HAL
open science

# Adding Decision Procedures to SMT Solvers using Axioms with Triggers

Claire Dross, Sylvain Conchon, Johannes Kanig, Andrei Paskevich

**To cite this version:**

Claire Dross, Sylvain Conchon, Johannes Kanig, Andrei Paskevich. Adding Decision Procedures to SMT Solvers using Axioms with Triggers. 2013. hal-00915931

**HAL Id: hal-00915931**

**https://inria.hal.science/hal-00915931v1**

Preprint submitted on 9 Dec 2013

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Adding Decision Procedures to SMT Solvers using Axioms with Triggers

Claire Dross Sylvain Conchon Johannes Kanig Andrei Paskevich

December 9, 2013

**Abstract**

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 believe that this effort is not greater than that of adding a single decision procedure to the same SMT solver. 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.

## 1 Introduction

It is often the case that SAT problems refer to elements to which a special meaning is associated, such as linear arithmetic, arrays, bit-vectors, etc. 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 [6], CVC3 [8], CVC4 [1], Z3 [18], and Alt-Ergo [4] 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 [19], which we will consider in greater detail below. More recently, the same work has been done for specification of more complex data-structures [14, 5].

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 [20], 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. Neither do we 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.

**Related Work.** Proving 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 Lynch et al. for a paramodulation-based procedure [12, 13]. The authors introduce *Schematic Saturation* as a means to over-approximate the inferences that paramodulation can generate while solving the satisfiability problem for a certain theory.

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 [11]. In the same way, Ge and de Moura [9] describe fragments of first-order logic that can be decided modulo theory by saturation. Both of these works define a restricted class of universally quantified formulas that can be finitely instantiated. We do not impose such restrictions *a priori* but rather require dedicated proofs of completeness and termination.

As for triggers, they are a commonly used heuristic in SMT solvers that handle quantifiers. 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 agreed that user guidance is useful in this domain [15]. 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 [6, 19], Z3 [16], and CVC3 [8]. Other heuristics for generating instances include model-based quantifier instantiation [9] and saturation processes close to the superposition calculus [17].

Triggers can also be used in semi-complete first-order theorem provers to guide the proof search and improve the prover's efficiency. This is done, for example, in the Princess [21] prover that combines a complete calculus for first-order logic with a decision procedure for linear arithmetic.

**Overview.** We start the technical development in Section 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 Section 3, we extend the well-known Abstract DPLL Modulo Theory framework [20] 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 Section 4, 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, which means that any solver implementing the extension of DPLL presented in the previous section is complete and terminating for that theory.

Finally, we present in Section 5 an implementation of our framework inside the first-order SMT solver Alt-Ergo. We use it to show that, for our example theory of doubly-linked lists, not only we obtain completeness and termination, but the overall performance of the prover for typical proof obligations has been improved.

## 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 [2] 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 can be stated as follows:

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

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 [10].

In this section, we extend first-order logic with constructions for triggers. We define what it means for a formula with triggers to be true in a 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 Syntax and semantics

We work in classical untyped first-order logic and assume the standard notation for first-order formulas and terms. We denote formulas with letters $\varphi$ and $\psi$, literals with $l$, terms with $s$ and $t$, and substitutions with $\sigma$ and $\mu$. Other notational conventions will be introduced in the course of the text.

We introduce two new kinds of formulas. A formula $\varphi$ under a trigger $l$ is written $[l]\varphi$. It can be read *if the literal $l$ is true and all its sub-terms are known then assume $\varphi$*. A dual construct for $[l]\varphi$, which we call a *witness*, is written $\langle l\rangle\varphi$. It can be read *assume that the literal $l$ is true and all its sub-terms are known and assume $\varphi$*. Notice that neither triggers nor witnesses are required to be tied to a quantifier.

To simplify the presentation, we work with formulas in negative normal form (NNF). Thus the extended syntax of formulas, literals, and atoms can be summarized as follows:

$$\begin{aligned}
\varphi &::= l \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \forall x.\varphi \mid \exists x.\varphi \mid [l]\varphi \mid \langle l\rangle\varphi \\
l &::= A \mid \neg A \\
A &::= \top \mid t_1 \approx t_2 \mid \ldots
\end{aligned}$$

The dots in the definition of atoms stand for other forms of predicates specific to background theories, e.g. comparison for linear arithmetic. Negation of non-atomic formulas is defined in a usual way. On the new constructs, $\neg\langle l\rangle\varphi$ is $[l]\neg\varphi$ and $\neg[l]\varphi$ is $\langle l\rangle\neg\varphi$.

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 $\langle l_1\rangle\ldots\langle l_n\rangle\varphi$.

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 the symbol $\mathscr{T}$ to denote the set of all terms that occur in a term, a literal, or a set of terms or literals.

*Example* 2.1. Here is an axiomatization for the theory of non-extensional arrays as defined by Greg Nelson [19]. 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 need only be instantiated with three terms $a$, $i$, and $e$ if the term $set(a,i,e)$ appears in the problem. For the second axiom, there are two different cases where it should be instantiated: if $get(set(a,i,e),j)$ appears in the problem or if both $set(a,i,e)$ and $get(a,j)$ appear in the problem. 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. This evaluation is made modulo some background theory $T$, which we assume to be fixed for the rest of this paper. In the simplest case, $T$ can be the theory of equality and non-interpreted function symbols (EUF). We write $\vDash_T$ for the usual implication in $T$ and we call a closed first-order formula (without triggers and witnesses) *$T$-satisfiable* if it has a model in $T$. 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.

**Definition 2.1** (World)**.** 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. A world $L$ is *complete* if for any ground literal $l$ in the signature of $T$, either $l \in L$ or $\neg l \in L$.

The key intuition about worlds 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 modulo $T$, that is to say there is $t' \in \mathscr{T}(L)$ such that $L \vDash_T t \approx t'$. 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. To say that a term $t$ is known in $L$ modulo $T$, we write $L \cup \bigwedge_{s \in \mathscr{T}(L)} known(s) \vDash_T known(t)$. We abbreviate the conjunction $\bigwedge_{t \in S} known(t)$ as $known(S)$, where $S$ is any set of ground terms.

**Definition 2.2** (Truth value)**.** Given a world $L$ and a closed formula $\varphi$, we define what it means for $\varphi$ to be *true* in $L$, written $L \rhd_T \varphi$, as follows:

| | |
|---|---|
| $L \rhd_T l$ | $L \vDash_T l$ and $L \cup known(\mathscr{T}(L)) \vDash_T known(\mathscr{T}(l))$ |
| $L \rhd_T \varphi_1 \vee \varphi_2$ | $L \rhd_T \varphi_1$ 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 $\varphi$ is *false* in $L$ whenever $L \rhd_T \neg\varphi$. We call $\varphi$ *feasible* if there exists a world in which $\varphi$ 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, is it is impossible for a formula to be both true and

false in the same world. In other words, there is no closed formula $\varphi$ 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 $\varphi$.

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

Let us establish the relation between the traditional first-order logic and our extension. Let $\varphi$ be a closed formula and $\varphi'$ be $\varphi$ 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 $L \rhd_T \varphi'$. It is also easy to see that $L \rhd_T \varphi'$ if and only if $L \vDash_T \varphi'$. 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 $\varphi$ if and only if it is a Herbrand model of $\varphi'$ in $T$. Consequently, $\varphi$ is satisfiable in the sense of Definition 2.3 if and only if $\varphi'$ is $T$-satisfiable, which justifies our reuse of the term.

For ground literals or conjunctions thereof the properties of feasibility, satisfiability, and $T$-satisfiability are all 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 2.1.** *Let $\varphi$ be a closed first-order formula without triggers and witnesses. Let $L$ be an inhabited world such that $L \rhd_{\mathrm{EUF}} \varphi$. Then $\varphi$ is satisfiable in first-order logic with equality (and therefore has a model in the sense of Definition 2.3).*

*Proof.* We define an encoding $[\![\ ]\!]$ that explicitly restricts instantiation of first-order formulas in negative normal form to known terms. It uses the predicate symbol *known* to represent the set of known terms:

$$
\begin{aligned}
[\![\varphi_1 \wedge \varphi_2]\!] &\triangleq [\![\varphi_1]\!] \wedge [\![\varphi_2]\!] & [\![\varphi_1 \vee \varphi_2]\!] &\triangleq [\![\varphi_1]\!] \vee [\![\varphi_2]\!] \\
[\![\forall x.\varphi]\!] &\triangleq \forall x.\mathit{known}(x) \rightarrow [\![\varphi]\!] & [\![\exists x.\varphi]\!] &\triangleq \exists x.\mathit{known}(x) \wedge [\![\varphi]\!] \\
[\![l]\!] &\triangleq \mathit{known}(\mathscr{T}(l)) \wedge l
\end{aligned}
$$

**Lemma 2.1.** *Let $\varphi$ be a closed first-order formula without triggers and witnesses. If $\varphi$ is feasible, then $[\![\varphi]\!]$ satisfiable.*

*Proof.* There is a satisfiable set of ground literals $L$ such that $L \rhd \varphi$. Let us consider the set of literals $L' = L \cup \mathit{known}(\mathscr{T}(L)) \cup \{\neg \mathit{known}(t) \mid L \cup \mathit{known}(\mathscr{T}(L)) \not\models \mathit{known}(t)\}$. Since the predicate symbol *known* is fresh and we work in EUF, $L'$ is satisfiable. We show by structural induction that, for every formula $\varphi$ such that $L \rhd \varphi$, we have $L' \models [\![\varphi]\!]$.

- $l$: We have $L \models l$ and, by construction of $L'$, $L' \models l$.
- $\varphi_1 \vee \varphi_2$: Either $L \rhd \varphi_1$ or $L \rhd \varphi_2$. By induction hypothesis, $L' \models [\![\varphi_1]\!]$ or $L' \models [\![\varphi_2]\!]$. Thus, $L' \models [\![\varphi_1]\!] \vee [\![\varphi_2]\!] = [\![\varphi_1 \vee \varphi_2]\!]$.
- $\varphi_1 \wedge \varphi_2$: Both $L \rhd \varphi_1$ and $L \rhd \varphi_2$. By induction hypothesis, $L' \models [\![\varphi_1]\!]$ and $L' \models [\![\varphi_2]\!]$. Thus, $L' \models [\![\varphi_1]\!] \wedge [\![\varphi_2]\!] = [\![\varphi_1 \wedge \varphi_2]\!]$.
- $\forall x.\varphi$: Let $t$ be a term. If $L \cup \mathit{known}(\mathscr{T}(L)) \models \mathit{known}(t)$ then there is $t' \in \mathscr{T}(L)$ such that $L \models t \approx t'$. By definition of $\rhd$, we have $L \rhd \varphi[x \leftarrow t']$ and hence $L' \models [\![\varphi[x \leftarrow t']]\!]$ by induction hypothesis. Therefore $L' \models \forall x.\mathit{known}(x) \rightarrow [\![\varphi]\!]$.
- $\exists x.\varphi$: There is a term $t$ in $\mathscr{T}(L)$ such that $L \rhd \varphi[x \leftarrow t]$. By construction of $L'$, $L' \models \mathit{known}(t)$. Furthermore, by induction hypothesis, $L' \models [\![\varphi[x \leftarrow t]]\!]$. As a consequence, $L' \models \exists x.\mathit{known}(x) \wedge [\![\varphi]\!]$.

Thanks to this encoding, we can reformulate our theorem:

**Lemma 2.2.** *Let $\varphi$ be a closed first-order formula without triggers and witnesses. Let $L$ be a Herbrand model of $[\![\varphi]\!]$ such that at least for one ground term $\omega$, $L \models \mathit{known}(\omega)$. Then $\varphi$ is satisfiable.*

*Proof.* We can assume that for every non-constant term $t$, $L$ contains an equality $t \approx c$, where $c$ is a constant (such equalities can always be added to a model if needed). We can also assume that $\omega$ is a constant.

We define a set of literals $L_1 = \{l \mid \mathit{known}$ does not occur in $l$ and $L \models [\![l]\!]\}$ and another $L_2 = L_1 \cup \{t \approx \omega \mid L_1 \cup \mathit{known}(\mathscr{T}(L_1)) \not\models \mathit{known}(t)$ and, for every proper subterm $t'$ of $t$, $L_1 \cup \mathit{known}(\mathscr{T}(L_1)) \models \mathit{known}(t')\}$. We show that:

(i) $L_2$ is satisfiable,

(ii) for every ground term $t$, there is $t' \in \mathscr{T}(L_1)$ such that $L_2 \models t \approx t'$, and

(iii) $L_2 \models \varphi$.

**Proof of (i):** Since $L$ is satisfiable, so is $L_1$. In EUF, for every set of literals $L$ and every pair of terms $t_1$ and $t_2$, if $L \models t_1 \not\approx t_2$ then $L \cup \mathit{known}(\mathscr{T}(L)) \models \mathit{known}(t_1) \wedge \mathit{known}(t_2)$. Thus, adding an equality between a known term and an unknown term to a set of literals cannot lead to inconsistency. Let $t_1$ and $t_2$ be two terms such that $L_1 \not\models t_1 \approx t_2$, $L_1 \cup \mathit{known}(\mathscr{T}(L_1)) \not\models \mathit{known}(t_i)$, and, for every proper subterm $t'$ of $t_1$ or $t_2$, $L_1 \cup \mathit{known}(\mathscr{T}(L_1)) \models \mathit{known}(t')$. Since $t_1$ cannot be equal modulo $L_1$ to a subterm of $t_2$, we have $L_1 \cup \{t_1 \approx \omega\} \cup \mathit{known}(\mathscr{T}(L_1)) \cup \mathit{known}(t_1) \not\models$

$known(t_2)$. Thus, no matter the order in which the equalities are added to $L_1$, they always involve a previously unknown term. As a consequence, $L_2$ is satisfiable.

**Proof of (ii):** We show that, for every ground term $t$, there is $t' \in \mathscr{T}(L_1)$ such that $L_2 \vDash t \approx t'$ by structural induction over $t$. We write $t$ as $f(t_1 \dots t_n)$ where $n$ can be zero for constants. By induction hypothesis, for every $i$ in $1..n$, there is $t'_i \in \mathscr{T}(L_1)$ such that $L_2 \vDash t_i \approx t'_i$. If $L_1 \cup known(\mathscr{T}(L_1)) \vDash known(t)$, $L_2 \cup known(\mathscr{T}(L_2)) \vDash known(t)$. Otherwise, consider $f(t'_1 \dots t'_n)$. By construction, $L_2 \vDash f(t'_1 \dots t'_n) \approx t$. If there is $t' \in \mathscr{T}(L_1)$ such that $L_1 \vDash f(t'_1 \dots t'_n) \approx t'$, then the proof is over. Otherwise, $L_1 \cup known(\mathscr{T}(L_1)) \nvDash known(f(t'_1 \dots t'_n))$ and, for every $i$ in $1..n$, $L_1 \cup known(\mathscr{T}(L_1)) \vDash known(t'_i)$. By construction of $L_2$, $f(t'_1 \dots t'_n) \approx \omega \in L_2$. Since $L_2 \vDash f(t'_1 \dots t'_n) \approx t$, we have $L_2 \vDash t \approx \omega$ which concludes the proof.

**Proof of (iii):** We show that, for every ground formula $\psi$ without triggers and witnesses such that $L \vDash [\![\psi]\!]$, $L_2 \vDash \psi$ by structural induction over $\psi$.

