Generic decision procedures for axiomatic first-order theories

Claire Dross

UNIVERSITÉ PARIS SUD

Comprendre le monde, construire l'avenir®

AdaCore
The GNAT Pro Company

informatics mathematics
Inria

LABORATOIRE DE RECHERCHE EN INFORMATIQUE

UNIVERSITÉ PARIS-SUD

ECOLE DOCTORALE : EDIPS
LABORATOIRE DE RECHERCHE EN INFORMATIQUE

DISCIPLINE : INFORMATIQUE

THÈSE DE DOCTORAT

Soutenue le 1 avril 2014 par

Claire DROSS

Procédures de Décision Génériques pour des Théories Axiomatiques du Premier Ordre

---

Directeur de thèse :	M. Claude MARCHÉ	Directeur de Recherche (Inria Saclay–Île-de-France)
Co-directeur de thèse :	M. Andrei PASKEVICH	Maitre de Conférences (Université Paris-Sud)
Composition du jury :		
Président du jury :	M. Joffroy BEAUQUIER	Professeur (Université Paris-Sud)
Rapporteurs :	M. Nikolaj BJØRNER	Principal Researcher (Microsoft Research)
	M. Albert RUBIO	Professeur (Universitat Politècnica de Catalunya)
Examinateur :	M. Stephan MERZ	Directeur de Recherche (Inria Nancy)
Invités :	M. Johannes KANIG	Ingénieur de Recherche (AdaCore)
	M. Yannick MOY	Ingénieur de Recherche (AdaCore)

Tout d'abord, je souhaite remercier Andrei, pour le temps qu'il m'a consacré au jour le jour, en particulier en période de rédaction, pour me conseiller, corriger mes erreurs et relire toutes mes preuves, même les plus mal écrites. J'ai surtout beaucoup apprécié nos discussion, à la fois professionnellement et personnellement. Je souhaite également remercier Claude, pour son aide et ses conseils, ainsi que pour ses encouragements jusqu'à la soutenance. Son tact, son calme et sa vision positive m'ont beaucoup aidée à appréhender sereinement les dernières étapes du parcours. Merci aussi à Sylvain, qui a orienté mes recherches et m'a encadrée durant toute ma première année. Ses conseils techniques m'ont été très utiles tout au long de ma thèse.

Cette thèse ayant été faite en convention CIFRE, j'ai également passé une grande partie de mon temps en entreprise. Je souhaite donc remercier Yannick et Johannes, qui m'ont accompagnée et encouragée pendant ces 3 ans. Ils m'ont laissée libre de choisir l'organisation de mon temps tout en m'intégrant comme membre à part entière de l'équipe. Nous avons eu de nombreuses discussions enrichissantes (techniques ou non), en particulier devant la machine à café.

Merci également à mes rapporteurs, Nikolaj Bjørner et Albert Rubio, et ainsi qu'aux autres membres de mon jury, Stephan Merz et Joffroy Beauquier, pour leurs remarques et leurs conseils qui à la fois m'ont permis d'améliorer la qualité de mon mémoire et m'ont donné de nouvelles pistes pour aller plus loin.

Je tiens aussi à remercier tous ceux avec qui j'ai travaillé et discuté ces trois dernières années, à la fois au laboratoire et en entreprise. Que ce soit d'un coté ou de l'autre, la bonne ambiance m'a toujours aidée à venir travailler, même dans les moments de fatigue ou de baisse de moral. En particulier, je souhaite remercier mes collègues de bureau, Raphaël et Jérôme, et Asma, Catherine et Stéfania pour nos conversations. Une dédicace spéciale à Elie, avec qui j'ai partagé de nombreuses discussions sur les mérites d'une thèse en CIFRE.

Finalement, pour le soutien moral qu'ils m'ont apporté tout au long de ma thèse, je souhaite remercier ma famille, en particulier mes parents, Fred, Camille et François, qui m'ont encouragée pendant la soutenance et aidée pour le pot. Merci aussi à Emmanuel, mon petit ange, qui a égayé la dernière année de ma thèse et m'a aidée à toujours bien séparer vie professionnelle et vie privée. Le mot de la fin est pour Xavier, mon mari, qui m'a toujours soutenue, a gardé Emmanuel quand je faisais des heures sup le soir avec Andrei et a relu maintes fois mes écrits à la recherche des nombreuses fautes d'orthographe. Nos longues discussions m'ont très souvent aidée à faire la part des choses et à continuer à avancer.

Résumé

Les solveurs SMT sont des outils dédiés à la vérifications d'un ensemble de formules mathématiques, en général sans quantificateurs, utilisant un certain nombre de théories prédéfinies, telles que la congruence, l'arithmétique linéaire sur les entiers, les rationnels ou les réels, les tableaux de bits ou les tableaux. Ajouter une nouvelle théorie à un solveur SMT nécessite en général une connaissance assez profonde du fonctionnement interne du solveur, et, de ce fait, ne peut en général être exécutée que par ses développeurs.

Pour de nombreuses théories, il est également possible de fournir une axiomatisation finie en logique du premier ordre. Toutefois, si les solveurs SMT sont généralement complets et efficaces sur des problèmes sans quantificateurs, ils deviennent imprévisibles en logique du premier ordre. Par conséquent, cette approche ne peut pas être utilisée pour fournir une procédure de décision pour ces théories.

Dans cette thèse, nous proposons un cadre d'application permettant de résoudre ce problème en utilisant des *déclencheurs*. Les déclencheurs sont des annotations permettant de spécifier la forme des termes avec lesquels un quantificateur doit être instancié pour obtenir des instances utiles pour la preuve. Ces annotations sont utilisées par la majorité des solveurs SMT supportant les quantificateurs et font partie du format SMT-LIB v2.

Dans notre cadre d'application, l'utilisateur fournit une axiomatisation en logique du premier ordre de sa théorie, ainsi qu'une démonstration de sa correction, de sa complétude et de sa terminaison, et obtient en retour un solveur correct, complet et qui termine pour sa théorie. Dans cette thèse, nous décrivons comment un solveur SMT peut être étendu à notre cadre nous basant sur l'algorithme DPLL modulo théories, utilisé traditionnellement pour modéliser ls solveurs SMT. Nous prouvons également que notre extension a bien les propriétés attendues.

L'effort à fournir pour implémenter cette extension dans un solveur SMT existant ne doit être effectué qu'une fois et le mécanisme peut ensuite être utilisé sur de multiple théories axiomatisées. De plus, nous pensons que, en général, cette implémentation n'est pas plus compliquée que l'ajout d'une unique théorie au solveur. Nous avons fait ce travail pour le solveur SMT Alt-Ergo, nous en présentons certains détails dans la thèse.

Pour valider l'utilisabilité de notre cadre d'application, nous avons prouvé la complétude et la terminaison de plusieurs axiomatizations, dont une pour les listes impératives doublement chaînée, une pour les ensembles applicatifs et une pour les vecteurs de Ada. Nous avons ensuite utilisé notre implémentation dans Alt-Ergo pour discuter de l'efficacité de notre système dans différents cas.

Generic Decision Procedures for Axiomatic First-Order Theories

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 thesis, 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 this thesis.

To show that our framework can handle complex theories, we prove completeness and termination of three axiomatization, one for doubly-linked lists, one for applicative sets, and one for Ada's vectors. Our tests show that, when the theory is heavily used, our approach results in a better performance of the solver on goals that stem from the verification of programs manipulating these data-structures.

Contents

1 Introduction **1**

1.1 Context: Deductive Verification of Programs 1
1.1.1 The Ada 2012 and SPARK 2014 Languages 2
1.1.2 Deductive Verification in SPARK 2014 3
1.1.3 Challenges in Deductive Verification of Programs 5
1.2 Landscape in Deductive Verification of Programs 6
1.2.1 Deductive Verification Tools for Mainstream Languages 6
1.2.2 Verification Condition Generation 7
1.2.3 Automatic Theorem Provers and SMT Solvers 8
1.3 Problem and Contributions 10
1.3.1 First Order Axiomatizations as Decision Procedures 10
1.3.2 Overview of the Contributions 11

2 First-Order Logic with Triggers **15**

2.1 Formalization 15
2.1.1 Preliminary Notions 16
2.1.2 Logic with Triggers (Syntax and Semantics) 17
2.1.3 Relation with Traditional First-Order Logic 18
2.1.4 Soundness and Completeness 21
2.1.5 Termination 22
2.2 Case Study: Imperative Doubly-Linked Lists 26
2.2.1 Presentation of the Theory 27
2.2.2 Description of the Axiomatization 28
2.2.3 Proofs of Soundness, Completeness, and Termination 30
2.2.4 Assessment of Adequacy with Respect to Existing Trigger Heuristics . 33
2.3 Designing Terminating and Complete Axiomatizations 35
2.3.1 Proving Termination of an Axiomatization 35
2.3.2 Designing a Complete Axiomatization 40
2.3.3 An Automatable Debugger for Completeness 42
2.4 Conclusion 43

3 A Black-Box Decision Procedure **45**
3.1 Description . . . 45
3.1.1 Preliminaries . . . 45
3.1.2 Deduction Rules for First-Order Logic with Triggers . . . 47
3.1.3 Soundness, Completeness, and Termination . . . 48
3.2 Implementation . . . 53
3.2.1 Description . . . 53
3.2.2 Benchmarks . . . 54
3.3 Conclusion . . . 56

4 A White-Box Decision Procedure **57**
4.1 Description . . . 58
4.1.1 Preliminaries . . . 58
4.1.2 Description of DPLL(T) with triggers . . . 63
4.1.3 Termination Related Constraints . . . 66
4.1.4 Soundness and Completeness . . . 68
4.1.5 Progress and Termination . . . 70
4.2 Implementation . . . 80
4.2.1 E-Matching on Uninterpreted Sub-Terms . . . 80
4.2.2 Different Notions of Termination . . . 82
4.2.3 Inclusion into the Theory Combination Mechanism . . . 83
4.2.4 Comparison with Alt-Ergo's Built-In Quantifier Handling . . . 83
4.3 Conclusion . . . 85

5 Case Study: Set Theory of Why3 **87**
5.1 Context: the B Method . . . 87
5.2 A Decision Procedure for Why3's Sets . . . 88
5.2.1 Presentation of the Theory . . . 88
5.2.2 Description of the Axiomatization . . . 89
5.2.3 Proofs of Soundness, Completeness, and Termination . . . 90
5.3 Benchmarks . . . 92
5.3.1 Assessment of the Adequacy Between our Framework and Usual E-matching Techniques . . . 93
5.3.2 Comparison Between our Implementation and the Built-in Quantifiers Handling of Alt-Ergo . . . 96
5.4 Conclusion . . . 96

6 Case Study: Formal Bounded Vectors of SPARK 2014 **97**
6.1 Context: the SPARK 2014 Tool . . . 97
6.2 Ada's Formal Vectors Package . . . 99
6.2.1 Description of the Ada Package . . . 99
6.2.2 Translation into WhyML . . . 101
6.3 A Decision Procedure for Formal Vectors . . . 101
6.3.1 Presentation of the Theory . . . 102

6.3.2 Description of the Axiomatization . . . 103
6.3.3 Proofs of Soundness, Completeness, and Termination . . . 107
6.4 Benchmarks . . . 111
6.4.1 Assessment of the Adequacy Between our Framework and Usual E-Matching Techniques . . . 112
6.4.2 Comparison Between our Implementation and the Built-in Quantifiers Handling of Alt-Ergo . . . 112
6.5 Conclusion . . . 112

