F \vDash^\star_T C$.*

Proof Let L be a model of $[\![F]\!]$. We have four cases to consider. Assume that C is a unit user clause and $\text{LIT}(F) \cup \lfloor \text{CLO}(F) \rfloor \vDash_T C$. Since $L \vDash_T [\![F]\!]$, $L \vDash_T \text{LIT}(F)$ and $L \vDash_T \lfloor \text{CLO}(F) \rfloor$. As a consequence, $L \vDash_T C$. The case where C is a non-unit user clause is handled similarly.

Otherwise, C is a theory clause $D \cdot \sigma$. Assume that there is $l \in D$ such that $F \vdash^\star_T l\sigma$ and $F \cup \mathit{known}(\mathscr{T}(\text{CLO}(F))) \vdash^\star_T \mathit{known}(\mathscr{T}(l\sigma))$. Since $L \vDash_T [\![F]\!]$, $L \vDash_T \text{LIT}(F) \cup \lfloor \text{CLO}(F) \rfloor$. As a consequence, $L \vDash_T l\sigma$ and $L \vDash_T [\![C]\!]$.

Otherwise, there is a theory clause $C' \cdot \sigma' \in F$ such that $C' \subseteq D$, $F \vdash^\star_T \sigma|_{Dom(\sigma')} \dot{\approx} \sigma'$, and $F \cup \mathit{known}(\mathscr{T}(\text{CLO}(F))) \vdash^\star_T \mathit{known}(\mathscr{T}(\sigma|_{Dom(\sigma')}))$. Since $L \vDash_T [\![F]\!]$, $L \vDash_T \sigma|_{Dom(\sigma')} \dot{\approx} \sigma'$ as per the previous case. Since $L \vDash_T [\![C' \cdot \sigma']\!]$, we have $L \vDash_T [\![C' \cdot \sigma]\!]$. Thus, $L \vDash_T [\![D \cdot \sigma]\!]$. □

Lemma 7 *Let C be a super-clause and F_1 and F_2 be two sets of super-clauses. If $F_1 \vdash^\star_T F_2$ and $F_2 \vdash^\star_T C$, then $F_1 \vdash^\star_T C$.*

Proof Assume that C is a unit user clause l and $\text{LIT}(F_2) \cup \lfloor \text{CLO}(F_2) \rfloor \models_T C$. Since $F_1 \vdash^\star_T F_2$, we have $F_1 \vdash^\star_T \text{LIT}(F_2)$ and $F_1 \vdash^\star_T \lfloor \text{CLO}(F_2) \rfloor$. By definition of $\vdash^\star_T$ on user clauses, $\text{LIT}(F_1) \cup \lfloor \text{CLO}(F_1) \rfloor \models_T \text{LIT}(F_2) \cup \lfloor \text{CLO}(F_2) \rfloor$. Thus, $F_1 \vdash^\star_T C$. The case where C is a non-unit user clause is handled in the same way, except that instead of $\text{LIT}(F_1)$ and $\text{LIT}(F_2)$ we consider the sets of all user clauses in F_1 and F_2, respectively.

Otherwise, C is a theory clause $D \cdot \sigma$. Assume that there is $l \in D$ such that $F_2 \vdash^\star_T l\sigma$ and $F_2 \cup \mathit{known}(\mathcal{T}(\text{CLO}(F_2))) \vdash^\star_T \mathit{known}(\mathcal{T}(l\sigma))$. Like in the previous case, $F_1 \vdash^\star_T l\sigma$. Since $F_1 \vdash^\star_T F_2$, $F_1 \cup \mathit{known}(\mathcal{T}(\text{CLO}(F_1))) \vdash^\star_T \mathit{known}(\mathcal{T}(\varphi \cdot \sigma))$ for every closure $\varphi \cdot \sigma \in \text{CLO}(F_2)$. As a consequence, $\text{LIT}(F_1) \cup \lfloor \text{CLO}(F_1) \rfloor \cup \mathit{known}(\mathcal{T}(\text{CLO}(F_1))) \models_T \text{LIT}(F_2) \cup \lfloor \text{CLO}(F_2) \rfloor \cup \mathit{known}(\mathcal{T}(\text{CLO}(F_2)))$ and $F_1 \vdash^\star_T C$.

Otherwise, there is a theory clause $C' \cdot \sigma' \in F_2$ such that $C' \subseteq D$, $F_2 \vdash^\star_T \sigma|_{Dom(\sigma')} \dot{\approx} \sigma'$, and $F_2 \cup \mathit{known}(\mathcal{T}(\text{CLO}(F_2))) \vdash^\star_T \mathit{known}(\mathcal{T}(\sigma|_{Dom(\sigma')}))$. Like in the previous case, $F_1 \vdash^\star_T \sigma|_{Dom(\sigma')} \dot{\approx} \sigma'$ and $F_1 \cup \mathit{known}(\mathcal{T}(\text{CLO}(F_1))) \vdash^\star_T \mathit{known}(\mathcal{T}(\sigma|_{Dom(\sigma')}))$.

Assume that there is a literal $l \in C'$ such that $F_1 \vdash^\star_T l\sigma'$ and $F_1 \cup \mathit{known}(\mathcal{T}(\text{CLO}(F_1))) \vdash^\star_T \mathit{known}(\mathcal{T}(l\sigma'))$. Since $F_1 \vdash^\star_T \sigma|_{Dom(\sigma')} \dot{\approx} \sigma'$, $F_1 \vdash^\star_T l\sigma$. With $F_1 \cup \mathit{known}(\mathcal{T}(\text{CLO}(F_1))) \vdash^\star_T \mathit{known}(\mathcal{T}(\sigma|_{Dom(\sigma')}))$, we deduce $F_1 \cup \mathit{known}(\mathcal{T}(\text{CLO}(F_1))) \vdash^\star_T \mathit{known}(\mathcal{T}(l\sigma))$. Therefore, $F_1 \vdash^\star_T C$.

Otherwise, there is a theory clause $C'' \cdot \sigma'' \in F_1$ such that $C'' \subseteq C'$, $F_1 \vdash^\star_T \sigma'|_{Dom(\sigma'')} \dot{\approx} \sigma''$, and $F_1 \cup \mathit{known}(\mathcal{T}(\text{CLO}(F_1))) \vdash^\star_T \mathit{known}(\mathcal{T}(\sigma'|_{Dom(\sigma'')}))$. Since $F_1 \vdash^\star_T \sigma|_{Dom(\sigma')} \dot{\approx} \sigma'$, we have $F_1 \vdash^\star_T \sigma|_{Dom(\sigma'')} \dot{\approx} \sigma''$. Hence, $F_1 \vdash^\star_T C$. □

Given a set of super-literals M, we write $M \vdash^\star_T C$ as an abbreviation for $\text{LIT}(M) \cup \text{CLO}(M) \vdash^\star_T C$. In other words, we treat literals and closures in M as unit user clauses and theory clauses, respectively, and we ignore the anti-closures. According to this definition, $M \vdash^\star_T \bot$ whenever the set $\text{LIT}(M) \cup \lfloor \text{CLO}(M) \rfloor$ is T-unsatisfiable.

In our algorithm, we use terms coming from the user clauses to instantiate universally quantified formulas and to unfold triggers. To make these terms usable for the $\vdash^\star_T$ relation, we need to convert the literals in the set of assumed facts to closures, as follows. Given a set of super-literals M, we define $\lceil M \rceil$ to be $M \cup \{l \cdot \varnothing \mid l \in \text{LIT}(M)\}$. Thus, for every term $t \in \mathcal{T}(M)$, $t \in \mathcal{T}(\text{CLO}(\lceil M \rceil))$.

Lemma 8 *Let M be a set of super-literals and e a super-literal. If $\lceil M \rceil \vdash^\star_T e$ then $M \models^\star_T e$.*

Proof If L is a model of $[\![M]\!]$ then L is also a model of $[\![\lceil M \rceil]\!]$. □

5.4 Description of DPLL(T) on Formulas with Triggers

The method introduced below adapts the principles of abstract DPLL modulo theories to super-literals and guarded clauses. We call it DPLL$^\star(T)$. The rules are given in Figs. 2 and 3. The set F now contains guarded clauses and the partial model M super-literals. This is transparent for the rules `Decide`, `UnitPropagate`, `Fail`, and `Restart` which are kept unchanged.

The rule T`-Propagate` is extended to allow propagation of elements of guards. Note that, to ensure termination, the propagation of an element e from a guard can only occur if the terms of e are already known in M. This restriction is motivated in Sect. 5.5 using an example.

Unlike the classical DPLL, we impose different conditions on the clauses that can be learned and the clauses that can be forgotten. We allow to learn any clause $H \rightarrow C$ if

UnitPropagate:

$$M \parallel F, H \to C \vee e \implies Me \parallel F, H \to C \vee e \quad \textbf{if} \begin{cases} H \wedge \neg C \subseteq M \\ e \text{ is undefined in } M \end{cases}$$

Decide:

$$M \parallel F \implies Me^{\mathrm{d}} \parallel F \quad \textbf{if} \begin{cases} e \text{ or } \neg e \text{ occurs in } \textsc{Avb}(F, M) \\ e \text{ is undefined in } M \end{cases}$$

Fail:

$$M \parallel F, H \to C \implies \mathit{fail} \quad \textbf{if} \begin{cases} H \wedge \neg C \subseteq M \\ M \text{ contains no decision literals} \end{cases}$$

Restart:

$$M \parallel F \implies \varnothing \parallel F$$

T-Propagate:

$$M \parallel F \implies Me \parallel F \quad \textbf{if} \begin{cases} e \notin M \text{ and either:} \\ \quad M \models_T^\star e \text{ and } e \text{ or } \neg e \text{ occurs in } \textsc{Avb}(F, M), \text{ or} \\ \quad \lceil M \rceil \vdash_T^\star e \text{ and } e \text{ occurs in } \textsc{Grd}(F) \end{cases}$$

T-Learn:

$$M \parallel F \implies M \parallel F, H \to C \quad \textbf{if} \begin{cases} \text{every atom of } H \text{ occurs in } \textsc{Grd}(F) \cup \lceil M \rceil \\ \text{every atom of } C \text{ occurs in } \textsc{Avb}(F, H) \cup \textsc{Lit}(M) \\ F, H \models_T^\star C \end{cases}$$

T-Forget:

$$M \parallel F, H \to C \implies M \parallel F \quad \textbf{if} \begin{cases} \text{each closure of } C \text{ defined in } M \text{ occurs in } \textsc{Avb}(F, H) \\ \text{and either } \textsc{Avb}(F, H) \vdash_T^\star C \text{ or} \\ \qquad F, H \models_T^\star C \text{ and } H \neq \varnothing \end{cases}$$

T-Backjump:

$$Me^{\mathrm{d}}N \parallel F \implies Me' \parallel F \quad \textbf{if} \begin{cases} \text{there is } H \to C \in F \text{ such that } H \wedge \neg C \subseteq Me^{\mathrm{d}}N \\ F, M \models_T^\star e', \\ e' \text{ is undefined in } M, \text{ and} \\ e' \text{ or } \neg e' \text{ occurs in } \textsc{Avb}(F, M) \cup \textsc{Lit}(Me^{\mathrm{d}}N) \end{cases}$$

Fig. 2 Transition rules of Abstract DPLL Modulo Theories on guarded clauses

Instantiate:

$$M \parallel F \implies M \parallel F, \forall x.C \cdot \sigma \wedge x \approx x \cdot [x \mapsto t] \to C \cdot (\sigma \cup [x \mapsto t]) \quad \textbf{if} \begin{cases} \forall x.C \cdot \sigma \text{ is in } M \\ t \in \mathscr{T}(M) \\ \textsc{Avb}(F, M) \not\vdash_T^\star C \cdot (\sigma \cup [x \mapsto t]) \end{cases}$$

Witness-Unfold:

$$M \parallel F \implies M \parallel F, \langle l \rangle C \cdot \sigma \to l \cdot \sigma, \langle l \rangle C \cdot \sigma \to C \cdot \sigma \quad \textbf{if} \begin{cases} \langle l \rangle C \cdot \sigma \text{ is in } M \end{cases}$$

Trigger-Unfold:

$$M \parallel F \implies M \parallel F, [l] C \cdot \sigma \wedge l \cdot \sigma \to C \cdot \sigma \quad \textbf{if} \begin{cases} [l] C \cdot \sigma \text{ is in } M \\ \lceil M \rceil \vdash_T^\star l \cdot \sigma \end{cases}$$

Fig. 3 Additional transition rules for Abstract DPLL Modulo Theories on guarded clauses

$F, H \models_T^\star C$, and thus every model of $[\![F]\!]$ is also a model of $[\![H \to C]\!]$. However, we are more restrictive with respect to which clauses can be forgotten. Namely, for a guarded clause with an empty guard $\varnothing \to C$ to be forgotten, we require $\textsc{Avb}(F, \varnothing) \vdash_T^\star C$. We show in Sect. 5.5 that this distinction is necessary for termination.

The rule T-Backjump is pretty much kept as is, except that we forbid adding a literal occurring in N but not in $\textsc{Avb}(F, M)$. As is the case for the previous restrictions, this is necessary for termination, as we explain in Sect. 5.5.

Specific rules are needed to retrieve information from closures. They are described in Fig. 3. The formulas added by these rules to the set of guarded clauses F are tautologies in the semantics of formulas with triggers. The rule Instantiate creates a new instance of

a universally quantified formula of M with a sub-term of M. The rule `Witness-Unfold` handles a witness $\langle l \rangle C$ as a conjunction $l \wedge C$. The rule `Trigger-Unfold` uses the guard mechanism to protect the pseudo-literals under the trigger so that they cannot be decided upon or propagated until the guard is unfolded. An application of one of these three rules is said to be *redundant* in F, if the added guarded clauses are redundant in F, and a guarded clause $H \to C$ is said to be *redundant* in F if $\textsc{Avb}(F, H) \vdash_T^\star C$.

Like for classical DPLL(T), a solver is allowed to deduce the unsatisfiability of a set of ground clauses G modulo an extension of the background theory T described as an axiomatization W if it can compute a derivation to *fail* from the input state $\varnothing \parallel W \cdot \varnothing \cup G$.

We now define when a solver is allowed to deduce the satisfiability of G modulo W.

Property 3 The solver can return *Sat* on G if $\varnothing \parallel W \cdot \varnothing \cup G \Longrightarrow^\star M \parallel F$ where:

(i) $M \vdash_T^\star \textsc{Avb}(F, M)$,
(ii) $M \not\vdash_T^\star \bot$, and
(iii) if $H \to C$ can be added by `Instantiate`, `Witness-Unfold`, or `Trigger-Unfold` then $\textsc{Avb}(F, M) \vdash_T^\star C$.

The first two requirements mimic those of classical DPLL(T). They are required for M to form a proper model of $\textsc{Avb}(F, M)$. The last one requires the set of clauses $\textsc{Avb}(F, M)$ to be saturated with respect to trigger, witness, and quantifier semantics.

Remark 6 When there are no closures involved, the calculus above coincides with classical abstract DPLL modulo theories as long as unit clauses are only forgotten if they are entailed by unit clauses. As a consequence, the changes in abstract DPLL can be implemented as an extension outside an existing DPLL implementation.

Remark 7 In classical abstract DPLL modulo theories, conflict-driven lemmas, namely formulas allowing to deduce the added element e' in M after an application $Me^{\mathrm{d}}N \parallel F \Longrightarrow Me' \parallel F$ of T`-Backjump`, can be added to F using T`-Learn`. In our framework, this is permitted when e' is a user literal or a closure, but not when e' is an anti-closure since super-clauses cannot contain anti-closures. This restriction can be removed by allowing to deduce guarded clauses $H \to C$ such that $F, H \models_T^\star C$ where C may contain super-literals of all three kinds: literals, closures, and anti-closures. With this modification, if there is $D \subseteq M$ such that $F, D \models_T^\star e'$ and e' or $\neg e'$ occurs in $\textsc{Avb}(F, M) \cup \textsc{Lit}(Me^{\mathrm{d}}N)$, $\textsc{Clo}(M) \to \{\neg e \mid e \text{ is an anti-closure or a literal of } D\} \vee e'$ can be added to F using T`-Learn`.

5.5 Termination Related Constraints

In this section, we motivate the constraints on T`-Propagate`, T`-Backjump`, T`-Learn`, T`-Forget`, and `Instantiate` using examples. These constraints are closely related to the definition of termination in Sect. 2.4. They aim at forbidding:

- The addition to M of a super-literal that should be protected by a trigger. This requires keeping track of guards that should be protecting a new clause when learning it. This idea motivates the constraints on T`-Propagate`, T`-Backjump`, and T`-Learn`.
- The loss of a unit clause that is entailed by non-unit clauses. In the definition of the termination property, we only require that an element of a unit clause is added to truth assignments. Indeed, we do not want to ask for an application of `Decide` if there is another rule, for example, `Instantiate`, that can be applied. This motivates the constraints on T`-Forget`.

– The generation of an instance that is redundant as far as truth assignments are concerned. Indeed, the construction of instantiation trees stops as soon as a final truth assignment is reached. This motivates the constraints on `Instantiate`.

In the rule T-`Propagate`, we only allow $e \in \text{GRD}(F)$ to be added to M if $\lceil M \rceil \vdash^\star_T e$. Indeed, a trigger $[l]C \cdot \sigma$ is supposed to protect elements of C until l is true in M and all its sub-terms are known in M. This is exactly what we get by requesting $\lceil M \rceil \vdash^\star_T l\sigma$, namely $\text{LIT}(M) \cup \{l'\sigma' \mid l' \cdot \sigma' \in M\} \vDash_T l\sigma$ and $\text{LIT}(M) \cup \{l'\sigma' \mid l' \cdot \sigma' \in M\} \cup \mathit{known}(\mathscr{T}(M)) \vDash_T \mathit{known}(\mathscr{T}(l\sigma))$. Only requesting that $M \vDash^\star_T l\sigma$ would not have been enough. For example, consider the axiomatization $W_1 = \{\forall x.[f(x)]p(f(x)) \approx \top\}$. We can easily check that W_1 is terminating. Indeed, every sub-term of the form $f(t')$ of every truth assignment of $[f(x)]p(f(x)) \approx \top \cdot [x \mapsto t] \cup L \cdot \varnothing$ is either a sub-term of L or a sub-term of t. Still, $M \vDash^\star_T (f(x) \approx f(x)) \cdot [x \mapsto t]$ for every term $t \in \mathscr{T}(M)$. As a consequence, for any term t in M, $p(f(x)) \approx \top \cdot [x \mapsto t]$ and then $p(f(x)) \approx \top \cdot [x \mapsto f(t)]$, $p(f(x)) \approx \top \cdot [x \mapsto f(f(t))] \ldots$ can be added to M.