- $L \vDash [\![l]\!]$. By definition of $L_1$, $l \in L_1$ and $L_2 \vDash l$.
- $L \vDash [\![\psi_1 \wedge \psi_2]\!] = [\![\psi_1]\!] \wedge [\![\psi_2]\!]$. By induction hypothesis, $L_2 \vDash \psi_1 \wedge \psi_2$.
- $L \vDash [\![\psi_1 \vee \psi_2]\!] = [\![\psi_1]\!] \vee [\![\psi_2]\!]$. Since $L$ is a model, $L \vDash [\![\psi_1]\!]$ or $L \vDash [\![\psi_2]\!]$. By induction hypothesis, $L_2 \vDash \psi_1$ or $L_2 \vDash \psi_2$. Thus $L_2 \vDash \psi_1 \vee \psi_2$.
- $L \vDash [\![\forall x.\psi]\!] = \forall x.known(x) \rightarrow [\![\psi]\!]$. Let $t$ be a ground term. By (ii), there is $t' \in \mathscr{T}(L_1)$ such that $L_2 \vDash t \approx t'$. By construction of $L_1$, $L \vDash known(\mathscr{T}(t'))$. We then have $L \vDash [\![\psi]\!][x \leftarrow t']$ and, by immediate induction over $\psi$, $L \vDash [\![\psi[x \leftarrow t']]\!]$. By induction hypothesis, we have $L_2 \vDash \psi[x \leftarrow t']$. Therefore, $L_2 \vDash \psi[x \leftarrow t]$ and $L_2 \vDash \forall x.\psi$.
- $L \vDash [\![\exists x.\psi]\!] = \exists x.known(x) \wedge [\![\psi]\!]$. Since every ground term is equal to a constant in $L$, there is a constant $c \in \mathscr{T}(L)$ such that $L \vDash known(c)$ and $L \vDash [\![\psi]\!][x \leftarrow c]$. By immediate induction over $\psi$, $L \vDash [\![\psi[x \leftarrow c]]\!]$. By induction hypothesis, $L_2 \vDash \psi[x \leftarrow c]$ and $L_2 \vDash \exists x.\psi$.

## 2.2 Soundness, completeness, and termination

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

**Definition 2.5** (Completeness)**.** 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.

Quite often, $T'$ is the theory defined by the same set of axioms $W$ where all triggers and witnesses are erased. More precisely, 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, to prove soundness, we must show that the added witnesses do not allow us to deduce statements beyond the initial set of first-order axioms. 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* 2.2. The proof that the set of axioms $W_{array}$ shown in Example 2.1 is complete modulo EUF closely resembles the proof by Greg Nelson in [19]. 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 with $get(set(a,i,e),i)$. Consider the set of ground 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 $get(set(a,i,e_1),i) \approx e_1$ and $get(set(a,i,e_2),i) \approx e_2$. Still, $W^1_{array} \cup L_1$ is true in $L_1$, 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 $get(set(a,i_2,e),j) \approx get(a,j)$. Still, $W^2_{array} \cup L_2$ is true in $L_2 \cup \{get(set(a,i_1,e),i_1) \approx e,\, get(set(a,i_2,e),i_2) \approx e\}$.

- 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 $get(set(a_2,i,e),j) \approx get(a_2,j)$. Yet $W^3_{array} \cup L_3$ is true in $L_3 \cup \{get(set(a_1,i,e),i) \approx e,\, get(set(a_2,i,e),i) \approx e\}$.

Once it has been established that a given set of axioms with triggers is sound and complete for our theory, we must show the solver equipped with this axiomatization terminates on any ground satisfiability problem. We call such axiomatizations *terminating* and the rest 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 Section 3. In Section 4, we prove that a non-trivial axiomatization of imperative doubly-linked lists is terminating according to this definition. Finally, in Section 5.2, 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.

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

$$\textsc{Sko}(\exists x.\varphi) \triangleq \langle c(\bar{y})\rangle\, \textsc{Sko}(\varphi[x \leftarrow c(\bar{y})]),$$

where $\bar{y}$ is the set of free variables of $\exists x.\varphi$ and $c$ is a fresh function symbol. Skolemization preserves feasibility and satisfiability, as can be proved by straightforward induction over $\varphi$. We construct a world for $\textsc{Sko}(\varphi)$ by giving the Skolem terms the same interpretation as for the corresponding ground terms in the original world for $\varphi$. In the opposite sense, if $\textsc{Sko}(\varphi)$ is feasible, then $\varphi$ is true in the same world. The use of the witness is crucial here. Indeed, $\textsc{Sko}(\exists x.[x]\bot)$ is $\langle c\rangle[c]\bot$ which preserves infeasibility, whereas the formula $[c]\bot$ is true in any world where $c$ is unknown.

Skolemization may not preserve the soundness and completeness of a set of axioms. For example, if $T'$ is the theory $\exists x.P(x)$, then the skolemized axiom $\langle c\rangle P(c)$ is not a sound representation of $T'$. Indeed, 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 Section 3 do not require skolemized axiomatizations.

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. It is easy to check that every set of skolemized formulas can be transformed to an equivalent set of pseudo-clauses. In particular, $[l](\varphi_1 \wedge \varphi_2)$ (respectively, $\langle l\rangle(\varphi_1 \wedge \varphi_2)$) is equivalent to $[l]\varphi_1 \wedge [l]\varphi_2$ (respectively, $\langle l\rangle\varphi_1 \wedge \langle l\rangle\varphi_2$).

Before we pass to 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 we can assume. 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 $\varphi$ and a substitution $\sigma$ mapping every free variable of $\varphi$ to a ground term. We write $\varphi\sigma$ for the application of $\sigma$ to $\varphi$, and $\varnothing$ for the empty substitution. If two substitutions $\sigma$ and $\sigma'$ have the same domain $D$, we write $\sigma \mathrel{\dot{\approx}} \sigma'$ 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 $\sigma'$ is $\sigma$ restricted to the free variables of $\varphi\}$. Such disjunctive sets of closures are called *theory clauses*, as they come from the axiomatization of our theory of interest.

Given a set of theory clauses $V$, we define $\lfloor V \rfloor \triangleq \{l\sigma \mid l \cdot \sigma$ is a unit clause in $V\}$. Informally speaking, this operation provides us with the facts that are readily available, without us needing to eliminate triggers or witnesses, to instantiate a variable, or to decide which part of a disjunction to assume.

**Definition 2.6** (Truth assignment)**.** A *truth assignment* of a set of theory clauses $V$ is any set $A$ 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 any 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 $\sigma'$ such that $C \cdot \sigma' \in A$ and $\lfloor A \rfloor \vDash_T (\sigma \cup [x \mapsto t]) \mathrel{\dot{\approx}} \sigma'$. In what follows, we write $\mathscr{T}(A)$ for $\mathscr{T}(\lfloor A \rfloor)$ and $A \vDash_T l$ for $\lfloor A \rfloor \vDash_T l$.

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

Notice 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.

**Definition 2.7** (Instantiation tree)**.** 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 2.8** (Termination)**.** A set of pseudo-clauses $W$ is *terminating* if, for every finite set of ground literals $L$, $W \cup L$ admits a finite instantiation tree.

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* 2.3. The proof of termination of the theory of arrays described in Examples 2.1 and 2.2 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 $\lfloor A \rfloor \rhd_T set(a,i,e)$. Then, for every term $t$ in $\mathscr{T}(get(set(a,i,e),i) \approx e)$, $\lfloor A \rfloor \cup \{get(set(a,i,e),i) \approx e\} \cup known(\mathscr{T}(\lfloor A \rfloor)) \vDash_T known(t)$. Indeed, since $\lfloor A \rfloor \rhd_T set(a,i,e)$, it 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* 2.4. Let us look at a more interesting proof of termination. 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 $e$.

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

We show that axiomatization $W_{conv}$ is terminating. 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 two first axioms with a term of $L$ if there is one that is not redundant in $A$. If there are no more of them, add an instance of one of the two last axioms of $W_{conv}$ 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 term is $conv_{E\to e}(conv_{e\to E}(t))$ which is equal to $t$. 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 the first two axioms in $A$ can only produce new terms of the form $unfit_{E\to e}(t)$ with $t$ already in $\mathscr{T}(A)$. For example, assume that there is a non-redundant instance of the first axiom with a term $t$ such that $A \vDash_T valid_E(t) \approx \mathtt{t}$. 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 $t' \in \mathscr{T}(A)$

such that $A \vDash_T valid_e(t') \approx \mathtt{t}$ and $A \vDash_T t \approx conv_{e\to E}(t')$. Therefore, the last axiom of $W_{conv}$ has been instantiated with $t'$ with the second step and $A \vDash_T conv_{E\to e}(conv_{e\to E}(t')) \approx t'$ and thus $A \vDash_T valid_e(conv_{E\to e}(t))$. Consequently, the result of an instance of the first axiom with $t$ can only produce new terms of the form $unfit_{E\to e}(t)$ with $t$ already in $\mathscr{T}(A)$. Since such terms cannot trigger new instances, 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.

# 3 Extension of DPLL($T$) to the Logic with Triggers

In this section, we introduce an extension of abstract DPLL modulo theories [20] that handles formulas with triggers and witnesses. We show that if a set of axioms is sound and complete with respect to a theory $T'$ which 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.

## 3.1 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 Section 2.2. 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 $T'$, 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.

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 $T$, 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 Section 2.1.

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 closures and anti-closures as follows:

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

Non-literal closures $\varphi \cdot \sigma$, where $\varphi$ 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 $\sigma$. 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. We also use the set of guards of $F$, defined as $\textsc{Grd}(F) \triangleq \{H \mid H \to C \in F\}$.

We now extend Definitions 2.2 and 2.3 onto super-literals and guarded clauses.

**Definition 3.1** (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* is $L$, 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 2.2 and 2.3, even though they have different meanings for ordinary literals; in this section, we follow Definition 3.1.

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

**Definition 3.2.** We define an encoding $[\![\ ]\!]$ of super-literals and guarded clauses into literals and clauses. In the rules below, $P_\varphi$ is a fresh predicate symbol that we associate to every pseudo-literal $\varphi$. The arity of $P_\varphi$ is the number of free variables in $\varphi$.

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

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 implication $\vDash_T$ onto super-literals and guarded clauses.

**Lemma 3.1.** *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 $L' = L \cup \{[\![e]\!] \mid e \text{ is a super-literal such that } L \blacktriangleright_T e\}$. The set $L'$ is satisfiable and complete. Indeed, for every closure $\varphi \cdot \sigma$ and every substitution $\sigma'$ such that $L \vDash_T \sigma \dot{\approx} \sigma'$, $L \blacktriangleright_T \varphi \cdot \sigma$ 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.

We also need a weaker version of implication that preserves feasibility as well as satisfiability. We want this implication to be as close as possible to the usual implication $\vDash_T$ on user clauses while remaining computable by a working solver on theory clauses:

**Definition 3.3.** Let $F$ be a set of super-clauses and $C$ a super-clause. We write $F \vdash_T^\star C$ if and only if one of the following conditions holds:

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

Remark that $\vdash_T^\star$ does not coincide with implication modulo $T$ on unit user clauses. Indeed, $\vdash_T^\star$ 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 implies 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 $F \setminus \{f(c) \approx c\} \cdot \varnothing \cup G \vdash_T^\star f(c) \approx c \cdot \varnothing$, 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 $F \setminus \{f(c) \approx c\}$. 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 3.3, known terms are only provided by the closures (that is, 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.

**Lemma 3.2.** *Let $C$ be a super-clause and $F$ be a set of super-clauses such that $F \vdash_T^\star C$. 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 $\textsc{Lit}(F) \cup \lfloor \textsc{Clo}(F) \rfloor \vDash_T C$. Since $L \blacktriangleright F$, $L \vDash_T \textsc{Lit}(F)$ and $L \vDash_T \lfloor \textsc{Clo}(F) \rfloor$. 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 $F \vdash_T^\star l\sigma$ and $F \cup \mathit{known}(\mathscr{T}(\textsc{Clo}(F))) \vdash_T^\star \mathit{known}(\mathscr{T}(l\sigma))$. Since $L \blacktriangleright F$, $L \vDash_T \textsc{Lit}(F) \cup \lfloor \textsc{Clo}(F) \rfloor$ and $L \cup \mathit{known}(\mathscr{T}(L)) \vDash_T \mathit{known}(\mathscr{T}(\textsc{Clo}(F)))$. As a consequence, $L \blacktriangleright l \cdot \sigma|_{\mathit{vars}(l)}$ and $L \blacktriangleright C$.

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

**Lemma 3.3.** *Let $C$ be a super-clause and $F$ be a set of super-clauses such that $F \vdash_T^\star C$. We have $F \vDash_T^\star C$.*

*Proof.* Let $L$ be a model of $[\![F]\!]$. We have four cases to consider. Assume that $C$ is a unit user clause and $\textsc{Lit}(F) \cup \lfloor \textsc{Clo}(F) \rfloor \vDash_T C$. Since $L \vDash_T [\![F]\!]$, $L \vDash_T \textsc{Lit}(F)$ and $L \vDash_T \lfloor \textsc{Clo}(F) \rfloor$. 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 $F \vdash_T^\star l\sigma$ and $F \cup \mathit{known}(\mathscr{T}(\textsc{Clo}(F))) \vdash_T^\star \mathit{known}(\mathscr{T}(l\sigma))$. Since $L \vDash_T [\![F]\!]$, $L \vDash_T \textsc{Lit}(F) \cup \lfloor \textsc{Clo}(F) \rfloor$. 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 $C' \subseteq D$, $F \vdash_T^\star \sigma|_{\mathit{Dom}(\sigma')} \dot{\approx} \sigma'$, and $F \cup \mathit{known}(\mathscr{T}(\textsc{Clo}(F))) \vdash_T^\star \mathit{known}(\mathscr{T}(\sigma|_{\mathit{Dom}(\sigma')}))$. Since $L \vDash_T [\![F]\!]$, $L \vDash_T \sigma|_{\mathit{Dom}(\sigma')} \dot{\approx} \sigma'$ 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 3.4.** *Let $C$ be a super-clause and $F_1$ and $F_2$ be two sets of super-clauses. If $F_1 \vdash_T^\star F_2$ and $F_2 \vdash_T^\star C$, then $F_1 \vdash_T^\star C$.*

*Proof.* Assume that $C$ is a unit user clause $l$ and $\textsc{Lit}(F_2) \cup \lfloor \textsc{Clo}(F_2) \rfloor \vDash_T C$. Since $F_1 \vdash_T^\star F_2$, we have $F_1 \vdash_T^\star \textsc{Lit}(F_2)$ and $F_1 \vdash_T^\star \lfloor \textsc{Clo}(F_2) \rfloor$. By definition of $\vdash_T^\star$ on user clauses, $\textsc{Lit}(F_1) \cup \lfloor \textsc{Clo}(F_1) \rfloor \vDash_T \textsc{Lit}(F_2) \cup \lfloor \textsc{Clo}(F_2) \rfloor$. Thus, $F_1 \vdash_T^\star C$. The case where $C$ is a non-unit user clause is handled in the same way, except that instead of $\textsc{Lit}(F_1)$ and $\textsc{Lit}(F_2)$ 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 $F_2 \vdash_T^\star l\sigma$ and $F_2 \cup \mathit{known}(\mathscr{T}(\textsc{Clo}(F_2))) \vdash_T^\star \mathit{known}(\mathscr{T}(l\sigma))$. Like in the previous case, $F_1 \vdash_T^\star l\sigma$. Since $F_1 \vdash_T^\star F_2$, $F_1 \cup \mathit{known}(\mathscr{T}(\textsc{Clo}(F_1))) \vdash_T^\star \mathit{known}(\mathscr{T}(\varphi \cdot \sigma))$ for every closure $\varphi \cdot \sigma \in \textsc{Clo}(F_2)$. As a consequence, $\textsc{Lit}(F_1) \cup \lfloor \textsc{Clo}(F_1) \rfloor \cup \mathit{known}(\mathscr{T}(\textsc{Clo}(F_1))) \vDash_T \textsc{Lit}(F_2) \cup \lfloor \textsc{Clo}(F_2) \rfloor \cup \mathit{known}(\mathscr{T}(\textsc{Clo}(F_2)))$ and $F_1 \vdash_T^\star C$.