7 Conclusion **117**
7.1 Summary of the Contributions . . . 117
7.2 Related Work . . . 118
7.3 Perspectives . . . 120

Bibliography **123**

Index **129**

Glossary **131**

Appendix **133**

A Imperative Doubly-Linked Lists **133**
A.1 Axiomatization . . . 133
A.2 Tests in WhyML . . . 139
A.2.1 API of program functions . . . 139
A.2.2 Tests using this API . . . 140

B Why3 Sets **145**
B.1 Axiomatization for Sets . . . 145

C SPARK 2014 Vectors **147**
C.1 Axiomatization for Formal Vectors . . . 147
C.2 Tests in SPARK 2014 . . . 156
C.2.1 Two_Way_Sort . . . 156
C.2.2 N_Queens . . . 157
C.2.3 Ring_Buffer . . . 160
C.2.4 Amortized_Queue . . . 163

List of Figures

1.1 Deductive verification in SPARK 2014 . . . 3
1.2 Formal verification of the `Max_Array` function . . . 4
1.3 Formal verification of the `Max_Array` function with a loop invariant . . . 5

2.1 Comparison of solvers' efficiency with and without explicit triggers on doubly-linked lists . . . 34

3.1 Deduction rules for the black-box implementation of our framework. . . . 47
3.2 Execution time of the black-box implementation on the theory of arrays. . . . 54
3.3 Execution time of the black-box implementation on doubly-linked lists. . . . 56

4.1 Transition rules of DPLL$^\star(T)$ on guarded clauses . . . 64
4.2 Additional transition rules for DPLL$^\star(T)$ . . . 64
4.3 Comparison between our theory mechanism's implementation in Alt-Ergo and Alt-Ergo's built-in quantifiers handling on doubly-linked lists . . . 84

5.1 Number of goals discharged by solvers with our theory mechanism's implementation and with solvers' built-in quantifier handling with and without triggers on Why3 Set case study . . . 93
5.2 Comparison of solvers' efficiency with our theory mechanism's implementation and with solvers' built-in quantifier handling with and without triggers on Why3 Set case study without additional lemmas . . . 94
5.3 Comparison of solvers' efficiency with our theory mechanism's implementation and with solvers' built-in quantifier handling with and without triggers on Why3 Set case study with additional lemmas . . . 95
5.4 Number of goals discharged by our theory mechanism's implementation in Alt-Ergo and Alt-Ergo's built-in quantifiers handling on Why3 Set case study . . . 96

6.1 Comparison of SMT solvers' efficiency with and without explicit triggers on SPARK Vectors case study . . . 113
6.2 Comparison between our theory mechanism's implementation in Alt-Ergo and Alt-Ergo's built-in quantifiers handling on SPARK Vectors case study . . . 114

6.3 Comparison between our theory mechanism's implementation in Alt-Ergo and Alt-Ergo's built-in quantifiers handling with an additional axiom on SPARK Vectors case study 115

1 Introduction

Contents

1.1 Context: Deductive Verification of Programs 1
1.1.1 The Ada 2012 and SPARK 2014 Languages 2
1.1.2 Deductive Verification in SPARK 2014 3
1.1.3 Challenges in Deductive Verification of Programs 5
1.2 Landscape in Deductive Verification of Programs 6
1.2.1 Deductive Verification Tools for Mainstream Languages 6
1.2.2 Verification Condition Generation 7
1.2.3 Automatic Theorem Provers and SMT Solvers 8
1.3 Problem and Contributions 10
1.3.1 First Order Axiomatizations as Decision Procedures 10
1.3.2 Overview of the Contributions 11

1.1 Context: Deductive Verification of Programs

Software often replaces or helps humans in critical domains such as transportation, avionics, space, or medical areas. Since its failure may cost a lot, both materially and in terms of human lives, this software is thoroughly verified so that it offers sufficient levels of confidence that it operates correctly. Historically, this verification is mostly done by testing. Still, since testing is expensive, there is a trend to replace or to complement it by methods based on automated mathematical analysis on source code, called formal verification.

Ada is a programming language targeted at real-time embedded software which requires a high level of safety, security, and reliability. In particular, it provides a wide range of run-time checks, for example for buffer overflows, and has a verbose syntax that makes it easy to read and debug. For these reasons, Ada is nowadays used in domains where software cannot be allowed to fail.

SPARK [6], co-developed by Altran and AdaCore, is a subset of Ada targeted at formal verification. Its restrictions ensure that the behavior of a SPARK program is unambiguously defined (unlike Ada). It excludes constructions that cannot easily be verified by automatic tools. The SPARK language and toolset for static verification has been applied for many years in on-board aircraft systems, control systems, cryptographic systems, and rail systems [62].

1.1.1 The Ada 2012 and SPARK 2014 Languages

Ada 2012 [5] is the latest version of the Ada language. It contains new features for specifying the behavior of programs, such as subprogram contracts and type invariants. When given a specific compilation switch, the Ada compiler can turn these constructs into assertions to check at run time. Thanks to this switch, the conformance of the implementation of a program to its specification can be checked dynamically during the process of unit testing.

Complex contracts can be expressed thanks to the new constructs that have been introduced in Ada 2012. They include in particular conditional expressions and universally and existentially quantified expressions over finite ranges of integers.

Here is the specification of an Ada 2012 function, named `Max_Array`:

```
function Max_Array (A    : Elt_Array;
                    EMin : Element) return Element with
  Post => Max_Array'Result >= EMin and
    (for all J in A'Range => Max_Array'Result >= A (J)) and
    (if Max_Array'Result /= EMin then
      (for some J in A'Range => Max_Array'Result = A (J)));
```

It takes as input an array `A` and a minimal bound `EMin` and returns the maximum of `EMin` and the elements stored in `A`. It has a postcondition that uses a universally quantified expression, an existentially quantified expression, and a conditional expression. It states that the result of `Max_Array` is bigger than both `EMin` and the elements of `A` and that, if `Max_Array` does not return `EMin` then it returns an element of `A`.

Here is a possible implementation of this function. It uses a loop to go through the array `A` and stores in `Result` the biggest element encountered so far:

```
function Max_Array (A    : Elt_Array;
                    EMin : Element) return Element
is
  Result : Element := EMin;
begin
  for J in A'Range loop
    if A (J) > Result then
      Result := A (J) ;
    end if ;
  end loop;
  return Result;
end Max_Array;
```

Testing is the most common way to verify that a program is safe, that is, that it does not fail at run-time, and that it behaves as is required by its contract. Still, formal verification can also be used to increase confidence in a program, in particular since it gives guarantees for every execution of the program.

The SPARK 2014 language is a subset of Ada 2012, augmented with features specific to formal verification. It comprises most of the Ada 2012 language excluding constructs which are

not easily amenable to sound static verification. Features such as pointers, side effects in expressions, aliasing, goto statements, controlled types (e.g. types with finalization) and exception handling are excluded.

The SPARK 2014 language is designed so that both the flow analysis – checking that there is no access to uninitialized variables and that global variables and subprogram parameters are accessed appropriately – and the proof of program – checking the absence of run-time errors and the conformance to the contract – can be checked. It provides dedicated features that are not part of Ada 2012. In particular, contracts can also contain information about data dependencies, information flows, state abstraction, and data and behavior refinement that can be checked by the SPARK 2014 tools. Essential constructs for formal verification such as loop variants and invariants have also been introduced.

1.1.2 Deductive Verification in SPARK 2014

As an alternative to testing, deductive verification, also called proof of programs, can be used to check that a program is run-time error free and that it complies with its specifications. As described in Figure 1.1, the SPARK 2014 tool for proof of program, named GNATprove, uses a specialized intermediate language named WhyML [15]. It is an ML-like programming language with support for proof-only features such as abstract types and functions with no definition, non executable contracts and assertions, as well as lemmas and axioms. The Why3 tool can be used to extract logical statements out of WhyML code, whose validity implies the soundness of the WhyML program. It can then be used as a frontend for a large set of both automatic and interactive provers.

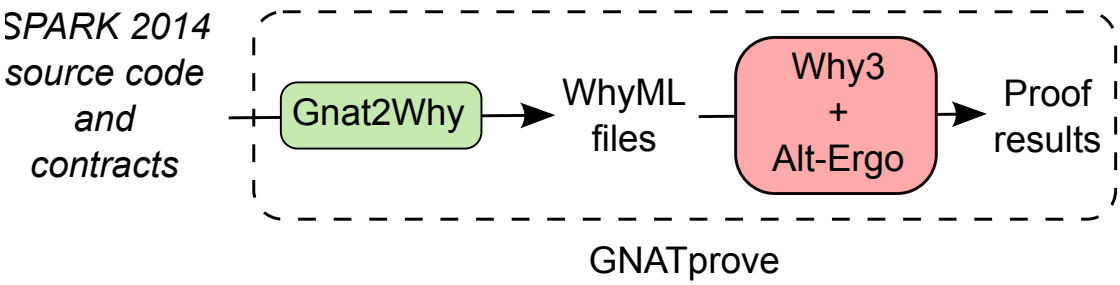


Figure 1.1: Deductive verification in SPARK 2014

We can use GNATprove to verify our implementation of the `Max_Array` function. It first checks that there is no flow error in the design of the program, for example, no use of uninitialized variable and no unintended flow of information. Then, it formally verifies both the absence of run-time errors and the validity of assertions and contracts. We see in Figure 1.2 that the postcondition of `Max_Array` cannot be verified by GNATprove. Indeed, deductive verification techniques used inside the SPARK 2014 tool sometimes require additional annotations. For the proof of the postcondition of `Max_Array` to go through, we need to specify a loop invariant, that is, a property that is true at every iteration of the loop. It allows one to describe how variables

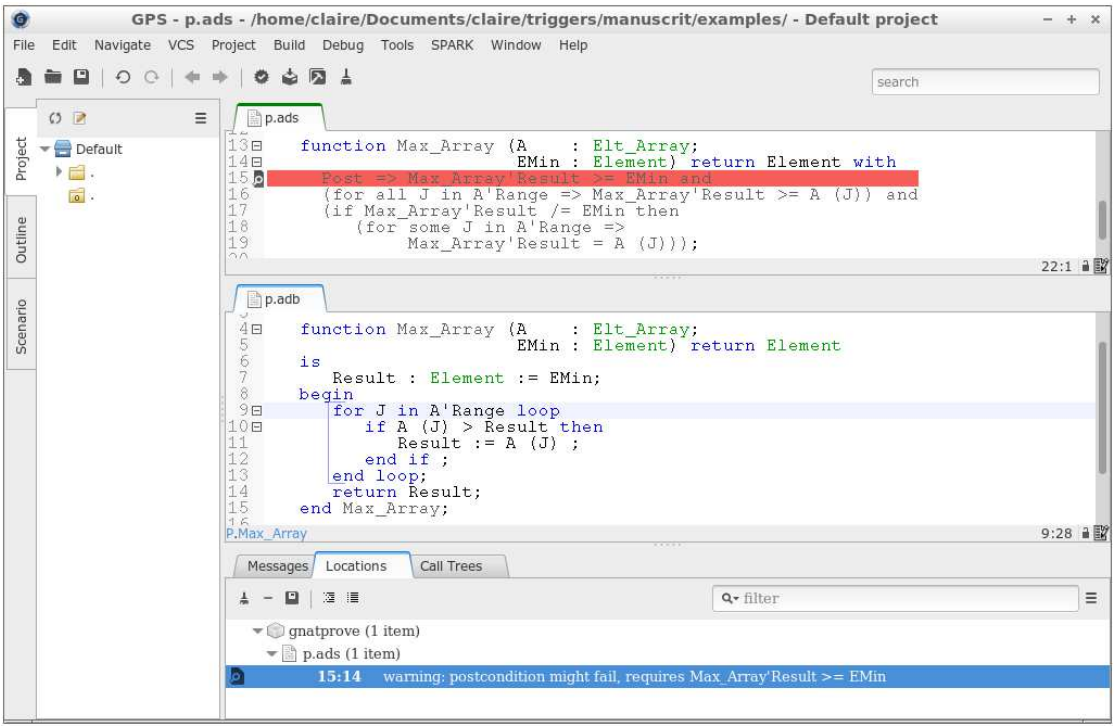


Figure 1.2: Formal verification of the `Max_Array` function

have been modified since the beginning of the loop. Here, the invariant simply states that the postcondition holds for the iterations up to index `J`:

```
function Max_Array (A    : Elt_Array;
                    EMin : Element) return Element
is
  Result : Element := EMin;
begin
  for J in A'Range loop
    pragma Loop_Invariant
      (Result >= EMin and
       (for all K in A'First .. J - 1 => Result >= A (K)) and
       (Result = EMin or
         (for some K in A'First .. J - 1 => Result = A (K))));
    if A (J) > Result then
      Result := A (J) ;
    end if ;
  end loop;
  return Result;
end Max_Array;
```

Thanks to this loop invariant, the SPARK 2014 tool can verify `Max_Array`. We see in Figure 1.3 that absence of run-time errors and conformance of `Max_Array` to its contract are successfully verified by GNATprove.