In the rule T-`Backjump`, we require that e' or $\neg e'$ occurs in $\text{AVB}(F, M) \cup \text{LIT}(Me^{\mathrm{d}}N)$. Assume that e' or $\neg e'$ is allowed to appear in $Me^{\mathrm{d}}N$ and consider the axiomatization $W_2 = \{\forall y.[p(y) \approx \top]\forall x.f(x, y) \approx x,\ \forall y.[p(y) \approx \top]\forall x.f(x, y) \approx f(x, y),\ c \approx c\}$. This axiomatization is terminating because as long as we have some $p(t) \approx \top$ to generate new terms $f(t', t)$ using the second axiom, we can also use the first axiom to collapse them to t'. Assume we launch the solver on a set of user clauses $G_2 = \{p(a) \approx \top,\ p(a) \not\approx \top \vee p(b) \approx \top,\ p(c) \approx \top \vee a \approx a,\ p(a) \not\approx \top \vee a \approx c\}$. We can add $p(a) \approx \top$ to M using `UnitPropagate`. We instantiate the first formula of W_2 with $a \in \mathscr{T}(M)$ and apply T-`Propagate`, `UnitPropagate`, and `Trigger-Unfold` so that $(\forall x.f(x, y) \approx f(x, y)) \cdot [y \mapsto a]$ is in M. Then we can make a bad choice and decide $p(b) \not\approx \top$. We now add $p(c) \approx \top$ to M using `Decide`, instantiate the first formula of W_2 with $c \in \mathscr{T}(M)$ and apply T-`Propagate`, `UnitPropagate`, and `Trigger-Unfold` so that $(\forall x.f(x, y) \approx f(x, y)) \cdot [y \mapsto c]$ is in M. Since we have a conflict clause in M, we can use T-`Backjump` but, instead of adding $p(b) \approx \top$, we make another bad choice and add $(\forall x.f(x, y) \approx f(x, y)) \cdot [y \mapsto c]$. Indeed, since $(\forall x.f(x, y) \approx f(x, y)) \cdot [y \mapsto a] \in M$ and $G_2 \vDash_T a \approx c$, $G_2 \cup M \vDash^\star_T (\forall x.f(x, y) \approx f(x, y)) \cdot [y \mapsto c]$. Because of this closure, we can produce an infinite number of terms $f(t, c)$, $f(f(t, c), c) \ldots$ Since we do not have $M \vDash_T p(c) \approx \top$, they are not all equal to t in M. Indeed, we are not bound to add $a \approx c$ to M until there is nothing else to do even if it is entailed by G_2.

In the rule T-`Learn`, for a new guarded clause $H \to C$ to be learned, every atom of C must occur in $\text{AVB}(F, H) \cup \text{LIT}(M)$. Even asking that $\text{AVB}(F, H) \vdash^\star_T C$ is not enough to prevent elements that are protected by a trigger in F from occurring in C without their trigger. When they are in C, they can be added to M, through `Decide` for example, and prevent the solver from terminating. The following example closely resembles the previous one. Assume that closures of C are allowed to occur in M and consider the axiomatization W_2 and the set of user clauses G_2 from the previous paragraph. We can add $p(a) \approx \top$ and $p(c) \approx \top$ to M using `UnitPropagate` and `Decide`. We instantiate the first formula of W_2 with a and $c \in \mathscr{T}(M)$ and apply T-`Propagate`, `UnitPropagate`, and `Trigger-Unfold` so that $([p(y) \approx \top]\forall x.f(x, y) \approx f(x, y)) \cdot [y \mapsto a] \wedge (p(y) \approx \top) \cdot [y \mapsto a] \to (\forall x.f(x, y) \approx f(x, y)) \cdot [y \mapsto a]$ is in F and $(\forall x.f(x, y) \approx f(x, y)) \cdot [y \mapsto c]$ is in M. If the condition of T-`Learn` were relaxed, the guarded clause $([p(y) \approx \top]\forall x.f(x, y) \approx f(x, y)) \cdot [y \mapsto a] \wedge (p(y) \approx \top) \cdot [y \mapsto a] \to (\forall x.f(x, y) \approx f(x, y)) \cdot [y \mapsto c]$ could be added to F using T-`Learn`. Indeed, $G_3 \cup \{c \approx c \cdot \varnothing,\ (\forall x.f(x, y) \approx f(x, y)) \cdot [y \mapsto a]\} \subseteq \text{AVB}(F, \{([p(y) \approx \top]\forall x.f(x, y) \approx f(x, y)) \cdot [y \mapsto a],\ (p(y) \approx \top) \cdot$

$[y \mapsto a]\}$) and, since $G_3 \vDash_T a \approx c$, $G_3 \cup \{c \approx c \cdot \varnothing,\ (\forall x.f(x,y) \approx f(x,y)) \cdot [y \mapsto a]\} \vdash_T^\star (\forall x.f(x,y) \approx f(x,y)) \cdot [y \mapsto c]$. Now, we remove everything from M using `Restart`. Using T`-Propagate` and `UnitPropagate`, we can add $p(a) \approx \top, ([p(y) \approx \top]\forall x.f(x,y) \approx f(x,y)) \cdot [y \mapsto a]$, $(p(y) \approx \top) \cdot [y \mapsto a]$ and finally $(\forall x.f(x,y) \approx f(x,y)) \cdot [y \mapsto c]$ to M. Because of this closure, we can produce an infinite number of terms $f(t,c)$, $f(f(t,c),c)\ldots$ Since we do not have $p(c) \approx \top$, they are not all equal to t in M. Indeed, we are not bound to add $a \approx c$ to M until there is nothing else to do even if it is entailed by G_3.

In the rule T`-Forget`, we forbid the deletion of a guarded clause $H \to C \in F$ if, after the deletion, there is a closure defined in M that no longer appears in $\text{AVB}(F, H)$. This is needed so that the solver can progress despite the additional constraints on T`-Backjump` and T`-Learn`. For example, assume F contains a redundant guarded clause $H \to C$ such that $H \subseteq M$ and there is a tautology $\varphi \cdot \sigma \in C$ such that $\varphi \cdot \sigma$ does not appear in $F \backslash \{H \to C\}$. The anti-closure $\neg(\varphi \cdot \sigma)$ can be added to M using `Decide`. If $H \to C$ is then erased from F with T`-Forget`, the rule T`-Backjump` can no longer be applied to revert $\varphi \cdot \sigma$. Alternatively, we can forbid the deletion of theory clauses from $W \cdot \varnothing$ and instead allow forgetting any clause that was added to F during the derivation, as long as there is no closure in M that is no longer in $\text{AVB}(F, M)$ after the deletion.

If H is $\varnothing$, we also require that $\text{AVB}(F, H) \vdash_T^\star C$. Assume that we can forget $H \to C$ as soon as we have $F, H \vDash_T^\star C$. Consider the axiomatization $W_4 = \{[p(a) \approx \top]\forall x.f(x,a) \approx x,\ [p(c) \approx \top]\forall x.f(x,c) \approx f(x,c),\ a \approx c,\ a \approx a, c \approx c\}$. Like W_2, W_4 is terminating. We launch the solver on the set of user clauses $G_4 = \{p(a) \approx \top,\ p(c) \approx \top,\ p(a) \not\approx \top \vee a \approx c\}$. We can easily check that $W_4 \cdot \varnothing \backslash \{a \approx c \cdot \varnothing\} \cup G_4 \vDash_T^\star a \approx c \cdot \varnothing$, and therefore we forget it. We can add $p(c) \approx \top$ and the second axiom of W_4 to M using `UnitPropagate`. With `Trigger-Unfold` and then T`-Propagate` and `UnitPropagate` we can add $\forall x.f(x,c) \approx f(x,c)$ to M. Because of this closure, we can produce an infinite number of terms $f(t,c)$, $f(f(t,c),c)\ldots$ Since we do not have $a \approx c$, they are not all equal to t in M.

In the rule `Instantiate`, an instance of a formula $\forall x.C \cdot \sigma$ with a term t cannot be added to F if $\text{AVB}(F, M) \vdash_T^\star C \cdot (\sigma \cup [x \mapsto t])$. This constraint is needed for termination so that redundant instances are forbidden.

5.6 Soundness and Completeness

We show that DPLL* is compliant with the semantics defined in Sect. 2.2.

Lemma 9 *For every derivation $M_1 \parallel F_1 \Longrightarrow^\star M_2 \parallel F_2$, every model L of F_1 is a model of F_2.*

Proof We proceed by case analysis over the rule applied for the step $M_1 \parallel F_1 \Longrightarrow M_2 \parallel F_2$.

- In `UnitPropagate`, `Decide`, `Restart`, T`-Propagate`, and T`-Backjump`, F_1 and F_2 are equal.
- For T`-Learn`, we have $F_1, H \vDash_T^\star C$. Since L is complete, by Lemma 4, if $L \blacktriangleright F_1$ and $L \blacktriangleright H$, then $L \blacktriangleright C$.
- For T`-Forget`, $F_2 \subseteq F_1$. Thus, if we have $L \blacktriangleright F_1$ then $L \blacktriangleright F_2$.
- For the rule `Witness-Unfold`, assume that $L \blacktriangleright \langle l \rangle C \cdot \sigma$. By definition of $\blacktriangleright$, $L \blacktriangleright l \cdot \sigma$ and $L \blacktriangleright C \cdot \sigma$.
- For the rule `Trigger-Unfold`, assume that $L \blacktriangleright [l]C \cdot \sigma \wedge l \cdot \sigma$. By definition of $\blacktriangleright$, $L \blacktriangleright C \cdot \sigma$.

– For the rule `Instantiate`, assume that $L \blacktriangleright \forall x.C \cdot \sigma \wedge x \approx x \cdot [x \mapsto t]$. By definition of $\blacktriangleright$, $L \cup known(\mathscr{T}(L)) \vDash_T known(\mathscr{T}(t))$ and so there is $t' \in \mathscr{T}(L)$ such that $L \vDash_T t \approx t'$. Therefore, $L \blacktriangleright C \cdot (\sigma \cup [x \mapsto t])$. □

Lemma 10 *For every derivation $M_1 \parallel F_1 \Longrightarrow^\star M_2 \parallel F_2$, we have $F_2 \vDash_T^\star F_1$.*

Proof Let L be a complete and satisfiable set of literals. We proceed by induction over the number of applications of T-`Forget` $M \parallel F, H \to C \Longrightarrow M \parallel F$. If there are none then $F_1 \subseteq F_2$. Otherwise, consider the last application $M \parallel F, H \to C \Longrightarrow M \parallel F$. We have that $F \subseteq F_2$ and, by induction hypothesis, $F \cup H \to C \vDash_T^\star F_1$. Either $F, H \vDash_T^\star C$ or $\textsc{Avb}(F, H) \vdash_T^\star C$ and $\textsc{Avb}(F, H) \vDash_T^\star C$ by Lemma 6. In both cases, $F \vDash_T^\star H \to C$ and $F_2 \vDash_T^\star F_1$. □

Theorem 6 (Soundness) *If the solver returns* Unsat *on a set of user clauses G with a sound axiomatization Ax of an extension T' of T then G has no model in the theory T'.*

Proof Let W be the result of the skolemization and the clausification of Ax. Every model of Ax can be extended to a model of W by adding the interpretations of the Skolem functions. As a consequence, since Ax is sound, for every T'-satisfiable set of literals G' that only contains literals of G, there is a model of $W \cup G'$. □

We first need an intermediate lemma. It states that every element of a set of super-literals M constructed in a derivation is either a decision or entailed by the input problem and previous decisions:

Lemma 11 *Let $M_0 e_1^{\mathrm{d}} M_1 \dots e_n^{\mathrm{d}} M_n$ be a sequence of super-literals where $e_1, \dots, e_n$ are the only decision literals. If $\varnothing \parallel G \cup W \cdot \varnothing \Longrightarrow^\star M_0 e_1^{\mathrm{d}} M_1 \dots e_n^{\mathrm{d}} M_n \parallel F$, L is a model of $G \cup W \cdot \varnothing$ and $L \blacktriangleright e_1, \dots, e_i$, then $L \blacktriangleright M_i$ for every i in $0 \dots n$.*

Proof Let L be a model of $G \cup W \cdot \varnothing$, such that $L \blacktriangleright M$. We show that, for every rule that adds a new super-literal e to M from $M \parallel F$ (except `Decide`), $L \blacktriangleright e$.

First note that, by Lemma 9, $L \blacktriangleright F$. For the rule `UnitPropagate`, $L \blacktriangleright H \to C \vee e$ and $L \blacktriangleright H \cup \neg C$. By definition of $\blacktriangleright$, $L \blacktriangleright e$. For the rule T-`Propagate`, $M \vDash_T^\star e$ and, since L is complete, $L \blacktriangleright e$ by Lemma 4. The only remaining rule is T-`Backjump`. We have $F \cup M \vDash_T^\star e$. Since $L \blacktriangleright M$ and $L \blacktriangleright F$, since L is complete, $L \blacktriangleright e$ by Lemma 4. □

Now, we can prove the soundness of the DPLL* framework. If the solver returns *Unsat* on G with W then there is a derivation $\varnothing \parallel G \cup W \cdot \varnothing \Longrightarrow^\star M \parallel F, H \to C \Longrightarrow \mathit{fail}$ such that M contains no decision literals and $H \wedge \neg C \subseteq M$. By contradiction, assume G has a model in T'. There is a T'-satisfiable set of literals G' such that $G' \vDash_T G$. Since Ax is sound, $W \wedge G'$ has a model L. By Lemma 11, $L \blacktriangleright M$. What is more, by Lemma 9, $L \blacktriangleright F, H \to C$. With $H \wedge \neg C \subseteq M$, we get a contradiction.

Theorem 7 (Completeness) *If the solver returns* Sat *on a set of clauses G with a complete axiomatization Ax of T' then G is T'-satisfiable.*

Proof Let W be the result of the skolemization and the clausification of Ax. If W is feasible then so is Ax. As a consequence, since Ax is complete, every set of literals L such that $W \cup L$ is feasible is T'-satisfiable.

We show that, if the solver returns *Sat* on a set of clauses G with the theory W then there is a T-satisfiable set of literals L such that $L \vDash_T G$ and $W \cup L$ is feasible. Since Ax is complete, L is T'-satisfiable. Since $L \vDash_T G$, so is G.

Let F be a set of guarded clauses and M a set of literals and closures such that $\varnothing \parallel G \cup W \cdot \varnothing \Longrightarrow^\star M \parallel F$ and:

(i) $M \vdash_T^\star \textsc{Avb}(F, M)$,
(ii) $M \not\vdash_T^\star \bot$, and
(iii) if $H \rightarrow C$ can be added by `Instantiate`, `Witness-Unfold`, or `Trigger-Unfold` then $\textsc{Avb}(F, M) \vdash_T^\star C$.

Consider $L = \textsc{Lit}(M) \cup \{l\sigma \mid l \cdot \sigma \in M\} \cup \{t \approx t \mid t \in \mathscr{T}(M)\}$. By (ii), L is T-satisfiable. We need to show that $L \vDash_T G$ and $L \rhd W$, which is the same as $L \blacktriangleright \textsc{Avb}(W \cdot \varnothing \cup G, \varnothing)$. It is sufficient to prove that $L \blacktriangleright \textsc{Avb}(F, \varnothing)$. Indeed, the only rule that can remove an element of $W \cdot \varnothing \cup G$ is T`-Forget` and, if $L \blacktriangleright \textsc{Avb}(F, \varnothing)$ and $\textsc{Avb}(F, \varnothing) \vdash_T^\star C$, by Lemma 5, $L \blacktriangleright C$.

Now, we only need to show that $L \blacktriangleright \textsc{Clo}(M)$. Indeed, $M \vdash_T^\star \textsc{Avb}(F, M)$ is the same as $\textsc{Lit}(M) \cup \textsc{Clo}(M) \vdash_T^\star \textsc{Avb}(F, M)$ and therefore $L \blacktriangleright \textsc{Clo}(M)$ implies $L \blacktriangleright \textsc{Avb}(F, M)$ by Lemma 5. For every closure $\varphi \cdot \sigma \in M$, we prove that $L \rhd \varphi\sigma$ by induction over the size of the formula φ.

- $l \cdot \sigma \in M$. By definition of L, $L \rhd l\sigma$.
- $\forall x.C \cdot \sigma \in M$. Let t be a ground term of L. By definition of L, t is a ground term of M. By (iii), $\textsc{Avb}(F, M) \vdash_T^\star C \cdot (\sigma \cup [x \mapsto t])$. Since $M \vdash_T^\star \textsc{Avb}(F, M)$, by Lemma 7, $M \vdash_T^\star C \cdot (\sigma \cup [x \mapsto t])$. Therefore, there is $\varphi \in C$ such that either $\varphi \cdot \sigma' \in M$ and $L \vDash_T (\sigma \cup [x \mapsto t]) \doteq \sigma'$ or φ is a literal, $M \vdash_T^\star \varphi\sigma$, and $M \cup \mathit{known}(\mathscr{T}(\textsc{Clo}(M))) \vdash_T^\star \mathit{known}(\mathscr{T}(\varphi\sigma))$. In the first case, since φ is strictly smaller than $\forall x.C$, we have $L \rhd \varphi\sigma'$ by induction hypothesis and, hence, $L \rhd \varphi\sigma$. In the second case, $L \rhd \varphi\sigma$ by definition of L. By definition of $\rhd$ on universally quantified formulas, $L \rhd (\forall x.C)\sigma$.
- $\langle l \rangle C \cdot \sigma \in M$. By (iii), we have $\textsc{Avb}(F, M) \vdash_T^\star l \cdot \sigma$ and $\textsc{Avb}(F, M) \vdash_T^\star C \cdot \sigma$. Since $M \vdash_T^\star \textsc{Avb}(F, M)$, by Lemma 7, $M \vdash_T^\star l \cdot \sigma$ and $M \vdash_T^\star C \cdot \sigma$. Hence, there is $\varphi \in C$ such that either there is a substitution σ' such that $\varphi \cdot \sigma' \in M$ and $M \vdash_T^\star \sigma \doteq \sigma'$ or φ is a literal, $M \vdash_T^\star \varphi\sigma$ and, $M \cup \mathit{known}(\mathscr{T}(\textsc{Clo}(M))) \vdash_T^\star \mathit{known}(\mathscr{T}(\varphi\sigma))$. In both cases, $L \rhd \varphi\sigma$ with the same reasoning as for universal quantifiers. In the same way, $L \rhd l\sigma$. Therefore, $L \rhd (\langle l \rangle C)\sigma$.
- $[l]C \cdot \sigma \in M$. Assume $L \rhd l\sigma$. By definition of $\rhd$, we have both $M \vdash_T^\star l\sigma$ and $M \cup \mathit{known}(\mathscr{T}(M)) \vdash_T^\star \mathit{known}(\mathscr{T}(l\sigma))$. Thus $\lceil M \rceil \vdash_T^\star l \cdot \sigma$. By (iii), $\textsc{Avb}(F, M) \vdash_T^\star C \cdot \sigma$. As a consequence, $L \rhd C\sigma$ like in the two previous cases and, by definition of $\rhd$, $L \rhd ([l]C)\sigma$. □

5.7 Termination and Progress

We have shown that DPLL$^\star(T)$ only allows derivations that are compliant with the semantics of Sect. 2. In this section, we show that, if some restrictions are applied, there cannot be infinite DPLL* derivations for terminating axiomatization. We also show that, within the same restrictions, every derivation that can not continue is terminal, *i.e.*, the solver can return *Sat* or *Unsat*.

For termination, we require instantiation to be *fair*, that is to say that every possible instance should be generated at some point in the search. To define fairness, we use a notion of *instantiation level* [19]. An instantiation level n for a term t indicates that t become known after n rounds of instantiation. More formally, if M is a set of super-literals, the instantiation level $\mathit{level}_M(t)$ (resp. $\mathit{level}_M(e)$) of a term t (resp. a super-literal e) is either a non-negative

integer or a special element ∞. It is defined as the limit of the sequence $level^i_M$ computed in the following manner:

on a term t	$level^i_M(t) \triangleq \min\{level^i_M(e) \mid e \in M \text{ and } t \in \mathscr{T}(e)\}$
on a literal l	$level^i_M(l) \triangleq 0$
on a closure or anti-closure	$level^0_M(e) \triangleq 0$ if σ is empty and ∞ otherwise
$\varphi \cdot \sigma$ or $\neg(\varphi \cdot \sigma)$	$level^{i+1}_M(e) \triangleq 1 + \max\{level^i_M(x\sigma) \mid x \in Dom(\sigma)\}$

Operations min, max and $+$ are such that, if S is a non-empty set, $\min(S \cup \infty) = \min(S)$, $\min(\varnothing) = \infty$, $\max(S \cup \infty) = \infty$, $\max(\varnothing) = -1$, and $1 + \infty = \infty$. This sequence always converges since the level of every term or super-literal either stays infinite forever or becomes finite at some i and does not change after that.