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

Assume that there is a literal $l \in C'$ such that $F_1 \vdash_T^\star l\sigma'$ and $F_1 \cup known(\mathscr{T}(\textsc{Clo}(F_1))) \vdash_T^\star known(\mathscr{T}(l\sigma'))$. Since $F_1 \vdash_T^\star \sigma|_{Dom(\sigma')} \dot{\approx} \sigma'$, $F_1 \vdash_T^\star l\sigma$. With $F_1 \cup known(\mathscr{T}(\textsc{Clo}(F_1))) \vdash_T^\star known(\mathscr{T}(\sigma|_{Dom(\sigma')}))$, we deduce $F_1 \cup known(\mathscr{T}(\textsc{Clo}(F_1))) \vdash_T^\star known(\mathscr{T}(l\sigma))$. Therefore, $F_1 \vdash_T^\star C$.

Otherwise, there is a theory clause $C'' \cdot \sigma'' \in F_1$ such that $C'' \subseteq C'$, $F_1 \vdash_T^\star \sigma'|_{Dom(\sigma'')} \dot{\approx} \sigma''$, and $F_1 \cup known(\mathscr{T}(\textsc{Clo}(F_1))) \vdash_T^\star known(\mathscr{T}(\sigma'|_{Dom(\sigma'')}))$. Since $F_1 \vdash_T^\star \sigma|_{Dom(\sigma')} \approx \sigma'$, we have $F_1 \vdash_T^\star \sigma|_{Dom(\sigma'')} \dot{\approx} \sigma''$. Hence, $F_1 \vdash_T^\star C$.

Given a set of super-literals $M$, we write $M \vdash_T^\star C$ as an abbreviation for $\textsc{Lit}(M) \cup \textsc{Clo}(M) \vdash_T^\star C$. 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, $M \vdash_T^\star \bot$ whenever the set $\textsc{Lit}(M) \cup \lfloor \textsc{Clo}(M) \rfloor$ 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 $\vdash_T^\star$ 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 $M \cup \{l \cdot \varnothing \mid l \in \textsc{Lit}(M)\}$. Thus, for every term $t \in \mathscr{T}(M)$, $t \in \mathscr{T}(\textsc{Clo}(\lceil M \rceil))$.

**Lemma 3.5.** *Let $M$ be a set of super-literals and $e$ a super-literal. If $\lceil M \rceil \vdash_T^\star e$ then $M \vDash_T^\star e$.*

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

## 3.2 Description of DPLL($T$)

The method introduced below adapts the principles of abstract DPLL modulo theories (following [20]) to super-literals and guarded clauses. The rules are given in Figures 1 and 2. They attempt to construct a model of a set of guarded clauses $F$. The partial model is represented as a set of super-literals $M$ that are assumed to be true. We call *state* of the procedure the pair $M \parallel F$ and we say that a super-literal $e$ is *defined* in $M$ if either $e$ or $\neg e$ is in $M$.

The elements of an available clause can be given an arbitrary truth value using the rule `Decide`. Super-literals of $M$ whose truth value was chosen arbitrarily are labeled with a letter `d` and called decision super-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 using `UnitPropagate`. If every element of an available clause is false then the corresponding guarded clause 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 could be found for $F$. The rule `Restart` can be used to restart the search from scratch. If there is a super-literal $e$ that appears in available clauses or guards of $F$ whose negation leads to a contradiction in $M$, it can be propagated using $T$`-Propagate`.

The set of guarded clauses $F$ can be modified during the search using $T$`-Learn` and $T$`-Forget`. 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 $H \to C$ if $F, H \vDash_T^\star C$, and thus every model of $[\![F]\!]$ is also a model of $[\![H \to C]\!]$. However, we are more restrictive with respect to what clauses can be forgotten. Namely, we require that for a guarded clause $H \to C$ to be forgotten, $\textsc{Avb}(F, H) \vdash_T^\star C$. We show below that this distinction is necessary for termination.

`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`:

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

$T$-`Learn`:

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

$T$-`Forget`:

$$M \parallel F, H \to C \implies M \parallel F \quad \textbf{if} \begin{cases} \text{each closure of } C \text{ defined in } M \text{ occurs in } \textsc{Avb}(F, H) \\ \textsc{Avb}(F, H) \vdash^{\star}_{T} C \end{cases}$$

$T$-`Backjump`:

$$Me^{\mathrm{d}}N \parallel F \implies Me' \parallel F \quad \textbf{if} \begin{cases} \text{there is } H \to C \in F \text{ such that } H \wedge \neg C \subseteq Me^{\mathrm{d}}N \\ \text{there is } D \subseteq M \text{ such that:} \\ \quad F, D \vDash^{\star}_{T} e', \\ \quad e' \text{ is undefined in } M, \text{ and} \\ \quad e' \text{ or } \neg e' \text{ occurs in } \textsc{Avb}(F, M) \cup \textsc{Lit}(Me^{\mathrm{d}}N) \end{cases}$$

Figure 1: Transition rules of Abstract DPLL Modulo Theories on guarded clauses

`Instantiate`:

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

`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`:

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



Figure 2: Additional transition rules for Abstract DPLL Modulo Theories on guarded clauses

Finally, if every element of an available 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 element that can be added to $M$. An element can be added to $M$ if it is implied by $M$ and $F$.

Specific rules are needed to retrieve information from closures. They are described in Fig. 2. 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 elements of 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 $\textsc{Avb}(F,H) \vdash_T^\star C$.

We finally define when a solver implementing DPLL is allowed to deduce the satisfiability or unsatisfiability of a set of ground clauses $G$ modulo an extension of the background theory $T$ described as an axiomatization $W$:

**Property 3.1.** The solver can return *Unsat* on $G$ if $\varnothing \parallel W \cdot \varnothing \cup G \Longrightarrow^\star _ \parallel \mathit{fail}$.

**Property 3.2.** 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 $\textsc{Avb}(F,M) \vdash_T^\star C$.

*Remark* 3.1. 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 implied by unit clauses. As a consequence, the changes in abstract DPLL can be implemented as an extension outside an existing DPLL implementation.

*Remark* 3.2. The relation $\vDash_T^\star$ on guarded clauses cannot be computed inside the solver, but it is not needed to implement DPLL. Indeed, like in classical abstract DPLL($T$), conflict analysis allows to deduce enough applications of $T$-`Backjump` and $T$-`Learn` to ensure progress. This is explained below in Lemma 3.11 and Corollary 3.2.

*Remark* 3.3. In classical abstract DPLL modulo theories, conflict driven lemmas, namely formulas allowing to deduce the added element $e'$ in $M$ after an application $Me^{\mathtt{d}}N \parallel F \Longrightarrow Me' \parallel F$ of $T$-`Backjump`, can be added to $F$ using $T$-`Learn`. In our framework, this is not the case. This restriction can be removed by allowing to deduce guarded clauses $H \to C$ such that $F,H \vDash_T^\star C$ 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 $F,D \vDash_T^\star e'$ and $e'$ or $\neg e'$ occurs in $\textsc{Avb}(F,M) \cup \textsc{Lit}(Me^{\mathtt{d}}N)$, $\textsc{Clo}(M) \to \{\neg e \mid e \text{ is an anti-closure or a literal of } D\} \vee e'$ can be added to $F$ using $T$-`Learn`.

### 3.3 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 Section 2.2. They aim at forbidding:

- The addition into $M$ of a super-literal that should be protected by a trigger. It 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 implied 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 $e \in \textsc{Grd}(F)$ to be added to $M$ if $\lceil M \rceil \vdash_T^\star e$. 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 $\lceil M \rceil \vdash_T^\star l\sigma$, namely $\textsc{Lit}(M) \cup \{l'\sigma' \mid l' \cdot \sigma' \in M\} \vDash_T l\sigma$ and $\textsc{Lit}(M) \cup \{l'\sigma' \mid l' \cdot \sigma' \in M\} \cup \mathit{known}(\mathscr{T}(M)) \vDash_T \mathit{known}(\mathscr{T}(l\sigma))$. Only requesting that $M \vDash_T^\star l\sigma$ 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, $M \vDash_T^\star (f(x) \approx f(x)) \cdot [x \mapsto t]$ 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 $p(f(x)) \approx \top \cdot [x \mapsto f(t)]$, $p(f(x)) \approx \top \cdot [x \mapsto f(f(t))]$... can be added to $M$.

In the rule $T$-`Backjump`, we require that $e'$ or $\neg e'$ occurs in $\textsc{Avb}(F,M) \cup \textsc{Lit}(Me^{\mathrm{d}}N)$. 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 $G_2 \vDash_T a \approx c$, $G_2 \cup M \vDash_T^\star (\forall x.f(x,y) \approx f(x,y)) \cdot [y \mapsto c]$. Because of this closure, we can produce an infinite

number of terms $f(t,c)$, $f(f(t,c),c)\dots$ 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 implied 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 $\textsc{Avb}(F,H) \cup \textsc{Lit}(M)$. Even asking that $\textsc{Avb}(F,H) \vdash_T^\star C$ 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, $G_3 \cup \{c \approx c \cdot \varnothing,\ (\forall x.f(x,y) \approx f(x,y)) \cdot [y \mapsto a]\} \subseteq \textsc{Avb}(F, \{([p(y) \approx \top]\forall x.f(x,y) \approx f(x,y)) \cdot [y \mapsto a],\ (p(y) \approx \top) \cdot [y \mapsto a]\})$ 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 $f(t,c)$, $f(f(t,c),c)\dots$ 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 implied 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 $\textsc{Avb}(F,H)$. This is needed so that we have a progress property in spite of 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 $F \setminus \{H \to C\}$. 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 $\varphi \cdot \sigma$.

We also require that $\textsc{Avb}(F,H) \vdash_T^\star C$. 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 $W_4 \cdot \varnothing \setminus \{a \approx c \cdot \varnothing\} \cup G_4 \vDash_T^\star a \approx c \cdot \varnothing$, 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 $f(t,c)$, $f(f(t,c),c)\dots$ 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 $\textsc{Avb}(F,M) \vdash_T^\star C \cdot (\sigma \cup [x \mapsto t])$. This constraint is needed for termination so that redundant instances are forbidden.

### 3.4 Soundness and Completeness

We show that our variant of abstract DPLL modulo theories is compliant with the semantics defined in Section 2.1.

**Lemma 3.6.** *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 3.1, 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 $\blacktriangleright$, $L \cup \mathit{known}(\mathscr{T}(L)) \vDash_T \mathit{known}(\mathscr{T}(t))$ 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 3.7.** *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, $F \cup H \to C \vDash_T^\star F_1$. Since $F \vdash_T^\star C$, $F \vDash_T^\star C$ by Lemma 3.3.

**Theorem 3.1** (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.* We define $W$ to 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 implied by the input problem and previous decisions:

**Lemma 3.8.** *If $\varnothing \parallel G \cup W \cdot \varnothing \Longrightarrow^\star M_0 e_1^{\mathrm{d}} M_1 \ldots e_n^{\mathrm{d}} M_n \parallel F$, $L$ is a model of $G \cup W \cdot \varnothing$ and $L \blacktriangleright e_1, \ldots, e_i$ then $L \blacktriangleright M_i$ for every $i$ in $0 \ldots n$.*

*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 3.6, $L \blacktriangleright F$. 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 3.1. The only remaining rule is $T$-`Backjump`. There is a subset $D$ of $M$ such that $F \cup D \vDash_T^\star e$. Since $L \blacktriangleright M$ and $L \blacktriangleright F$, since $L$ is complete, $L \blacktriangleright e$ by Lemma 3.1.

Thanks to the previous lemma, we can prove the soundness of the DPLL framework modulo $T$. 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 3.8, $L \blacktriangleright M$. What is more, by Lemma 3.6, $L \blacktriangleright F, H \to C$. With $H \wedge \neg C \subseteq M$, we get a contradiction.

**Theorem 3.2** (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.* We define $W$ to 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) $M \nvdash_T^\star \bot$, and

(iii) if $H \to C$ can be added by `Instantiate`, `Witness-Unfold`, or `Trigger-Unfold` then $\textsc{Avb}(F, M) \vdash_T^\star C$.

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 3.2, $L \blacktriangleright C$.

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 3.2. For every closure $\varphi \cdot \sigma \in M$, we prove that $L \rhd \varphi\sigma$ by induction over the size of the formula $\varphi$.

- $l \cdot \sigma \in M$. 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 3.4, $M \vdash_T^\star C \cdot (\sigma \cup [x \mapsto t])$. Therefore, there is $\varphi \in C$ such that either $\varphi \cdot \sigma' \in M$ and $L \vDash_T (\sigma \cup [x \mapsto t]) \approx \sigma'$

or $\varphi$ is a literal, $M \vdash_T^\star \varphi\sigma$, and $M \cup known(\mathscr{T}(\textsc{Clo}(M))) \vdash_T^\star known(\mathscr{T}(\varphi\sigma))$. In the first case, since $\varphi$ 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 3.4, $M \vdash_T^\star l \cdot \sigma$ and $M \vdash_T^\star C \cdot \sigma$. Hence, there is $\varphi \in C$ such that either there is a substitution $\sigma'$ such that $\varphi \cdot \sigma' \in M$ and $M \vdash_T^\star \sigma \approx \sigma'$ or $\varphi$ is a literal, $M \vdash_T^\star \varphi\sigma$ and, $M \cup known(\mathscr{T}(\textsc{Clo}(M))) \vdash_T^\star known(\mathscr{T}(\varphi\sigma))$. 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 $M \cup known(\mathscr{T}(M)) \vdash_T^\star known(\mathscr{T}(l\sigma))$. 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 $\rhd$, $L \rhd ([l]C)\sigma$.

### 3.5 Progress and Termination

We have shown that our version of abstract DPLL modulo theories only allows derivations that are compliant with the semantics of Section 2. In this section, we show that, if some restrictions are applied, there cannot be infinite DPLL derivations. 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*. An instantiation level $n$ for a term $t$ indicates that $t$ is the result of $n$ rounds of instantiation. More formally, if $M$ is a set of super-literals, the instantiation level $level_M(t)$ (resp. $level_M(e)$) of a term $t$ (resp. a super-literal $e$) is either a non-negative integer or a special element $\infty$. It is defined as the limit of the sequence $level_M^i$ computed in the following manner:

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

Operations min, max and $+$ are so 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 Section 2.2, 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 3.4** (Fairness)**.** We say that a derivation is *fair* if, for every step $_ \parallel F \Longrightarrow Me \parallel F$ 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) $N \not\vdash_T^\star \bot$,

(b) for every unit super-clause $e \in \textsc{Avb}(F, \varnothing)$, we have $\lceil N \rceil \vdash_T^\star e$,

(c) for every closure $\langle l \rangle C \cdot \sigma \in N$, $\lceil N \rceil \vdash_T^\star l \cdot \sigma$ and, if $C$ is a unit clause, $\lceil N \rceil \vdash_T^\star C \cdot \sigma$,

(d) for every closure $[l]\varphi \cdot \sigma \in N$ such that $\varphi$ is a unit clause, if $\lceil N \rceil \vdash_T^\star l \cdot \sigma$ then we have $\lceil N \rceil \vdash_T^\star \varphi \cdot \sigma$,

(e) for every closure $\forall x.\varphi \cdot \sigma \in M'$ such that $\varphi$ 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 $\lceil N \rceil \vdash_T^\star \varphi \cdot (\sigma \cup [x \mapsto t])$, 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 $level_M(H) \leq level(M)$, $\textsc{Avb}(F, M) \vdash_T^\star C$.

*Remark* 3.4. Note that, in a fair derivation, the current instantiation level of every partial model $M$ is finite.

*Remark* 3.5. Dealing with instantiation levels 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.