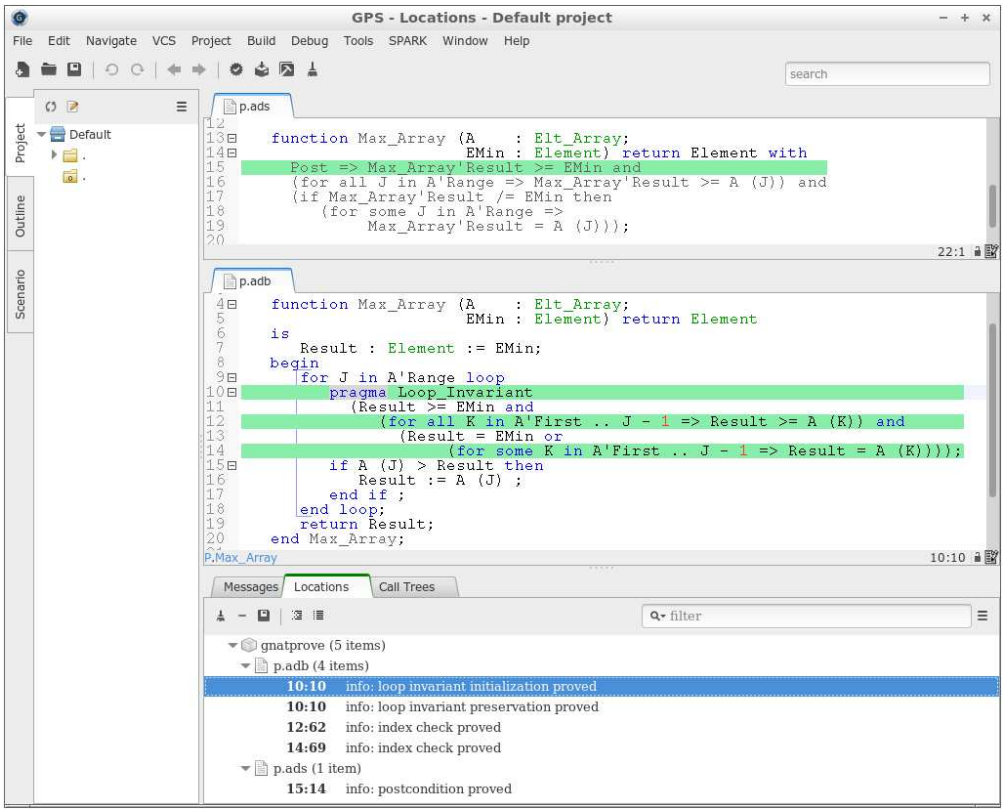


Figure 1.3: Formal verification of the `Max_Array` function with a loop invariant

1.1.3 Challenges in Deductive Verification of Programs

As critical software becomes more and more complex, it becomes difficult to gain enough confidence on an industrial software by testing only without prohibitively increasing the development costs. Formal methods are a possible alternative to tests for verification of critical software: instead of executing the source code, they proceed to mathematical analyses that establish guarantees about every possible execution of the software. These techniques have already been used successfully in the industry for example by Airbus [70] and Siemens [11]. Thanks to the support of corresponding certification authorities, formal methods, and in particular the B-method, are widely used in the railway domain. As formal methods were recently adopted by avionic certification authorities as an alternative to testing (see DO-333), a new market is open for them.

Deductive verification raises several scientific challenges. First, the specification language must be defined so that it is expressive enough to allow the specification of complex programs

while remaining amenable to static verification. Ease of use can also be improved by automatically generating annotations, for example loop invariants [13, 41, 48, 49, 57]. Then, there is ongoing work on efficiently extracting the logical propositions that express the soundness of the program, in particular in presence of aliasing as it is subject to combinatorial explosion [50]. A survey of what has been done for aliasing can be found for example in either of Asma Tafat's thesis [72] (in French) or Romain Bardou's thesis [4] (in English). Finally, efficient deductive verification requires efficient solvers to prove the validity of the logical propositions extracted from programs. Not all these formulas can be proved automatically as they are usually expressed in a combination of first-order logic and non-trivial theories such as integer arithmetic which is undecidable. It is still worthwhile to improve the solvers so that more can be proved. My thesis works toward this last goal.

1.2 Landscape in Deductive Verification of Programs

Popular techniques for formal verification of programs include bounded model checking, abstract interpretation, and deductive verification. The idea behind abstract interpretation [25] is to model every possible execution of a program using abstract domains that only describe the part of program states in which we are interested. The abstract interpreter uses this abstraction to compute an over-approximation of the values that a variable may take at a given program point. This over-approximation can then be used to verify that a safety property holds during every execution of the program. These abstract domains can be more or less precise depending on which property we want to check, precision being at the expense of efficiency and sometimes termination. For example, the values that can be taken by an integer variable can be modeled by a sign, an interval, or more complex structures like convex polyhedron used to model dependencies between several integer variables.

Bounded model checking [20] consists in systematic exploration of all possible executions of a program up to a predefined bound. The program is not executed but its execution is simulated so that the state of the program at each program point is determined for every input within the bounds. The model checker can then verify that a safety property holds for every execution of the program up to the bound. Since it needs to simulate every program execution, model checking techniques may become unpractical for large bounds on its input.

Deductive verification is based on the generation of mathematical formulas, called verification conditions or proof obligations, which express the correctness of the program.

1.2.1 Deductive Verification Tools for Mainstream Languages

There are several tools for deductive verification of C programs. The verifier VCC [21], developed at Microsoft Research, can be used to prove correctness of a concurrent C program annotated with contracts, such as pre- and postconditions and type invariants. The HAVOC [63] verifier, also developed at Microsoft Research, is specialized in the verification of systems software. In particular, it provides an accurate memory model for C accounting for low-level operations such as pointer arithmetic, address-of operations, and casts. The Frama-C [26] tool-suite, developed jointly by the LSL laboratory at CEA and the Toccata team at Inria Saclay, provides

several static analysis tools to verify a C program's correctness. In particular, it includes tools based on abstract interpretation and on deductive verification of C programs annotated using a formal specification language named ACSL. Both VCC and Frama-C are based on classical first-order logic. Verifast [43] is a verifier for single-threaded and multi-threaded C and Java programs annotated with contracts written in separation logic [64].

Apart from Verifast, there are several other tools for deductive verification of Java code. The Extended Static Checker for Java version 2 (ESC/Java2) [22], developed at IT University of Copenhagen, applies deductive verification to Java programs annotated with the Java Modeling Language (JML). The KeY system [1] allows formal verification of Java code specified in dynamic logic. An additional layer allows to alternatively specify programs using JML or the object constraint language that is part of the UML standard. The proof obligations generated for this verification are discharged by a theorem prover for first-order dynamic logic. The KIV tool [3], developed at University of Ulm, also verifies formal requirements specifications written in a higher-order algebraic specification language. In particular, it supports specifications coming from UML for Java. Deduction is based on a sequent calculus.

The language Spec# [7] is a formal language for contract-based specification of C# programs developed at Microsoft Research. It supports specification of object invariants in multi-threaded programs.

Some programming languages have been designed specifically to facilitate formal verification. For example, Dafny [52], developed at Microsoft Research, is an imperative object-based language designed for formal verification. It supports generic classes and dynamic allocation, but it also provides built-in specification constructs. These constructs include pre- and postconditions as well as some additional constructs to facilitate program annotation such as updatable ghost variables that are ignored by the compiler. As another example, F* [71], is a higher order, effectful programming language based on $F^\sharp$, that was designed with program verification in mind.

1.2.2 Verification Condition Generation

Tools for deductive verification of programs work by transforming program properties into logical formulas called *verification conditions* or *proof obligations*. The validity of those logical formulas, which implies the correctness of the program, can then be checked by automatic provers. One of the most common ways to extract verification conditions is the Weakest Precondition (WP) calculus [35, 59, 50] .

The WP calculus is based on Hoare logic [42] which associates both a precondition P and a postcondition Q to every statement S of the program. It infers a precondition $WP(S,Q)$ for a statement S of a program, its postcondition Q being given. The WP calculus generates the weakest precondition strong enough to enforce the validity of Q after the execution of S. Verifying a program S with a precondition P and postcondition Q then amounts to verifying that $WP(S,Q)$ is implied by the user-supplied precondition P. Sometimes, it is impossible to compute the weakest precondition for a statement. It is the case in particular for loop statements. In this case, it can be necessary to manually supply additional information to the calculus. Loop invariants are properties that can be used as the postcondition for a loop body. The WP calculus then simply checks that the loop invariant is inductive – if it holds at an iteration then it also holds at the next

iteration – and that it is true before the first iteration.

Generating verification conditions for mainstream languages is a complex task that can be alleviated by splitting it into two steps. The program and its annotations are translated into an intermediate language on which the verification conditions are generated. The transformation into the intermediate language is made easier by the fact that there are still primitive program constructs and, since the intermediate language is simpler than a classical language, verification condition generation is facilitated. What is more, the verification condition generator can then easily be shared between different tools.

Boogie [51], developed at Microsoft Research, is a non executable verification language. It mixes imperative components with pure, mathematical ones, such as logical quantification. It is used as a back-end for several verifiers such as Dafny, VCC, HAVOC, and Spec#. Boogie is also the name of the tool that generates verification conditions from Boogie programs. This tool can also infer some loop invariants.

The Why3 platform [15], developed in Toccata team, can be used as a front-end for provers and a middle-end for analysis of realistic programming languages. Its language WhyML is used as intermediate language by Frama-C and GNATprove. It is made of two parts. The logical part can be used to express mathematical properties. It includes type polymorphism, recursive algebraic data types, recursive function symbols, inductive predicates, and pattern matching. It is used to express the verification conditions that can be given to the provers. The program part is an ML-like programming language with annotations such as loop-variant and invariant and pre- and postconditions. The tool that generates verification conditions from WhyML programs is also called Why3.