Using this definition, we define the current instantiation level of a set of super-literals M as $level(M) = \max\{level_M(e) \mid e \in M\}$. We enforce fairness by requiring that new instances of level strictly bigger than the current instantiation level are only possible when:

- a truth assignment, as defined in Sect. 2.4, has been reached, and
- every previously available instance of a smaller instantiation level has already been handled.

These two requirements are obtained by a restriction on derivations:

Definition 20 (*Fairness*) We say that a derivation is *fair* if, for every step $M \parallel F \Longrightarrow Me \parallel F$ where $level_M(e) > level(M)$, e has form $x \approx x \cdot [x \mapsto t]$ and `Instantiate` can be applied to some universal formula $\forall x.\varphi \cdot \sigma$ and the term t in $M \parallel F$. For every such step, if M' is the minimal prefix of M such that $t \subseteq \mathscr{T}(M')$, then there is a prefix N of M containing M' and $\forall x.\varphi \cdot \sigma$ such that:

(a) $N \not\vdash^\star_T \bot$,
(b) for every unit super-clause $e \in \text{AVB}(F, \varnothing)$, we have $\lceil N \rceil \vdash^\star_T e$,
(c) for every closure $\langle l \rangle C \cdot \sigma \in N$, $\lceil N \rceil \vdash^\star_T l \cdot \sigma$ and, if C is a unit clause, $\lceil N \rceil \vdash^\star_T C \cdot \sigma$,
(d) for every closure $[l]\varphi \cdot \sigma \in N$ such that φ is a unit clause, if $\lceil N \rceil \vdash^\star_T l \cdot \sigma$ then we have $\lceil N \rceil \vdash^\star_T \varphi \cdot \sigma$,
(e) for every closure $\forall x.\varphi \cdot \sigma \in M'$ such that φ is a unit clause and for every term $t \in \mathscr{T}(M')$ such that $level_M(\varphi \cdot (\sigma \cup [x \mapsto t])) \leq level(M)$, we have $\lceil N \rceil \vdash^\star_T \varphi \cdot (\sigma \cup [x \mapsto t])$, and
(f) for every guarded clause $H \to C$ that can be added to F by applying `Instantiate`, `Witness-Unfold` or `Trigger-Unfold` on a closure of M', if $level_M(H) \leq level(M)$, $\text{AVB}(F, M) \vdash^\star_T C$.

Remark 8 Note that, in a fair derivation, the current instantiation level of every partial model M is finite.

Remark 9 Dealing with instantiation levels in the implementation is not mandatory. To ensure fairness, it suffices to handle unit clauses, triggers and witnesses before generating new instances and to select instances in the order in which they become possible.

Lemma 12 *Let W be a terminating axiomatization and let G be a set of user clauses. There is a finite set of super-literals $\overline{M}$ such that, at every state $M \parallel F$ in a fair derivation from $\varnothing \parallel G \cup W \cdot \varnothing$, we have $M \subseteq \overline{M}$.*

Proof The idea of the proof is the following. During the search, the algorithm goes through possible instantiation trees of $L \cup W$, where L is a set of literals from G. Fairness will prevent

us from generating an unbounded number of instances before generating the one instance that will allow each tree to grow. Since, for any L, there exists a finite instantiation tree of $L \cup W$ by the termination property of W, the number of generated instances is bounded and we have an upper bound on M. □

Introduction of Z_i: Let us first construct a sequence of sets of super-literals Z_i that will be used to bound M during the search. We call sub-formula of W, an element of the smallest set containing $\{\varphi \mid \varphi \in C \text{ and } C \in W\}$ and such that, if $\forall x.C$, $\langle l \rangle C$ or $[l]C$ is a sub-formula of W, l and every element of C are sub-formulas of W.

We define the sequence Z_i such that $Z_0 = \{l,\ l \cdot \varnothing,\ \neg(l \cdot \varnothing) \mid l \text{ or } \neg l \text{ occurs in } G\} \cup \{\varphi \cdot \varnothing,\ \neg(\varphi \cdot \varnothing) \mid \varphi \text{ is a closed sub-formula of } W\}$ and $Z_{n+1} = Z_n \cup \{\varphi \cdot \sigma, \neg(\varphi \cdot \sigma) \mid \varphi$ is a sub-formula of W or the equality $x \approx x$ and $\mathscr{T}(\sigma) \subseteq \mathscr{T}(Z_n)\}$.

Remark 10 By construction of the sequence Z_i, if an element $e \in M$ has an instantiation level n in M then $e \in Z_n$.

Introduction of A^M, the biggest truth assignment in M: Since W is terminating, for every subset L of the finite set of literals $\{l \mid l \cdot \varnothing \in Z_0\}$, we can choose a finite instantiation tree of $W \cup L$. We define what is the biggest truth assignment A^M occurring in these trees that is entailed by the set M at some point in the search. If A is a set of theory clauses, we define $\text{UNIT}(A)$ to be the set of unit super-clauses of A. The truth assignment A^M will be used during the proof to link the current partial model to a node in the finite set of finite instantiation trees of $W \cup L$, with $L \subseteq \{l \mid l \cdot \varnothing \in Z_0\}$.

For every set of super-literals M, we compute a sequence A_i^M of sets of theory clauses as follows. A_0^M is the biggest subset of $\{l \cdot \varnothing \mid l \cdot \varnothing \in Z_0\}$ such that $\lceil M \rceil \vdash_T^\star A_0^M$. A_1^M is the biggest truth assignment of $A_0^M \cup W \cdot \varnothing$ such that $\lceil M \rceil \vdash_T^\star \text{UNIT}(A_1^M)$. Such a truth assignment may not exist, for instance, if W contains a unit theory clause $\langle l \rangle C$ and $\lceil M \rceil \nvdash_T^\star l \cdot \varnothing$. Let $\mathbb{T}^M$ be the finite instantiation tree of $\{l \mid l \cdot \varnothing \in A_0^M\} \cup W$. If $\forall x.C \cdot \sigma$, t is the new instance added to A_i^M in $\mathbb{T}^M$, then A_{i+1}^M is the biggest truth assignment of $A_i^M \cup C \cdot (\sigma \cup [x \mapsto t])$ such that $\lceil M \rceil \vdash_T^\star \text{UNIT}(A_{i+1}^M)$, if any. We call d^M the maximal i for which A_i^M exists and we define A^M as $A_{d^M}^M$.

Introduction of n^M, the size of A^M: For $i \in 0 \ldots d^M$, let n_i^M be the number of closures that are in A_i^M but not in A_{i-1}^M, if any. We define n_{max} and d_{max} to be integers such that, for every subset L of $\{l \mid l \cdot \varnothing \in Z_0\}$, the height of the chosen finite instantiation tree $\mathbb{T}$ of $L \cup W$ is less than d_{max} and there is less than n_{max} closures in every truth assignment of $\mathbb{T}$. We have that $d^M < d_{max}$ and $n_i^M < n_{max}$ for every M and every $i \in 0 \ldots d^M$. We call n^M the integer $\sum_{i \in 0 \ldots d^M} (n_i^M + 1) \times (n_{max} + 1)^{(d_{max} - i)}$. Note that n^M models a lexicographic order on the finite sequence $n_0^M \ldots n_{d^M}^M$.

Remark 11 By definition, n^M depends only on A^M and, if A^{Me} is different from A^M, then $n^{Me} > n^M$.

Let us now do the proof of Lemma 12. Let m be $(n_{max} + 2)^{d_{max}+1}$. We show that, for every state $M \parallel F$ in the derivation, the current instantiation level in M is at most $n^M + 1$. Thus, if $\varnothing \parallel W \cdot \varnothing \cup G \Longrightarrow M \parallel F$, elements of M have an instantiation level of at most $m + 1$ in M. By Remark 10, $M \subseteq Z_{m+1}$ and Lemma 12 holds.

We prove the folowing properties by induction over the derivation of $M \parallel F$:

Property 1. The current instantiation level in M is at most $n^M + 1$.
Property 2. There is a prefix M' of M such that elements of M' have an instantiation level smaller or equal to n^M in M and $\lceil M' \rceil \vdash^\star_T \textsc{Unit}(A^M)$.

We only consider steps that introduce new element to the current partial model M. If an inference step does not modify M or remove elements from M, we necessarily return to a previous state of M in the derivation, where the two properties hold by induction hypothesis. It suffices to show that, for every step $\_ \parallel F \Longrightarrow Me \parallel F$, e has at most an instantiation level of $n^M + 1$ in M. Indeed, there are two cases: either A^{Me} is A^M and the two properties hold by induction hypothesis or A^{Me} is different from A^M. In this second case, by Remark 11, we have $n^{Me} > n^M$. As a consequence, the current instantiation level in Me must be at most n^{Me}. Since, by construction, $\lceil Me \rceil \vdash^\star_T \textsc{Unit}(A^{Me})$, we can conclude.

From now on, we concentrate on showing that, for every step $\_ \parallel F \Longrightarrow Me \parallel F$ such that properties 1 and 2 hold for M, e has at most an instantiation level of $n^M + 1$ in M.

In an application $Me^d N \parallel F \Longrightarrow Me' \parallel F$ of `T-Backjump`, we have $e' \in \textsc{Avb}(F, M) \cup \textsc{Lit}(Me^d N)$. Thus, the instantiation level of e' in M is smaller than the current instantiation level in M.

Let us now consider the steps $M \parallel F \Longrightarrow Me \parallel F$ that simply add an element to M. By contradiction, we assume that e has an instantiation level of $n^M + 2$ in M. Since the derivation is fair, e has form $x \approx x \cdot [x \mapsto t]$, some universal formula $\forall x.C \cdot \varphi$ can be instantiated with t, and, if M' is the minimal prefix of M such that $t \subseteq \mathscr{T}(M')$, then there is a prefix N of M containing M' and $\forall x.C \cdot \varphi$ such that:

(a) $N \not\vdash^\star_T \bot$,
(b) for every unit super-clause $e \in \textsc{Avb}(F, \varnothing)$, we have $\lceil N \rceil \vdash^\star_T e$,
(c) for every closure $\langle l \rangle C \cdot \sigma \in N$, $\lceil N \rceil \vdash^\star_T l \cdot \sigma$ and, if C is a unit clause, $\lceil N \rceil \vdash^\star_T C \cdot \sigma$,
(d) for every closure $[l]\varphi \cdot \sigma \in N$ such that φ is a unit clause, if $\lceil N \rceil \vdash^\star_T l \cdot \sigma$ then we have $\lceil N \rceil \vdash^\star_T \varphi \cdot \sigma$,
(e) for every closure $\forall x.\varphi \cdot \sigma \in M'$ such that φ is a unit clause, and for every term $t \in \mathscr{T}(M')$ such that $level_M(\varphi \cdot (\sigma \cup [x \mapsto t])) \leq level(M)$, we have $\lceil N \rceil \vdash^\star_T \varphi \cdot (\sigma \cup [x \mapsto t])$, and
(f) for every guarded clause $H \to C$ that can be added to F by applying `Instantiate`, `Witness-Unfold` or `Trigger-Unfold` on a closure of M', if $level_M(H) \leq level(M)$, $\textsc{Avb}(F, M) \vdash^\star_T C$.

For the rest of the proof, we need three intermediate lemmas. To facilitate reading, we prove these lemmas at the end of the main proof.

Lemma 13 *N contains a truth assignment of $A^M_0 \cup W \cdot \varnothing$.*

Lemma 14 *For every closure $\varphi \cdot \sigma$ such that $\lceil N \rceil \vdash^\star_T \varphi \cdot \sigma$, there is a truth assignment A of $A^M \cup \varphi \cdot \sigma$ such that $\lceil N \rceil \vdash^\star_T \textsc{Unit}(A)$.*

Lemma 15 *If A^M is final, for every $\varphi \cdot \sigma \in N$, we have $\textsc{Unit}(A^M) \vdash^\star_T \varphi \cdot \sigma$.*

Since $t \in \mathscr{T}(M')$, there must be an element of M' that has an instantiation level in M of $n^M + 1$ at least and, by induction hypothesis, of $n^M + 1$ exactly. As a consequence, by property 2, there is a prefix M'' of M' such that $\lceil M'' \rceil \vdash^\star_T \textsc{Unit}(A^M)$. By Lemma 13, M contains a truth assignment of $A^M_0 \cup W \cdot \varnothing$. Thus, A^M_1 is defined, d^M is at least one, and A^M is a truth assignment.

Since A^M is a truth assignment of $\mathbb{T}^M$, we have three possibilities, it can be either T-unsatisfiable, T-satisfiable and non-final, or final in $\mathbb{T}^M$. Since $N \not\vdash^\star_T \bot$ by (a), A^M cannot be T-unsatisfiable. Let us consider the two other cases.

Case 1: A^M is not final. Let $\forall x.C \cdot \sigma, t$ be the new instance added to A^M in the instantiation tree $\mathbb{T}^M$. We show that there is a truth assignment A of $A^M \cup C \cdot (\sigma \cup [x \mapsto t])$ such that $\lceil N \rceil \vdash^\star_T \textsc{Unit}(A)$, which contradicts the maximality of A^M in M.

We have that $\forall x.C \cdot \sigma \in A^M$ and $t \in \mathscr{T}(\textsc{Unit}(A^M))$. If C is not a unit clause, $A^M \cup C \cdot (\sigma \cup [x \mapsto t])$ is a truth assignment of $A^M \cup C \cdot (\sigma \cup [x \mapsto t])$ such that $\lceil N \rceil \vdash^\star_T \textsc{Unit}(A^M \cup C \cdot (\sigma \cup [x \mapsto t]))$ which contradicts the definition of A^M. Therefore, C is a unit clause. Since $\lceil M'' \rceil \vdash^\star_T \textsc{Unit}(A^M)$, there is a substitution σ' and a term $t' \in \mathscr{T}(M'')$ such that $\forall x.C \cdot \sigma' \in M''$, $M'' \vdash^\star_T \sigma \mathrel{\hat{\approx}} \sigma'$, $\mathit{known}(\mathscr{T}(M'')) \cup M'' \vdash^\star_T \mathit{known}(\mathscr{T}(\sigma)) \cup \mathit{known}(\mathscr{T}(t))$ and $M'' \vdash^\star_T t \approx t'$. Since $\forall x.C \cdot \sigma'$ and t' are in M'', this instance has an instantiation level smaller or equal to $n^M + 1$. By (e), $\lceil N \rceil \vdash^\star_T C \cdot (\sigma' \cup [x \mapsto t'])$. Since $M'' \subseteq N$, $\lceil N \rceil \vdash^\star_T C \cdot (\sigma \cup [x \mapsto t])$. By Lemma 14, there is a truth assignment A of $A^M \cup C \cdot (\sigma \cup [x \mapsto t])$ such that $\lceil N \rceil \vdash^\star_T \textsc{Unit}(A)$ which contradicts the definition of A^M.

Case 2: A^M is final. No new instance is possible in A^M. Consider the universal formula $\forall x.C \cdot \sigma \in N$ that we can instantiate with the term $t \in \mathscr{T}(M')$ by fairness. Let us show that we have $\textsc{Avb}(F, M) \vdash^\star_T C \cdot (\sigma \cup [x \mapsto t])$, which contradicts the non-redundancy condition of `Instantiate`.

By Lemma 15 and since $\textsc{Lit}(M) \subseteq A^M$ by construction, there is a term $t' \in \mathscr{T}(A^M)$ and a substitution σ' such that $\forall x.C \cdot \sigma' \in A^M$, $\textsc{Unit}(A^M) \vdash^\star_T \sigma \mathrel{\hat{\approx}} \sigma'$, $\mathit{known}(\mathscr{T}(\textsc{Unit}(A^M))) \cup \textsc{Unit}(A^M) \vdash^\star_T \mathit{known}(\mathscr{T}(\sigma)) \cup \mathit{known}(\mathscr{T}(t))$, and $\textsc{Unit}(A^M) \vdash^\star_T t \approx t'$. Since A^M is final, there is $C \cdot \sigma'' \in A^M$ such that $\textsc{Unit}(A^M) \vdash^\star_T \sigma'' \mathrel{\dot{\approx}} (\sigma' \cup [x \mapsto t'])$. We only need to show that $\textsc{Avb}(F, M) \vdash^\star_T C \cdot \sigma''$. Indeed, since $\lceil M'' \rceil \vdash^\star_T \textsc{Unit}(A^M)$, we can deduce $\textsc{Avb}(F, M) \vdash^\star_T C \cdot (\sigma \cup [x \mapsto t])$. By construction of A^M, we are in one of three cases:

- There is $\langle l \rangle C \cdot \sigma'' \in A^M$. Since $\lceil M'' \rceil \vdash^\star_T \textsc{Unit}(A^M)$, there is $\langle l \rangle C \cdot \tau \in M''$ such that $M'' \vdash^\star_T \sigma'' \mathrel{\dot{\approx}} \tau$ and $M'' \cup \mathit{known}(\mathscr{T}(M'')) \vdash^\star_T \mathit{known}(\mathscr{T}(\sigma''))$. As a consequence, by (f), $\textsc{Avb}(F, M) \vdash^\star_T C \cdot \sigma''$.
- There is $[l]C \cdot \sigma'' \in A^M$ such that $\textsc{Unit}(A^M) \vdash^\star_T l\sigma''$. Since $\lceil M'' \rceil \vdash^\star_T \textsc{Unit}(A^M)$, there is $[l]C \cdot \tau \in M''$ such that $M'' \vdash^\star_T \sigma'' \mathrel{\dot{\approx}} \tau$, $M'' \cup \mathit{known}(\mathscr{T}(M'')) \vdash^\star_T \mathit{known}(\mathscr{T}(\sigma''))$, and $M'' \vdash^\star_T l\sigma''$. Therefore, by (f), $\textsc{Avb}(F, M) \vdash^\star_T C \cdot \sigma''$.
- There is $\forall y.C \cdot (\sigma''\backslash[y \mapsto y\sigma'']) \in A^M$ and $y\sigma'' \in \mathscr{T}(\textsc{Unit}(A^M))$. Since we have $\lceil M'' \rceil \vdash^\star_T \textsc{Unit}(A^M)$, there is $\forall y.C \cdot \tau \in M''$ and $s \in \mathscr{T}(M'')$ such that $M'' \vdash^\star_T (\sigma''\backslash[y \mapsto y\sigma'']) \mathrel{\dot{\approx}} \tau$, $M'' \vdash^\star_T y\sigma'' \approx s$, and $M'' \cup \mathit{known}(\mathscr{T}(M'')) \vdash^\star_T \mathit{known}(\mathscr{T}(\sigma''))$. As a consequence, by (f), $\textsc{Avb}(F, M) \vdash^\star_T C \cdot \sigma''$.

Consequently, e has an instantiation level of at most $n^M + 1$ in M and the proof of Lemma 12 is complete. We can go back to the proofs of the intermediate lemmas.

Proof of Lemma 13 N contains a truth assignment of $A_0^M \cup W \cdot \varnothing$.