Using this definition of fairness, we state some restrictions on derivation that enforce termination:

**Theorem 3.3** (Termination)**.** *There is no infinite derivation Der from a state $\varnothing \parallel G \cup W \cdot \varnothing$ where $W$ is terminating such that:*

- *Der has no infinite sub-derivation made only of `T-Learn`, `T-Forget`, and redundant `Witness-Unfold`, `Trigger-Unfold` and `Instantiate` steps,*
- *the derivation is fair,*
- *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 three `Restart` steps are the ones producing $S_i$, $S_j$ and $S_k$, either:*
  - *there are more DPLL steps that are neither `T-Learn` or `T-Forget` steps nor redundant applications of `Witness-Unfold`, `Trigger-Unfold` or `Instantiate` in $S_j \Longrightarrow \ldots \Longrightarrow S_k$ than in $S_i \Longrightarrow \ldots \Longrightarrow S_j$, or*

– *a guarded clause including only literals and closures with empty substitutions is learned in* $S_j \Longrightarrow \ldots \Longrightarrow S_k$ *and is not forgotten in Der.*

*Remark* 3.6. 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 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 steps that are neither $T$-`Learn` or $T$-`Forget` steps nor redundant applications of `Witness-Unfold`, `Trigger-Unfold` or `Instantiate`. Since there is no infinite sub-derivation of *Der* made only of $T$-`Learn`, $T$-`Forget`, and redundant `Witness-Unfold`, `Trigger-Unfold` and `Instantiate` steps, for every integer $n$, there is a sub-derivation of *Der* containing no `Restart` steps and more than $n$ steps that are neither $T$-`Learn` or $T$-`Forget` steps nor redundant applications of `Witness-Unfold`, `Trigger-Unfold` or `Instantiate`. With the two following properties, we reach a contradiction:

(i) Let $M_W$ be a finite set of super-literals. There is an integer $max_{step}$ such that, for every sub-derivation $Der'$ of *Der* containing no `Restart`, if, for every state $M \parallel F$ in $Der'$, $M \subseteq M_W$, then $Der'$ contains no more than $max_{step}$ steps that are neither $T$-`Learn`, or $T$-`Forget` steps nor redundant applications of `Witness-Unfold`, `Trigger-Unfold`, or `Instantiate`.

(ii) If $W$ is terminating, then there is a finite set of super-literals $M_W$ such that, at every state $M \parallel F$ in *Der*, $M \subseteq M_W$.

**Proof of (i):** Let $Der'$ be a sub-derivation of *Der* containing no `Restart` such that, for every state $M \parallel F$ in $Der'$, $M \subseteq M_W$. We first need an order on partial models $M$. Every partial model $M$ can be written $M_1 e_1^{\mathtt{d}} M_2 \ldots M_n e_n^{\mathtt{d}} M_{n+1}$ where $e_1^{\mathtt{d}} \ldots e_n^{\mathtt{d}}$ 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 produce a strictly greater partial model. The other rules do not change the partial model. Since $M_W$ is finite and a partial model cannot contain the same super-literal twice, the size of strictly increasing sequences of partial models is bounded.

Thus, we only have to consider sub-derivations that consist of $T$-`Learn`, $T$-`Forget`, `Instantiate`, `Witness-Unfold`, and `Trigger-Unfold` steps. Since $M_W$ is finite, there can only be a finite number of distinct applications of `Instantiate`, `Witness-Unfold` and `Trigger-Unfold` in the derivation. Therefore, if $\sharp\textsc{Clo}(M_W)$ is the number of closures in $M_W$, there can only be $\sharp\textsc{Clo}(M_W)$ non-redundant applications of `Instantiate`, `Witness-Unfold`, and `Trigger-Unfold` in our sub-derivation.

As a conclusion, there is an integer $max_{step}$ such that every derivation $Der'$ contains no more than $max_{step}$ steps that are neither $T$-`Learn` or $T$-`Forget` steps nor redundant applications of `Witness-Unfold`, `Trigger-Unfold` or `Instantiate`.

**Proof of (ii):** The idea of the proof is the following. During the search, the algorithm will go through the instantiation trees of $L \cup W$, where $L$ is a set of literals from $G$. Fairness will prevent it from generating too many instances before generating the one instance that will allow the tree to grow. Note that, since the derivation is fair, every element of $M$ has a finite instantiation level. Indeed, if `Instantiate` can be applied to some universal formula and some term $t$ in $M \parallel F$ then $t \in \mathscr{T}(M)$.

Let us first construct the 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 $\{\varphi \mid \varphi \in C$ 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 $Z_0 = \{l,\ l \cdot \varnothing,\ \neg(l \cdot \varnothing) \mid l$ or $\neg l$ occurs in $G\} \cup \{\varphi \cdot \varnothing,\ \neg(\varphi \cdot \varnothing) \mid \varphi$ 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* 3.7. By construction of the sequence $Z_i$, if an element $e \in M$ has an instantiation level $n$ in $M$ then $e \in Z_n$.

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 implied by the set $M$ at some point in the search. If $A$ is a set of super-clauses, we define $\textsc{Unit}(A)$ to be the set of unit super-clauses of $A$.

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 $\lceil M \rceil \vdash_T^\star \textsc{Unit}(A_1^M)$. 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 $\lceil M \rceil \vdash_T^\star \textsc{Unit}(A_{i+1}^M)$, if any. We call $d^M$ the maximal $i$ for which $A_i^M$ exists and we define $A^M$ as $A_{d^M}^M$.

For $i \in 0..d^M$, 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 $i \in 0..d^M$. We call $n^M$ the integer $\sum_{i \in 0..d^M} (n_i^M + 1) \times (n_{max} + 1)^{(d_{max} - i)}$. Note that $n^M$ models a lexicographic order on the finite sequence $n_0^M \ldots n_{d^M}^M$.

*Remark* 3.8. By definition, $n^M$ depends only on $A^M$ and, if $A^{Me}$ is different from $A^M$, then $n^{Me} > n^M$.

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 3.7, $M \subseteq Z_{m+1}$.

Let us now do the proof. We show by induction over the derivation of $M \parallel F$ that:

1. the current instantiation level in $M$ is at most $n^M + 1$, and

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 $\lceil M' \rceil \vdash_T^\star \textsc{Unit}(A^M)$.

If we remove elements from $M$, we necessarily return to some previous state of $M$ in the derivation, where the two properties hold by induction hypothesis. In an application of $T$-`Backjump` $Me^dN \parallel F \Longrightarrow Me' \parallel F$, $e' \in \textsc{Avb}(F,M) \cup \textsc{Lit}(Me^dN)$. Thus, the instantiation level of $e'$ in $M$ is smaller than the current instantiation level in $M$. As a consequence, since the current instantiation level in $M$ is at most $n^M + 1$ by induction hypothesis, the current level in $Me'$ is also at most $n^M + 1$.

For a step $M \parallel F \Longrightarrow Me \parallel F$, we show that $e$ has an instantiation level of at most $n^M + 1$ in $M$. The both properties then follow from remark 3.8. By contradiction, assume that $e$ has an instantiation level of $n^M + 2$ in $M$. Since the derivation if 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) $N \not\vdash_T^\star \bot$,

(b) for every unit super-clause $e \in \textsc{Avb}(F,\varnothing)$, we have $\lceil N \rceil \vdash_T^\star e$,

(c) for every closure $\langle l \rangle C \cdot \sigma \in N$, $\lceil N \rceil \vdash_T^\star l \cdot \sigma$ and, if $C$ is a unit clause, $\lceil N \rceil \vdash_T^\star C \cdot \sigma$,

(d) for every closure $[l]\varphi \cdot \sigma \in N$ such that $\varphi$ is a unit clause, if $\lceil N \rceil \vdash_T^\star l \cdot \sigma$ then we have $\lceil N \rceil \vdash_T^\star \varphi \cdot \sigma$,

(e) for every closure $\forall x.\varphi \cdot \sigma \in M'$ such that $\varphi$ 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 $\lceil N \rceil \vdash_T^\star \varphi \cdot (\sigma \cup [x \mapsto t])$, 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 $level_M(H) \leq level(M)$, $\textsc{Avb}(F,M) \vdash_T^\star C$.

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 $\lceil M'' \rceil \vdash_T^\star \textsc{Unit}(A^M)$. We need two intermediate lemmas:

**Lemma 3.9.** *$N$ 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 $\lceil N \rceil \vdash_T^\star l \cdot \varnothing$. Since $\textsc{Unit}(\textsc{Avb}(F,\varnothing)) \vdash_T^\star \textsc{Unit}(W \cdot \varnothing)$, by (b), if $l \in W$ then $\lceil N \rceil \vdash_T^\star l \cdot \varnothing$. Moreover, for every closure $\varphi \cdot \varnothing \in \textsc{Unit}(W \cdot \varnothing)$ such that $\varphi$ is not a literal, $\varphi \cdot \varnothing \in \textsc{Unit}(\textsc{Avb}(F,\varnothing))$. By (b), $\lceil N \rceil \vdash_T^\star \textsc{Unit}(\textsc{Avb}(F,\varnothing))$. Therefore, for every closure $\varphi \cdot \varnothing \in \textsc{Unit}(W \cdot \varnothing)$ such that $\varphi$ is not a literal, $\lceil N \rceil \vdash_T^\star \varphi \cdot \varnothing$ and $\varphi \cdot \varnothing \in N$. What is more, we have:

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

- For every $[l]\varphi \in \textsc{Unit}(W)$ such that $N \vdash_T^\star [l]C \cdot \varnothing$ and $N \vdash_T^\star l \cdot \varnothing$, we have $\lceil N \rceil \vdash_T^\star \varphi \cdot \varnothing$ by (d).

As a consequence, $d^M$ is at least one and $A^M$ is a truth assignment.

**Lemma 3.10.** *For every closure $\varphi \cdot \sigma$ such that $\lceil N \rceil \vdash_T^\star \varphi \cdot \sigma$, there is a truth assignment $A$ of $A^M \cup \varphi \cdot \sigma$ such that $\lceil N \rceil \vdash_T^\star \textsc{Unit}(A)$.*

*Proof.* We do the proof by structural induction over $\varphi$.

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

If $\varphi$ is a witness $\langle l \rangle C$ then $\lceil N \rceil \vdash_T^\star l \cdot \sigma$ 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, $\lceil N \rceil \vdash_T^\star C \cdot \sigma$ by (c). By induction hypothesis, there is a truth assignment $A$ of $A^M \cup C \cdot \sigma$ such that $\lceil N \rceil \vdash_T^\star \textsc{Unit}(A)$. As a consequence, $A \cup \varphi \cdot \sigma \cup l \cdot \sigma$ is a truth assignment of $A^M \cup \varphi \cdot \sigma$ and $\lceil N \rceil \vdash_T^\star \textsc{Unit}(A) \cup \varphi \cdot \sigma \cup l \cdot \sigma$.

If $\varphi$ is a trigger $[l]C$ and $\{l\tau \mid l \cdot \tau \in A^M\} \rhd_T l\sigma$ then $\lceil N \rceil \vdash_T^\star l \cdot \sigma$ since $\lceil M' \rceil \vdash_T^\star \textsc{Unit}(A^M)$ 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 $\lceil N \rceil \vdash_T^\star C \cdot \sigma$ by (d), and, by induction hypothesis, there is a truth assignment $A$ of $A^M \cup C \cdot \sigma$ such that $\lceil N \rceil \vdash_T^\star \textsc{Unit}(A)$. As a consequence, $A \cup \varphi \cdot \sigma$ is a truth assignment of $A^M \cup \varphi \cdot \sigma$ and $\lceil N \rceil \vdash_T^\star \textsc{Unit}(A) \cup \varphi \cdot \sigma$.

**Corollary 3.1.** *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 $\textsc{Unit}(A^M) \vdash_T^\star H$ then $\textsc{Unit}(A^M) \vdash_T^\star \varphi \cdot \sigma$. If $\textsc{Unit}(A^M)$ 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 $\sigma'$ such that $A^M$ is a truth assignment of $\varphi \cdot \sigma'$, $\textsc{Unit}(A^M) \vdash_T^\star \sigma \approx \sigma'$ and $\textsc{Unit}(A^M) \cup \mathscr{T}(\textsc{Unit}(A^M)) \vdash_T^\star \mathit{known}(\mathscr{T}(\sigma))$.

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 $\sigma$ is $\mu|_{\mathit{vars}(\varphi)}$. Thus, since $\textsc{Unit}(A^M) \vdash_T^\star H$, there is $\langle l \rangle (C \vee \varphi) \cdot \mu' \in A^M$, such that $\textsc{Unit}(A^M) \vdash_T^\star \mu \approx \mu'$ and $\textsc{Unit}(A^M) \cup \mathscr{T}(\textsc{Unit}(A^M)) \vdash_T^\star \mathit{known}(\mathscr{T}(\mu))$. Since $\lceil N \rceil \vdash_T^\star \textsc{Unit}(A^M)$, we have both $\lceil N \rceil \vdash_T^\star \mu \approx \mu'$ and $\lceil N \rceil \cup \mathscr{T}(N) \vdash_T^\star \mathit{known}(\mathscr{T}(\mu))$. Hence, $\lceil N \rceil \vdash_T^\star \varphi \cdot \mu'|_{\mathit{vars}(\varphi)}$ and, by Lemma 3.10, there is a truth assignment $A$ of $A^M \cup \varphi \cdot \mu'|_{\mathit{vars}(\varphi)}$ such that $\lceil N \rceil \vdash_T^\star \textsc{Unit}(A)$. 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 $l \cdot \mu|_{\mathit{vars}(l)}$ in $H$ such that $\sigma = \mu|_{\mathit{vars}(\varphi)}$. Like for `Witness-Unfold`, there is $[l](C \vee \varphi) \cdot \mu' \in A^M$ such that $\textsc{Unit}(A^M) \vdash_T^\star \mu \approx \mu'$. Since $\textsc{Unit}(A^M) \vdash_T^\star l \cdot \mu|_{\mathit{vars}(l)}$, $\textsc{Unit}(A^M) \vdash_T^\star l \cdot \mu'|_{\mathit{vars}(l)}$ and there is a truth assignment $A$ of $A^M \cup \varphi \cdot \mu'|_{\mathit{vars}(\varphi)}$ such that $\lceil N \rceil \vdash_T^\star \textsc{Unit}(A)$. Since $\textsc{Unit}(A^M) \vdash_T^\star l \cdot \mu'|_{\mathit{vars}(l)}$, $A$ is a truth assignment of $A^M$ and, by maximality of $A^M$, $A = A^M$.

If $\textsc{Unit}(A^M)$ 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 $\sigma = (\mu \cup [x \mapsto t])|_{\mathit{vars}(\varphi)}$. Since $\textsc{Unit}(A^M) \vdash_T^\star H$, there is $\forall x.C \vee \varphi \cdot \mu' \in A^M$ and $t' \in \mathscr{T}(\textsc{Unit}(A^M))$ such that $\textsc{Unit}(A^M) \vdash_T^\star (\mu \cup [x \mapsto t]) \approx (\mu' \cup [x \mapsto t'])$ and $\textsc{Unit}(A^M) \cup \mathscr{T}(\textsc{Unit}(A^M)) \vdash_T^\star \mathit{known}(\mathscr{T}(\mu \cup [x \mapsto t]))$. Since $A^M$ is final, there is $C'' \vee \varphi \cdot \sigma'' \in A^M$ such that $\textsc{Unit}(A^M) \vdash_T^\star (\mu' \cup [x \mapsto t']) \approx \sigma''$. Since $\lceil N \rceil \vdash_T^\star \textsc{Unit}(A^M)$, we have both $\lceil N \rceil \vdash_T^\star (\mu \cup [x \mapsto t])|_{\mathit{vars}(\varphi)} \approx \sigma''$ and $\lceil N \rceil \cup \mathscr{T}(N) \vdash_T^\star \mathit{known}(\mathscr{T}(\mu \cup [x \mapsto t]))$. Thus, $\lceil N \rceil \vdash_T^\star \varphi \cdot \sigma''$ and, by Lemma 3.10, there is a truth assignment $A$ of $A^M \cup \varphi \cdot \sigma''$ such that

$\lceil N \rceil \vdash_T^\star \textsc{Unit}(A)$. By construction, $A$ is a truth assignment of $A^M$ and, by maximality of $A^M$, $A = A^M$.