1.2.3 Automatic Theorem Provers and SMT Solvers

Verification conditions coming from program verification can be discharged using either interactive theorem provers, automatic theorem provers, or Satisfiability Modulo Theory (SMT) solvers. Since we are interested in automated program verification, we only consider the latter two.

Automatic theorem provers are able to decide in a semi-complete way the satisfiability of a set of first-order formulas. Vampire [65] is an automatic theorem prover developed at Manchester University. It uses resolution and superposition calculus to decide the satisfiability of first-order propositional calculus with equality. The E theorem prover [68], developed at Technishe Universität München, also decides first-order logic with equality using resolution and superposition calculus. We can also cite iProver [46], based on an instantiation calculus, also developed at Manchester University, and SPASS [73], an automatic theorem prover for first-order logic with equality developed at Max Planck Institute for Computer Science that combines superposition calculus and splitting for explicit case analysis.

SMT solvers [9, 33] try to decide satisfiability of a set of, usually ground, formulas modulo a background theory. Most classical theories include:

- Equality with Uninterpreted Functions, also known as free functions, which is congruence closure over uninterpreted function symbols,

- Linear Arithmetic, that is, arithmetic with only $+$, $-$, and multiplication with a constant, over integers, rationals, or reals,
- Difference Arithmetic, where predicates must be of the form $x - y \leq c$,
- Non-Linear Arithmetics, with both $+$ and $\times$,
- Bit-Vectors, that is, arithmetics over machine integers, and
- Arrays, that are functional maps with an access function *get* and an update function *set*.

Solving an SMT problem amounts to combining an algorithm for resolution of satisfiability over propositional logic (SAT) with solvers for the various background theories. Most SMT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm for solving the SAT problem [28].

This algorithm works on a set of formulas including only logical variables combined using conjunction, disjunction, and negation operators. It tries to assign each logical variable to either true or false in a way that sets every formula in the input set to true. If there is such a truth assignment, the input set of formulas is said to be satisfiable. For example, the set of formulas $\{a_1 \vee a_2, \neg a_1 \vee a_2\}$ is satisfiable as it is true whenever a_2 is assigned to true. To determine that a formula is valid, that is, it is true for every value of its logical variables, a SAT solver checks that the negation of the formula is unsatisfiable. For example, $\neg(\neg a_1 \vee a_2) \vee \neg a_1 \vee a_2$ is valid as there is no truth values of a_1 and a_2 that satisfy its negation $(\neg a_1 \vee a_2) \wedge a_1 \wedge \neg a_2$.

The search for the appropriate truth values of the logical variables is done by systematic exploration. The algorithm chooses arbitrarily a truth value for a logical variable. Then it propagates this information in the propositional formulas from the input problem. If it finds a formula that is set to false by the chosen truth values of logical variables, it undoes the last choice and starts over. There are several improvements [61] to this naive search strategy. For example, when a conflict is found, it can be analyzed to determine which decisions were responsible for it so that the algorithm can go back at once to the last decision that is involved in the conflict.

Most SMT solvers provide multiple built-in background theories and it is often the case that the problem we want to solve includes elements coming from several of them. The solvers for these theories need therefore to be combined, and so, without losing termination nor completeness of procedures that decide satisfiability in these theories. There are several methods that can be used to combine theories. The most common one is the Nelson-Oppen combination method [60]. For example, assume we want to decide the satisfiability of the set of formulas $L = \{f(a) \not\approx f(b+c), b \approx a - c\}$ modulo the theories of linear integer arithmetics and uninterpreted functions. The idea is to separate the problem into smaller problems containing only function symbols coming from one of the theories. If one of the smaller problems is unsatisfiable then the original problem is unsatisfiable. Otherwise, the solvers for the smaller problems must find an agreement on a set of equivalence classes for the variables that are shared among theories. If there is such a set of equivalence classes then the problem is satisfiable, otherwise, it is unsatisfiable. On our example, we split L into two sets $L_1 = \{f(a) \not\approx f(d)\}$ and $L_2 = \{d \approx b + c, b \approx a - c\}$. Both L_1 and L_2 are satisfiable. As a consequence, the two solvers must agree on a set of equivalence classes for the set $\{a, d\}$. This is impossible. Indeed, if we

choose $a \approx d$ then L_1 is unsatisfiable in the theory of uninterpreted functions and if we choose $a \not\approx d$ then L_2 is unsatisfiable in the theory of linear integer arithmetics. As a consequence, L is unsatisfiable in the combination of uninterpreted functions and linear integer arithmetics.

The SMT solver Z3 [31], developed at Microsoft Research, is the default solver used by Boogie. It includes, among others, theories for linear real and integer arithmetic, fixed-sized bit-vectors, uninterpreted functions, and extensional arrays. It supports quantifiers and model generation. The open-source SMT solver CVC4 [8], the successor of CVC3 [38], is a joint project of New York University and University of Iowa. It includes theories for rational and integer linear arithmetic, arrays, tuples, records, inductive data types, bit-vectors, and uninterpreted functions. Like Z3, it supports quantifiers and model generation. VeriT [16] is an open-source SMT solver developed by University of Nancy, Inria and Federal University of Rio Grande do Norte. It supports uninterpreted functions and difference logic on real numbers and integers. It also includes quantifier reasoning capabilities through the integration of a first-order prover.

The verification conditions generated from SPARK 2014 programs are discharged by an open-source SMT solver named Alt-Ergo [14]. Alt-Ergo, developed in Toccata team, is dedicated to the proof of theorems coming from the verification of programs. It provides a built-in support for linear arithmetic over integers and rationals, non-linear arithmetic, functional arrays, enumerated data-types, record data-types, associative and commutative symbols, and fixed-size bit-vectors. It also supports first-order logic which is essential for program verification, as well as type polymorphism. In Alt-Ergo, theory combination is not done using the Nelson-Oppen method but rather using a variant of Shostak combination mechanism [27, 69] called CC(X) [24].

1.3 Problem and Contributions

1.3.1 First Order Axiomatizations as Decision Procedures

Verification conditions generated in program verification use a number of theories. Some of them are supported specifically by the prover used, we call them *background theories* of the prover. SMT solvers usually decide in a terminating and complete way the satisfiability of quantifier-free formulas in the theories they support. We say that they are *decision procedures* for the satisfiability of such formulas. Of course, not every useful theory is supported by every SMT solver and 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 [34], CVC3, CVC4, Z3, and Alt-Ergo support first-order logic. Since it is undecidable, any automated prover is at best semi-complete on first-order problems and even semi-completeness is unattainable when non-trivial background theories, like integer arithmetic, are involved. To improve the chance 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. Pattern matching modulo equality was introduced in the Stanford Pascal Verifier [58] and is now used in most SMT solvers supporting quantifiers. 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 [58], which we will consider in greater detail below. More recently, the same work has been done for specification of more complex data-structures [55, 19].

Example 1.1. Here is an axiomatization for the theory of non-extensional arrays as defined by Greg Nelson. 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 the term $get(set(a,i,e),j)$ 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.

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.

1.3.2 Overview of the Contributions

We propose in Chapter 2 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. 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. We give a fairly exhaustive axiomatization for imperative doubly-linked lists as an example and we provide completeness and termination proofs of this axiomatization in our framework.

In Chapter 3 we describe how a solver for first-order formulas with triggers can be built on top of a ground SMT solver. We show that, when given an axiomatization that meets the three conditions of soundness, completeness, and termination, such a solver yields a decision procedure for this axiomatization. Although good in theory, in practice, we show on some simple examples that this black-box extension mechanism results in poor performances and therefore does not yet give a practical way for adding a new theory to an SMT solver, calling for a better integrated approach.

We consider in Chapter 4 the well-known Abstract DPLL Modulo Theory framework [61], 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 the 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 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. To validate our approach, we compare, on several case studies, the efficiency of a first-order axiomatization with and without hand-written triggers, using the built-in quantifier handling in SMT solvers. In Chapter 5, we describe a theory for mathematical sets. We then use it for the verification of mathematical formulas coming from the development of programs using the B method [67] through Atelier B. In Chapter 6, we describe a theory that models some aspects of vectors in SPARK 2014. We design an axiomatization for this theory and prove that it is sound,

complete, and terminating on SPARK programs.

2 First-Order Logic with Triggers

Contents

2.1 Formalization 15
2.1.1 Preliminary Notions 16
2.1.2 Logic with Triggers (Syntax and Semantics) 17
2.1.3 Relation with Traditional First-Order Logic 18
2.1.4 Soundness and Completeness 21
2.1.5 Termination 22
2.2 Case Study: Imperative Doubly-Linked Lists 26
2.2.1 Presentation of the Theory 27
2.2.2 Description of the Axiomatization 28
2.2.3 Proofs of Soundness, Completeness, and Termination 30
2.2.4 Assessment of Adequacy with Respect to Existing Trigger Heuristics 33
2.3 Designing Terminating and Complete Axiomatizations 35
2.3.1 Proving Termination of an Axiomatization 35
2.3.2 Designing a Complete Axiomatization 40
2.3.3 An Automatable Debugger for Completeness 42
2.4 Conclusion 43

In this chapter, we introduce a formalization for a first-order logic with a notation for triggers that restrict instantiation. In this logic, we then define properties – soundness, completeness, and termination – that a set of formulas with triggers should guarantee in order to provide a decision procedure for a theory T' that extends a background theory T. We assess in Section 2.2 the usability of these definitions on an axiomatization for imperative doubly-linked lists. We show that this axiomatization is sound, complete, and terminating following our definitions. Finally, in Section 2.3, we give some advices for designing a complete and terminating axiomatization.

2.1 Formalization

In first-order SMT solvers, triggers are used to favor instantiation of universally quantified formulas with "known" terms that have a given form. Intuitively, a term is said to be known when it appears in a ground fact assumed by the solver. Here is an example of a formula with a trigger in SMT-LIB version 2 [10] notation:

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

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

`(= (f t) c)` *holds 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 [40]:

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

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

2.1.1 Preliminary Notions

We work in classical untyped first-order logic and assume the standard notation for first-order formulas and terms. We denote formulas with letters φ and ψ, literals with l, terms with s and t, and substitutions with σ and μ. Other notational conventions will be introduced in the course of the text.

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

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

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

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

We say that a formula is *closed* if it has no free variables, and that a term, literal, or formula is *ground* if it has no free variables and no quantifiers. We use $\mathscr{T}(t)$, $\mathscr{T}(l)$, $\mathscr{T}(S)$ to denote the set of all terms that occur in, respectively, a term t, a literal l, or a set of terms or literals S.

We reason modulo some background theory T, which we assume to be fixed for the rest of this section. In the simplest case, T can be the theory of Equality With Uninterpreted Functions (EUF). We assume that the signature of T contains at least one constant symbol to allow constructing the Herbrand universe and can be extended at will with uninterpreted function symbols to allow Skolemization. We use the following definition for atoms, were $\approx$ is the symbol for equality:

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

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

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

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

Definition 2.1 (T-satisfiability)**.** We say that a first-order formula φ is valid in a model L modulo T, written $L \vDash_T \varphi$ if φ is valid in L and L is also a model of T. If a first-order formula φ has a model modulo T then we say that it is T-satisfiable or, equivalently, that is is satisfiable modulo T.