Proof By definition of A_0^M, for every literal $l \cdot \varnothing \in \textsc{Unit}(A_0^M)$, we have that $\lceil N \rceil \vdash^\star_T l \cdot \varnothing$. Since $\textsc{Unit}(\textsc{Avb}(F, \varnothing)) \vdash^\star_T \textsc{Unit}(W \cdot \varnothing)$, by (b), if $l \in W$ then $\lceil N \rceil \vdash^\star_T l \cdot \varnothing$. Moreover, for every closure $\varphi \cdot \varnothing \in \textsc{Unit}(W \cdot \varnothing)$ such that φ is not a literal, $\varphi \cdot \varnothing \in \textsc{Unit}(\textsc{Avb}(F, \varnothing))$. By (b), $\lceil N \rceil \vdash^\star_T \textsc{Unit}(\textsc{Avb}(F, \varnothing))$. Therefore, for every closure $\varphi \cdot \varnothing \in \textsc{Unit}(W \cdot \varnothing)$ such that φ is not a literal, $\lceil N \rceil \vdash^\star_T \varphi \cdot \varnothing$ and $\varphi \cdot \varnothing \in N$. What is more, we have:

- For every $\langle l \rangle C$ such that $N \vdash^\star_T \langle l \rangle C \cdot \varnothing$, $\lceil N \rceil \vdash^\star_T l \cdot \varnothing$ and, if C is a unit clause φ then $\lceil N \rceil \vdash^\star_T \varphi \cdot \varnothing$ by (c).

– For every $[l]\varphi \in \text{UNIT}(W)$ such that $N \vdash^\star_T [l]C \cdot \varnothing$ and $N \vdash^\star_T l \cdot \varnothing$, we have $\lceil N \rceil \vdash^\star_T \varphi \cdot \varnothing$ by (d). □

Proof of Lemma 14 For every closure $\varphi \cdot \sigma$ such that $\lceil N \rceil \vdash^\star_T \varphi \cdot \sigma$, there is a truth assignment A of $A^M \cup \varphi \cdot \sigma$ such that $\lceil N \rceil \vdash^\star_T \text{UNIT}(A)$.

Proof We do the proof by structural induction over φ.

If φ is a universally quantified formula, a literal, or a trigger $[l]C \cdot \sigma$ such that $\{l'\sigma' \mid l' \cdot \sigma' \in A^M\} \not\models_T l\sigma$, then $A^M \cup \varphi \cdot \sigma$ is a truth assignment of $A^M \cup \varphi \cdot \sigma$.

If φ is a witness $\langle l \rangle C$ then $\lceil N \rceil \vdash^\star_T l \cdot \sigma$ by (c). If C is not a unit clause, $A^M \cup \varphi \cdot \sigma \cup l \cdot \sigma \cup C \cdot \sigma$ is a truth assignment of $A^M \cup \varphi \cdot \sigma$. Otherwise, $\lceil N \rceil \vdash^\star_T C \cdot \sigma$ by (c). By induction hypothesis, there is a truth assignment A of $A^M \cup C \cdot \sigma$ such that $\lceil N \rceil \vdash^\star_T \text{UNIT}(A)$. As a consequence, $A \cup \varphi \cdot \sigma \cup l \cdot \sigma$ is a truth assignment of $A^M \cup \varphi \cdot \sigma$ and $\lceil N \rceil \vdash^\star_T \text{UNIT}(A) \cup \varphi \cdot \sigma \cup l \cdot \sigma$.

If φ is a trigger $[l]C$ and $\{l\tau \mid l \cdot \tau \in A^M\} \rhd_T l\sigma$ then $\lceil N \rceil \vdash^\star_T l \cdot \sigma$ since $\lceil M' \rceil \vdash^\star_T \text{UNIT}(A^M)$ and $M' \subseteq N$. If C is not a unit clause, $A^M \cup \varphi \cdot \sigma \cup C \cdot \sigma$ is a truth assignment of $A^M \cup \varphi \cdot \sigma$. Otherwise, we deduce that $\lceil N \rceil \vdash^\star_T C \cdot \sigma$ by (d), and, by induction hypothesis, there is a truth assignment A of $A^M \cup C \cdot \sigma$ such that $\lceil N \rceil \vdash^\star_T \text{UNIT}(A)$. As a consequence, $A \cup \varphi \cdot \sigma$ is a truth assignment of $A^M \cup \varphi \cdot \sigma$ and $\lceil N \rceil \vdash^\star_T \text{UNIT}(A) \cup \varphi \cdot \sigma$. □

For the proof of the last intermediate lemma, we use a corollary of Lemma 14:

Corollary 1 *For every guarded clause $H \to C \vee \varphi \cdot \sigma$ that can be obtained by applying either* `Witness-Unfold` *or* `Trigger-Unfold` *such that $\varphi \cdot \sigma \in N$, if $\text{UNIT}(A^M) \vdash^\star_T H$ then $\text{UNIT}(A^M) \vdash^\star_T \varphi \cdot \sigma$. If $\text{UNIT}(A^M)$ is final, then the same is true for any $H \to C \vee \varphi \cdot \sigma$ that can be obtained by applying* `Instantiate`.

Proof We show that, in each case, there is σ' such that A^M is a truth assignment of $\varphi \cdot \sigma'$, $\text{UNIT}(A^M) \vdash^\star_T \sigma \mathrel{\dot{\approx}} \sigma'$ and $\text{UNIT}(A^M) \cup \mathscr{T}(\text{UNIT}(A^M)) \vdash^\star_T \mathit{known}(\mathscr{T}(\sigma))$.

If $H \to C \vee \varphi \cdot \sigma$ can be obtained by `Witness-Unfold`, H is $\langle l \rangle (C \vee \varphi) \cdot \mu$ such that σ is $\mu|_{vars(\varphi)}$. Thus, since $\text{UNIT}(A^M) \vdash^\star_T H$, there is $\langle l \rangle (C \vee \varphi) \cdot \mu' \in A^M$, such that $\text{UNIT}(A^M) \vdash^\star_T \mu \mathrel{\dot{\approx}} \mu'$ and $\text{UNIT}(A^M) \cup \mathscr{T}(\text{UNIT}(A^M)) \vdash^\star_T \mathit{known}(\mathscr{T}(\mu))$. Since $\lceil N \rceil \vdash^\star_T \text{UNIT}(A^M)$, we have both $\lceil N \rceil \vdash^\star_T \mu \mathrel{\dot{\approx}} \mu'$ and $\lceil N \rceil \cup \mathscr{T}(N) \vdash^\star_T \mathit{known}(\mathscr{T}(\mu))$. Hence, $\lceil N \rceil \vdash^\star_T \varphi \cdot \mu'|_{vars(\varphi)}$ and, by Lemma 14, there is a truth assignment A of $A^M \cup \varphi \cdot \mu'|_{vars(\varphi)}$ such that $\lceil N \rceil \vdash^\star_T \text{UNIT}(A)$. By construction, A is a truth assignment of A^M and, by maximality of A^M, $A = A^M$.

If $H \to C \vee \varphi \cdot \sigma$ can be obtained by `Trigger-Unfold`, there is $[l](C \vee \varphi) \cdot \mu$ and $l \cdot \mu|_{vars(l)}$ in H such that $\sigma = \mu|_{vars(\varphi)}$. Like for `Witness-Unfold`, there is $[l](C \vee \varphi) \cdot \mu' \in A^M$ such that $\text{UNIT}(A^M) \vdash^\star_T \mu \mathrel{\dot{\approx}} \mu'$. Since $\text{UNIT}(A^M) \vdash^\star_T l \cdot \mu|_{vars(l)}$, $\text{UNIT}(A^M) \vdash^\star_T l \cdot \mu'|_{vars(l)}$ and there is a truth assignment A of $A^M \cup \varphi \cdot \mu'|_{vars(\varphi)}$ such that $\lceil N \rceil \vdash^\star_T \text{UNIT}(A)$. Since $\text{UNIT}(A^M) \vdash^\star_T l \cdot \mu'|_{vars(l)}$, A is a truth assignment of A^M and, by maximality of A^M, $A = A^M$.

If $\text{UNIT}(A^M)$ is final and $H \to C \vee \varphi \cdot \sigma$ can be obtained by `Instantiate`, there is $\forall x.(C \vee \varphi) \cdot \mu$ and $x \approx x \cdot [x \mapsto t] \in H$ such that $\sigma = (\mu \cup [x \mapsto t])|_{vars(\varphi)}$. Since $\text{UNIT}(A^M) \vdash^\star_T H$, there is $\forall x.C \vee \varphi \cdot \mu' \in A^M$ and $t' \in \mathscr{T}(\text{UNIT}(A^M))$ such that $\text{UNIT}(A^M) \vdash^\star_T (\mu \cup [x \mapsto t]) \mathrel{\dot{\approx}} (\mu' \cup [x \mapsto t'])$ and $\text{UNIT}(A^M) \cup \mathscr{T}(\text{UNIT}(A^M)) \vdash^\star_T \mathit{known}(\mathscr{T}(\mu \cup [x \mapsto t]))$. Since A^M is final, there is $C'' \vee \varphi \cdot \sigma'' \in A^M$ such that $\text{UNIT}(A^M) \vdash^\star_T (\mu' \cup [x \mapsto t']) \mathrel{\dot{\approx}} \sigma''$. Since $\lceil N \rceil \vdash^\star_T \text{UNIT}(A^M)$, we have both $\lceil N \rceil \vdash^\star_T (\mu \cup [x \mapsto t])|_{vars(\varphi)} \mathrel{\dot{\approx}} \sigma''$ and $\lceil N \rceil \cup \mathscr{T}(N) \vdash^\star_T \mathit{known}(\mathscr{T}(\mu \cup [x \mapsto t]))$. Thus, $\lceil N \rceil \vdash^\star_T \varphi \cdot \sigma''$ and, by Lemma 14, there is a truth assignment A of $A^M \cup \varphi \cdot \sigma''$ such that $\lceil N \rceil \vdash^\star_T \text{UNIT}(A)$. By construction, A is a truth assignment of A^M and, by maximality of A^M, $A = A^M$. □

Proof of Lemma 15 If A^M is final, for every $\varphi \cdot \sigma \in N$, we have $\text{UNIT}(A^M) \vdash^\star_T \varphi \cdot \sigma$.

Proof Thanks to Corollary 1, we can show that, for every $\varphi \cdot \sigma \in N$, $\text{UNIT}(A^M) \vdash^\star_T \varphi \cdot \sigma$. By contradiction, let $\varphi \cdot \sigma$ be the first closure of N such that $\text{UNIT}(A^M) \not\vdash^\star_T \varphi \cdot \sigma$. Let $M^\circ(\varphi \cdot \sigma) \parallel F^\circ$ be the state after $\varphi \cdot \sigma$ was added. If $\varphi \cdot \sigma \in \text{GRD}(F^\circ)$ was added to M° using T`-Propagate`, then $\text{UNIT}(A^M) \vdash^\star_T \varphi \cdot \sigma$. Indeed, $\text{UNIT}(A^M)$ entails every closure $\varphi \cdot \sigma$ in M° and also $l \cdot \varnothing$ for every user literal in $l \in M$. By construction, if $\varphi \cdot \sigma$ was added by any other rule, $\varphi \cdot \sigma$ occurs in $\text{AVB}(F^\circ, M^\circ)$. As a consequence, either $\varphi \cdot \sigma \in C'$, for some $C' \in W \cdot \varnothing$, or there is a guarded clause $H \to C$ that can be obtained by either `Witness-Unfold`, `Trigger-Unfold`, or `Instantiate` such that $\varphi \cdot \sigma \in C$ and $H \subseteq M^\circ$. If $\varphi \cdot \sigma \in C'$, for some $C' \in W \cdot \varnothing$, then $\varphi \cdot \sigma \in A^M$, by construction of A^M_1. Otherwise, there is a guarded clause $H \to C$ that can be obtained by either `Witness-Unfold`, `Trigger-Unfold`, or `Instantiate` such that $\varphi \cdot \sigma \in C$ and $\text{UNIT}(A^M) \vdash^\star_T H$. By Corollary 1, $\text{UNIT}(A^M) \vdash^\star_T \varphi \cdot \sigma$. □

The proof of Lemma 12 is now complete.

Theorem 8 (Termination) *Let W be a terminating axiomatization and G a finite set of user clauses. There is no infinite fair derivation Der from $\varnothing \parallel G \cup W \cdot \varnothing$ such that:*

- *Der has no infinite sub-derivation made only of applications of T`-Learn`, T`-Forget`, `Witness-Unfold`, `Trigger-Unfold`, and `Instantiate`,*
- *for every sub-derivation of the form: $S_{i-1} \Longrightarrow S_i \Longrightarrow \ldots \Longrightarrow S_j \Longrightarrow \ldots \Longrightarrow S_k$ where the only `Restart` steps are the ones producing S_i, S_j and S_k, either:*
  - *there are more applications of the rules `UnitPropagate`, `Decide`, T`-Propagate`, and T`-Backjump` in $S_j \Longrightarrow \ldots \Longrightarrow S_k$ than in $S_i \Longrightarrow \ldots \Longrightarrow S_j$, or*
  - *a guarded clause containing only literals and closures with empty substitutions is learned in $S_j \Longrightarrow \ldots \Longrightarrow S_k$ and is not forgotten in Der.*

Remark 12 If the axiomatization W is empty, those are exactly the restrictions needed for termination of classical abstract DPLL modulo theories.

Proof Assume that there is an infinite fair derivation Der that satisfies these restrictions. Since there is only a finite number of literals and closures with empty substitutions in $G \cup W \cdot \varnothing$, after a finite number of steps, `Restart` steps in Der are separated by an increasing number of applications of `UnitPropagate`, `Decide`, T`-Propagate`, and T`-Backjump`. Since there is no infinite sub-derivation of Der made only of applications of T`-Learn`, T`-Forget`, `Witness-Unfold`, `Trigger-Unfold` and `Instantiate`, for every integer n, there is a sub-derivation of Der containing no `Restart` steps and more than n applications of `UnitPropagate`, `Decide`, T`-Propagate`, and T`-Backjump`.

Let us introduce an order on partial models. Every partial model M can be written $M_1 e_1^{\text{d}} M_2 \ldots M_n e_n^{\text{d}} M_{n+1}$ where $e_1^{\text{d}} \ldots e_n^{\text{d}}$ are the only decision super-literals in M. The order is defined as the lexicographic order on sequences $\sharp M_1 \ldots \sharp M_{n+1}$ where $\sharp M_k$ is the length of M_k. An inspection of `UnitPropagate`, `Decide`, T`-Propagate`, and T`-Backjump` shows that they all produce a strictly greater partial model. The rules T`-Learn`, T`-Forget`, `Witness-Unfold`, `Trigger-Unfold` and `Instantiate` do not change the partial model. Since a partial model cannot contain the same super-literal twice, the cardinality of partial models in Der has no finite upper bound. This contradicts Lemma 12, which states the existence of a finite set $\overline{M}$ containing every super-literal of every partial model in Der. □

At this point, we have shown three properties:

- *Soundness:* If the solver returns *Unsat* on a set of clauses G with a sound axiomatization of T' then G is T'-unsatisfiable (Theorem 6).
- *Completeness:* If the solver returns *Sat* on a set of clauses G with a complete axiomatization of T' then G is T'-satisfiable (Theorem 7).
- *Termination:* There cannot be infinite derivations with a terminating axiomatization abiding by a set of constraints (Theorem 8).

To be sure that the solver always returns the right answer on a set of clauses G, we need to make sure that it cannot get stuck, that is, if, after a derivation Der, the solver can neither return *Sat* nor *Unsat* on a set of clauses G, then the derivation Der can be extended without breaking the constraints required for termination. This property is called Progress.

We need an intermediate lemma:

Lemma 16 (Conflict Analysis) *If there is a conflict clause in the state $M \parallel F$ and M contains at least a decision literal, then there is a possible application $M \parallel F \Longrightarrow M'e \parallel F$ of T-`Backjump`.*

Proof Let $H \to C$ be a conflict clause in the state $M \parallel F$. By definition, $H \wedge \neg C \subseteq M$ and $H \to C \in F$. We define a sequence e_i of literals and a sequence M_i of subsequences of M such that M can be written $M_1 e_1^{\mathrm{d}} \dots M_n e_n^{\mathrm{d}} M_{n+1}$ and M_i contain no decision super-literals. We write $\overline{M}_i$ for the prefix $\dots M_i$ of M.

Let us show that, for every $D \subseteq M$ such that $F \cup D \vDash_T^\star \bot$, there is an application $M \parallel F \Longrightarrow M_j \neg e_i \parallel F$ of T-`Backjump`. We do this proof by induction on position of the last and the before-last element of D in M. In other words, we can use the induction hypothesis on a set of super-literals D' if either there is an element of D that appears strictly after every element of D' in M or if the last element e of D in M is in D' and the before-last element of D in M appears strictly after every element of $D' \backslash e$ in M.

If every element of D is in M_1 then there is an application of T-`Backjump` $M \parallel F \Longrightarrow \overline{M}_1 \neg e_1 \parallel F$.

If the element of D that occurs last in M is a decision literal e_i, let $j \leq i$ be the smallest index such that $D \backslash e_i \subseteq \overline{M}_j$ and e_i occurs in $\textsc{Avb}(F, \overline{M}_j)$ (by definition of `Decide` such a j always exists). If $j = 1$ or $e_{j-1} \in D$ or e_i does not occur in $\textsc{Avb}(F, \overline{M}_{j-1})$[1] then $F \cup (D \backslash e_i) \vDash_T^\star \neg e_i$ and e_i is undefined in $\overline{M}_j$. As a consequence, there is a T-`Backjump` step $M \parallel F \Longrightarrow \overline{M}_j \neg e_i \parallel F$.

Otherwise, let e be the element of D that occurs last in M if it is not a decision super-literal and the element of D that occurs before last in M otherwise. Let M' be such that $M = M'e \dots$. By hypothesis, e is not a decision literal. Thus, the super-literal e must have been added to the partial model by one of the rules `UnitPropagate`, T-`Propagate`, or T-`Backjump`. We show that, in each case, there is a set of super-literals $D' \subseteq M'$ such that $F \cup D' \vDash_T^\star e$. We then consider the set $D'' = (D \backslash e) \cup D'$ on which we can apply the induction hypothesis.

- If e was added to M' using `UnitPropagate`, then there is a clause $H \to C \vee e$ such that $H \cup \neg C \subseteq M'$ and $F \vDash_T^\star H \to C \vee e$ by Lemma 10. Thus, $F \cup H \cup \neg C \vDash_T^\star e$.
- If e was added to M' using T-`Propagate`, then $M' \vDash_T^\star e$ by Lemma 8. Let S be a minimal subset of M' such that $S \vDash_T^\star e$. We have $F \cup S \vDash_T^\star e$.

[1] These three hypothesis are not needed to apply T-`Backjump`. Still, to implement conflict analysis, we want to make j as small as possible.

– If e was added to M' using T-Backjump, there is a set of super-literals $D \subseteq M'$ and a set of guarded clauses F' such that $F' \cup D \models^{\star}_{T} e$ and $F \models^{\star}_{T} F'$ by Lemma 10. Thus, $F \cup D \models^{\star}_{T} e$. □

Remark 13 Compared to usual DPLL, back-jumping is restricted by the requirement on T-Backjump that e' must appear in $\text{AVB}(F, M)$. This restriction is needed in general but it can be alleviated by allowing to add a subsequence of $Me^{d}N$ to M using UnitPropagate and T-Propagate before e' is added with T-Backjump.