Since $N \not\vdash_T^\star \bot$ by (a), $A^M$ cannot be $T$-unsatisfiable. If $A^M$ 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 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 $\lceil N \rceil \vdash_T^\star \textsc{Unit}(A^M \cup C \cdot (\sigma \cup [x \mapsto t]))$ which contradicts the definition of $A^M$. Therefore, $C$ is a unit clause. Since $\lceil M'' \rceil \vdash_T^\star \textsc{Unit}(A^M)$, there is a substitution $\sigma'$ and a term $t' \in \mathscr{T}(M'')$ such that $\forall x.C \cdot \sigma' \in M''$, $M'' \vdash_T^\star \sigma \mathrel{\dot{\approx}} \sigma'$, $\mathit{known}(\mathscr{T}(M'')) \cup M'' \vdash_T^\star \mathit{known}(\mathscr{T}(\sigma)) \cup \mathit{known}(\mathscr{T}(t))$ and $M'' \vdash_T^\star t \approx t'$. 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), $\lceil N \rceil \vdash_T^\star C \cdot (\sigma' \cup [x \mapsto t'])$. Since $M'' \subseteq N$, $\lceil N \rceil \vdash_T^\star C \cdot (\sigma \cup [x \mapsto t])$. By Lemma 3.10, there is a truth assignment $A$ of $A^M \cup C \cdot (\sigma \cup [x \mapsto t])$ such that $\lceil N \rceil \vdash_T^\star \textsc{Unit}(A)$ which contradicts the definition of $A^M$.

Therefore $A^M$ is final and 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 $\textsc{Avb}(F, M) \vdash_T^\star C \cdot (\sigma \cup [x \mapsto t])$, which contradicts the non-redundancy condition of `Instantiate`.

We first show that, for every $\varphi \cdot \sigma \in N$, $\textsc{Unit}(A^M) \vdash_T^\star \varphi \cdot \sigma$. By contradiction, let $\varphi \cdot \sigma$ be the first closure of $N$ such that $\textsc{Unit}(A^M) \not\vdash_T^\star \varphi \cdot \sigma$. Let $M^\circ(\varphi \cdot \sigma) \parallel F^\circ$ be the state after $\varphi \cdot \sigma$ was added. If $\varphi \cdot \sigma \in \textsc{Grd}(F^\circ)$ was added to $M^\circ$ using $T$-`Propagate`, then $\textsc{Unit}(A^M) \vdash_T^\star \varphi \cdot \sigma$. Indeed, $\textsc{Unit}(A^M)$ implies every closure $\varphi \cdot \sigma$ in $M^\circ$ 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 $\textsc{Avb}(F^\circ, M^\circ)$. 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 $A_1^M$. 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 $\textsc{Unit}(A^M) \vdash_T^\star H$. By Corollary 3.1, $\textsc{Unit}(A^M) \vdash_T^\star \varphi \cdot \sigma$.

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

- There is $\langle l \rangle C \cdot \sigma'' \in A^M$. Since $\lceil M'' \rceil \vdash_T^\star \textsc{Unit}(A^M)$, there is $\langle l \rangle C \cdot \tau \in M''$ such that $M'' \vdash_T^\star \sigma'' \mathrel{\dot{\approx}} \tau$ and $M'' \cup \mathit{known}(\mathscr{T}(M'')) \vdash_T^\star \mathit{known}(\mathscr{T}(\sigma''))$. As a consequence, by (f), $\textsc{Avb}(F, M) \vdash_T^\star C \cdot \sigma''$.

- There is $[l] C \cdot \sigma'' \in A^M$ such that $\textsc{Unit}(A^M) \vdash_T^\star l\sigma''$. Since $\lceil M'' \rceil \vdash_T^\star \textsc{Unit}(A^M)$, there is $[l] C \cdot \tau \in M''$ such that $M'' \vdash_T^\star \sigma'' \mathrel{\dot{\approx}} \tau$, $M'' \cup \mathit{known}(\mathscr{T}(M'')) \vdash_T^\star \mathit{known}(\mathscr{T}(\sigma''))$, and $M'' \vdash_T^\star l\sigma''$. Therefore, by (f), $\textsc{Avb}(F, M) \vdash_T^\star C \cdot \sigma''$.

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

Consequently, property 1 holds.

We finally need a progress property. Every derivation that does not allow the solver to terminate can be extended without breaking the restrictions requested for termination. What is more, we only require the rule $T$-`Propagate` to be applied when $\lceil M \rceil \vdash_T^\star e$. We need an intermediate lemma:

**Lemma 3.11** (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 $M_1 e_1^{\mathsf{d}} \ldots M_n e_n^{\mathsf{d}} M_{n+1}$ and $M_i$ contain no decision super-literals. We write $\overline{M}_i$ for the prefix $\ldots M_i$ 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 $D' \setminus e$ 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 $D \setminus e_i \subseteq \overline{M}_j$ 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 $F \cup (D \setminus e_i) \vDash_T^\star \neg e_i$ 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 $M = M'e \ldots$. 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 $D'' = (D \setminus e) \cup D'$ 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 3.7. 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 3.5. 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 $F' \cup D \vDash_T^\star e$ and $F \vDash_T^\star F'$ by Lemma 3.7. Thus, $F \cup D \vDash_T^\star e$.

*Remark* 3.9. Compared to usual DPLL, back-jumping is restricted by the requirement on $T$-`Backjump` that $e'$ must appear in $\textsc{Avb}(F,M)$. This restriction is needed in general but it can be alleviated by allowing to add a subsequence of $Me^dN$ to $M$ using `UnitPropagate` and $T$-`Propagate` before $e'$ is added with $T$-`Backjump`.

**Corollary 3.2.** *If here is a closure or a literal $e$ such that $\neg e \in M$ and $\lceil M \rceil \vdash_T^\star e$, then a conflict clause can be learned so that either `Fail` or $T$-`Backjump` can be applied.*

*Proof.* Since $\lceil M \rceil \vdash_T^\star e$, there is a set of closures $S \subseteq \lceil M \rceil$ such that $S \vdash_T^\star e$. 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 $\textsc{Avb}(F,M)$. 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 $\textsc{Avb}(F \setminus H \to C, H)$. Let $H \subseteq \lceil M \rceil$ be a superset of $S$ such that $e$ occurs in $\textsc{Avb}(F,H)$. 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 3.11, $T$-`Backjump` can be applied on $M \parallel F$.

**Theorem 3.4** (Progress)**.** *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 fair derivation $M \parallel F \Longrightarrow^+ S$ containing no `Restart` step and at least one step that is neither an application of $T$-`Learn` or $T$-`Forget` nor a redundant application of `Witness-Unfold`, `Trigger-Unfold` or `Instantiate`.*

*Remark* 3.10. 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 properties is false:

(i) $M \vdash_T^\star \textsc{Avb}(F,M)$,

(ii) $M \nvdash_T^\star \bot$, and

(iii) if $H \to C$ can be added by `Instantiate`, `Witness-Unfold`, or `Trigger-Unfold` then $\textsc{Avb}(F,M) \vdash_T^\star C$.

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 3.11, either `Fail` or $T$-`Backjump` can be applied. Otherwise, there is an undefined super-literal $e$ that occurs in $\textsc{Avb}(F,M)$. Since $e \in \textsc{Avb}(F,M)$, $level_M(e) \leq level(M)$ and `Decide` can be applied on $e$.

If (ii) is false, then $\textsc{Lit}(M) \cup \{l\sigma \mid l \cdot \sigma \in M\} \vDash_T \bot$. Like in the proof of Corollary 3.2, 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 $\textsc{Avb}(F,H) \not\vdash_T^\star C$ or $\textsc{Avb}(F,H) \vdash_T^\star C$ and $H \not\subseteq M$. If $\textsc{Avb}(F,H) \not\vdash_T^\star C$, the application of `Instantiate`, `Witness-Unfold`, or `Trigger-Unfold` is non-redundant in $F$. Otherwise, $\lceil M \rceil \vdash_T^\star H$ and, for some $l \cdot \sigma \in H \setminus M$, $l \cdot \sigma \in \textsc{Grd}(F)$. If $\neg(l \cdot \sigma) \in M$, we conclude using Corollary 3.2. 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) $M \not\vdash_T^\star \bot$,

(b) for every unit super-clause $e \in \textsc{Avb}(F,\varnothing)$, $\lceil M \rceil \vdash_T^\star e$,

(c) for every closure $\langle l \rangle C \cdot \sigma \in M$, $\lceil M \rceil \vdash_T^\star l \cdot \sigma$ and, if $C$ is a unit clause, $\lceil M \rceil \vdash_T^\star C \cdot \sigma$,

(d) for every closure $[l]\varphi \cdot \sigma \in M$ such that $\varphi$ is a unit clause, if $\lceil M \rceil \vdash_T^\star l \cdot \sigma$ then we have $\lceil M \rceil \vdash_T^\star \varphi \cdot \sigma$,

(e) for every closure $\forall x.\varphi \cdot \sigma \in M$ such that $\varphi$ 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 $\lceil M \rceil \vdash_T^\star \varphi \cdot (\sigma \cup [x \mapsto t])$, and

(f) for every guarded clause $H \to C$ that can be added to $F$ by either `Instantiate`, `Witness-Unfold` or `Trigger-Unfold` on $M$, if $level_M(H) \leq level(M)$, $\textsc{Avb}(F,M) \vdash_T^\star C$.

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

If (b) is false then there is a unit super-clause $e \in \textsc{Avb}(F,\varnothing)$ 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 $\lceil M \rceil \not\vdash_T^\star l \cdot \sigma$ or $C$ is a unit clause and $\lceil M \rceil \not\vdash_T^\star C \cdot \sigma$. Therefore, the rule `Witness-Unfold` can be applied with $\langle l \rangle C \cdot \sigma$. If it is redundant, $\textsc{Avb}(F,M) \vdash_T^\star l \cdot \sigma$ and $\textsc{Avb}(F,M) \vdash_T^\star C \cdot \sigma$. 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 $\lceil M \rceil \vdash_T^\star l \cdot \sigma$ and $\lceil M \rceil \not\vdash_T^\star \varphi \cdot \sigma$. 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 $\lceil M \rceil \not\vdash_T^\star \varphi \cdot (\sigma \cup [x \mapsto t])$. First assume that $\textsc{Avb}(F,M) \not\vdash_T^\star \varphi \cdot (\sigma \cup [x \mapsto t])$. 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 $\textsc{Avb}(F,M) \vdash_T^\star \varphi \cdot (\sigma \cup [x \mapsto t])$ then $\textsc{Unit}(\textsc{Avb}(F,M)) \vdash_T^\star \varphi \cdot (\sigma \cup [x \mapsto t])$. By Lemma 3.4, $\lceil M \rceil \not\vdash_T^\star \textsc{Unit}(\textsc{Avb}(F,M))$ (otherwise, $\lceil M \rceil \vdash_T^\star \varphi \cdot (\sigma \cup [x \mapsto t])$). Then, there is a guarded clause

$H \to e \in F$ such that $H \subseteq M$ and $M \not\vdash_T^\star e$. 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 $\text{AVB}(F,H) \not\vdash_T^\star C$ or $\text{AVB}(F,H) \vdash_T^\star C$ and $H \not\subseteq M$. If $\text{AVB}(F,H) \not\vdash_T^\star C$, the application of `Instantiate`, `Witness-Unfold`, or `Trigger-Unfold` is non-redundant in $F$. Otherwise, $\lceil M \rceil \vdash_T^\star H$ and, for some $l \cdot \sigma \in H \setminus M$, $l \cdot \sigma \in \text{GRD}(F)$. If $\neg(l \cdot \sigma) \in M$, we conclude using Corollary 3.2. Otherwise, $T$-`Propagate` can be applied to $l \cdot \sigma$. Indeed, since $level_M(H) \leq level(M)$, it is not forbidden by fairness.

# 4 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 $T$ contains integer linear arithmetic and booleans. For the sake of simplicity, we assume that we can use sorts. The axiomatized extension $T'$ of $T$ 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 from the API of lists in the Ada standard library [7].

## 4.1 Theory

Lists are ordered containers of elements on which an equivalence, named *equal_elements*, is defined. 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 *co* is modeled by an integer $length(co)$ representing the length of the sequence together with a total function *position* so that, for every cursor *cu*, $position(co, cu)$ returns the position of *cu* in *co* if it appears in the sequence and 0 otherwise. The content part of *co* is modeled by a function *element* so that $element(co, cu)$ returns the element associated to *cu* in *co* if any:

$$\begin{array}{lcccc} elements: & \star & \star & & \star \\ & \uparrow & \uparrow & \dots & \uparrow \\ cursors: & \bullet & \bullet & & \bullet \\ positions: & 1 & 2 & & length \end{array}$$

Thanks to this description, we can define several other functions. $has_element(co, cu)$ returns true if and only if *cu* appears in the imperative part of *co* and $is_empty(co)$ returns true if *co* is an empty list. The functions *last*, *first*, *previous* and *next* are used to iterate through the iterative part of *co*. If *co* is empty, $last(co)$ and $first(co)$ return a special cursor, named *no_element* that never appears in any list. *no_element* is also added at both ends of the iterative part of *co* so that

$previous(co, first(co))$, $previous(co, no_element)$, $next(co, last(co))$ and $next(co, no_element)$ are $no_element$:

$$\underset{no_element}{\circlearrowright \circ \leftarrow} \quad \underset{sequence}{\bullet \rightleftarrows \bullet \cdots \bullet \rightleftarrows \bullet} \quad \underset{no_element}{\rightarrow \circ \circlearrowleft}$$

We define two functions $left$ and $right$, that are used to split the list. If $cu$ appears in the imperative part of $co$ or is $no_element$, $left(co, cu)$ (resp. $right(co, cu)$) returns the prefix (resp. suffix) of $co$ that stops (resp. starts) before $cu$:

$$\underbrace{\begin{array}{ccc} \star & & \star \\ \uparrow & \cdots & \uparrow \\ \bullet & & \bullet \end{array}}_{left(co,cu)} \qquad \underbrace{\begin{array}{ccc} \star & & \star \\ \uparrow & \cdots & \uparrow \\ cu & & \bullet \end{array}}_{right(co,cu)}$$

A special empty list $empty$ is returned by $left(co, cu)$ (resp. $right(co, cu)$) if the cursor $cu$ is $first(co)$ (resp. $no_element$). On $no_element$, $left(co, cu)$ returns $co$.

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

We add a notion of equality on list: $equal_lists(co_1, co_2)$ is true if and only if both parts of $co_1$ and $co_2$ are equal.

The last three functions are designed to describe how a list $co$ is modified when an element is either inserted, deleted or replaced in $co$. If $insert(co, cu, e, r)$ is true then $r$ can be obtained by inserting a cursor before $cu$ in the list $co$ (or at the end if $cu$ is $no_element$) and mapping it to $e$. If $delete(co, cu, r)$ is true then $r$ can be obtained by deleting the cursor $cu$ from the list $co$. If $replace_element(co, cu, e, r)$ is true then $r$ can be obtained by replacing the element associated to $cu$ in $co$ by $e$.

## 4.2 Axiomatization

We give in this section an overview of the axiomatization of the extension of Section 4.1. For readability, we extend our logic to sort. We specify the sort of every quantified variable. Here we only give a few axioms. The whole axiomatization is available in Appendix A.

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:

$$\forall co : list.[length(co)]length(co) \geq 0$$

POSITION_GTE_ZERO:

$$\begin{array}{l} \forall co : list, cu : cursor.[position(co, cu)] \\ \quad length(co) \geq position(co, cu) \wedge position(co, cu) \geq 0 \end{array}$$

POSITION_EQ:

$$\forall co : list, cu_1 cu_2 : cursor.[position(co, cu_1), position(co, cu_2)]$$
$$position(co, cu_1) > 0 \rightarrow position(co, cu_1) \approx position(co, cu_2) \rightarrow cu_1 \approx cu_2$$

Functions on lists such as *right*, *previous*, $first$ or $find$ are described on their domain of definition. We only present $find$. The function $find_first(co, e)$ returns the result of $find(co, e, no_element)$, that is to say the first cursor of *co* that is mapped to an element equivalent to *e*. The result $find(co, e, cu)$ can then be defined to be the result of $find_first$ on the cursors following *cu* in *co* that is to say $right(co, cu)$.