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

2.1.2 Logic with Triggers (Syntax and Semantics)

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

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

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

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

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

Definition 2.2 (World modulo T)**.** We call *world* a T-satisfiable set of ground literals. A world L is *inhabited* if there is at least one term occurring in it, i.e. $\mathscr{T}(L)$ is non-empty. A world L is *complete* if for any ground literal l in the signature of T, either $l \in L$ or $\neg l \in L$.

Definition 2.3 (Known term modulo T)**.** A term t is *known* in a world L if and only if there is a term $t' \in \mathscr{T}(L)$ such that $L \models_T t \approx t'$

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

Definition 2.4 (Truth value modulo T)**.** Given a world L and a closed formula φ, we define what it means for φ to be *true* in L, written $L \rhd_T \varphi$, by induction on φ as follows:

$L \rhd_T l$	$L \models_T l$ and $L \cup known(\mathscr{T}(L)) \models_T known(\mathscr{T}(l))$
$L \rhd_T \varphi_1 \vee \varphi_2$	$L \rhd_T \varphi_1$ 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 φ is *false* in L whenever $L \rhd_T \neg\varphi$. We call φ *feasible* if there exists a world in which φ is true.

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

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

Definition 2.5 (Model modulo T)**.** A world L is said to be a *model* of a closed formula φ whenever L is complete and $L \rhd_T \varphi$. We call φ *satisfiable* if it has a model.

2.1.3 Relation with Traditional First-Order Logic

Let φ be a closed formula and φ' be φ where all triggers are replaced with implications and all witnesses with conjunctions. As noted above, in any complete world L, $L \rhd_T \varphi$ if and only if

$L \rhd_T \varphi'$. Moreover, $L \rhd_T \varphi'$ if and only if $L \models_T \varphi'$. Indeed, since every ground term is known in a complete world, the truth value of quantified formulas and ground literals in our logic coincides with that in the usual first-order logic. Thus, L is a model of φ if and only if it is a Herbrand model of φ' in T. Consequently, φ is satisfiable in the sense of Definition 2.5 if and only if φ' is T-satisfiable, which justifies our reuse of the term.

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

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

It is thus all the more remarkable that the following implication holds in the background theory EUF (Equality with Uninterpreted Functions):

Theorem 2.1. *Let φ be a closed first-order formula without triggers and witnesses. Let L be an inhabited world such that $L \rhd_{\mathrm{EUF}} \varphi$. Then φ is satisfiable in first-order logic with equality (and therefore has a model in the sense of Definition 2.5).*

Proof. We start with some preliminaries. From now on and until the end of this section, we write $\rhd$ for $\rhd_{\mathrm{EUF}}$ and $\models$ for $\models_{\mathrm{EUF}}$. We define an encoding $\| \ \|$ that explicitly restricts instantiation of first-order formulas in negative normal form to known terms.

$$\|\varphi_1 \wedge \varphi_2\| \triangleq \|\varphi_1\| \wedge \|\varphi_2\| \qquad \|\varphi_1 \vee \varphi_2\| \triangleq \|\varphi_1\| \vee \|\varphi_2\|$$
$$\|\forall x.\varphi\| \triangleq \forall x.known(x) \rightarrow \|\varphi\| \qquad \|\exists x.\varphi\| \triangleq \exists x.known(x) \wedge \|\varphi\|$$
$$\|l\| \triangleq known(\mathscr{T}(l)) \wedge l$$

Lemma 2.1. *Let φ be a closed first-order formula without triggers and witnesses. If there is an inhabited world L such that $L \rhd \varphi$, then there is a Herbrand model L' of $\|\varphi\|$ such that, for at least one ground term ω, $L' \models known(\omega)$.*

Proof. Let us consider the following set of literals L':

$$L \cup known(\mathscr{T}(L)) \cup \{\neg known(t) \mid L \cup known(\mathscr{T}(L)) \not\models known(t)\}$$

Since L is inhabited, for at least one ground term ω, $L' \models known(\omega)$. We show that the set of literals L' is satisfiable. By contradiction, assume L' is unsatisfiable. This can only be the case if there is a set of equalities $t_1 \approx t'_1 \ldots$ with $t_i \in \mathscr{T}(L)$ and $t'_i \in \{\neg known(t) \mid L \cup known(\mathscr{T}(L)) \not\models known(t)\}$ such that $L \models t_1 \approx t'_1 \vee \ldots$. In the theory EUF, if a set of literals implies a disjunction of equalities then it implies one of them. As a consequence, there must be an equality $t \approx t'$ such that $t \in \mathscr{T}(L)$ and $t' \in \{\neg known(t) \mid L \cup known(\mathscr{T}(L)) \not\models known(t)\}$, which is impossible.

We show by structural induction on φ that, for every formula φ 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, $L' \models \exists x.\mathit{known}(x) \wedge \|\varphi\|$.

□

Using Lemma 2.1, we can reformulate our theorem:

Lemma 2.2. *Let φ 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 ω, $L \models \mathit{known}(\omega)$. Then φ is EUF-satisfiable.*

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 ω 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 $L \cup \mathit{known}(\mathscr{T}(L)) \models \mathit{known}(t_1) \wedge \mathit{known}(t_2)$. Indeed, if t_1 is not known in L modulo EUF then we can choose any value for it and, in particular, the value associated to 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 that $L_1 \cup \{t_1 \approx \omega\} \cup \mathit{known}(\mathscr{T}(L_1)) \cup \mathit{known}(t_1) \not\models \mathit{known}(t_2)$. 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 $L_1 \cup \mathit{known}(\mathscr{T}(L_1)) \vDash \mathit{known}(t)$, there is $t' \in \mathscr{T}(L_1)$ such that $L_1 \vDash t \approx t'$. 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, $L_1 \cup \mathit{known}(\mathscr{T}(L_1)) \nvDash \mathit{known}(f(t'_1 \dots t'_n))$ and, for every i in $1..n$, $L_1 \cup \mathit{known}(\mathscr{T}(L_1)) \vDash \mathit{known}(t'_i)$. 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 ψ without triggers and witnesses such that $L \vDash \|\psi\|$, $L_2 \vDash \psi$ by structural induction over ψ.

- $L \vDash \|l\|$. 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 $L_2 \vDash \psi_1 \vee \psi_2$.
- $L \vDash \|\forall x.\psi\| = \forall x.\mathit{known}(x) \to \|\psi\|$. 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, $L \vDash \mathit{known}(\mathscr{T}(t'))$. We then have $L \vDash \|\psi\|\,[x \leftarrow t']$ and, by immediate induction over ψ, $L \vDash \|\psi[x \leftarrow t']\|$. By induction hypothesis, we have $L_2 \vDash \psi[x \leftarrow t']$. Therefore, $L_2 \vDash \psi[x \leftarrow t]$ and $L_2 \vDash \forall x.\psi$.
- $L \vDash \|\exists x.\psi\| = \exists x.\mathit{known}(x) \wedge \|\psi\|$. Since every ground term is equal to a constant in L, there is a constant $c \in \mathscr{T}(L)$ such that $L \vDash \mathit{known}(c)$ and $L \vDash \|\psi\|\,[x \leftarrow c]$. By immediate induction over ψ, $L \vDash \|\psi[x \leftarrow c]\|$. By induction hypothesis, $L_2 \vDash \psi[x \leftarrow c]$ and $L_2 \vDash \exists x.\psi$.

□

2.1.4 Soundness and Completeness

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

Definition 2.6 (Soundness modulo T)**.** An axiomatization W is *sound* with respect to T' if, for every T'-satisfiable set of ground literals L, $W \cup L$ is T-satisfiable.

Definition 2.7 (Completeness modulo T)**.** An axiomatization W is *complete* with respect to T' if, for every set of ground literals L such that $W \cup L$ is feasible, L is T'-satisfiable.

Remark 2.1. Note that the two definitions of soundness and completeness are not symmetrical. Indeed, we want a solver to be allowed to stop whenever it has found a world L in which the axiomatization is true while returning satisfiable. In particular, we want the solver to instantiate universal quantifiers using only terms of L and to ignore formulas protected by a trigger l if l is not true in L.

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 2.1. The proof that the set of axioms W_{array} shown in Example 1.1 is complete modulo EUF closely resembles the proof by Greg Nelson in [58]. We do not give this proof here but show that simpler or more "intuitive" variants of that axiomatization are incomplete.

- Let W^1_{array} be W_{array} where the trigger in the first axiom is replaced by $get(set(a,i,e),i)$. Consider the set of literals $L_1 = \{set(a,i,e_1) \approx set(a,i,e_2), e_1 \not\approx e_2\}$. It is unsatisfiable in the theory of arrays since we have both $get(set(a,i,e_1),i) \approx e_1$ and $get(set(a,i,e_2),i) \approx e_2$. Still, $W^1_{array} \cup L_1$ is true in the world L_1, since we have no term in L_1 to match the trigger.

- Let W^2_{array} be W_{array} without the second axiom. Consider the set of ground literals $L_2 = \{get(set(a,i_1,e),j) \not\approx get(set(a,i_2,e),j), i_1 \not\approx j, i_2 \not\approx j\}$. It is unsatisfiable in the theory of arrays since $get(set(a,i_1,e),j) \approx get(a,j)$ and $get(set(a,i_2,e),j) \approx get(a,j)$. Yet, $W^2_{array} \cup L_2$ is true in the world $L_2 \cup \{get(set(a,i_1,e),i_1) \approx e, get(set(a,i_2,e),i_2) \approx e\}$.

- Let W^3_{array} be W_{array} without the third axiom. Consider the set of ground literals $L_3 = \{set(a_1,i,e) \approx set(a_2,i,e), i \not\approx j, get(a_1,j) \not\approx get(a_2,j)\}$. It is unsatisfiable in the theory of arrays since $get(set(a_1,i,e),j) \approx get(a_1,j)$ and $get(set(a_2,i,e),j) \approx get(a_2,j)$. Still $W^3_{array} \cup L_3$ is true in the world $L_3 \cup \{get(set(a_1,i,e),i) \approx e, get(set(a_2,i,e),i) \approx e\}$.

2.1.5 Termination

Once it has been established that a given set of axioms with triggers is sound and complete for our theory, we must show that the solver equipped with this axiomatization terminates on any ground satisfiability problem. We call such axiomatizations *terminating* and the 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 Chapter 4. In Section 2.2, we prove that a non-trivial axiomatization of imperative doubly-linked lists is terminating according to this definition. Finally, in Section 4.2.2, 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.

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

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

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

Lemma 2.3. *Skolemization preserves feasibility and satisfiability.*

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

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

Skolemization 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 Chapters 3 and 4 do not require Skolemized axiomatizations.

Clausification:

Definition 2.8 (Pseudo-clause)**.** We say that a formula is a *pseudo-literal* if it is a literal l, a trigger $[l]C$, a witness $\langle l\rangle C$, or a universally quantified formula $\forall x.C$, where C is a disjunction of pseudo-literals, called *pseudo-clause*.

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

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

The adversary's moves are represented by so-called *truth assignments*. Intuitively, given a current set of assumed facts, a truth assignment is any set of further facts that the solver may assume using only propositional reasoning, without instantiation. Once this completion is done,

we may choose an assumed universal formula and a known term to perform instantiation, allowing for the next stage of completion and so on. A tree that inspects all possible truth assignments for certain instantiation choices (i.e. all possible adversary's responses to a particular strategy of ours) is called *instantiation tree*. An axiomatization is terminating if for any ground satisfiability problem we can construct a finite instantiation tree.