Corollary 2 *If there is a closure or a literal e such that $\neg e \in M$ and $\lceil M \rceil \vdash^{\star}_{T} e$, then a conflict clause can be learned so that either Fail or T-Backjump can be applied.*

Proof Since $\lceil M \rceil \vdash^{\star}_{T} e$, there is a set of closures $S \subseteq \lceil M \rceil$ such that $S \vdash^{\star}_{T} e$. We construct a guarded clause $H \to e$ that can be added to F using T-Learn. If e is a literal, let H be S itself. Otherwise, since $\neg e \in M$ is an anti-closure, e occurs in $\text{AVB}(F, M)$. Indeed, a guarded clause $H \to C$ of F cannot be forgotten if there is a closure of C defined in M that does not occur in $\text{AVB}(F \backslash H \to C, H)$. Let $H \subseteq \lceil M \rceil$ be a superset of S such that e occurs in $\text{AVB}(F, H)$. Now, we can add $H \to e$ to F using T-Learn. By definition of $\lceil M \rceil$, closures of H either are already in M or can be propagated using T-Propagate without breaking the fairness property. As a consequence, $H \to e$ is a conflict clause and either Fail or, by Lemma 16, T-Backjump can be applied on $M \parallel F$. □

Theorem 9 (Progress) *For every set of ground clauses G and every set of pseudo-clauses W, if the solver can not return after a fair derivation $\varnothing \parallel G \cup W \cdot \varnothing \Longrightarrow M \parallel F$, then there is a rule other than Restart, T-Learn or T-Forget, that can be applied to continue the derivation in a fair way.*

Remark 14 This proof also shows that the definition of fairness does not constrain the choice of instantiating eagerly or lazily, namely after or before deciding on literals of a disjunction. If a decision is possible, then it is allowed and, if an instance is possible, then it will be allowed or redundant after some steps that do not involve any decision.

Proof If the solver cannot return on $M \parallel F$ then at least one of the following is false:

(i) $M \vdash^{\star}_{T} \text{AVB}(F, M)$,
(ii) $M \not\vdash^{\star}_{T} \bot$, and
(iii) if $H \to C$ can be added by Instantiate, Witness-Unfold, or Trigger-Unfold then $\text{AVB}(F, M) \vdash^{\star}_{T} C$.

Assume (i) is false in $M \parallel F$. If there is a guarded clause $H \to C \in F$ such that $H \cup \neg C \subseteq M$, then $H \to C$ is a conflict clause in $M \parallel F$, and, by Lemma 16, either Fail or T-Backjump can be applied. Otherwise, there is an undefined super-literal e that occurs in $\text{AVB}(F, M)$. Since $e \in \text{AVB}(F, M)$, $level_M(e) \leq level(M)$ and Decide can be applied on e.

If (ii) is false, then $\text{LIT}(M) \cup \{l\sigma \mid l \cdot \sigma \in M\} \models_T \bot$. Like in the proof of Corollary 2, a conflict clause can be learned so that either Fail or T-Backjump can be applied.

If (iii) is false in $M \parallel F$, there is a guarded clause $H \to C$ that can be added to F using either Instantiate, Witness-Unfold, or Trigger-Unfold on M such that either $\text{AVB}(F, H) \not\vdash^{\star}_{T} C$ or $\text{AVB}(F, H) \vdash^{\star}_{T} C$ and $H \not\subseteq M$. If $\text{AVB}(F, H) \not\vdash^{\star}_{T} C$, the application of Instantiate, Witness-Unfold, or Trigger-Unfold is non-redundant in F. Otherwise, $\lceil M \rceil \vdash^{\star}_{T} H$ and, for some $l \cdot \sigma \in H \backslash M$, $l \cdot \sigma \in \text{GRD}(F)$. If $\neg(l \cdot \sigma) \in M$, we

conclude using Corollary 2. Otherwise, T-`Propagate` can be applied to $l \cdot \sigma$ if it is not forbidden by fairness.

Assume the application of T-`Propagate` is forbidden by fairness. The application adding $H \to C$ to F must be an application of `Instantiate` and $l \cdot \sigma$ must be of the form $x \approx x \cdot [x \mapsto t]$. At least one of the following properties is false:

(a) $M \not\vdash^\star_T \bot$,
(b) for every unit super-clause $e \in \text{AVB}(F, \varnothing)$, $\lceil M \rceil \vdash^\star_T e$,
(c) for every closure $\langle l \rangle C \cdot \sigma \in M$, $\lceil M \rceil \vdash^\star_T l \cdot \sigma$ and, if C is a unit clause, $\lceil M \rceil \vdash^\star_T C \cdot \sigma$,
(d) for every closure $[l]\varphi \cdot \sigma \in M$ such that φ is a unit clause, if $\lceil M \rceil \vdash^\star_T l \cdot \sigma$ then we have $\lceil M \rceil \vdash^\star_T \varphi \cdot \sigma$,
(e) for every closure $\forall x.\varphi \cdot \sigma \in M$ such that φ is a unit clause and for every term $t \in \mathscr{T}(M)$ such that $level_M(\varphi \cdot (\sigma \cup [x \mapsto t])) \leq level(M)$, we have $\lceil M \rceil \vdash^\star_T \varphi \cdot (\sigma \cup [x \mapsto t])$, and
(f) for every guarded clause $H \to C$ that can be added to F by either `Instantiate`, `Witness-Unfold` or `Trigger-Unfold`, if $level_M(H) \leq level(M)$, $\text{AVB}(F, M) \vdash^\star_T C$.

Condition (a) can not be false if (i) is true.

If (b) is false then there is a unit super-clause $e \in \text{AVB}(F, \varnothing)$ such that $e \notin M$. If $\neg e \in M$, $\varnothing \to e$ is a conflict clause in F. Otherwise, by construction, e is of level 0 in M and e can be added to M using `UnitPropagate`.

If (c) is false, there is a closure $\langle l \rangle C \cdot \sigma \in M$ such that $\lceil M \rceil \not\vdash^\star_T l \cdot \sigma$ or C is a unit clause and $\lceil M \rceil \not\vdash^\star_T C \cdot \sigma$. Therefore, the rule `Witness-Unfold` can be applied with $\langle l \rangle C \cdot \sigma$. If it is redundant, $\text{AVB}(F, M) \vdash^\star_T l \cdot \sigma$ and $\text{AVB}(F, M) \vdash^\star_T C \cdot \sigma$. Therefore, either $\neg(l \cdot \sigma)$ or $\neg(C \cdot \sigma)$ (if it is a unit clause) is in M and there is a conflict clause in F after the application of `Witness-Unfold` or one of $l \cdot \sigma$, $C \cdot \sigma$ can be added to M using `UnitPropagate`.

If (d) is false, there is a closure $[l]\varphi \cdot \sigma \in M$ such that $\lceil M \rceil \vdash^\star_T l \cdot \sigma$ and $\lceil M \rceil \not\vdash^\star_T \varphi \cdot \sigma$. Hence `Trigger-Unfold` can be be applied with $[l]\varphi \cdot \sigma$. If it is redundant, either $l \cdot \sigma$ can be added to M using T-`Propagate`, there is a conflict clause, or $\varphi \cdot \sigma$ can be added to M using `UnitPropagate`.

If (e) is false, there is a closure $\forall x.\varphi \cdot \sigma \in M$ and a term $t \in \mathscr{T}(M)$ such that $\varphi \cdot (\sigma \cup [x \mapsto t])$ has an instantiation level smaller than the current instantiation level in M and $\lceil M \rceil \not\vdash^\star_T \varphi \cdot (\sigma \cup [x \mapsto t])$. First assume that $\text{AVB}(F, M) \not\vdash^\star_T \varphi \cdot (\sigma \cup [x \mapsto t])$. Then, `Instantiate` can be applied with $\forall x.\varphi \cdot \sigma$. If it is redundant then either $x \approx x \cdot [x \mapsto t]$ can be added to M using T-`Propagate`, there is a conflict clause, or $\varphi \cdot (\sigma \cup [x \mapsto t])$ can be added to M using `UnitPropagate`.

If $\text{AVB}(F, M) \vdash^\star_T \varphi \cdot (\sigma \cup [x \mapsto t])$ then $\text{UNIT}(\text{AVB}(F, M)) \vdash^\star_T \varphi \cdot (\sigma \cup [x \mapsto t])$. By Lemma 7, $\lceil M \rceil \not\vdash^\star_T \text{UNIT}(\text{AVB}(F, M))$ (otherwise, $\lceil M \rceil \vdash^\star_T \varphi \cdot (\sigma \cup [x \mapsto t])$). Then, there is a guarded clause $H \to e \in F$ such that $H \subseteq M$ and $M \not\vdash^\star_T e$. Since $H \subseteq M$, e has an instantiation level smaller than the current instantiation level in M and can be added to M using `UnitPropagate`.

Otherwise, (f) is false and either `Instantiate`, `Witness-Unfold` or `Trigger-Unfold` can be applied with $level_M(H) \leq level(M)$ so that either $\text{AVB}(F, H) \not\vdash^\star_T C$ or $\text{AVB}(F, H) \vdash^\star_T C$ and $H \not\subseteq M$. If $\text{AVB}(F, H) \not\vdash^\star_T C$, the application of `Instantiate`, `Witness-Unfold`, or `Trigger-Unfold` is non-redundant in F. Otherwise, $\lceil M \rceil \vdash^\star_T H$ and, for some $l \cdot \sigma \in H \backslash M$, $l \cdot \sigma \in \text{GRD}(F)$. If $\neg(l \cdot \sigma) \in M$, we conclude by Corollary 2. Otherwise, T-`Propagate` can be applied to $l \cdot \sigma$. Indeed, since $level_M(H) \leq level(M)$, it is not forbidden by fairness. □

Theorem 9 shows that the restrictions posed by Theorem 8 are not prohibitive, since a solver that respects them can not get stuck. Indeed, any fair derivation that does not allow the solver to return an answer can be continued without recurring to `Restart`, T`-Learn` or T`-Forget`. Furthermore, the solver cannot be forced to apply indefinitely `Instantiate`, `Trigger-Unfold`, and `Witness-Unfold`, since the number of such steps is limited by the number of closures in the current partial model M.

6 Implementation in the Alt-Ergo Theorem Prover

In this section, we explain how the framework of Sect. 5.4 can be implemented in a SMT solver and the choices we made when implementing it in the Alt-Ergo theorem prover.

6.1 Implementation of Our Extension

For ease of explanation, we first describe a naive implementation of DPLL(T) and then explain how it can be modified to support our framework.

6.1.1 Usual DPLL(T) Implementation

As is usually done for implementations of DPLL(T), we rely on a theory solver $Solver_T$ for theory reasoning. This solver maintains a state composed of a set of assumed literals M along with a set of background literals S for which no truth value has been specified yet but in the validity of which the user is interested. We assume that $Solver_T$ supports the following operations:

- The DPLL engine can notify $Solver_T$ that a new literal is assumed to be true.
- $Solver_T$ can decide the satisfiability of the set M of assumed literals modulo T. If it is unsatisfiable, it also computes a (hopefully small) subset of M that is unsatisfiable.
- When asked, $Solver_T$ can detect which are the undefined literals appearing in the set S of background literals that are entailed by the current set of assumed literals M. For each such literal l, it can compute a subset of M that entails l modulo T.
- $Solver_T$ can be backtracked to a previous state.

Remark 15 Remark that, most of the time, we do not need $Solver_T$ to be complete when checking satisfiability of the current model and generating entailed literals. We only require completeness before either returning or, in our extension, before generating new instances. This last case is necessary to ensure termination of the process.

These operations are those that are usually used to describe possible implementations of DPLL(T) [31]. We now describe a naive implementation GS of DPLL(T) using the theory solver $Solver_T$ for theory reasoning.

Description of the main loop As in the DPLL(T) framework, the solver GS attempts to compute a partial model for a set of ground clauses G. This partial model M is represented as a sequence of literals $M_1 l_1^{\mathrm{d}} \dots M_n l_n^{\mathrm{d}} M_{n+1}$ where $l_1 \dots l_n$ are the only decision literals in M.

During a run of GS, the set of ground clauses F on which the solver works can be modified, either by adding or forgetting redundant clauses. Throughout the run, neither the satisfiability of F nor the set of literals appearing in it will be modified.

To allow an easy analysis when a conflict is encountered, GS maintains a mapping from every element $l \in M_i$ to a set $exp(l)$ of literals appearing before l in M such that $exp(l) \cup F \vDash_T l$. This set is called the *explanation* of l.

When launched on a set of ground clauses G, *GS* initializes F with G, and M with the empty sequence, and enters a loop, where, on each iteration, it applies the first possible rule from the list below, or stops and returns *Sat* if none of the rules are applicable:

1. Check whether there is a conflict clause in F, that is, a clause C in F such that, for all $l \in C$, $\neg l \in M$. If such a clause can be found, then analyze the conflict and either revert a decision (DPLL rule T`-Backjump`), or stop and return *Unsat* if none can be reverted (rule `Fail`). This step, called conflict analysis, is explained in more detail below.
2. Pick a clause $C \vee l \in F$ such that l does not occur in M and, for all $l \in C$, $\neg l \in M$ if any. Add l to M with the explanation $\neg C$ (rule `UnitPropagate`).
3. Check satisfiability of M using the theory solver $Solver_T$. If it is unsatisfiable, use the unsatisfiable subset of M returned by $Solver_T$ to apply conflict analysis, resulting in an application of T`-Backjump` or `Fail`.
4. Use $Solver_T$ to compute the set of literals from G that are entailed by M. For every literal l in this set, use $Solver_T$ to compute a subset M' of M such that $M' \vDash_T l$ and add l to M with the explanation M' (rule T`-Propagate`).
5. Choose a clause of G such that there is no literal $l \in M$ appearing positively in G. Pick a literal l of G such that $\neg l \notin M$ and add l^{d} to M (rule `Decide`).

If none of these rules applies, then *GS* stops and returns *Sat*. Once in a while, *GS* restarts the search from scratch. It also forgets clauses learned during conflict analysis based on a heuristics that we do not describe here.

Description of conflict analysis Conflict analysis is used to find the last decision in a partial model M, if any, that is useful in explaining a conflict. More precisely, it starts from a subset M' of M such that $M' \cup F$ is unsatisfiable. It computes a decision literal l_i in M and a prefix $M_1 l_1^{\mathrm{d}} \dots M_j$ of M with $j \leq i$ such that $M_1 l_1^{\mathrm{d}} \dots M_j \cup F \vDash_T \neg l_i$. For the implementation to be more efficient, it tries to choose i and j as small as possible.

An implementation can work in two steps. First, the smallest possible i is computed. For this, we start from the set of literals S containing the negation of every literal in the conflict clause. By definition, every literal of S appears in M. To compute i, we modify S iteratively in the following way:

We consider the literal $l^{max} \in S$ that appears last in M. If $l^{max} \in M_1$, then no decision literal from M participate in the conflict. Thus, the solver stops and returns *Unsat*. If l^{max} is a decision literal l_k^{d}, we define i to be k. Otherwise, we remove l^{max} from S and add $exp(l^{max})$ to S instead.

Once we have settled for a decision literal l_i^{d} to revert, we try to find the smallest prefix $M_1 l_1^{\mathrm{d}} \dots M_j$ such that $M_1 l_1^{\mathrm{d}} \dots M_j \cup F \vDash_T \neg l_i$ We start from the set S computed in the first step from which we remove l_i. We then continue applying the same process to S until either $S \subseteq M_1$ in which case $j = 1$ or S contains a decision literal l_k^{d} and $S \subseteq M_1 l_1^{\mathrm{d}} \dots M_d$ in which case $j = d$.

To end our analysis, we can now set M to be $M_1 l_1^{\mathrm{d}} \dots M_j \neg l_i$ The explanation $exp(\neg l_i)$ associated to $\neg l_i$ is the set S computed in the last step. We also add the explanation of the conflict to F for further use in the form of a clause $\neg S \vee \neg l_i$.

6.1.2 Extension to Our Logic with Triggers

In this section, we extend the solver *GS* from the previous section to match our logic with triggers. We need two additional operations to communicate with $Solver_T$:

- The DPLL* engine can notify $Solver_T$ of the presence of a new background literal in the problem. This new operation is needed so that $Solver_T$ can support the addition of new literals into the input problem, which is bound to happen when quantifiers are involved.
- $Solver_T$ can be asked to normalize a substitution σ from variables to terms of the problem. It returns a substitution σ' which is σ where each term as been replaced with a representative of its equivalence class. We assume that the representative is always as old as possible, that is, there is no term t in an equivalence class that used to appear in the set of background literals S at a time where its representative was not. This capability can be implemented using a union-find algorithm. It is used to avoid generating instances that are redundant in F modulo T.

Encode the Semantics of Closures Since DPLL$^\star(T)$ works on theory clauses, the partial model M will contain not only literals but closures. As $Solver_T$ does not support closures, we encode them into opaque atoms as described in Sect. 5.

Support our Semantics of Guards To model our semantics, we choose not to use guarded clauses but rather to backtrack the set of clauses F whenever we remove an element from the partial model M. This allows us to ensure that, throughout the derivation, F always contains clauses that are implied by the conjunction of the input problem and the current partial model M.

To enforce this behavior, we split the set F of theory clauses and usual clauses into a sequence $F_1 \dots F_n$. More precisely, every time a decision is added to M, a new set is created. In this way, the new clauses added to F during the search can be backtracked when a decision is reversed. Intuitively, clauses stored in F_i are guarded by elements of $M_1 \dots M_i$ and thus should not be used anymore when elements from this set are removed.

We need to adapt conflict analysis to this semantics of guards. Indeed, after conflict analysis, we add a literal $\neg l_i$ to a prefix $M_1 l_1^{\mathrm{d}} \dots M_j$ of M such that $j \leq i$. Since we removed elements of $l_j^{\mathrm{d}} \dots M_n$ from M, we should also revert F to be $F_1 \dots F_j$. As a consequence, we should take care while computing j to make sure that l_i still occurs in F after the revert. For this, it is enough to stop the second phase of conflict analysis whenever l_i does not occur in $F_1 \dots F_{j-1}$.

For backtracking to be as efficient as possible, we only decide on a literal of a clause C of F once the partial model contains a literal from every clause that was added to F before C.

Then, when we learn a conflict clause C, we have to decide where to add it in F. Indeed, if we add it in the last set F_j of F, then it will be of no use as it will be removed from F the next time an element of $\neg C$ is removed from M. To be as useful as possible, it should be inserted in the first set F_k such that every element of C occurs in $F_1 \dots F_k$.

Remark 16 Conflict clauses may contain negations of closures. In our theoretical framework, we do not support learning such clauses as explained in Remark 7. In the special case of this implementation, it does not cause any additional difficulty though. Indeed, we only use these additional clauses for propagating literals into the partial model and not to forget other clauses or check redundancy of instantiated formulas.

Remark 17 As explained in Sect. 5.5, since we never forget clauses from the input problem, we can safely forget any previously learned clause.

Encode the Semantics of Closures We now explain how our extension handles quantifiers and triggers. Every time a trigger $[l]C$ is added to the partial model M, the theory solver $Solver_T$

is notified that we are now interested in the truth value of the literal l. Then, if l is already in M, we add the clause C to F. Otherwise, $[l]C$ is stored in a structure which is used to check, whenever a literal is added to the partial model M, if it allows to unfold a trigger from M. Nothing is done for triggers $[l]C$ which are propagated by theory reasoning though, as they are obviously redundant.