FIND_FIRST_RANGE:

$$\forall co : list, e : element_type.[find_first(co, e)]$$
$$find_first(co, e) \approx no_element \vee position(co, find_first(co, e)) > 0$$

FIND_FIRST_NOT:

$$\forall co : list, e : element_type, cu : cursor.[find_first(co, e), element(co, cu)]$$
$$find_first(co, e) \approx no_element \rightarrow position(co, cu) > 0 \rightarrow$$
$$equal_elements(element(co, cu), e) \not\approx \top$$

FIND_FIRST_FIRST:

$$\forall co : list, e : element_type, cu : cursor.[find_first(co, e), element(co, cu)]$$
$$0 < position(co, cu) < position(co, find_first(co, e)) \rightarrow$$
$$equal_elements(element(co, cu), e) \not\approx \top$$

FIND_FIRST_ELEMENT:

$$\forall co : list, e : element_type.[find_first(co, e)]0 < position(co, find_first(co, e)) \rightarrow$$
$$equal_elements(element(co, find_first(co, e)), e) \approx \top$$

FIND_FIRST:

$$\forall co : list, e : element_type.[find(co, e, no_element)]$$
$$find(co, e, no_element) \approx find_first(co, e)$$

FIND_OTHERS:

$$\forall co : list, e : element_type, cu : cursor.[find(co, e, cu)]$$
$$position(co, cu) > 0 \rightarrow find(co, e, cu) \approx find_first(right(co, cu), e)$$

The predicates describing a modification of the 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 links between the two lists using functions *length*, *position* and *element*.

DELETE_RANGE:

$$\forall co_1 co_2 : list, cu : cursor.[delete(co_1, cu, co_2)]$$
$$delete(co_1, cu, co_2) \approx \top \rightarrow position(co_1, cu) > 0$$

DELETE_LENGTH:

$$\forall co_1 co_2 : list, cu : cursor.[delete(co_1, cu, co_2)]$$
$$delete(co_1, cu, co_2) \approx \top \rightarrow length(co_2) + 1 \approx length(co_1)$$

DELETE_POSITION_BEFORE:

$$\begin{array}{l}\forall co_1 co_2 : list, cu_1 cu_2 : cursor.[delete(co_1, cu_1, co_2), position(co_1, cu_2)] \\ \quad (delete(co_1, cu_1, co_2) \approx \top \wedge position(co_1, cu_2) < position(co_1, cu_1)) \rightarrow \\ \quad\quad position(co_1, cu_2) \approx position(co_2, cu_2) \\ \forall co_1 co_2 : list, cu_1 cu_2 : cursor.[delete(co_1, cu_1, co_2), position(co_2, cu_2)] \\ \quad (delete(co_1, cu_1, co_2) \approx \top \wedge 0 < position(co_2, cu_2) < position(co_1, cu_1)) \rightarrow \\ \quad\quad position(co_1, cu_2) \approx position(co_2, cu_2)\end{array}$$

DELETE_POSITION_AFTER:

$$\begin{array}{l}\forall co_1 co_2 : list, cu_1 cu_2 : cursor.[delete(co_1, cu_1, co_2), position(co_1, cu_2)] \\ \quad (delete(co_1, cu_1, co_2) \approx \top \wedge position(co_1, cu_2) > position(co_1, cu_1)) \rightarrow \\ \quad\quad position(co_1, cu_2) \approx position(co_2, cu_2) + 1 \\ \forall co_1 co_2 : list, cu_1 cu_2 : cursor.[delete(co_1, cu_1, co_2), position(co_2, cu_2)] \\ \quad (delete(co_1, cu_1, co_2) \approx \top \wedge position(co_2, cu_2) \geq position(co_1, cu_1)) \rightarrow \\ \quad\quad position(co_1, cu_2) \approx position(co_2, cu_2) + 1\end{array}$$

DELETE_POSITION_NEXT:

$$\forall co_1 co_2 : list, cu : cursor.[delete(co_1, cu, co_2)] delete(co_1, cu, co_2) \approx \top \rightarrow \langle next(co_1, cu) \rangle \top$$

DELETE_ELEMENT:

$$\begin{array}{l}\forall co_1 co_2 : list, cu_1 cu_2 : cursor.[delete(co_1, cu_1, co_2), element(co_1, cu_2)] \\ \quad (delete(co_1, cu, co_2) \approx \top \wedge position(co_2, cu_2) > 0) \rightarrow \\ \quad\quad element(co_1, cu_2) = element(co_2, cu_2)\end{array}$$

## 4.3 Properties

In this section, we illustrate how a proof of termination, soundness and completeness can be conducted on the axiomatization of doubly-linked lists.

**Theorem 4.1.** *The axiomatization in Section 4.2 is terminating, sound and complete with respect to the theory of doubly-linked lists in Section 4.1.*

### 4.3.1 Proof of Termination

Every universal quantification is done on lists, cursors or elements. As a consequence, if we show that only a finite number of terms of type list, cursor and element can be created, we can deduce that the axiomatization 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 $find(co, e, cu)$. Such a term cannot be created by the axiomatization. Since the symbol $find$ is not interpreted, $known(find(co, e, cu))$ can only be deduced if we have $known(find(co', e', cu'))$ and equalities between each sub-term. These equalities are enough to ensure that the new term $right(co, cu)$ is equal to the already known term $right(co', cu')$. As a consequence, there can only be one new term of type list per terms of the form $find(co, e, cu)$ in the initial problem.

Then we concentrate on terms of type cursor. The axioms CONTAINS_DEF, FIND_FIRST, FIND_OTHERS, 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 as for the lists terms. Indeed, the symbols $contains(co,e)$, $find(co,e,cu)$, $insert(co_1,cu,e,co_2)$, $delete(co_1,cu,co_2)$ and $equal_lists(co_1,co_2)$ are all uninterpreted and cannot be created by the axiomatization.

Finally, let us look at terms of type element. There are a great deal of those because the function *element* is often used. However, most of the time, new terms of type element 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 solved with the same reasoning we did for terms of type list and cursor. Indeed, $find_first(co,e)$ is uninterpreted and can only be created once per $contains(co,e)$ and twice per $find(co',e,cu)$ which themselves cannot be created.

### 4.3.2 Proof of Soundness

We show that, if a set of literals $G$ has a model in the theory of doubly-linked lists then there is a total model of $G$ and the axiomatization. If $I$ is a model of a set of literals $G$ in the theory of doubly-linked lists, we define $L = \{l \mid I(l) = \top\}$. By construction of $L$, $L$ is a total model of $G$. It is straightforward to show that, for every axiom $\varphi$ of the axiomatization, $L \rhd_T \varphi$.

### 4.3.3 Proof of Completeness

We first need a lemma that states that equalities between integers can safely be added to partial models of the axiomatization:

**Lemma 4.1.** *If the axiomatization is true in a world $L$, $t_1,t_2 \in \mathscr{T}(L)$ have type integer and $L \not\models_T t_1 \not\approx t_2$ then the axiomatization is also true in $L \cup t_1 \approx t_2$.*

*Proof.* Triggers of the axiomatization either have no (non-variable) sub-term of type integer or can be written $t \approx t$ where $t$ is of type integer and has no proper (non-variable) sub-term 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 the axiomatization, if $L \not\rhd_T l$, $t_1,t_2 \in \mathscr{T}(L)$ 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 the axiomatization is true in $L \cup t_1 \approx t_2$.

Let $G$ be a set of literals and $L$ a world in which $G$ and the axiomatization are true. 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 the axiomatization. As a consequence, the axiomatization is true in $L \cup \{i \approx i \mid i$ is an integer

constant}. For every term $t \in \mathscr{T}(L)$ of the form $length(co)$ or $position(co, cu)$, we add $t \approx I_T(t)$ to $L$. By Lemma 4.1, the axiomatization is still true in $L$.

For every term $co$ of type list in $L$, if $length(co)$ is not in $\mathscr{T}(L)$ modulo $T$, we add $length(co) \approx 0$ to $L$ and, for every term $cu$ of type cursor, if $position(co, cu)$ is not in $L$, we add $position(co, cu) \approx 0$. 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, $length(co)$ is in $\mathscr{T}(L)$ whenever there is a cursor $cu$ such that $position(co, cu)$ is in $\mathscr{T}(L)$.

We now need 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 $co$ of type list in $L$ to be a sublist $co[i, j]$, with $i$ and $j$ are in $0..length(co)$ such that either $i = 0$ or there is a term $cu$ of type cursor in $\mathscr{T}(L)$ such that $L \vDash_T position(co, cu) \approx i$ and, for all $k$ such that $i < k \leq j$, there is no term $cu$ of type cursor in $L$ such that $L \vDash_T position(co, cu) \approx k$. 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 are the rules for deletion.
For every $co[i, j] \in eq(z)$:

$$
\begin{array}{lcllcrl}
L \vDash_T delete(co, cu, co') \approx \top & \text{and} & L \vDash_T j < position(co, cu) & \rightarrow & co'[i, j] & \in eq(z) \\
L \vDash_T delete(co, cu, co') \approx \top & \text{and} & L \vDash_T j > position(co, cu) & \rightarrow & co'[i-1, j-1] & \in eq(z) \\
L \vDash_T delete(co', cu, co) \approx \top & \text{and} & L \vDash_T j < position(co', cu) & \rightarrow & co'[i, j] & \in eq(z) \\
L \vDash_T delete(co', cu, co) \approx \top & \text{and} & L \vDash_T position(co', cu) \leq j & \rightarrow & co'[i+1, j+1] & \in eq(z) \\
 & & & & & \\
L \vDash_T left(co, cu) \approx co' & \text{and} & L \vDash_T 0 < j < position(co, cu) & \rightarrow & co'[i, j] & \in eq(z) \\
L \vDash_T left(co', cu) \approx co & & & \rightarrow & co'[i, j] & \in eq(z)
\end{array}
$$

The set $eq(z)$ has some good properties:

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 $position(co, cu) > 0$ and $co[position(co, cu), _] \in eq(z)$ then, for every zone $co'[i, _] \in eq(z)$, $L \vDash_T position(co', cu) \approx i$.

From the last two properties, we deduce that each list $co$ 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 $co[i, j]$, with $co$ known and $i < j$:

- We compute the set of zones $eq(co[i, j])$.
- To each $k$ such that $i < k \leq j$, we associate a fresh cursor $cu_k$.

- For each $co'[i', j']$ and each $k'$ such that $i' < k' \leq j'$, $position(co', cu_{k'-i'+i}) \approx k'$ is added to $L$.

Once there is no more zone $co[i, j]$, with $co \in \mathscr{T}(L)$ and $i < j$, for every term of type list and every term of type cursor of $L$, we can add $position(co, cu) \approx 0$ to $L$. We can check straightforwardly that the axiomatization 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 term of type element. We map $element(co, cu)$ to $e$ for any term $co$ of type list and any term $cu$ of type cursor in $L$ such that $element(co, cu)$ is not in $L$ modulo $T$. Each axiom with $element(l, cu)$ 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, the axiomatization is still true in $L$.

*Remark* 4.1. 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 the axiomatization 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 $position(co, find_first(co, e_i)) > n$ for every $i \in 1..n$ and $equal_elements(e_i, e_j) \not\approx \top$ for every $i$ and $j \in 1..n$ such that $i \neq j$.

We have constructed a model for $L$ in the theory of doubly linked list described in Section 4.1. As a consequence, the axiomatization from Section 4.2 is complete for this theory.

# 5 Implementation in the Alt-Ergo Theorem Prover

In this section, we present our implementation of the framework of Section 3.2 in the Alt-Ergo theorem prover. We discuss various specificities of this implementation and finally give some bench-marks using the theory of doubly-linked lists given in Section 4.1.

## 5.1 $E$-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 that 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 allow 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, that 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. For every closure $\varphi \cdot \sigma$ where $\varphi$ is a universally quantified formula $\forall \overline{x}.[l_1, \ldots, l_n]\varphi'$ where $\varphi'$ is not a trigger, we compute a triplet made of a set of literals $l_\varphi$ and two sets of terms, $p_\varphi$ and $k_\varphi$. It has the following properties:

(i) every free variable (that is not in the domain of $\sigma$) in $l_\varphi$ or $k_\varphi$ is also in $p_\varphi$,

(ii) terms of $p_\varphi$ only contain variables and uninterpreted symbols,

(iii) if $\tau$ is a mapping from free variables of $p_\varphi$ to terms containing only variables that are in the domain of $\sigma$, 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 $\tau$ from free variables of $p_\varphi$ 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 $p_\varphi$ to be the set of all the patterns $p_t$ constructed above; $k_\varphi$ 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 $p_\varphi$; 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 $k_\varphi$.

## 5.2 Different Notions of Termination

The notion of termination in Section 2.2 may turn out be too constraining for some axiomatization. Let us start with an example. Assume that we want to add to our theory in Section 4.1 a notion of structural equality of lists modulo equivalence of elements named *equivalent_lists*. The function $equivalent_lists(co_1, co_2)$ returns true if and only if $co_1$ and $co_2$ contain equivalent elements in the same order. In an axiomatization of this concept, we could have:

EQUIVALENT_LISTS_ELEMENT:

$$\begin{aligned}
&\forall co_1 co_2 : list.[equivalent_lists(co_1, co_2)]equivalent_lists(co_1, co_2) \approx \top \rightarrow \\
&\quad (\forall cu_1 : cursor.[element(co_1, cu_1)]position(co_1, cu_1) > 0 \rightarrow \\
&\quad\quad \exists cu_2 : cursor.position(co_1, cu_1) = position(co_2, cu_2) \wedge \\
&\quad\quad\quad equivalent_lists_elements(element(co_1, cu_1), element(co_2, cu_2)) \approx \top)
\end{aligned}$$

Unfortunately, such an axiom would introduce a loop. If the set of literals includes both $equivalent_lists(co_1, co_2) \approx \top$ and $equivalent_lists(co_2, co_1) \approx \top$ and $element(co_1, cu)$ is known for some $cu$, there is a branch deducing the term $element(co_2, sko(co_1, co_2, cu))$ which itself allows the deduction of the term $element(co_1, sko(co_2, co_1, sko(co_1, co_2, cu)))$ and so on. We can

see that the term $sko(co_2, co_1, sko(co_1, co_2, cu))$ is in fact equal to $cu$, using POSITION_EQ. However, our definition of truth assignment is not restrictive enough to enforce this deduction.

The definition of truth assignment given in Section 2.2 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 Section 3.5. 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 Section 2.2, 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 literal have been assigned a truth value by the model to imply 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 $\varphi$ 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 *equivalent_lists* example to go through. The first one has the drawback of forbidding an 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. This gives a more constraining version of termination that can still be proved for the axiomatization of the theory of doubly-linked list. As compensation, new rules can be added to our version of abstract DPLL to handle anti-closures. They have to work in a compatible way with the semantics of negations of closures defined for the proofs of Section 3.4. 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.

## 5.3 Accommodating Non-Convex Theories

The notion of satisfiability in Section 2.1 can also turn out to be too constraining. For example, in the proof of completeness of Section 4.3, we need to show that adding equalities on known terms of type integer do not break the partial model. This lemma is needed because we cannot assume that, for a partial model $L$ and two known terms $t_1$ and $t_2$ such that $L \not\models_T t_1 \approx t_2$, we can find an interpretation $I$ of $L$ such that, $I(t_1) \neq I(t_2)$. Indeed, linear integer arithmetic is not a convex theory, which means that there can be a set of literals $L$ and a set of terms $t_1, s_1, \ldots, t_n, s_n \subseteq \mathscr{T}(L)$ such that $L \models_T t_1 \approx s_1 \vee \cdots \vee t_n \approx s_n$ and, for every $i \in 1..n$, $L \not\models_T t_i \approx s_i$. As a consequence, if an axiomatization does not have the above property, it will not be complete in our framework.