To avoid applying substitutions, we use *closures*. A closure is a pair $\varphi \cdot \sigma$ made of a pseudo-literal φ and a substitution σ mapping every free variable of φ to a ground term. We write $\varphi\sigma$ for the application of σ to φ, and $\varnothing$ for the empty substitution. If two substitutions σ and σ' have the same domain D, we write $\sigma \doteq\!\approx \sigma'$ 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 $\{\varphi \cdot \sigma' \mid \varphi \in C \text{ and } \sigma' \text{ is } \sigma \text{ restricted to the free variables of } \varphi\}$. Disjunctive sets of closures are sometimes called *theory clauses*, as they come from the axiomatization of our theory of interest.

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

Definition 2.9. Given a set of theory clauses V, we define the set of literals $\lfloor V \rfloor \triangleq \{l\sigma \mid l \cdot \sigma \text{ is a unit clause in } V\}$.

Definition 2.10 (Truth assignment modulo T)**.** A *truth assignment* of a set of theory clauses V is any set of theory clauses A that can be constructed starting from V by exhaustive application of the following rules:

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

We say that a truth assignment A is *T-satisfiable* if the set of literals $\lfloor A \rfloor$ is T-satisfiable. A T-satisfiable truth assignment A is said to be *final* if every possible instantiation is *redundant* in A, that is for every closure $\forall x.C \cdot \sigma$ in A and every term $t \in \mathscr{T}(\lfloor A \rfloor)$, there is a ground substitution σ' such that $C \cdot \sigma' \in A$ and $\lfloor A \rfloor \vDash_T (\sigma \cup [x \mapsto t]) \doteq\!\approx \sigma'$. 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 assignments only decomposes formula and do not introduce new terms, any finite set of theory clauses has a finite number of possible truth assignments.

Remark that, in terms of solver implementation, this definition means that, while we require the solver to eliminate triggers and witnesses eagerly, it is permitted to postpone the decision over disjunctions. Such postponing corresponds to adding no closures at all in the first case of the definition above. In this way, the solver is not urged to make choices which it will have to backtrack later, and can instead wait until subsequent instantiations reduce the choice space.

Definition 2.11 (Instantiation tree modulo T)**.** An *instantiation tree* of a set of pseudo-clauses W is any tree where the root is labeled by $W \cdot \varnothing$, every node is labeled by a set of theory clauses, and every edge is labeled by a non-final truth assignment such that:

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

We have two notions of termination; the strongest one can be applied to less axiomatizations of theories but leads to easier proof of termination of solvers' implementation:

Definition 2.12 (Strong Termination modulo T)**.** A set of pseudo-clauses W is *strongly terminating* if, for every finite set of ground literals L, every instantiation tree of $W \cup L$ is finite.

Definition 2.13 (Weak Termination modulo T)**.** A set of pseudo-clauses W is *weakly terminating* or simply *terminating* if, for every finite set of ground literals L, $W \cup L$ admits at least one finite instantiation tree.

Remark 2.2. In the definition of weak termination, we only require that W has one finite instantiation tree and not that every instantiation tree of W is finite. Indeed, we rely on the solver's implementation to be fair and to do all the instances required by one particular finite instantiation tree. That it may also do other instances is not a problem as, in a finite amount of time, it will either reach an unsatisfiable state or all these unaccounted instances will be redundant.

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

Example 2.2. The proof of termination of the theory of arrays described in Examples 1.1 and 2.1 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 $set(a,i,e)$ is known in A, that is $\lfloor A \rfloor \cup known(\mathscr{T}(A)) \vDash_T known(set(a,i,e))$. Then, for every term t in $\mathscr{T}(get(set(a,i,e),i) \approx e)$, $\lfloor A \rfloor \cup \{get(set(a,i,e),i) \approx e\} \cup known(\mathscr{T}(\lfloor A \rfloor)) \vDash_T known(t)$. Indeed, since $set(a,i,e)$ is known in A, 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 axioms. Therefore, every instantiation tree of $W_{array} \cup L$ is finite.

Example 2.3. 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. For example, this could be used to model conversion between mathematical integers and machine 32 bit integers. The axiomatization contains five function symbols. If $valid_E(x)$ (resp. $valid_e(x)$) returns `t` (for 'true") 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 $\mathit{unfit}_{E\to e}(x)$ returns $\mathtt{t}$ when x cannot be converted to e.

$$W_{conv} = \left\{ \begin{array}{l} \forall x.[\mathit{valid}_E(x) \approx \mathtt{t}]\ \mathit{valid}_e(\mathit{conv}_{E\to e}(x)) \approx \mathtt{t} \vee \mathit{unfit}_{E\to e}(x) \approx \mathtt{t} \\ \forall x.[\mathit{valid}_e(x) \approx \mathtt{t}]\ \mathit{valid}_E(\mathit{conv}_{e\to E}(x)) \approx \mathtt{t} \\ \forall x.[\mathit{valid}_E(x) \approx \mathtt{t}, \mathit{valid}_e(\mathit{conv}_{E\to e}(x)) \approx \mathtt{t}]\ \mathit{conv}_{e\to E}(\mathit{conv}_{E\to e}(x)) \approx x \\ \forall x.[\mathit{valid}_e(x) \approx \mathtt{t}, \mathit{conv}_{e\to E}(x)]\ \mathit{conv}_{E\to e}(\mathit{conv}_{e\to E}(x)) \approx x \end{array} \right\}$$

The axiomatization W_{conv} is not strongly terminating. Indeed, if we start from a set of literals $L = \{\mathit{valid}_E(c)\}$, we can create an infinite instantiation tree by applying in priority the first two axioms. We start by instantiating the first axiom with c. In branches labeled by truth assignments containing $\mathit{valid}_e(\mathit{conv}_{E\to e}(x)) \approx \mathtt{t} \cdot [x \mapsto c]$, we instantiate the second axiom with $\mathit{conv}_{E\to e}(c)$ and so on. As there is an infinite branch in which we can always apply one of the first two axioms, W_{conv} is not strongly terminating.

Still, the axiomatization W_{conv} is weakly 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 $\mathit{conv}_{E\to e}(\mathit{conv}_{e\to E}(t))$ 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 $\mathit{unfit}_{E\to e}(t)$ with t already in $\mathscr{T}(A)$. By contradiction, assume that there is a non-redundant instance of the first axiom with a term t such that $A \vDash_T \mathit{valid}_E(t) \approx \mathtt{t}$. By construction of A, t cannot occur in L, otherwise, the instance has been already produced in the first step. As a consequence, $\mathit{valid}_E(t) \approx \mathtt{t}$ was deduced using the second axiom and there is $t' \in \mathscr{T}(A)$ such that $A \vDash_T \mathit{valid}_e(t') \approx \mathtt{t}$ and $A \vDash_T t \approx \mathit{conv}_{e\to E}(t')$. Therefore, the last axiom of W_{conv} has been instantiated with t' in the second step and $A \vDash_T \mathit{conv}_{E\to e}(\mathit{conv}_{e\to E}(t')) \approx t'$ and thus $A \vDash_T \mathit{valid}_e(\mathit{conv}_{E\to e}(t))$. Consequently, the result of an instance of the first axiom with t can only produce new terms of the form $\mathit{unfit}_{E\to e}(t)$ 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.

2.2 Case Study: Imperative Doubly-Linked Lists

In this section, we give a rather large axiomatization as an example (more than 50 axioms). We assume that the background theory T is the combination of integer linear arithmetic and booleans or any conservative extension of it. The axiomatized theory T' contains a definition of imperative doubly-linked lists with a definition for iterators (named *cursors*), an equality function, several

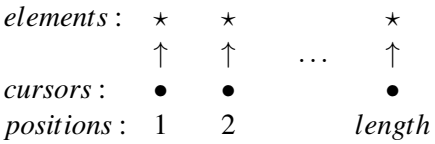
modification functions and so on. We prove that this axiomatization is sound, complete and strongly terminating. It is inspired by the API of lists in the Ada standard library [36].

2.2.1 Presentation of the Theory

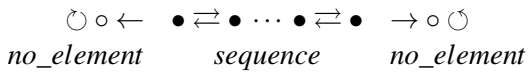
Lists are ordered containers of elements on which an equivalence is defined using the function $equal\_elements(e_1 : element\_type, e_2 : element\_type) : bool$. We represent imperative lists of elements as pairs of:

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

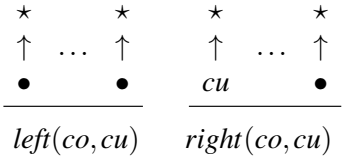
The iterative part of an imperative list co is modeled by an integer $length(co : list) : int$ representing the length of the sequence together with a total function $position(co : list, cu : cursor) : int$ so that, for every cursor cu, $position(co, cu)$ returns the position of cu in co if it appears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



Thanks to this description, we can define several other functions. $has\_element(co : list, cu : cursor) : bool$ returns true if and only if cu appears in the iterative part of co and $is\_empty(co : list) : bool$ returns true if co is an empty list. The functions $last(co : list) : cursor$, $first(co : list) : cursor$, $previous(co : list, cu : cursor) : cursor$ and $next(co : list, cu : cursor) : cursor$ 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 $no\_element$:



We define two functions $left(co : list, cu : cursor) : list$ and $right(co : list, cu : cursor) : list$, that are used to split the list. If cu appears in the iterative part of co or is $no\_element$, $left(co, cu)$ (resp. $right(co, cu)$) returns the prefix (resp. suffix) of co that stops before (resp. starts at) 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 $find(co : list, e : element\_type, cu : cursor) : cursor$. 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 $first(co)$. $contains(co : list, e : element\_type) : bool$ is true if and only if *co* contains an element equivalent to *e*.

We add a notion of equality on list: $equal\_lists(co_1 : list, co_2 : list)$ 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*. They are represented as predicates relating the value of the list before and after the modification as there may be several possible results for the modification. $insert(co : list, cu : cursor, e : element\_type, r : list) : bool$ 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 $delete(co : list, cu : cursor, r : list) : bool$ is true then *r* can be obtained by deleting the cursor *cu* from the list *co*. If $replace\_element(co : list, cu : cursor, e : element\_type, r : list) : bool$ is true then *r* can be obtained by replacing the element associated to *cu* in *co* by *e*.

2.2.2 Description of the Axiomatization

In this section, we formalize the theory of lists by translating it into a first-order axiomatization. We also add triggers and witnesses to make it adequate for integration in our framework. We only give a few axioms. The whole axiomatization is available in Appendix A.1.