Universally quantified formulas are handled by adding a new step in the solver's main loop. This step can be located either after or before the last step of *GS*. It consists in taking every universally quantified formula $\forall x.C \cdot \sigma$ and every term t from M and generating the instance $C \cdot \sigma'$ where σ' is the normalization of $(\sigma \cup [x \mapsto t])$. This step is required so that a universally quantified formula cannot be instantiated twice with the same term modulo T.

Remark 18 For efficiency reasons, we simplify the check for redundancy of instances. We only forbid to instantiate a universally quantified formula $\forall x.C \cdot \sigma$ with a term t if the resulting instance $C \cdot \sigma'$ already appears in F. Thus, we generate instances which are forbidden by our theoretical framework. It does not compromise the termination of our solver though. Indeed, these additional instances cannot influence the algorithm. If they contain new literals, these literals will never be decided on by our algorithm as we always decide on literals of older clauses first. Thus, they can only create terms that are equal to already existing terms. Since instances are normalized, these terms will never lead to new instances.

Remark 19 Deciding the relation $\vDash^{\star}_{T}$ on guarded clauses would require using an implementation of classical DPLL(T) inside our DPLL*(T) implementation. In practice, we rather use conflict analysis to deduce valid applications of T`-Backjump` and T`-Learn`.

Remark 20 If the instantiation step is done after the decision step, the solver applies lazy instantiation. Lazy instantiation is safer with non-terminating axiomatizations for which the solver may never be done with instantiating formulas.

Alternatively, the instantiation step can be moved before the decision step. The solver then applies eager instantiation. We discuss later the benefits of each approach on our benchmarks.

6.2 *E*-Matching on Uninterpreted Sub-Terms

Instantiating every universal quantifier with every known term is really inefficient. All the more since some instances are not usable because there is a trigger directly behind the universal quantifier. As a consequence, we would like to use as much as possible the powerful E-matching techniques that are commonly used in SMT solvers. However, we have a constraint which is not usually required in SMT solvers: we need the matching algorithm to be complete. Indeed, this is needed not only for completeness of our solver but also for termination which is mandatory to use an eager instantiation mechanism.

There are two possibilities to easily turn incomplete E-matching techniques into a complete instantiation mechanism. The first one is to restrict the input language so that axiomatizations can only use some restricted form of triggers on which E-matching is complete. The second one, which we have chosen, is to apply E-matching on parts of triggers on which it is complete and then to check the remaining ones.

More formally, assume that we have an E-matching implementation that is complete on terms that only contain uninterpreted symbols, that is, that produces at least a term per equality class that matches a trigger. For every closure $\varphi \cdot \sigma$ where φ is a universally quantified formula $\forall \overline{x}.[l_1, \ldots, l_n]\varphi'$ where φ' is not a trigger, we compute a triplet made of a set of literals l_φ and two sets of terms, p_φ and k_φ. It has the following properties:

(i) every free variable (that is not in the domain of σ) in l_φ or k_φ is also in p_φ,
(ii) terms of p_φ only contain variables and uninterpreted symbols,
(iii) if τ is a mapping from free variables of p_φ to terms containing only variables that are in the domain of σ, then, for i in $1..n$:

$$known(\mathscr{T}(\sigma) \cup \mathscr{T}(\tau\sigma) \cup \mathscr{T}(p_\varphi\tau\sigma) \cup k_\varphi\tau\sigma) \cup l_\varphi\tau\sigma \vDash_T known(\mathscr{T}(l_i))\tau\sigma$$

To instantiate the closure $\varphi \cdot \sigma$, we use the matching algorithm on $p_\varphi\sigma$ to get a substitution τ from free variables of p_φ to known terms. We then wait for every term in $k_\varphi\tau\sigma$ to appear in M, and every literal of $l_\varphi\tau\sigma \cup l_i\sigma$ to be true in M to do the actual instantiation.

To compute the triplet, we proceed in the following way. We associate a fresh variable x_t to every sub-term t of a literal l_i such that t begins with an interpreted function symbol. For every sub-term t of a literal l_i such that t begins with a uninterpreted function symbol, and t does not appear as an argument of a uninterpreted function symbol in l_i, we create a pattern p_t by replacing every sub-term t' of t that begins with an interpreted function symbol by the variable $x_{t'}$. We now define p_φ to be the set of all the patterns p_t constructed above; k_φ to be the set of all the sub-terms t of a literal l_i beginning with an interpreted function symbol such that x_t does not appear in p_φ; and l_s to be the set of all the equalities $x_t \approx t$ where t is a sub-term of l_i beginning with an interpreted function symbol and x_t is not in k_φ.

Example 11 Assume that we have a trigger $f(g(z, 2 \times (y + h(z)))) \approx f(y)$ where f, g, and h are uninterpreted and z and y are universally quantified variables. We can compute the three sets:

$$p = \{f(g(z, x_{2\times(y+h(z))})),\ h(z),\ y,\ f(y)\}$$
$$k = \{y + h(z)\}$$
$$l = \{x_{2\times(y+h(z))} \approx 2 \times (y + h(z))\}$$

Then, applying E-matching on p, we get substitutions $[x_{2\times(y+h(z))} \mapsto t_x,\ y \mapsto t_y,\ z \mapsto t_z]$ such that $f(g(t_z, t_x))$, $h(t_z)$, $f(t_y)$ and all their sub-terms are in $\mathscr{T}(M)$ modulo M. To finish the instance, we only need to wait for $t_y + h(t_z)$ to be known and $t_x \approx 2 \times (t_y + h(t_z))$ and $f(g(t_z, t_x)) \approx f(t_y)$ to be true.

6.3 Different Notions of Termination

The notion of termination in Sect. 2.4 may turn out to be too constraining for some axiomatizations. Let us start with an example. Assume that we want to extend our theory in Sect. 3.1 with a predicate symbol that states that a list is a palindrome $is\_palindrome(l{:}list){:}bool$. In an axiomatization of this concept, we could have:

IS_PALINDROME_DEF:

$$\begin{array}{l}\forall l{:}list.[is\_palindrome(l)]is\_palindrome(l) \approx \mathtt{t} \rightarrow \\ \quad (\forall c_1{:}cursor.[element(l, c_1)]position(l, c_1) > 0 \rightarrow \\ \quad\quad \exists c_2{:}cursor.\langle position(l, c_2) = length(l) - position(l, c_1) + 1\rangle \\ \quad\quad\quad equivalent\_elements(element(l, c_1), element(l, c_2)) \approx \mathtt{t})\end{array}$$

Unfortunately, such an axiom would introduce a loop. If the input set of literals includes the literal $is\_palindrome(l) \approx \mathtt{t}$ and the term $element(l, c)$ is known for some c, there is a branch deducing the term $element(l, sko(l, c))$, where sko is the Skolem function replacing c_2. This term permits to deduce of the term $element(l, sko(l, sko(l, c)))$ and so on. We can see that the term $sko(l, sko(l, c))$ is in fact equal to c, using POSITION_EQ. However, our definition of truth assignment is not restrictive enough to enforce this deduction.

The definition of truth assignment given in Sect. 2.5 can easily be made more or less restrictive. This results in a more or less constraining notion of fairness in the proof of termination of the solver Sect. 5.7. Here are a few examples of alternative choices:

1. Require that at least an element $\varphi_i \cdot \sigma_i$ is added to assignments containing a disjunction $\varphi_1 \cdot \sigma_1 \vee \cdots \vee \varphi_n \cdot \sigma_n$ (in the definition of Sect. 2.5, assignments are allowed to contain none). In practice, this amounts to enforcing a lazy instantiation approach, that is to say that new instances can only be generated when enough literals have been assigned a truth value by the model to entail every clause.
2. Require that, if $\varphi_1\sigma_1 \ldots \varphi_{n-1}\sigma_{n-1}$ are literals that are false in an assignment containing a disjunction $\varphi_1 \cdot \sigma_1 \vee \cdots \vee \varphi_n \cdot \sigma_n$ then $\varphi_n \cdot \sigma_n$ is added to the assignment. A compliant implementation could be obtained by requiring an eager application of T`-Propagate` and `UnitPropagate` in clauses.
3. Do not require that φ is added to assignments containing a witness $\langle l \rangle \varphi$ or a trigger $[l]\varphi$ with l true. The rules `Witness-Unfold` and `Trigger-Unfold` do no longer have to be applied eagerly (before new instances are made).

The first two alternatives would allow the proof of termination of the $is\_palindrome$ example to go through. The first one has the drawback of forbidding eager instantiation of universal quantifiers that can be profitable in practice. We have implemented the second alternative in Alt-Ergo.

Another possibility that we have implemented is allowing the solver to add to truth assignments negations of closures that occur in disjunctions. Termination becomes difficult to achieve with this modification since any universal quantifier can be turned into an existential one and then cause the creation of a new term. To alleviate it, we do not allow negations of closures to be added for the last, or the unique, element of a disjunction. This restriction is motivated by the fact that axioms are often written as implications, the guards being in general simple enough to avoid causing too many new instances.

In general, this gives a more constraining version of termination. Still, if every pseudo-clause occurring in an axiomatization can be written as $l_1 \rightarrow \ldots l_n \rightarrow \varphi$, which is the case for the theory of doubly-linked lists, then only anti-closures of literals can be added to truth assignments. Such anti-closures can only shorten branches in which they are introduced and, therefore, cannot compromise termination. As compensation for this generally more constraining version of termination, new rules can be added to DPLL$^\star(T)$ to handle anti-closures. They have to work in a compatible way with the semantics of negations of closures defined for the proofs of Sect. 5.6. Existential quantifiers arising from the negation of a universally quantified formula can be handled by associating *a priori* a Skolem constant to every universal quantifier in the axiomatization.

Using anti-closures, we have restrained termination further by requiring that, for an element $\varphi_i \cdot \sigma_i$ of a theory clause $\varphi_1 \cdot \sigma_1 \ldots \varphi_n \cdot \sigma_n$ or its negation to be added to a truth assignment A, $\neg(\varphi_k \sigma_k)$ must also be added for every k strictly smaller than i. This amounts to requiring that, in DPLL$^\star(T)$, decision on a closure of a theory clause C must only occur when C is not already true in the current model and must always decide on the first closure of C that is not assigned to false. Remark that we do not lose completeness since it does not depend on the order of decisions.

7 Benchmarks

In this section, we compare our implementation with the built-in quantifier handling available in Alt-Ergo. We aim at justifying that, on terminating axiomatizations, eager instantiation,

allowed by our restrictive semantics of axiomatizations, makes up for the cost of enforcing these restrictions and is in many cases more efficient than the built-in, lazy, quantifier handling.

We also use first-order SMT solvers Z3 and CVC4, in particular to show that our triggers are profitable for their built-in quantifier handling. Still, we do not intend to compare the results of our implementation to theirs. Such comparison would not make much sense, since there are a lot of other differences in implementation of Alt-Ergo, CVC4, and Z3 that influence their performance on our case studies.

7.1 Imperative Doubly-Linked Lists

We use the Why3 VC generator version 0.82 and the Alt-Ergo theorem prover version 0.95.2. The implementation instantiates every universally quantified formula of the theory before deciding on literals. We define some program functions for a program API of lists, using contracts. For example, an element can only be accessed on a valid cursor and, after an application of the modification function `insert`, the new version of the list is related to the old one by the predicate *insert*.

```
val element (l:list) (c:cursor) : element_type
      requires { has_element l c }
      ensures  { result = element l c }

val insert (l:ref list) (c:cursor) (e:element_type) : unit
      requires { has_element !l c \/ c = no_element }
      reads    { l }
      writes   { l }
      ensures  { insert (old !l) c e !l }
```

Here is an example of tests for using the theory of doubly-linked lists. The function `double_size` iterates through the list `li`, inserting the element `e` before each existing element of the list. If the list `li` is not empty at the beginning of the function, then `li` should be twice as long at the end of the function. Since there is a loop, we need to come up with a loop invariant powerful enough to deduce both that the post-condition is true and that the iteration can be resumed after the insertion. The loop invariant states that:

- the current cursor is valid in `li` and used to be valid in `li` at the beginning of the function or *no_element* was reached
- the length of the visited part was doubled, and
- the unvisited part of the list `li` has not been modified yet.

```
let double_size (li : ref list) (e : element_type)
  requires { not (is_empty !li) }
  ensures  { length !li = 2 * (length (old !li)) } =
  let c = ref (first !li) in
  'Loop_Entry:
  while has_element !li !c do
    invariant {
      (((has_element (at !li 'Loop_Entry) !c /\ has_element !li !c)
          \/ !c = no_element) /\
        length (left !li !c) = 2 * (length (left (at !li 'Loop_Entry)!c)) /\
        equal_lists (right !li !c) (right (at !li 'Loop_Entry) !c))
    }
    insert li !c e;
    c := next !li !c
  done
```

Table 1 Time (in seconds) needed to answer correctly on all tests with a timeout of 1000 s with and without triggers in the axiomatization

	Alt-Ergo*	Alt-Ergo 0.95.2		Z3 4.3.1		CVC4 1.3	
With triggers	Yes	Yes	No	Yes	No	Yes	No
`test_false`	0.00	0.00	134.08	TO	TO	TO	Error
`test_delete`	1.32	3.41	140.76	0.02	0.02	0.10	3.84
`test_insert`	41.20	186.70	TO	0.06	0.06	0.30	TO
`double_size`	15.74	13.78	TO	1.26	TO	2.96	TO
`filter`	21.03	19.02	TO	0.56	343.14	TO	Error
`my_contain`	0.36	2.34	TO	0.04	11.94	0.42	Error
`my_find`	6.62	89.08	TO	88.34	TO	2.00	TO
`map_f`	9.84	7.93	TO	0.18	189.44	TO	TO

We do the tests with Z3 4.3.1, CVC4 1.3, and with Alt-Ergo 0.95.2, with its built-in quantifier handling and with our mechanism (named Alt-Ergo*)

We see on results from Table 1 that, on our tests, SMT solvers are far less efficient when triggers are removed from the axiomatization. Indeed, even if triggers can be inferred by SMT solvers, efficient handling of first-order formulas usually requires user guidance in this choice [29]. This validates that, at least on this limited benchmarks, our semantics is rather consistent with the way triggers are handled in first-order SMT solvers.

The test named `test_false` attempts to prove $\bot$. The results on this line are the time needed for each solver to stop returning that it cannot discharge the verification condition. Out of the three solvers we tried, Alt-Ergo is the only one to terminate on this test in less than 1000 s. Indeed, trigger-driven instantiation is the only quantifier instantiation heuristics in Alt-Ergo, which is not the case for Z3 and CVC4.

We use a slightly different version of our axiomatization for Alt-Ergo* in which we have grouped the axioms for each function symbol into one axiom. For example, for *delete*, we now have only one axiom:

DELETE_DEF:

$$
\begin{aligned}
&\forall l_1 l_2 : list, c : cursor.[delete(l_1, c, l_2)]\ delete(l_1, c, l_2) \approx \mathtt{t} \rightarrow \\
&\quad position(l_1, c) > 0 \wedge length(l_2) + 1 \approx length(l_1) \wedge \langle next(l_1, c)\rangle \mathtt{t} \wedge \\
&\quad (\forall c_2 : cursor.[position(l_1, c_2)] \\
&\qquad (position(l_1, c_2) < position(l_1, c) \rightarrow \\
&\qquad\qquad position(l_1, c_2) \approx position(l_2, c_2)) \wedge \\
&\qquad (position(l_1, c_2) > position(l_1, c) \rightarrow \\
&\qquad\qquad position(l_1, c_2) \approx position(l_2, c_2) + 1)) \wedge \\
&\quad (\forall c_2 : cursor.[position(l_2, c_2)] \\
&\qquad (0 < position(l_2, c_2) < position(l_1, c) \rightarrow \\
&\qquad\qquad position(l_1, c_2) \approx position(l_2, c_2)) \wedge \\
&\qquad (position(l_2, c_2) \geq position(l_1, c) \rightarrow \\
&\qquad\qquad position(l_1, c_2) \approx position(l_2, c_2) + 1)) \wedge \\
&\quad (\forall c_2 : cursor.[element(l_1, c_2)] \\
&\qquad position(l_2, c_2) > 0 \rightarrow element(l_1, c_2) = element(l_2, c_2))
\end{aligned}
$$

Indeed, Alt-Ergo, on these examples, works better with separated axioms. It uses a lazy instantiation technique that matches every available trigger once in a while. With the separated

axioms, it is the case that it often can instantiate every needed axiom at once while, with the grouped version, it needs several matching rounds. On the other hand, the theory mechanism works better with the grouped version as it has less triggers to match.

We see that, on average, our implementation gives better results than the usual instantiation mechanism of Alt-Ergo. Indeed, on verification conditions where Alt-Ergo runs the longest we have a considerable improvement and we do not lose too much on others. This is mainly due to the fact that instances are now generated in an eager way. Note that this is also a drawback in some cases where a lot of work is done even if it was not necessary. This is all the more the case when there are things to prove that do not require theory handling and when the amount of work required by the theory is important.

These benchmarks are small, handwritten examples where not that much theory reasoning is needed. Let us now consider a larger, more theory-dependent set of benchmarks.

7.2 Set Theory of Why3

We apply our approach on the theory of sets provided in the Why3 standard library. This theory is a basis of a larger library of Why3 theories used to formalize the set theory of the B method, which is used in the ANR project BWare.[2] It is an industrial research project that aims to provide a framework allowing to automatically discharge verification conditions coming from the verification of safety critical industrial applications using the B method. A generic verification platform is designed that uses several solvers as a back-end. As part of this project, verification conditions coming from the B method are translated into the WhyML language using an axiomatization for polymorphic sets. They can then be discharged by SMT solvers like Alt-Ergo. A benchmark of more than 10,000 verification conditions is provided in this project.

We have applied our method on the core part of the set theory, including only union, disjunction, intersection, and subset. We have also added defining axioms for power sets and integer intervals to the theory. Although they are not complete, these axioms are terminating and therefore cannot hurt the mechanism. Leaving these two defining axioms to the solver's built-in quantifier handling indeed is not efficient as they are often used and should be treated along with other theory axioms. Other axioms about sets, such as those for functions and relations as well as definitions coming from the input problem are left to the built-in quantifier handling of Alt-Ergo.

Figure 4 compares the number of verification conditions discharged by Alt-Ergo, Z3 4.3.2, and CVC4 1.3 in a given amount of time. The set theory is polymorphic. As we use the SMT-lib format for Z3 and CVC4 which does not support polymorphism, an encoding is used for them, where an instance of each lemma is generated for each set type in the input problem. On the other hand, the input syntax of Alt-Ergo is polymorphic. Thus, we have launched Alt-Ergo (both with its built-in quantifier handling only and with our theory mechanism) on two versions of the bench, one with the polymorphic axioms of the set theory and one with the monomorphic encoding. This encoding, as expected, hinders Alt-Ergo both with and without our implementation but it does not prevent our mechanism from improving the solver's efficiency on these benchmarks.

Table 2 compares the time needed to discharge verification conditions with our theory mechanism's implementation in Alt-Ergo and with Alt-Ergo's built-in quantifiers handling only on the polymorphic benchmarks. We see that eager instantiation ends up being rather efficient on these tests.

[2] http://bware.lri.fr/index.php/BWare_project.