To have a less constraining notion of completeness, the definition of satisfiability can be modified. Let $F$ be an axiomatization. Instead of searching for any $T$-satisfiable set of literals

$L$ such that $L \rhd F$, we can restrict the definition of partial models so that we only search for $T$-satisfiable sets of literals $L$ such that, if $L \models_T t_1 \approx s_1 \vee \cdots \vee t_n \approx s_n$ for $t_1, s_1, \ldots, t_n, s_n \subseteq \mathscr{T}(L)$ then $L \models_T t_i \approx s_i$ for some $i \in 1..n$. Our implementation in Alt-Ergo complies with this second definition by using the disjunctions generated by the theory combination mechanism.

## 5.4 Benchmarks

We use the Why3 VC generator version 0.80 and the Alt-Ergo theorem prover version 0.95. 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 (co:list) (cu:cursor) : element_type
  requires { has_element co cu }
  ensures  { result = element co cu }

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

The tests for using the theory of doubly-linked lists are given in Appendix B. Here is only one of them. 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) /\
```

| Tests | Alt-Ergo | Alt-Ergo$^\star$ |
|---|---|---|
| `test_delete` | 14 | 5 |
| `test_insert` | 608 | 394 |
| `double_size` | 314 | 47 |
| `filter` | 196 | 207 |
| `my_contain` | 22 | 2 |
| `my_find` | 300 | 6 |
| `map_f` | 107 | 31 |

Figure 3: Time (in seconds) needed to solve all tests with Alt-Ergo giving the first-order axiomatization directly and Alt-Ergo through the theory mechanism (named Alt-Ergo$^\star$) .

```
        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 3 is a comparison between the results obtained by giving the axiomatization directly in input to Alt-Ergo and using it as a theory through the new mechanism. We see that, on average, our implementation gives better results than the usual instantiation mechanism of Alt-Ergo. This is mainly due to the fact that instances are now generated in an eager way.

The necessity of specifying appropriate triggers may seem to be a drawback of our approach. However, even if triggers can be inferred by SMT solvers, efficient handling of first-order formulas usually requires user guidance in this choice. Indeed, if we remove our triggers from the first-order axiomatization of lists before giving it to Alt-Ergo, the programs in Table 3 can no longer be verified in less than the time limit of 1000s.

# 6 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, cf. [3, 10, 22].

In future work, we would 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 whether it is possible to check automatically the completeness and termination of an axiomatization, at least in some restricted cases.

## References


[1] Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovi, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: G. Gopalakrishnan, S. Qadeer (eds.) Computer Aided Verification, *Lecture Notes in Computer Science*, vol. 6806, pp. 171–177. Springer Berlin Heidelberg (2011)

[2] Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard version 2.0. Technical report, University of Iowa (2010)

[3] Bjørner, N.: Engineering theories with Z3. Programming Languages and Systems pp. 4–16 (2011)

[4] Bobot, F., Conchon, S., Contejean, E., Lescuyer, S.: Implementing polymorphism in SMT solvers. In: SMT'08, *ACM ICPS*, vol. 367, pp. 1–5. ACM (2008)

[5] Chatterjee, S., Lahiri, S.K., Qadeer, S., Rakamarić, Z.: A low-level memory model and an accompanying reachability predicate. International journal on software tools for technology transfer **11**(2), 105–116 (2009)

[6] Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005)

[7] Dross, C., Filliâtre, J.C., Moy, Y.: Correct code containing containers. In: Proceedings of the 5th international conference on Tests and proofs, TAP'11, pp. 102–118. Springer-Verlag (2011)

[8] Ge, Y., Barrett, C., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. In: CADE-21, *LNCS*, vol. 4603, pp. 167–182. Springer (2007)

[9] Ge, Y., De Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Computer Aided Verification, *LNCS*, vol. 5643, pp. 306–320. Springer (2009)

[10] 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 (2008)

[11] Jacobs, S., Kuncak, V.: Towards complete reasoning about axiomatic specifications. In: Proceedings of VMCAI-12, *LNCS*, vol. 6538, pp. 278–293. Springer (2011)

[12] Lynch, C., Morawska, B.: Automatic decidability. In: Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on, pp. 7–16. IEEE (2002)

[13] Lynch, C., Ranise, S., Ringeissen, C., Tran, D.K.: Automatic decidability and combinability. pp. 1026–1047. Elsevier (2011)

[14] McPeak, S., Necula, G.C.: Data structure specifications via local equality axioms. In: Computer Aided Verification, pp. 476–490. Springer (2005)

[15] Moskal, M.: Programming with triggers. In: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories, ACM ICPS, pp. 20–29. ACM (2009)

[16] de Moura, L., Bjørner, N.: Efficient E-matching for SMT solvers. In: CADE-21, *LNCS*, vol. 4603, pp. 183–198. Springer (2007)

[17] de Moura, L., Bjørner, N.: Engineering DPLL(T) + saturation. In: IJCAR 2008, *LNCS*, vol. 5195, pp. 475–490. Springer (2008)

[18] de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS, *LNCS*, vol. 4963, pp. 337–340. Springer (2008)

[19] Nelson, G.: Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center (1981)

[20] Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). Journal of the ACM **53**(6), 937–977 (2006)

[21] Rümmer, P.: E-matching with free variables. In: Logic for Programming, Artificial Intelligence, and Reasoning: 18th International Conference, LPAR-18, *LNCS*, vol. 7180, pp. 359–374. Springer (2012)

[22] Suter, P., Steiger, R., Kuncak, V.: Sets with cardinality constraints in satisfiability modulo theories. In: R. Jhala, D. Schmidt (eds.) Verification, Model Checking, and Abstract Interpretation, *Lecture Notes in Computer Science*, vol. 6538, pp. 403–418. Springer Berlin Heidelberg (2011)

# A Axiomatization of Imperative Doubly-Linked Lists

LENGTH_GTE_ZERO:

$$\forall co : list.[length(co)]length(co) \geq 0$$

IS_EMPTY:

$$\forall co : list.[is_empty(co)]is_empty(co) \approx \top \leftrightarrow length(co) \approx 0$$

EMPTY_IS_EMPTY:

$$is_empty(empty)$$

EQUAL_ELEMENTS_REFL:

$$\forall e : element_type.[equal_elements(e,e)]equal_elements(e,e) \approx \top$$

EQUAL_ELEMENTS_SYM:

$$\begin{aligned}&\forall e_1 e_2 : element_type.[equal_elements(e_1,e_2)]\\ &\quad equal_elements(e_1,e_2) \approx equal_elements(e_2,e_1)\end{aligned}$$

EQUAL_ELEMENTS_TRANS:

$$\begin{aligned}&\forall e_1 e_2 e_3 : element_type.[equal_elements(e_1,e_2), equal_elements(e_2,e_3)]\\ &\quad equal_elements(e_1,e_2) \approx \top \rightarrow equal_elements(e_2,e_3) \approx \top \rightarrow\\ &\quad\quad equal_elements(e_1,e_3) \approx \top\\ &\forall e_1 e_2 e_3 : element_type.[equal_elements(e_1,e_2), equal_elements(e_1,e_3)]\\ &\quad equal_elements(e_1,e_2) \approx \top \rightarrow equal_elements(e_2,e_3) \approx \top \rightarrow\\ &\quad\quad equal_elements(e_1,e_3) \approx \top\end{aligned}$$

POSITION_GTE_ZERO:

$$\begin{aligned}&\forall co : list, cu : cursor.[position(co,cu)]\\ &\quad length(co) \geq position(co,cu) \wedge position(co,cu) \geq 0\end{aligned}$$

POSITIONNO_ELEMENT:

$$\forall co : list.[position(co, no_element)]position(co, no_element) \approx 0$$

POSITION_EQ:

$$\begin{aligned}&\forall co : list, cu_1 cu_2 : cursor.[position(co,cu_1), position(co,cu_2)]\\ &\quad position(co,cu_1) > 0 \rightarrow position(co,cu_1) \approx position(co,cu_2) \rightarrow cu_1 \approx cu_2\end{aligned}$$

PREVIOUS_IN:

$$\begin{aligned}&\forall co : list, cu : cursor.[previous(co,cu)]\\ &\quad (position(co,cu) > 1 \vee position(co, previous(co,cu)) > 0) \rightarrow\\ &\quad\quad position(co, previous(co,cu)) \approx position(co,cu) - 1\end{aligned}$$

PREVIOUS_EXT:

$$\forall co : list, cu : cursor.[previous(co,cu)]$$
$$(position(co,cu) \approx 1 \vee cu \approx no_element) \rightarrow previous(co,cu) \approx no_element$$

NEXT_IN:

$$\forall co : list, cu : cursor.[next(co,cu)]$$
$$(length(co) > position(co,cu) > 0 \vee position(co,next(co,cu)) > 0) \rightarrow$$
$$position(co,next(co,cu)) \approx position(co,cu) + 1$$

NEXT_EXT:

$$\forall co : list, cu : cursor.[next(co,cu)]$$
$$(position(co,cu) \approx length(co) \vee cu \approx no_element) \rightarrow next(co,cu) \approx no_element$$

LAST_EMPTY:

$$\forall co : list.[last(co)]length(co) \approx 0 \leftrightarrow last(co) \approx no_element$$

LAST_GEN:

$$\forall co : list.[last(co)]length(co) \approx position(co,last(co))$$

FIRST_EMPTY:

$$\forall co : list.[first(co)]length(co) \approx 0 \leftrightarrow first(co) \approx no_element$$

FIRST_GEN:

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

HAS_ELEMENT_DEF:

$$\forall co : list, cu : cursor.[has_element(co,cu)]position(co,cu) > 0 \leftrightarrow has_element(co,cu) \approx \top$$

LEFT_NO_ELEMENT:

$$\forall co : list.[left(co,no_element)]left(co,no_element) \approx co$$

LEFT_LENGTH:

$$\forall co : list, cu : cursor.[left(co,cu)]$$
$$position(co,cu) > 0 \rightarrow length(left(co,cu)) \approx position(co,cu) - 1$$

LEFT_POSITION_IN:

$$\forall co : list, cu_1 cu_2 : cursor.[position(left(co, cu_1), cu_2)]$$
$$(position(left(co, cu_1), cu_2) > 0 \vee position(co, cu_2) < position(co, cu_1)) \rightarrow$$
$$position(left(co, cu_1), cu_2) \approx position(co, cu_2)$$
$$\forall co : list, cu_1 cu_2 : cursor.[left(co, cu_1), position(co, cu_2)]$$
$$(position(left(co, cu_1), cu_2) > 0 \vee position(co, cu_2) < position(co, cu_1)) \rightarrow$$
$$position(left(co, cu_1), cu_2) \approx position(co, cu_2)$$

LEFT_POSITION_EXT:

$$\forall co : list, cu_1 cu_2 : cursor.[position(left(co, cu_1), cu_2)]$$
$$position(co, cu_2) \geq position(co, cu_1)) > 0 \rightarrow$$
$$position(left(co, cu_1), cu_2) \approx 0$$

LEFT_ELEMENT:

$$\forall co : list, cu_1 cu_2 : cursor.[element(left(co, cu_1), cu_2)]$$
$$(position(left(co, cu_1), cu_2) > 0 \vee 0 < position(co, cu_2) < position(co, cu_1)) \rightarrow$$
$$element(left(co, cu_1), cu_2) \approx element(co, cu_2)$$
$$\forall co : list, cu_1 cu_2 : cursor.[left(co, cu_1), element(co, cu_2)]$$
$$(position(left(co, cu_1), cu_2) > 0 \vee 0 < position(co, cu_2) < position(co, cu_1)) \rightarrow$$
$$element(left(co, cu_1), cu_2) \approx element(co, cu_2)$$

RIGHT_NO_ELEMENT:

$$\forall co : list.[right(co, no_element)] right(co, no_element) \approx empty$$

RIGHT_LENGTH:

$$\forall co : list, cu : cursor.[right(co, cu)]$$
$$position(co, cu) > 0 \rightarrow length(right(co, cu)) \approx length(co) - position(co, cu) + 1$$

RIGHT_POSITION_IN:

$$\forall co : list, cu_1 cu_2 : cursor.[position(right(co, cu_1), cu_2)]$$
$$(position(right(co, cu_1), cu_2) > 0 \vee 0 < position(co, cu_1) \leq position(co, cu_2)) \rightarrow$$
$$position(right(co, cu_1), cu_2) \approx position(co, cu_2) - position(co, cu_1) + 1$$
$$\forall co : list, cu_1 cu_2 : cursor.[right(co, cu_1), position(co, cu_2)]$$
$$(position(right(co, cu_1), cu_2) > 0 \vee 0 < position(co, cu_1) \leq position(co, cu_2)) \rightarrow$$
$$position(right(co, cu_1), cu_2) \approx position(co, cu_2) - position(co, cu_1) + 1$$

RIGHT_POSITION_EXT:

$$\forall co : list, cu_1 cu_2 : cursor.[position(right(co, cu_1), cu_2)]$$
$$position(co, cu_2) < position(co, cu_1)) \rightarrow$$
$$position(right(co, cu_1), cu_2) \approx 0$$

RIGHT_ELEMENT:

$$\forall co : list, cu_1 cu_2 : cursor.[element(right(co, cu_1), cu_2)]$$
$$(position(right(co, cu_1), cu_2) > 0 \vee 0 < position(co, cu_1) \leq position(co, cu_2)) \rightarrow$$
$$element(right(co, cu_1), cu_2) \approx element(co, cu_2)$$
$$\forall co : list, cu_1 cu_2 : cursor.[right(co, cu_1), element(co, cu_2)]$$
$$(position(right(co, cu_1), cu_2) > 0 \vee 0 < position(co, cu_1) \leq position(co, cu_2)) \rightarrow$$
$$element(right(co, cu_1), cu_2) \approx element(co, cu_2)$$

FIND_FIRST_RANGE:

$$\forall co : list, e : element_type.[find_first(co, e)]$$
$$find_first(co, e) \approx no_element \vee position(co, find_first(co, e)) > 0$$

FIND_FIRST_NOT:

$$\forall co : list, e : element_type, cu : cursor.[find_first(co, e), element(co, cu)]$$
$$find_first(co, e) \approx no_element \rightarrow position(co, cu) > 0 \rightarrow$$
$$equal_elements(element(co, cu), e) \not\approx \top$$

FIND_FIRST_FIRST:

$$\forall co : list, e : element_type, cu : cursor.[find_first(co, e), element(co, cu)]$$
$$0 < position(co, cu) < position(co, find_first(co, e)) \rightarrow$$
$$equal_elements(element(co, cu), e) \not\approx \top$$

FIND_FIRST_ELEMENT:

$$\forall co : list, e : element_type.[find_first(co, e)] 0 < position(co, find_first(co, e)) \rightarrow$$
$$equal_elements(element(co, find_first(co, e)), e) \approx \top$$

CONTAINS_DEF:

$$\forall co : list, e : element_type.[contains(co, e)]$$
$$contains(co, e) \leftrightarrow 0 < position(co, find_first(co, e))$$

FIND_FIRST:

$$\forall co : list, e : element_type.[find(co, e, no_element)]$$
$$find(co, e, no_element) \approx find_first(co, e)$$

FIND_OTHERS:

$$\forall co : list, e : element_type, cu : cursor.[find(co, e, cu)]$$
$$position(co, cu) > 0 \rightarrow find(co, e, cu) \approx find_first(right(co, cu), e)$$

REPLACE_ELEMENT_RANGE:

$$\forall co_1 co_2 : list, cu : cursor, e : element_type.[replace_element(co_1, cu, e, co_2)]$$
$$replace_element(co_1, cu, e, co_2) \approx \top \rightarrow position(co_1, cu) > 0$$