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:

$$\forall co : list, cu : cursor.[position(co,cu)] \\ length(co) \geq position(co,cu) \wedge position(co,cu) \geq 0$$

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 only described on their domain of definition. We only present *find*. The function $find\_first(co : list, e : element\_type) : cursor$ 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:

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

FIND_FIRST_FIRST:

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

FIND_FIRST_ELEMENT:

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

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:

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

DELETE_LENGTH:

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

DELETE_POSITION_BEFORE:

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

DELETE_POSITION_AFTER:

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

DELETE_POSITION_NEXT:

$$\begin{array}{l}\forall co_1, co_2 : list, cu : cursor.[delete(co_1, cu, co_2)] \\ \quad delete(co_1, cu, co_2) \approx \mathtt{t} \rightarrow \langle next(co_1, cu) \rangle \top\end{array}$$

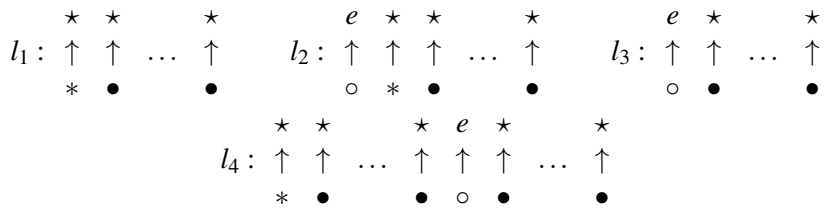
DELETE_ELEMENT:

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

Note that the axiom DELETE_POSITION_NEXT only introduces a new known term. This term is needed so that the axiomatization is complete. Indeed, consider the following set of ground formulas G:

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

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



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

2.2.3 Proofs of Soundness, Completeness, and Termination

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

Theorem 2.2. *The axiomatization in Section 2.2.2 is strongly terminating, sound, and complete with respect to thesame axiomatization without the triggers and witnesses.*

2.2.3.1 Proof of Strong 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_type* can be created, we can deduce that the axiomatization is strongly 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 all arguments. 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 FIND_FIRST, FIND_OTHERS, CONTAINS_DEF, INSERT_NEW, INSERT_NEW_NO_ELEMENT and DELETE_POSITION_NEXT all contain a literal in which there is a sub-term t of type *cursor* that does not appear in the trigger. Also, there is an existentially quantified cursor variable in EQUAL_LISTS_INV, which amounts to a term of type cursor after Skolemization. All these cases can be solved with the same arguments as for the terms of type list. Indeed, the symbols $contains(co,e)$, $find(co,e,cu)$, $insert(co_1,cu,e,co_2)$, $delete(co_1,cu,co_2)$, 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. The remaining axioms FIND_FIRST_ELEMENT and EQUAL_LISTS_INV 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.

2.2.3.2 Proof of Soundness

We show that, if a set of literals G has a model in the axiomatization without triggers and witnesses then there is a total model of G and the axiomatization. If I is a model of a set of literals G in the axiomatization without triggers, we define $L = \{l \mid I(l) = \top\}$. By construction of L, L is a total model of G. Since L is complete, for every axiom φ of the axiomatization, $L \rhd_T \varphi$.

2.2.3.3 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 2.4. *Let L be a world in which the axiomatization is true and t_1 and $t_2 \in \mathscr{T}(L)$ be two terms of type integer. If $L \not\models_T t_1 \not\approx t_2$ then 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 $L \not\triangleright_T l$, $t_1, t_2 \in \mathscr{T}(L)$ have type integer and $L \not\models_T t_1 \not\approx t_2$ then $L \cup t_1 \approx t_2 \not\triangleright_T l$. 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 $L$for the axiomatization without triggers and witnesses. Since $L \triangleright_T G$, 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 $L \cup \{i \approx i \mid i \text{ is an integer constant}\}$. 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 2.4, the axiomatization is still true in L.

For every term co of type list in L, if the term $length(co)$ 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 $0 \leq i < j \leq length(co)$ such that either $i = 0$ or there is a term cu of type cursor in $\mathscr{T}(L)$ such that $L \models_T position(co, cu) \approx i$ and, for all k such that $i < k \leq j$, there is no term cu of type cursor in L such that $L \models_T position(co, cu) \approx k$. 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 straightforward. For example, here are the rules for deletion.
For every $co[i, j] \in eq(z)$:

$$
\begin{aligned}
L \models_T delete(co, cu, co') \approx \mathtt{t} \text{ and } L \models_T j < position(co, cu) &\rightarrow & co'[i, j] &\in eq(z) \\
L \models_T delete(co, cu, co') \approx \mathtt{t} \text{ and } L \models_T j > position(co, cu) &\rightarrow & co'[i-1, j-1] &\in eq(z) \\
L \models_T delete(co', cu, co) \approx \mathtt{t} \text{ and } L \models_T j < position(co', cu) &\rightarrow & co'[i, j] &\in eq(z) \\
L \models_T delete(co', cu, co) \approx \mathtt{t} \text{ and } L \models_T j \geq position(co', cu) &\rightarrow & co'[i+1, j+1] &\in eq(z)
\end{aligned}
$$

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 we have both $L \vDash_T position(co, cu) > 0$ 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 $co'[i', j'] \in eq(co[i, j])$ 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 co of type list and every term of type cursor cu of L for which $position(co, cu)$ is not yet known in L, we can add $position(co, cu) \approx 0$ to L. We can check straight-forwardly that the axiomatization is still true in L. We now have a model in the axiomatization without triggers 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 $element(co, cu)$ 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 2.3. 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 anymore. For example, consider the finite set of literals L with n constants of type element $e_1 \dots e_n$ containing $position(co, find\_first(co, e_i)) > n$ for every $i \in 1..n$.

We have constructed a model for L the axiomatization described in Section 2.2.2 without the triggers. As a consequence, the axiomatization from Section 2.2.2 is complete for this theory.

2.2.4 Assessment of Adequacy with Respect to Existing Trigger Heuristics

In this section we assess the concordance between our semantics for formulas with triggers and the heuristics used for trigger instantiation in off-the-shelf SMT solvers. 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*. The examples have been written in WhyML so that several provers can be used to discharge the logical formulas arising from their verification using the Why3 tool.

	Alt-Ergo 0.95.2		Z3 3.2		CVC4 1.2	
With Triggers	Yes	No	Yes	No	Yes	No
`test_false`	**0.00**	135.06	TO	TO	TO	TO
`test_delete`	**3.04**	145.60	0.03	0.06	**0.86**	TO
`test_insert`	**34.31**	TO	0.07	0.08	**3.9**	TO
`double_size`	**13.67**	TO	**11.17**	TO	**13.3**	TO
`filter`	**12.24**	TO	**0.56**	TO	**27.05**	TO
`my_contain`	**0.76**	TO	**0.11**	880.05	**0.65**	449.31
`my_find`	**7.46**	TO	TO	TO	**9.33**	TO
`map_f`	**5.68**	TO	**10.42**	TO	Error	TO

Figure 2.1: Time (in seconds) needed to answer correctly on all tests with a timeout of 1000s with and without triggers in the axiomatization. Bold figures point to notably better results for each prover. We do the tests with Alt-Ergo 0.95.2, Z3 3.2, and CVC4 1.2.

```
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 A.2. 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 postcondition is true and that the iteration can be resumed after the insertion. The loop invariant states that:

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

```
let double_size (li : ref list) (e : element_type) =
  requires { not (is_empty !li) }
  ensures  { length !li = 2 * (length (old !li)) }
  let c = ref (first !li) in
  'Loop_Entry:
```