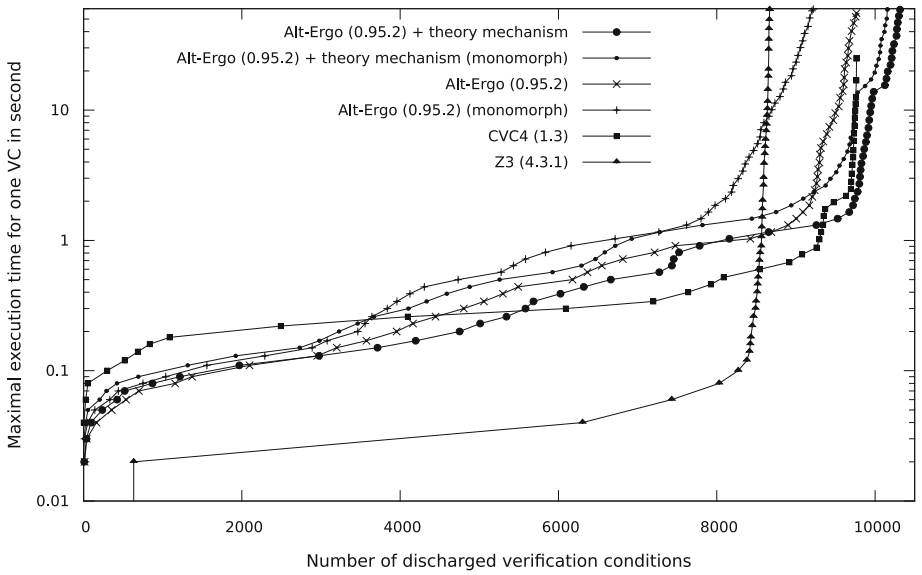


Fig. 4 Evolution of the number of verification conditions from the BWare benchmark discharged by Alt-Ergo 0.95.2, Z3 4.3.1, CVC4 1.3, and by Alt-Ergo through the theory mechanism

Table 2 Number of verification conditions from the BWare benchmark discharged by our theory mechanism's implementation in Alt-Ergo, named Alt-Ergo*, and Alt-Ergo's built-in quantifiers handling

Alt-Ergo*	Discharged	Undischarged	Total
Alt-Ergo			
Discharged	9690	71	9761
Undischarged	626	185	811
Total	10,316	256	10,572

Note that there are also several verification conditions that are discharged by Alt-Ergo's built-in quantifiers handling but not by our implementation. That can be because there are still a great deal of axioms describing additional constructs such as relations, functions, restrictions, finite sets, etc. that we do not include in the theory as well as definitions specific to the input problem. If instances from outside the theory are needed, they will only be performed once no more instance can be done using quantifiers that belong to the theory. If this theory reasoning was not necessary to solve the problem, it will slow down the solver.

On Fig. 5, we compare the time needed to solve the verification conditions using Alt-Ergo's built-in quantifier handling only and using our mechanism. We see that, on verification conditions that are easily discharged by both versions, our framework is a little slower since it has to verify more conditions in order to ensure termination. Still, as we have noted earlier, eager instantiation is profitable enough on an important number of verification conditions to balance out this disadvantage at the end.

8 Related Work

Instantiation with Triggers Triggers are a commonly used heuristic in first-order SMT solvers. User manuals of such solvers usually explain how they should be used to achieve the best performance. Triggers can be automatically computed by the solvers but it is commonly

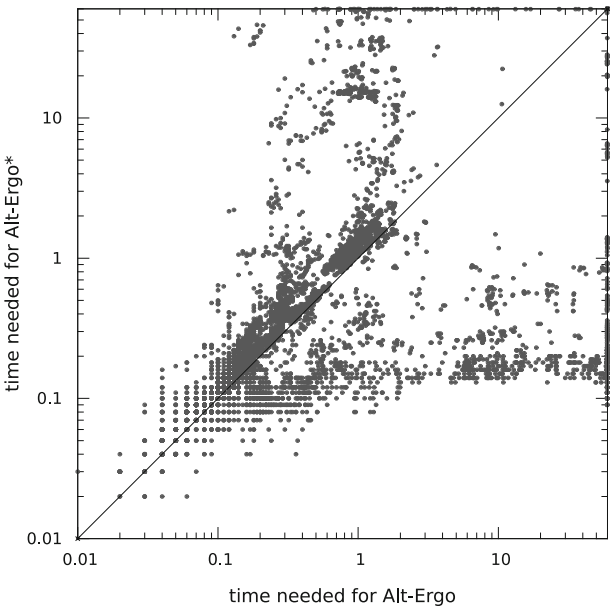


Fig. 5 Time needed to discharge each verification conditions from the BWare benchmark using our theory mechanism's implementation in Alt-Ergo, named Alt-Ergo*, and Alt-Ergo's built-in quantifiers handling

agreed that user guidance is useful in this domain [29]. We believe that our formalization along with the methods and practices described in Sect. 4 can be useful for manual choice of triggers. A lot of work has also been done on defining an efficient mechanism for finding the instances allowed by a trigger. These techniques, called E-matching, are described for Simplify [15,30], Z3 [11], and CVC3 [19].

Even though triggers are a heuristic, axiomatizations with triggers are sometimes used as a work-around to implementing a new decision procedure in an SMT solver. For example, Chatterjee et al. [10] resorted to writing an axiomatization with triggers to implement their decision procedure for well-founded reachability. Though they did pen-and-paper proofs of the termination and completeness of their procedure written as an inference system, they were not able to verify their actual implementation and noticed that, though it seemed to be complete in practice, it was not terminating.

Triggers can also be used in semi-complete first-order ATPs to guide the proof search and improve the prover's efficiency. For example, the Princess prover [33] that combines a complete calculus for first-order logic with a decision procedure for linear arithmetic can use triggers.

Other generic techniques for quantifier handling in SMT solvers Model-based quantifier instantiation [20] is another heuristic for generating instances. This instantiation mechanism generates new instances when the solver is about to return *Sat*, that is, when it has computed a *candidate model*. A candidate model is not always a model when the solver is not in its domain of completeness. It is the case in particular when quantifiers are involved. Model-based instantiation does not in general increase the solver's performance. On the other hand, it increases its accuracy since it allows to continue the search when otherwise the solver would have stopped with only a partial model. It goes through the candidate model M and

searches for an instance that is not already implied by M. If one such instance is found, the search goes on.

A saturation process, close to the superposition calculus, has also been integrated into abstract DPLL(T) by de Moura and Bjørner [12]. The idea is to add elements to the set of clauses that are handled by DPLL by using the superposition calculus. If superposition steps use the current partial assignment of DPLL, the inferred clauses must be prefixed by a guard so that they can be reversed when the current partial assignment is backtracked.

Specialized complete instantiation techniques In SMT solvers, the idea that a set of first-order formulas can be saturated with a finite set of ground instances has been explored previously. For example, decision procedures for universally quantified properties of functional programs can be designed using local model reasoning [22]. This property states that, for every input problem, it is enough to instantiate universal quantifiers with terms from the problem. Just as in our work, the reasoning is performed modulo an existing background theory and proofs of locality of axiomatizations are not in general automatic, though there are classes of universally quantified formulas for which locality can be decided automatically.

In the same way, Ge and de Moura [20] describe fragments of first-order logic for which one can automatically compute a finite set of ground instances that are enough to ensure completeness. As in the previous work, the automation of the termination checks is possible due to the limited extent of these fragments. Alberti et al. [1] also describe fragments of first-order logic that are decidable modulo the theory of arrays.

McPeak and Necula [28] designed a procedure to decide a particular form of first-order formulas. It is dedicated to descriptions of shape properties of data structures, involving pointers and predicates on scalar fields. This procedure is always terminating but is only complete for a more restricted form of formulas. Outside this restriction, they do not provide ways to check that a particular axiomatization is complete.

Our work is in this respect closer to the work done on the axioms for the theory of arrays. For this theory, a specialized instantiation technique achieves both termination and completeness [9,14]. The idea is to treat operations over arrays as uninterpreted functions but to supply additional information about them, that is, instances of array axioms, at appropriate times. In a more general way, some SMT solvers accept sets of rewriting rules as input. Such a set of rules can also produce a decision procedure in specific cases [23].

Work has also been done to reuse or adapt generic instantiation techniques used in SMT solvers to serve as decision procedures. Recently, Bansal et al. [3] showed that local theory extensions can be decided using E-matching techniques. Triggers are computed automatically for a set of quantified formulas. Just like in our work, termination is then ensured by restraining matching so that only new instances modulo the background theory are created.

In the same way, Piskac et al. [32] use first-order SMT solver in their tool GRASShopper to handle formulas from decidable fragments of separation logic. As the first-order formulas they produce are in the effectively propositional fragment of first-order logic, model-based quantifier instantiation yields a decision procedure for them.

Instantiation-based theorem provers such as iProver [24] are semi-complete procedures for first-order logic based on instantiation [25]. To choose appropriate instances, they use a heuristic based on models. A ground solver is launched on the set of ground facts contains in the input problem. If a model is found, they use it to generate instances of universally quantified formulas from the problem that contradict this model. Unlike in our framework, no specific proof is required to ensure the completeness of this procedure. On the other hand, this procedure only works modulo theories for which there is an "answer-complete" ground solver, that

is, a solver able to find a most general substitution for ground instances of a set of literals with free variables to be unsatisfiable [18]. For example, there can be no answer-complete solver for arithmetics.

Demonstrating that a solver always terminates on a given first-order axiomatization of a theory in order to obtain a decision procedure has been done by Armando et al. [2] on a paramodulation-based procedure. In this work, the termination of the superposition calculus on the conjunction of the axiomatization and any input problem was demonstrated manually.

Lynch et al. [26,27] then extended this work by introducing automatic procedures for deciding the termination. In particular, *Schematic Saturation* is used to over-approximate the inferences that paramodulation can generate while solving the satisfiability problem for a certain theory. As opposed to our work, they do not reason modulo a background theory. To allow combination with other decision procedures in a Nelson–Oppen framework, they explain how schematic saturation can also be used to check stably-infiniteness and complete deduction in some cases.

Other ways of integrating decision procedures in an SMT solver Apart from first-order axiomatizations, there are several ways that can be attempted to add a new decision procedure into an SMT solver. Bjørner [7] describes how several non-native theories were successfully supported using the SMT solver Z3. A rather usual way to support a new theory is to encode it in an already supported domain. This paper also presents an API that can be used to add decision procedure to Z3 using callbacks. Both systems require a lot of thinking to get an efficient design as well as manual proofs of completeness and termination, exactly like our axiomatization framework. We believe that each of these different techniques can be better suited than the others to specific theories.

9 Conclusion

We have introduced an abstract description of an SMT solver in which a new theory can be defined as a first-order axiomatization with triggers. If such an axiomatization can be proved to be sound, complete, and terminating in our framework, then the solver will behave as a decision procedure for this theory. We believe that this mechanism will be useful in proof of programs where domain-specific theories are often needed (for libraries, data structures, etc.), as is witnessed by a wide range of papers that deal with theories [7,21,34].

In future work, we would like to allow a more permissive definition of fairness that would support different priorities on parts of the axiomatization. We would require proofs that the axiomatization restricted to high-priority axioms is terminating in our framework so that these axioms can be instantiated before the others.

We would also like to investigate the combination of several theories defined as first-order axiomatizations in a Nelson-Oppen framework. This will require determining which requirements are needed to preserve termination.

Another area worth of investigation is automated checking of the completeness and termination of an axiomatization, at least in some restricted cases.

Acknowledgments We would like to thank the anonymous reviewers for their careful reading of our article, as well as their helpful comments which greatly contributed to the quality of this paper. This work is partially supported by the BWare Project (ANR-12-INSE-0010, http://bware.lri.fr/) and the Joint Laboratory ProofInUse (ANR-13-LAB3-0007, http://www.spark-2014.org/proofinuse) of the French national research organization.

Appendix: Axiomatization of Imperative Doubly-Linked Lists

LENGTH_GTE_ZERO:

$$\forall l\colon list.[length(l)]length(l) \geq 0$$

IS_EMPTY:

$$\forall l\colon list.[is\_empty(l)]is\_empty(l) \approx \mathtt{t} \leftrightarrow length(l) \approx 0$$

EMPTY_IS_EMPTY:

$$is\_empty(empty)$$

EQUAL_ELEMENTS_REFL:

$$\forall e\colon element\_type.[equal\_elements(e,e)]equal\_elements(e,e) \approx \mathtt{t}$$

EQUAL_ELEMENTS_SYM:

$$\forall e_1 e_2\colon element\_type.[equal\_elements(e_1,e_2)]$$
$$equal\_elements(e_1,e_2) \approx equal\_elements(e_2,e_1)$$

EQUAL_ELEMENTS_TRANS:

$$\forall e_1 e_2 e_3\colon element\_type.[equal\_elements(e_1,e_2), equal\_elements(e_2,e_3)]$$
$$equal\_elements(e_1,e_2) \approx \mathtt{t} \rightarrow equal\_elements(e_2,e_3) \approx \mathtt{t} \rightarrow$$
$$equal\_elements(e_1,e_3) \approx \mathtt{t}$$
$$\forall e_1 e_2 e_3\colon element\_type.[equal\_elements(e_1,e_2), equal\_elements(e_1,e_3)]$$
$$equal\_elements(e_1,e_2) \approx \mathtt{t} \rightarrow equal\_elements(e_2,e_3) \approx \mathtt{t} \rightarrow$$
$$equal\_elements(e_1,e_3) \approx \mathtt{t}$$

POSITION_GTE_ZERO:

$$\forall l\colon list, c\colon cursor.[position(l,c)]$$
$$length(l) \geq position(l,c) \wedge position(l,c) \geq 0$$

POSITIONNO_ELEMENT:

$$\forall l\colon list.[position(l, no\_element)]position(l, no\_element) \approx 0$$

POSITION_EQ:

$$\forall l\colon list, c_1 c_2\colon cursor.[position(l,c_1), position(l,c_2)]$$
$$position(l,c_1) > 0 \rightarrow position(l,c_1) \approx position(l,c_2) \rightarrow c_1 \approx c_2$$

PREVIOUS_IN:

$$\forall l\colon list, c\colon cursor.[previous(l,c)]$$
$$(position(l,c) > 1 \vee position(l, previous(l,c)) > 0) \rightarrow$$
$$position(l, previous(l,c)) \approx position(l,c) - 1$$

PREVIOUS_EXT:

$$\forall l\colon list, c\colon cursor.[previous(l,c)]$$
$$(position(l,c) \approx 1 \vee c \approx no\_element) \rightarrow previous(l,c) \approx no\_element$$

NEXT_IN:

$$\forall l{:}\,list, c{:}\,cursor.[next(l,c)]\\ (length(l) > position(l,c) > 0 \vee position(l,next(l,c)) > 0) \rightarrow\\ position(l,next(l,c)) \approx position(l,c) + 1$$

NEXT_EXT:

$$\forall l{:}\,list, c{:}\,cursor.[next(l,c)]\\ (position(l,c) \approx length(l) \vee c \approx no\_element) \rightarrow next(l,c) \approx no\_element$$

LAST_EMPTY:

$$\forall l{:}\,list.[last(l)]length(l) \approx 0 \leftrightarrow last(l) \approx no\_element$$

LAST_GEN:

$$\forall l{:}\,list.[last(l)]length(l) \approx position(l,last(l))$$

FIRST_EMPTY:

$$\forall l{:}\,list.[first(l)]length(l) \approx 0 \leftrightarrow first(l) \approx no\_element$$

FIRST_GEN:

$$\forall l{:}\,list.[first(l)]length(l) > 0 \rightarrow position(l, first(l)) \approx 1$$

HAS_ELEMENT_DEF:

$$\forall l{:}\,list, c{:}\,cursor.[has\_element(l,c)]position(l,c) > 0 \leftrightarrow has\_element(l,c) \approx \mathtt{t}$$

LEFT_NO_ELEMENT:

$$\forall l{:}\,list.[left(l,no\_element)]left(l,no\_element) \approx l$$

LEFT_LENGTH:

$$\forall l{:}\,list, c{:}\,cursor.[left(l,c)]\\ position(l,c) > 0 \rightarrow length(left(l,c)) \approx position(l,c) - 1$$

LEFT_POSITION_IN:

$$\forall l{:}\,list, c_1c_2{:}\,cursor.[position(left(l,c_1),c_2)]\\ (position(left(l,c_1),c_2) > 0 \vee position(l,c_2) < position(l,c_1)) \rightarrow\\ position(left(l,c_1),c_2) \approx position(l,c_2)\\ \forall l{:}\,list, c_1c_2{:}\,cursor.[left(l,c_1), position(l,c_2)]\\ (position(left(l,c_1),c_2) > 0 \vee position(l,c_2) < position(l,c_1)) \rightarrow\\ position(left(l,c_1),c_2) \approx position(l,c_2)$$

LEFT_POSITION_EXT:

$$\forall l{:}\,list, c_1c_2{:}\,cursor.[position(left(l,c_1),c_2)]\\ position(l,c_2) \geq position(l,c_1)) > 0 \rightarrow\\ position(left(l,c_1),c_2) \approx 0$$

LEFT_ELEMENT:

$$\forall l\colon list, c_1c_2\colon cursor.[element(left(l, c_1), c_2)]$$
$$(position(left(l, c_1), c_2) > 0 \vee 0 < position(l, c_2) < position(l, c_1)) \rightarrow$$
$$element(left(l, c_1), c_2) \approx element(l, c_2)$$
$$\forall l\colon list, c_1c_2\colon cursor.[left(l, c_1), element(l, c_2)]$$
$$(position(left(l, c_1), c_2) > 0 \vee 0 < position(l, c_2) < position(l, c_1)) \rightarrow$$
$$element(left(l, c_1), c_2) \approx element(l, c_2)$$

RIGHT_NO_ELEMENT:

$$\forall l\colon list.[right(l, no\_element)]right(l, no\_element) \approx empty$$

RIGHT_LENGTH:

$$\forall l\colon list, c\colon cursor.[right(l, c)]$$
$$position(l, c) > 0 \rightarrow length(right(l, c)) \approx length(l) - position(l, c) + 1$$

RIGHT_POSITION_IN:

$$\forall l\colon list, c_1c_2\colon cursor.[position(right(l, c_1), c_2)]$$
$$(position(right(l, c_1), c_2) > 0 \vee 0 < position(l, c_1) \leq position(l, c_2)) \rightarrow$$
$$position(right(l, c_1), c_2) \approx position(l, c_2) - position(l, c_1) + 1$$
$$\forall l\colon list, c_1c_2\colon cursor.[right(l, c_1), position(l, c_2)]$$
$$(position(right(l, c_1), c_2) > 0 \vee 0 < position(l, c_1) \leq position(l, c_2)) \rightarrow$$
$$position(right(l, c_1), c_2) \approx position(l, c_2) - position(l, c_1) + 1$$

RIGHT_POSITION_EXT:

$$\forall l\colon list, c_1c_2\colon cursor.[position(right(l, c_1), c_2)]$$
$$position(l, c_2) < position(l, c_1)) \rightarrow$$
$$position(right(l, c_1), c_2) \approx 0$$

RIGHT_ELEMENT:

$$\forall l\colon list, c_1c_2\colon cursor.[element(right(l, c_1), c_2)]$$
$$(position(right(l, c_1), c_2) > 0 \vee 0 < position(l, c_1) \leq position(l, c_2)) \rightarrow$$
$$element(right(l, c_1), c_2) \approx element(l, c_2)$$
$$\forall l\colon list, c_1c_2\colon cursor.[right(l, c_1), element(l, c_2)]$$
$$(position(right(l, c_1), c_2) > 0 \vee 0 < position(l, c_1) \leq position(l, c_2)) \rightarrow$$
$$element(right(l, c_1), c_2) \approx element(l, c_2)$$