REPLACE_ELEMENT_LENGTH:

$$\forall co_1 co_2 : list, cu : cursor, e : element_type.[replace_element(co_1, cu, e, co_2)]$$
$$replace_element(co_1, cu, e, co_2) \approx \top \rightarrow length(co_1) \approx length(co_2)$$

REPLACE_ELEMENT_POSITION:

$$\forall co_1 co_2 : list, cu_1 cu_2 : cursor, e : element_type.[replace_element(co_1, cu_1, e, co_2), position(co_1, cu_2]$$
$$replace_element(co_1, cu_1, e, co_2) \approx \top \rightarrow position(co_1, cu_2) \approx position(co_2, cu_2)$$
$$\forall co_1 co_2 : list, cu_1 cu_2 : cursor, e : element_type.[replace_element(co_1, cu_1, e, co_2), position(co_2, cu_2]$$
$$replace_element(co_1, cu_1, e, co_2) \approx \top \rightarrow position(co_1, cu_2) \approx position(co_2, cu_2)$$

REPLACE_ELEMENT_ELEMENT_IN:

$$\forall co_1 co_2 : list, cu : cursor, e : element_type.[replace_element(co_1, cu, e, co_2)]$$
$$replace_element(co_1, cu, e, co_2) \approx \top \rightarrow element(co_2, cu) \approx e$$

REPLACE_ELEMENT_ELEMENT_EXT:

$$\forall co_1 co_2 : list, cu_1 cu_2 : cursor, e : element_type.[replace_element(co_1, cu_1, e, co_2), element(co_1, cu_2)]$$
$$(replace_element(co_1, cu_1, e, co_2) \approx \top \wedge position(co_1, cu_2) > 0 \wedge cu_1 \not\approx cu_2) \rightarrow$$
$$element(co_1, cu_2) \approx element(co_2, cu_2)$$
$$\forall co_1 co_2 : list, cu_1 cu_2 : cursor, e : element_type.[replace_element(co_1, cu_1, e, co_2), element(co_2, cu_2)]$$
$$(replace_element(co_1, cu_1, e, co_2) \approx \top \wedge position(co_1, cu_2) > 0 \wedge cu_1 \not\approx cu_2) \rightarrow$$
$$element(co_1, cu_2) \approx element(co_2, cu_2)$$

INSERT_RANGE:

$$\forall co_1 co_2 : list, cu : cursor, e : element_type.[insert(co_1, cu, e, co_2)]$$
$$insert(co_1, cu, e, co_2) \approx \top \rightarrow cu \approx no_element \vee position(co_1, cu) > 0$$

INSERT_LENGTH:

$$\forall co_1 co_2 : list, cu : cursor, e : element_type.[insert(co_1, cu, e, co_2)]$$
$$insert(co_1, cu, e, co_2) \approx \top \rightarrow length(co_2) \approx length(co_1) + 1$$

INSERT_NEW:

$$\forall co_1 co_2 : list, cu : cursor, e : element_type.[insert(co_1, cu, e, co_2)]$$
$$(insert(co_1, cu, e, co_2) \approx \top \wedge position(co_1, cu) > 0) \rightarrow$$
$$position(co_1, previous(co_2, cu)) \approx 0 \wedge element(co_2, previous(co_2, cu)) \approx e$$

INSERT_NEW_NO_ELEMENT:

$$\forall co_1 co_2 : list, cu : cursor, e : element_type.[insert(co_1, no_element, e, co_2)]$$
$$insert(co_1, no_element, e, co_2) \approx \top \rightarrow$$
$$position(co_1, last(co_2)) \approx 0 \wedge element(co_2, last(co_2)) \approx e$$

INSERT_POSITION_BEFORE:

$$\forall co_1 co_2 : list, cu_1 cu_2 : cursor, e : element_type.[insert(co_1, cu_1, e, co_2), position(co_1, cu_2)]$$
$$(insert(co_1, cu_1, e, co_2) \approx \top \wedge 0 < position(co_1, cu_2) < position(co_1, cu_1)) \rightarrow$$
$$position(co_1, cu_2) \approx position(co_2, cu_2)$$
$$\forall co_1 co_2 : list, cu_1 cu_2 : cursor, e : element_type.[insert(co_1, cu_1, e, co_2), position(co_2, cu_2)]$$
$$(insert(co_1, cu_1, e, co_2) \approx \top \wedge position(co_2, cu_2) < position(co_1, cu_1)) \rightarrow$$
$$position(co_1, cu_2) \approx position(co_2, cu_2)$$

INSERT_POSITION_AFTER:

$$\forall co_1 co_2 : list, cu_1 cu_2 : cursor, e : element_type.[insert(co_1, cu_1, e, co_2), position(co_1, cu_2)]$$
$$(insert(co_1, cu_1, e, co_2) \approx \top \wedge position(co_1, cu_2) \geq position(co_1, cu_1) > 0) \rightarrow$$
$$position(co_1, cu_2) + 1 \approx position(co_2, cu_2)$$
$$\forall co_1 co_2 : list, cu_1 cu_2 : cursor, e : element_type.[insert(co_1, cu_1, e, co_2), position(co_2, cu_2)]$$
$$(insert(co_1, cu_1, e, co_2) \approx \top \wedge position(co_2, cu_2) > position(co_1, cu_1) > 0) \rightarrow$$
$$position(co_1, cu_2) + 1 \approx position(co_2, cu_2)$$

INSERT_POSITION_NO_ELEMENT:

$$\forall co_1 co_2 : list, cu : cursor, e : element_type.[insert(co_1, no_element, e, co_2), position(co_1, cu)]$$
$$(insert(co_1, no_element, e, co_2) \approx \top \wedge position(co_1, cu) > 0) \rightarrow$$
$$position(co_1, cu) \approx position(co_2, cu)$$
$$\forall co_1 co_2 : list, cu : cursor, e : element_type.[insert(co_1, no_element, e, co_2), position(co_2, cu)]$$
$$(insert(co_1, no_element, e, co_2) \approx \top \wedge position(co_2, cu_2) < length(co_2)) \rightarrow$$
$$position(co_1, cu) \approx position(co_2, cu)$$

INSERT_ELEMENT:

$$\forall co_1 co_2 : list, cu_1 cu_2 : cursor, e : element_type.[insert(co_1, cu_1, e, co_2), element(co_1, cu_2)]$$
$$(insert(co_1, cu_1, e, co_2) \approx \top \wedge position(co_1, cu_2) > 0) \rightarrow$$
$$element(co_1, cu_2) \approx element(co_2, cu_2)$$
$$\forall co_1 co_2 : list, cu_1 cu_2 : cursor, e : element_type.[insert(co_1, cu_1, e, co_2), element(co_2, cu_2)]$$
$$(insert(co_1, cu_1, e, co_2) \approx \top \wedge position(co_1, cu_2) > 0) \rightarrow$$
$$element(co_1, cu_2) \approx element(co_2, cu_2)$$

DELETE_RANGE:

$$\forall co_1 co_2 : list, cu : cursor.[delete(co_1, cu, co_2)]$$
$$delete(co_1, cu, co_2) \approx \top \rightarrow position(co_1, cu) > 0$$

DELETE_LENGTH:

$$\forall co_1 co_2 : list, cu : cursor.[delete(co_1, cu, co_2)]$$
$$delete(co_1, cu, co_2) \approx \top \rightarrow length(co_2) + 1 \approx length(co_1)$$

DELETE_POSITION_BEFORE:

$$\forall co_1 co_2 : list, cu_1 cu_2 : cursor.[delete(co_1, cu_1, co_2), position(co_1, cu_2)]$$
$$(delete(co_1, cu_1, co_2) \approx \top \wedge position(co_1, cu_2) < position(co_1, cu_1)) \rightarrow$$
$$position(co_1, cu_2) \approx position(co_2, cu_2)$$
$$\forall co_1 co_2 : list, cu_1 cu_2 : cursor.[delete(co_1, cu_1, co_2), position(co_2, cu_2)]$$
$$(delete(co_1, cu_1, co_2) \approx \top \wedge 0 < position(co_2, cu_2) < position(co_1, cu_1)) \rightarrow$$
$$position(co_1, cu_2) \approx position(co_2, cu_2)$$

DELETE_POSITION_AFTER:

$$\forall co_1 co_2 : list, cu_1 cu_2 : cursor.[delete(co_1, cu_1, co_2), position(co_1, cu_2)]$$
$$(delete(co_1, cu_1, co_2) \approx \top \wedge position(co_1, cu_2) > position(co_1, cu_1)) \rightarrow$$
$$position(co_1, cu_2) \approx position(co_2, cu_2) + 1$$
$$\forall co_1 co_2 : list, cu_1 cu_2 : cursor.[delete(co_1, cu_1, co_2), position(co_2, cu_2)]$$
$$(delete(co_1, cu_1, co_2) \approx \top \wedge position(co_2, cu_2) \geq position(co_1, cu_1)) \rightarrow$$
$$position(co_1, cu_2) \approx position(co_2, cu_2) + 1$$

DELETE_POSITION_NEXT:

$$\forall co_1 co_2 : list, cu : cursor.[delete(co_1, cu, co_2)] delete(co_1, cu, co_2) \approx \top \rightarrow \langle next(co_1, cu) \rangle \top$$

DELETE_ELEMENT:

$$\forall co_1 co_2 : list, cu_1 cu_2 : cursor.[delete(co_1, cu_1, co_2), element(co_1, cu_2)]$$
$$(delete(co_1, cu, co_2) \approx \top \wedge position(co_2, cu_2) > 0) \rightarrow$$
$$element(co_1, cu_2) = element(co_2, cu_2)$$

EQUAL_LISTS_POSITION:

$$\forall co_1 co_2 : list.[equal_lists(co_1, co_2)] equal_lists(co_1, co_2) \approx \top \rightarrow$$
$$(\forall cu : cursor.[position(co_1, cu)] position(co_1, cu) \approx position(co_2, cu)) \wedge$$
$$(\forall cu : cursor.[position(co_2, cu)] position(co_1, cu) \approx position(co_2, cu))$$

EQUAL_LISTS_ELEMENT:

$$\forall co_1 co_2 : list.[equal_lists(co_1, co_2)] equal_lists(co_1, co_2) \approx \top \rightarrow$$
$$(\forall cu : cursor.[element(co_1, cu)] position(co_1, cu) > 0 \rightarrow$$
$$element(co_1, cu) \approx element(co_2, cu)) \wedge$$
$$(\forall cu : cursor.[element(co_2, cu)] position(co_1, cu) > 0 \rightarrow$$
$$element(co_1, cu) \approx element(co_2, cu))$$

EQUAL_LISTS_INV:

$$\forall co_1 co_2 : list.[equal_lists(co_1, co_2)] equal_lists(co_1, co_2) \not\approx \top \rightarrow$$
$$(\exists cu : cursor.position(co_1, cu) > 0 \wedge$$
$$(position(co_2, cu) > 0 \rightarrow element(co_1, cu) \not\approx element(co_2, cu)))$$

EQUAL_LISTS_LENGTH:

$$\forall co_1 co_2 : list.[equal_lists(co_1, co_2)] equal_lists(co_1, co_2) \approx \top \rightarrow length(co_1) \approx length(co_2)$$

# B Tests in Why3 Language Using the Theory of Doubly-Linked Lists

API of program functions:

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

val replace_element (co:ref list) (cu:cursor) (e:element_type) : unit
  requires { has_element !co cu }
  writes   { co }
  ensures  { replace_element (old !co) cu e !co }

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

val prepend (co:ref list) (e:element_type) : unit
  reads   { co }
  writes  { co }
  ensures { insert (old !co) (first (old !co)) e !co }

val append (co:ref list) (e:element_type) : unit
  reads   { co }
  writes  { co }
  ensures { insert (old !co) no_element e !co }

val delete (co:ref list) (cu:cursor) : unit
  requires { has_element !co cu }
  reads    { co }
  writes   { co }
  ensures  { delete (old !co) cu !co }

val previous (co:list) (cu:cursor) : cursor
  requires { cu = no_element \/ has_element co cu }
  ensures  { result = previous co cu }

val next (co:list) (cu:cursor) : cursor
  requires { cu = no_element \/ has_element co cu }
  ensures  { result = next co cu }
```

Tests using this API:

```
(* take a list of 4 elements, prepend element e, remove all
   initial 4 elements, take the last element of the list, it is e *)
let test_delete (li : ref list) (e : element_type) =
  requires { length !li = 4 }
  ensures  { result = e }
  prepend li e;
  let c = ref (last !li) in
  delete li !c;
  c := first !li;
  c := next !li (first !li);
  delete li !c;
  c := last !li;
  delete li !c;
  c := last !li;
  delete li !c;
  element !li (last !li)

(* adding elements to a list does not invalidate an existing cursor *)
let test_insert (li : ref list) (c d f g h : cursor) (e : element_type) =
  requires { position !li c = 4 /\ has_element !li f /\ has_element !li h }
  ensures  { has_element !li c }
  insert li c e;
  append li e;
  if has_element !li d then
    insert li d e;
  insert li f e;
  if length !li > 5 then
    if g = (next !li c) then
      insert li g e
    else
      insert li h e

(* iterate through the list by adding element e at every position. This doubles
   the size of the list *)
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

(* Removes some elements from li, stores them in removed *)
function fun_test element_type : bool

let filter_one (li:ref list) (removed:ref list) (c:ref cursor) =
    requires { has_element !li !c }
    ensures  {
     ((has_element (old !li) !c /\ has_element !li !c) \/
            !c = no_element) /\
      (length (left !li !c)) + (length !removed) =
             (length (left (old !li) !c)) + (length (old !removed)) /\
       equal_lists (right !li !c) (right (old !li) !c) /\
             !c = next (old !li) (old !c
    }
    let c_int = next !li !c in
    append removed (element !li !c);
    delete li !c;
    c := c_int

let filter (li:ref list) (removed:ref list) =
  requires { not (is_empty !li) /\ is_empty !removed }
  ensures  { (length !li) + (length !removed) = 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)) + (length !removed) =
              length (left (at !li 'Loop_Entry) !c) /\
         equal_lists (right !li !c) (right (at !li 'Loop_Entry) !c))
    }
    if fun_test(element !li !c) then
        filter_one li removed c
  done
```

```
(* the usual implementation of contains indeed computes the awaited result *)
let my_contain (s:list) (e:element_type) =
    ensures { result = True <-> contains s e }
    let c = ref (first s) in
    let res = ref False in
    try
      while has_element s !c do
         invariant {
            ((has_element s !c \/ !c = no_element) /\
             (not contains (left s !c) e)) }
         if equal_elements e (element s !c) then
            raise Return
         else c:=next s !c
      done
    with Return -> res := True end;
    ! res

(* the usual implementation of find indeed computes the awaited result *)
let my_find (s : list) (e : element_type) (f : cursor) =
  requires { has_element s f }
  ensures  { result = find s e f }
  let c = ref f in
  try
    while has_element s !c do
      invariant {
        (has_element (right s f) !c \/ !c = no_element) /\
        find (left (right s  f) !c) e no_element = no_element
      }
      if equal_elements e (element s !c) then
        raise Return
      else c := next s !c
    done
  with Return -> () end;
  !c
  { result = find s e f }

(* after map l s, every element in s has been transformed through f *)
function f element_type : element_type

let map_f (s : ref list) (cu : cursor) =
  ensures { forall cu : cursor. has_element !s cu ->
                   element !s cu = f (element (old !s) cu) }
  'Loop_Entry :
```

```
let c = ref (first !s) in
while !c <> no_element do
  invariant {
   (has_element !s !c /\ has_element (at !s 'Loop_Entry) !c \/
         !c = no_element) /\
   (forall cu : cursor. has_element (left !s !c) cu ->
          element !s cu = f (element (at !s 'Loop_Entry) cu)) /\
   equal_lists (right (at !s 'Loop_Entry) !c) (right !s !c)
  }
  replace_element s !c (f(element !s !c));
  c := next !s !c
done
```