FIND_FIRST_RANGE:

$$\forall l\colon list, e\colon element\_type.[find\_first(l, e)]$$
$$find\_first(l, e) \approx no\_element \vee position(l, find\_first(l, e)) > 0$$

FIND_FIRST_NOT:

$$\forall l\colon list, e\colon element\_type, c : cursor.[find\_first(l, e), element(l, c)]$$
$$find\_first(l, e) \approx no\_element \rightarrow position(l, c) > 0 \rightarrow$$
$$equal\_elements(element(l, c), e) \not\approx \mathtt{t}$$

FIND_FIRST_FIRST:

$$\forall l\colon list, e\colon element\_type, c : cursor.[find\_first(l, e), element(l, c)]$$
$$0 < position(l, c) < position(l, find\_first(l, e)) \rightarrow$$
$$equal\_elements(element(l, c), e) \not\approx \mathtt{t}$$

FIND_FIRST_ELEMENT:

$$\forall l{:}\,list, e{:}\,element\_type.[find\_first(l,e)]0 < position(l, find\_first(l,e)) \rightarrow equal\_elements(element(l, find\_first(l,e)), e) \approx \mathtt{t}$$

CONTAINS_DEF:

$$\forall l{:}\,list, e{:}\,element\_type.[contains(l,e)] \quad contains(l,e) \leftrightarrow 0 < position(l, find\_first(l,e))$$

FIND_FIRST:

$$\forall l{:}\,list, e{:}\,element\_type.[find(l,e,no\_element)] \quad find(l,e,no\_element) \approx find\_first(l,e)$$

FIND_OTHERS:

$$\forall l{:}\,list, e{:}\,element\_type, c{:}\,cursor.[find(l,e,c)] \quad position(l,c) > 0 \rightarrow find(l,e,c) \approx find\_first(right(l,c),e)$$

REPLACE_ELEMENT_RANGE:

$$\forall l_1 l_2{:}\,list, c{:}\,cursor, e{:}\,element\_type.[replace\_element(l_1,c,e,l_2)] \quad replace\_element(l_1,c,e,l_2) \approx \mathtt{t} \rightarrow position(l_1,c) > 0$$

REPLACE_ELEMENT_LENGTH:

$$\forall l_1 l_2{:}\,list, c{:}\,cursor, e{:}\,element\_type.[replace\_element(l_1,c,e,l_2)] \quad replace\_element(l_1,c,e,l_2) \approx \mathtt{t} \rightarrow length(l_1) \approx length(l_2)$$

REPLACE_ELEMENT_POSITION:

$$\forall l_1 l_2{:}\,list, c_1 c_2{:}\,cursor, e{:}\,element\_type.[replace\_element(l_1,c_1,e,l_2), position(l_1,c_2)] \quad replace\_element(l_1,c_1,e,l_2) \approx \mathtt{t} \rightarrow position(l_1,c_2) \approx position(l_2,c_2)$$

$$\forall l_1 l_2{:}\,list, c_1 c_2{:}\,cursor, e{:}\,element\_type.[replace\_element(l_1,c_1,e,l_2), position(l_2,c_2)] \quad replace\_element(l_1,c_1,e,l_2) \approx \mathtt{t} \rightarrow position(l_1,c_2) \approx position(l_2,c_2)$$

REPLACE_ELEMENT_ELEMENT_IN:

$$\forall l_1 l_2{:}\,list, c{:}\,cursor, e{:}\,element\_type.[replace\_element(l_1,c,e,l_2)] \quad replace\_element(l_1,c,e,l_2) \approx \mathtt{t} \rightarrow element(l_2,c) \approx e$$

REPLACE_ELEMENT_ELEMENT_EXT:

$$\forall l_1 l_2{:}\,list, c_1 c_2{:}\,cursor, e{:}\,element\_type.[replace\_element(l_1,c_1,e,l_2), element(l_1,c_2)] \quad (replace\_element(l_1,c_1,e,l_2) \approx \mathtt{t} \wedge position(l_1,c_2) > 0 \wedge c_1 \not\approx c_2) \rightarrow element(l_1,c_2) \approx element(l_2,c_2)$$

$$\forall l_1 l_2{:}\,list, c_1 c_2{:}\,cursor, e{:}\,element\_type.[replace\_element(l_1,c_1,e,l_2), element(l_2,c_2)] \quad (replace\_element(l_1,c_1,e,l_2) \approx \mathtt{t} \wedge position(l_1,c_2) > 0 \wedge c_1 \not\approx c_2) \rightarrow element(l_1,c_2) \approx element(l_2,c_2)$$

INSERT_RANGE:

$$\forall l_1 l_2{:}\,list, c{:}\,cursor, e{:}\,element\_type.[insert(l_1,c,e,l_2)] \quad insert(l_1,c,e,l_2) \approx \mathtt{t} \rightarrow c \approx no\_element \vee position(l_1,c) > 0$$

INSERT_LENGTH:

$$\forall l_1 l_2 : list, c : cursor, e : element\_type.[insert(l_1, c, e, l_2)]$$
$$insert(l_1, c, e, l_2) \approx \mathtt{t} \rightarrow length(l_2) \approx length(l_1) + 1$$

INSERT_NEW:

$$\forall l_1 l_2 : list, c : cursor, e : element\_type.[insert(l_1, c, e, l_2)]$$
$$(insert(l_1, c, e, l_2) \approx \mathtt{t} \wedge position(l_1, c) > 0) \rightarrow$$
$$position(l_1, previous(l_2, c)) \approx 0 \wedge element(l_2, previous(l_2, c)) \approx e$$

INSERT_NEW_NO_ELEMENT:

$$\forall l_1 l_2 : list, c : cursor, e : element\_type.[insert(l_1, no\_element, e, l_2)]$$
$$insert(l_1, no\_element, e, l_2) \approx \mathtt{t} \rightarrow$$
$$position(l_1, last(l_2)) \approx 0 \wedge element(l_2, last(l_2)) \approx e$$

INSERT_POSITION_BEFORE:

$$\forall l_1 l_2 : list, c_1 c_2 : cursor, e : element\_type.[insert(l_1, c_1, e, l_2), position(l_1, c_2)]$$
$$(insert(l_1, c_1, e, l_2) \approx \mathtt{t} \wedge 0 < position(l_1, c_2) < position(l_1, c_1)) \rightarrow$$
$$position(l_1, c_2) \approx position(l_2, c_2)$$
$$\forall l_1 l_2 : list, c_1 c_2 : cursor, e : element\_type.[insert(l_1, c_1, e, l_2), position(l_2, c_2)]$$
$$(insert(l_1, c_1, e, l_2) \approx \mathtt{t} \wedge position(l_2, c_2) < position(l_1, c_1)) \rightarrow$$
$$position(l_1, c_2) \approx position(l_2, c_2)$$

INSERT_POSITION_AFTER:

$$\forall l_1 l_2 : list, c_1 c_2 : cursor, e : element\_type.[insert(l_1, c_1, e, l_2), position(l_1, c_2)]$$
$$(insert(l_1, c_1, e, l_2) \approx \mathtt{t} \wedge position(l_1, c_2) \geq position(l_1, c_1) > 0) \rightarrow$$
$$position(l_1, c_2) + 1 \approx position(l_2, c_2)$$
$$\forall l_1 l_2 : list, c_1 c_2 : cursor, e : element\_type.[insert(l_1, c_1, e, l_2), position(l_2, c_2)]$$
$$(insert(l_1, c_1, e, l_2) \approx \mathtt{t} \wedge position(l_2, c_2) > position(l_1, c_1) > 0) \rightarrow$$
$$position(l_1, c_2) + 1 \approx position(l_2, c_2)$$

INSERT_POSITION_NO_ELEMENT:

$$\forall l_1 l_2 : list, c : cursor, e : element\_type.[insert(l_1, no\_element, e, l_2), position(l_1, c)]$$
$$(insert(l_1, no\_element, e, l_2) \approx \mathtt{t} \wedge position(l_1, c) > 0) \rightarrow$$
$$position(l_1, c) \approx position(l_2, c)$$
$$\forall l_1 l_2 : list, c : cursor, e : element\_type.[insert(l_1, no\_element, e, l_2), position(l_2, c)]$$
$$(insert(l_1, no\_element, e, l_2) \approx \mathtt{t} \wedge position(l_2, c_2) < length(l_2)) \rightarrow$$
$$position(l_1, c) \approx position(l_2, c)$$

INSERT_ELEMENT:

$$\forall l_1 l_2 : list, c_1 c_2 : cursor, e : element\_type.[insert(l_1, c_1, e, l_2), element(l_1, c_2)]$$
$$(insert(l_1, c_1, e, l_2) \approx \mathtt{t} \wedge position(l_1, c_2) > 0) \rightarrow$$
$$element(l_1, c_2) \approx element(l_2, c_2)$$
$$\forall l_1 l_2 : list, c_1 c_2 : cursor, e : element\_type.[insert(l_1, c_1, e, l_2), element(l_2, c_2)]$$
$$(insert(l_1, c_1, e, l_2) \approx \mathtt{t} \wedge position(l_1, c_2) > 0) \rightarrow$$
$$element(l_1, c_2) \approx element(l_2, c_2)$$

DELETE_RANGE:

$$\forall l_1 l_2 : list, c : cursor.[delete(l_1, c, l_2)]$$
$$delete(l_1, c, l_2) \approx \mathtt{t} \rightarrow position(l_1, c) > 0$$

DELETE_LENGTH:

$$\forall l_1 l_2 : list, c : cursor.[delete(l_1, c, l_2)] \\ delete(l_1, c, l_2) \approx \mathtt{t} \rightarrow length(l_2) + 1 \approx length(l_1)$$

DELETE_POSITION_BEFORE:

$$\forall l_1 l_2 : list, c_1 c_2 : cursor.[delete(l_1, c_1, l_2), position(l_1, c_2)] \\ (delete(l_1, c_1, l_2) \approx \mathtt{t} \wedge position(l_1, c_2) < position(l_1, c_1)) \rightarrow \\ position(l_1, c_2) \approx position(l_2, c_2) \\ \forall l_1 l_2 : list, c_1 c_2 : cursor.[delete(l_1, c_1, l_2), position(l_2, c_2)] \\ (delete(l_1, c_1, l_2) \approx \mathtt{t} \wedge 0 < position(l_2, c_2) < position(l_1, c_1)) \rightarrow \\ position(l_1, c_2) \approx position(l_2, c_2)$$

DELETE_POSITION_AFTER:

$$\forall l_1 l_2 : list, c_1 c_2 : cursor.[delete(l_1, c_1, l_2), position(l_1, c_2)] \\ (delete(l_1, c_1, l_2) \approx \mathtt{t} \wedge position(l_1, c_2) > position(l_1, c_1)) \rightarrow \\ position(l_1, c_2) \approx position(l_2, c_2) + 1 \\ \forall l_1 l_2 : list, c_1 c_2 : cursor.[delete(l_1, c_1, l_2), position(l_2, c_2)] \\ (delete(l_1, c_1, l_2) \approx \mathtt{t} \wedge position(l_2, c_2) \geq position(l_1, c_1)) \rightarrow \\ position(l_1, c_2) \approx position(l_2, c_2) + 1$$

DELETE_POSITION_NEXT:

$$\forall l_1 l_2 : list, c : cursor.[delete(l_1, c, l_2)] delete(l_1, c, l_2) \approx \mathtt{t} \rightarrow \langle next(l_1, c) \rangle \mathtt{t}$$

DELETE_ELEMENT:

$$\forall l_1 l_2 : list, c_1 c_2 : cursor.[delete(l_1, c_1, l_2), element(l_1, c_2)] \\ (delete(l_1, c, l_2) \approx \mathtt{t} \wedge position(l_2, c_2) > 0) \rightarrow \\ element(l_1, c_2) = element(l_2, c_2)$$

EQUAL_LISTS_POSITION:

$$\forall l_1 l_2 : list.[equal\_lists(l_1, l_2)] equal\_lists(l_1, l_2) \approx \mathtt{t} \rightarrow \\ (\forall c : cursor.[position(l_1, c)] position(l_1, c) \approx position(l_2, c)) \wedge \\ (\forall c : cursor.[position(l_2, c)] position(l_1, c) \approx position(l_2, c))$$

EQUAL_LISTS_ELEMENT:

$$\forall l_1 l_2 : list.[equal\_lists(l_1, l_2)] equal\_lists(l_1, l_2) \approx \mathtt{t} \rightarrow \\ (\forall c : cursor.[element(l_1, c)] position(l_1, c) > 0 \rightarrow \\ element(l_1, c) \approx element(l_2, c)) \wedge \\ (\forall c : cursor.[element(l_2, c)] position(l_1, c) > 0 \rightarrow \\ element(l_1, c) \approx element(l_2, c))$$

EQUAL_LISTS_INV:

$$\forall l_1 l_2 : list.[equal\_lists(l_1, l_2)] equal\_lists(l_1, l_2) \not\approx \mathtt{t} \rightarrow \\ (\exists c : cursor.position(l_1, c) > 0 \wedge \\ (position(l_1, c) \approx position(l_2, c) \rightarrow element(l_1, c) \not\approx element(l_2, c)))$$

EQUAL_LISTS_LENGTH:

$$\forall l_1 l_2 : list.[equal\_lists(l_1, l_2)] equal\_lists(l_1, l_2) \approx \mathtt{t} \rightarrow length(l_1) \approx length(l_2)$$

References

1. Alberti, F., Ghilardi, S., Sharygina, N.: Decision procedures for flat array properties. In: Ábrahám, E., Havelund, K. (eds.) and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 8413, pp. 15–30. Springer, Berlin (2014)
2. Armando, A., Ranise, S., Rusinowitch, M.: A rewriting approach to satisfiability procedures. Inf. Comput. **183**(2), 140–164 (2003). In: 12th International Conference on Rewriting Techniques and Applications (RTA 2001)
3. Bansal, K., Reynolds, A., King, T., Barrett, C., Wies, T.: Deciding local theory extensions via E-matching. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification, CAV 2015, Lecture Notes in Computer Science. Vol. 9207, pp.87–105 Springer (2015)
4. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification. Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer, Berlin (2011)
5. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard version 2.0. Technical report, University of Iowa (2010)
6. Biere, A., Heule, M., van Maaren, H.: Handbook of Satisfiability, vol. 185. IOS Press, Amsterdam (2009)
7. Bjørner, N.: Engineering theories with Z3. In: Yang, H. (ed.) Proceedings of the 9th Asian Symposium on Programming Languages and Systems, APLAS 2011, Lecture Notes in Computer Science, vol. 7078, pp. 4–16. Springer, Berlin (2011)
8. Bobot, F., Conchon, S., Contejean, E., Lescuyer, S.: Implementing polymorphism in SMT solvers. In: SMT'08, ACM ICPS, vol. 367, pp. 1–5. ACM, New York (2008)
9. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2006). Lecture Notes in Computer Science, vol. 3855, pp. 427–442. Springer, Berlin (2006)
10. Chatterjee, S., Lahiri, S.K., Qadeer, S., Rakamarić, Z.: A low-level memory model and an accompanying reachability predicate. Int. J. Softw. Tools Technol. Transf. **11**(2), 105–116 (2009)
11. de Moura, L., Bjørner, N.: Efficient E-matching for SMT solvers. In: Pfenning, F. (ed.) CADE-21, LNCS, vol. 4603, pp. 183–198. Springer, Berlin (2007)
12. de Moura, L., Bjørner, N.: Engineering DPLL(T) + saturation. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008, LNCS, vol. 5195, pp. 475–490. Springer, Berlin (2008)
13. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C. R., Rehof, J. (eds.) TACAS 2008, LNCS, vol. 4963, pp. 337–340. Springer (2008)
14. de Moura, L., Bjørner, N.: Generalized, efficient array decision procedures. In: Biere, A., Pixley, C. (eds.) Formal Methods in Computer-Aided Design, 2009. FMCAD 2009, pp. 45–52. IEEE (2009)
15. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005)
16. Dross, C., Conchon, S., Kanig, J., Paskevich, A.: Reasoning with triggers. In: 10th International Workshop on Satisfiability Modulo Theories, SMT 2012, pp. 22–31 (2012)
17. Dross, C., Filliâtre, J.C., Moy, Y.: Correct code containing containers. In: Gogolla, M., Wolff, B. (eds.) Proceedings of the 5th International Conference on Tests and Proofs, TAP'11, Lecture Notes in Computer Science, vol. 6706, pp. 102–118. Springer, Berlin (2011)
18. Ganzinger, H., Korovin, K.: Theory instantiation. In: Hermann, M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2006, Lecture Notes in Computer Science, vol. 4246, pp. 497–511. Springer, Berlin (2006)
19. Ge, Y., Barrett, C., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. In: Pfenning, F. (ed.) CADE-21, LNCS, vol. 4603, pp. 167–182. Springer, Berlin (2007)
20. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiability modulo theories. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification, *LNCS*, vol. 5643, pp. 306–320. Springer (2009)
21. Goel, A., Krstić, S., Fuchs, A.: Deciding array formulas with frugal axiom instantiation. In: Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning, ACM ICPS, pp. 12–17. ACM, New York (2008)
22. Jacobs, S., Kuncak, V.: Towards complete reasoning about axiomatic specifications. In: Jhala, R., Schmidt, D. (eds.) Proceedings of VMCAI-12, LNCS, vol. 6538, pp. 278–293. Springer, Berlin (2012)
23. Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebras. In: Automation of Reasoning, Symbolic Computation, pp. 342–376. Springer, Berlin (1983)
24. Korovin, K.: iProver—an instantiation-based theorem prover for first-order logic (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) Proceedings of the 4th International Joint Conference

on Automated Reasoning, (IJCAR 2008), Lecture Notes in Computer Science, vol. 5195, pp. 292–298. Springer, Berlin (2008)
25. Korovin, K.: Instantiation-based reasoning: from theory to practice. In: Schmidt, R.A. (ed.) 22nd International Conference on Automated Deduction, CADE'22, Lecture Notes in Computer Science, vol. 5663, pp. 163–166. Springer, Berlin (2009)
26. Lynch, C., Morawska, B.: Automatic decidability. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, pp. 7–16. IEEE (2002)
27. Lynch, C., Ranise, S., Ringeissen, C., Tran, D.K.: Automatic decidability and combinability. Inf. Comput. **209**(7), 1026–1047 (2011)
28. McPeak, S., Necula, G.C.: Data structure specifications via local equality axioms. In: Etessami, K., Rajamani,S.K. (eds.) Computer Aided Verification, CAV 2005, Lecture Notes in Computer Science, vol. 3576, pp. 476–490. Springer, Berlin (2005)
29. Moskal, M.: Programming with triggers. In: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories, ACM ICPS, pp. 20–29. ACM, New York (2009)
30. Nelson, G.: Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center (1981)
31. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: from an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM **53**(6), 937–977 (2006)
32. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using SMT. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification, CAV 2013, Lecture Notes in Computer Science, Vol. 8044, pp. 773–789. Springer, Berlin (2013)
33. Rümmer, P.: E-matching with free variables. In: Voronkov, A., Bjørner, N. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning: 18th International Conference, LPAR-18, LNCS, vol. 7180, pp. 359–374. Springer, Berlin (2012)
34. Suter, P., Steiger, R., Kuncak, V.: Sets with cardinality constraints in satisfiability modulo theories. In: Jhala, R., Schmidt, D. (eds.) Verification, Model Checking, and Abstract Interpretation. Lecture Notes in Computer Science, vol. 6538, pp. 403–418. Springer, Berlin (2011)