```
while has_element !li !c do
  invariant {
    (((has_element (at !li 'Loop_Entry) !c /\ has_element !li !c)
      \/ !c = no_element) /\
      length (left !li !c) = 2 * (length (left (at !li 'Loop_Entry) !c)) /\
      equal_lists (right !li !c) (right (at !li 'Loop_Entry) !c))
  }
  insert li !c e;
  c := next !li !c
done
```

We see on results from Figure 2.1 that, in general, SMT solvers are far less efficient when triggers are removed from the axiomatization. This shows that, even if triggers can be inferred by SMT solvers, efficient handling of first-order formulas usually requires user guidance in this choice. This also validates the fact that our semantics is rather consistent with the way triggers are handled in first-order SMT solvers.

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

2.3 Designing Terminating and Complete Axiomatizations

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

2.3.1 Proving Termination of an Axiomatization

Here we list several techniques that can be attempted to do a proof of termination of a given axiomatization W. First, we can define a property stronger than termination which can be checked in an automatable way. The idea is to over-approximate the set of terms that can be created by the axiomatization. For this definition, we need to compute, for any set of ground literals L, the literals that will eventually be entailed in every truth assignment of an instantiation tree of $W \cup L$. More precisely, we consider the set of all literals l for which there is an instantiation tree Z of $W \cup L$ such that l is entailed by every truth assignment at a certain depth in Z. This set is not computable and can be infinite, but we can take some finite under-approximation of it, for example, by fixing the depth, *i.e.*, the number of consecutive instances on a branch. The greater the chosen depth, the more precise our approximation of the set of generated terms and the costlier the computation. In what follows, we assume some fixed depth n:

Definition 2.14. Let L be a set of ground literals. Let $F_1, \ldots, F_m$ be the sets of theory clauses that can be constructed from $(W \cup L) \cdot \varnothing$ in the following way:

- if $[l]C \cdot \sigma \in F$ and $\lfloor F \rfloor \rhd_T l$ then add $C \cdot \sigma$ to F,
- if $\langle l \rangle C \cdot \sigma \in F$ then add $l \cdot \sigma$ and $C \cdot \sigma$ to F, and
- if $\forall x.C \cdot \sigma \in F$, $t \in \mathscr{T}(\lfloor F \rfloor)$, and there are less than n instances that have already been added to F then add $C \cdot (\sigma \cup [x \mapsto t])$ to F.

We define the set of literals *reachable with n instances* from L written $\mathrm{RCH}^n_W(L)$ to be $\lfloor F_1 \cup \cdots \cup F_m \rfloor$.

Now, we can construct a single partial instantiation tree where all the literals from $\mathrm{RCH}^n_W(L)\rfloor$ are entailed in every branch after a certain number of instances. Here and below, a partial instantiation tree is an instantiation tree in which no instance has been chosen yet for some truth assignments.:

Lemma 2.5. *For every set V of theory clauses such that $W \cdot \varnothing \subseteq V$ and $\lfloor V \rfloor \rhd_T L$, there is a finite partial instantiation tree Z of V such that, for every truth assignment A that labels a orphan edge of Z, $\lfloor A \rfloor \rhd_T \mathrm{RCH}^n_W(L)$.*

Proof. Let $F_1, \ldots, F_m$ be the sets of theory clauses that were used for the construction of $\mathrm{RCH}^n_W(L)$. We show by induction over the construction of a set F_i that, for every set V' of theory clauses such that$V \subseteq V'$, there is a finite partial instantiation tree $Z_i(V')$ of V' such that, for every truth assignment A that labels a orphan edge of $Z_i(V')$ and every theory clause $C \cdot \sigma$ in $F_i \setminus (L \cdot \varnothing)$, we have $\lfloor A \rfloor \cup \mathit{known}(\mathscr{T}(\lfloor A \rfloor)) \vDash_T \mathit{known}(\mathscr{T}(\sigma))$ and there is $C \cdot \sigma' \in A$ such that $\lfloor A \rfloor \rhd_T \sigma \approx \sigma'$. This is enough to conclude as we can construct Z iteratively starting from $Z_0 = V$ and, for $i < m$, extending every orphan edge of Z_i labeled by a truth assignment A with $Z_{i+1}(A)$.

Let us now do the proof. By hypothesis, $W \cdot \varnothing \subseteq A$.

- If $C \cdot \sigma$ or $l \cdot \sigma$ was added to F using $\langle l \rangle C \cdot \sigma \in F$ then, by induction hypothesis, for every satisfiable truth assignment A at a leaf in Z, $\lfloor A \rfloor \cup \mathit{known}(\mathscr{T}(\lfloor A \rfloor)) \vDash_T \mathit{known}(\mathscr{T}(\sigma))$ and there is σ' such that $\langle l \rangle C \cdot \sigma' \in A$ and $\lfloor A \rfloor \vDash_T \sigma \approx \sigma'$. As a consequence, by definition of a truth assignment, $l \cdot \sigma'$ and $C \cdot \sigma'$ are in A. Thus, the property holds for the tree Z.
- If $C \cdot \sigma$ was added to F using $[l]C \cdot \sigma \in F$ then $\lfloor F \rfloor \rhd_T l$. By induction hypothesis, for every satisfiable truth assignment A at a leaf in Z, $\lfloor A \rfloor \cup \mathit{known}(\mathscr{T}(\lfloor A \rfloor)) \vDash_T \mathit{known}(\mathscr{T}(\sigma))$ and there is σ' such that $[l]C \cdot \sigma' \in A$ and $\lfloor A \rfloor \vDash_T \sigma \approx \sigma'$. What is more, for every $l' \cdot \sigma' \in F$, $\lfloor A \rfloor \rhd_T l'\sigma'$ by induction hypothesis. As a consequence, $\lfloor A \rfloor \rhd_T l$ and, by definition of a truth assignment, $C \cdot \sigma'$ is in A. Thus, the property holds for the tree Z.
- If $C \cdot (\sigma \cup [x \mapsto t])$ was added to F using $\forall x.C \cdot \sigma \in F$ and $t \in \mathscr{T}(\lfloor F \rfloor)$ then, by induction hypothesis, for every satisfiable truth assignment A at a leaf in Z, we have $\lfloor A \rfloor \cup \mathit{known}(\mathscr{T}(\lfloor A \rfloor)) \vDash_T \mathit{known}(\mathscr{T}(\sigma))$ and there is σ' such that $\forall x.C \cdot \sigma' \in A$ and $\lfloor A \rfloor \vDash_T \sigma \approx \sigma'$. What is more, by induction hypothesis, for every $l' \cdot \sigma' \in F$, $\lfloor A \rfloor \rhd_T l'\sigma'$. Thus, $\lfloor A \rfloor \cup \mathit{known}(\mathscr{T}(\lfloor A \rfloor)) \vDash_T \mathit{known}(\mathscr{T}(t))$ and there is t' in $\mathscr{T}(\lfloor A \rfloor)$ such that $\lfloor A \rfloor \vDash_T t \approx t'$.

We construct a new tree Z' starting from Z by adding a new instance $\varphi \cdot (\sigma' \cup [x \mapsto t'])$ for every satisfiable truth assignment A at a leaf in Z such that $\varphi \cdot (\sigma' \cup [x \mapsto t'])$ is not redundant in A. By construction, the property holds for the tree Z'.

□ □

We can then define an over-approximation of all the terms that can be learned by instantiating axioms and removing triggers from W. The idea is to use fresh constant symbols to model terms that have been used to instantiate a universal quantifier:

Definition 2.15. For every pseudo-clause C and every set of literals G, we define an over-approximation $\mathscr{N}_W(C,G)$ of the set of new terms that can be deduced from C assuming that the elements of G are true and all their subterms are known:

$$\begin{aligned}
\mathscr{N}_W(l,G) &= \{t \in \mathscr{T}(l) \mid \\
&\qquad \mathrm{RCH}^n_W(G \cup l) \cup \mathit{known}(\mathscr{T}(\mathrm{RCH}^0_W(G))) \nvDash_T \mathit{known}(t)\} \\
\mathscr{N}_W(\langle l \rangle C,G) &= \mathscr{N}_W(l,G) \cup \mathscr{N}_W(C,G \cup l) \\
\mathscr{N}_W([l]C,G) &= \mathscr{N}_W(C,G \cup l) \\
\mathscr{N}_W(\forall x.C,G) &= \mathscr{N}_W(C[x \mapsto a], G \cup a \approx a) \qquad a \text{ is a fresh constant symbol} \\
\mathscr{N}_W(C,G) &= \bigcup_{\varphi \in C} \mathscr{N}_W(\varphi,G)
\end{aligned}$$

We then define the set of new terms of W to be $\bigcup_{C \in W} \mathscr{N}_W(C,\varnothing)$.

Remark 2.4. We do not consider terms that have allowed us to instantiate universal quantifiers and to remove triggers as they must already be known to reach the current sub-formula nor for terms that can be equated to one of those in every truth assignment in less than n instances.

Theorem 2.3. *If $\bigcup_{\varphi \in W} \mathscr{N}_W(\varphi,\varnothing)$ does not contain any newly introduced constant then W is terminating.*

Proof. Let L be a set of literals. We show that $(W \cup L) \cdot \varnothing$ has a finite instantiation tree. Let S be $\mathscr{T}(L) \cup \{t \mid t \text{ is a ground term of } W\}$. We define the set $\overline{S}$ of the ground terms that appear in an instance of a sub-formula of W with terms of S. More precisely, $\overline{S}$ is the smallest set of ground terms such that, for every literal l or every witness $\langle l \rangle \_$ that appears as a sub-formula of W and every substitution σ from free variables of l to terms of S, $\mathscr{T}(l\sigma) \subseteq \overline{S}$. Let N be the set of pairs (C,G) of a pseudo-clause and a set of literals on which $\mathscr{N}_W$ is called during the computation of $\bigcup_{\varphi \in W} \mathscr{N}_W(\varphi,\varnothing)$.

We construct a finite instantiation tree of $(W \cup L) \cdot \varnothing$ in the following way. For a satisfiable, non-final truth assignment A, if we have $\lfloor A \rfloor \cup \mathit{known}(S) \vDash_T \mathit{known}(\mathscr{T}(\lfloor A \rfloor))$, then we add any non-redundant instance to A. Otherwise, we show that there is a term $t \in \overline{S}$ such that $\lfloor A \rfloor \cup \mathit{known}(S) \nvDash_T \mathit{known}(t)$ and there is a finite partial instantiation tree Z starting from A such that, for every truth assignment A' at a orphan edge of Z, $\lfloor A' \rfloor \cup \mathit{known}(S) \vDash_T \mathit{known}(t)$. We attach Z to A and continue.

This method terminates and gives a finite instantiation tree. Indeed, at each step, one of the following conditions holds:

- There is one new non-redundant instance of a universally quantified sub-formula of W with a term from S. There is only a finite number of such instances.
- There are more terms of $\overline{S}$ that are equal to a term of S modulo A' than modulo A. Of course the number of terms in $\overline{S}$ is also finite.

In the rest of the proof, we show that, if $\lfloor A \rfloor \cup \mathit{known}(S) \not\models_T \mathit{known}(\mathscr{T}(\lfloor A \rfloor))$, there is a term $t \in \overline{S}$ such that $\lfloor A \rfloor \cup \mathit{known}(S) \not\models_T \mathit{known}(t)$ and there is a finite partial instantiation tree Z starting from A such that, for every truth assignment A' at a orphan edge of Z, $\lfloor A' \rfloor \cup \mathit{known}(S) \models_T \mathit{known}(t)$. We need an intermediate lemma:

Lemma 2.6. *Let A be a truth assignment in an instantiation tree Z of $(W \cup L) \cdot \varnothing$. If $\lfloor A \rfloor \cup \mathit{known}(S) \not\models_T \mathit{known}(\mathscr{T}(\lfloor A \rfloor))$, then there is a pair $(l, G) \in N$ and a substitution μ from constant symbols introduced during the computation of $\mathscr{N}_W$ to terms of S such that $\lfloor A \rfloor \cup \mathit{known}(S) \not\models_T \mathit{known}(\mathscr{T}(l\mu))$, $\lfloor A \rfloor \cup \mathit{known}(S) \models_T \mathit{known}(\mathscr{T}(G\mu))$, and $\lfloor A \rfloor \models_T G\mu \cup l\mu$.*

Proof. We show by induction over the construction of A that, for every element $C \cdot \sigma \notin L \cdot \varnothing$ that is added to a truth assignment A' above A in Z, there is $(C\tau, G) \in N$ where τ maps free variables to constant symbols and a substitution μ from constant symbols to terms such that $\tau\mu|_{\mathit{vars}(C)}$ is σ and $\lfloor A' \rfloor \rhd_T G\mu$. For simplicity, we divide handling of a witness $\langle l \rangle C \cdot \sigma$ in the construction of a truth assignment into two parts: we first add $l \cdot \sigma$ and then $C \cdot \sigma$.

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

- $\varphi \cdot \sigma$ was added to a truth assignment A' in Z such that $\varphi \cdot \sigma \vee C' \cdot \sigma' \in A'$. By induction hypothesis, there is $((\varphi \vee C')\tau, G) \in N$ where τ maps free variables to constant symbols and a substitution μ from constant symbols to terms such that $\tau\mu|_{\mathit{vars}(\varphi \vee C')}$ is $\sigma \cup \sigma'$ and $\lfloor A' \rfloor \rhd_T G\mu$. By definition of $\mathscr{N}_W$, we have $(\varphi\tau, G) \in N$.
- $l \cdot \sigma$ was added to a truth assignment A' in Z such that $\langle l \rangle C \cdot \sigma \in A'$. By induction hypothesis, there is $(\langle l \rangle C\tau, G) \in N$ where τ maps free variables to constant symbols and a substitution μ from constant symbols to terms such that $\tau\mu|_{\mathit{vars}(\langle l \rangle C)}$ is σ and $\lfloor A' \rfloor \rhd_T G\mu$. By definition of $\mathscr{N}_W$, $(l\tau, G) \in N$.
- $C \cdot \sigma$ was added to a truth assignment A' in Z such that $\langle l \rangle C \cdot \sigma \in A'$. By induction hypothesis, there is $(\langle l \rangle C\tau, G) \in N$ where τ maps free variables to constant symbols and a substitution μ from constant symbols to terms such that $\tau\mu|_{\mathit{vars}(\langle l \rangle C)}$ is σ and $\lfloor A' \rfloor \rhd_T G\mu$. By definition of $\mathscr{N}_W$, $(C\tau, G \cup l\tau) \in N$. What is more, since $\langle l \rangle C \cdot \sigma \in A'$, we have $l \cdot \sigma \in A'$ and, thus, $\lfloor A' \rfloor \rhd_T (G \cup l\tau)\mu$.
- $C \cdot \sigma$ was added to a truth assignment A' in Z such that $[l]C \cdot \sigma \in A'$ and $\lfloor A' \rfloor \rhd_T l\sigma$. By induction hypothesis, there is $([l]C\tau, G) \in N$ where τ maps free variables to constant symbols and a substitution μ from constant symbols to terms such that $\tau\mu|_{\mathit{vars}([l]C)}$ is σ and $\lfloor A' \rfloor \rhd_T G\mu$. By definition of $\mathscr{N}_W$, $(C\tau, G \cup l\tau) \in N$. Since $\lfloor A' \rfloor \rhd_T l\sigma$, we have $\lfloor A' \rfloor \rhd_T (G \cup l\tau)\mu$.
- the new instance $C \cdot (\sigma \cup [x \mapsto t])$ was added to a truth assignment A' in Z such that $\forall x.C \cdot \sigma \in A'$ and $t \in \mathscr{T}(\lfloor A' \rfloor)$. By induction hypothesis, there is $(\forall x.C\tau, G) \in N$ where τ maps

free variables to constant symbols and a substitution μ from constant symbols to terms such that $\tau\mu|_{vars(\forall x.C)}$ is σ and $\lfloor A' \rfloor \rhd_T G\mu$. By definition of N, there is a new constant symbol a such that $(C(\tau \cup [x \mapsto a]), G \cup a \approx a) \in N$. Since a is fresh in $C\tau$ and G, we extend μ with $[a \mapsto t]$. Since $t \in \mathscr{T}(\lfloor A' \rfloor)$, we have $\lfloor A' \rfloor \rhd_T (G \cup a \approx a)(\mu \cup [a \mapsto t])$.

Now, we only need to find a closure $(l \cdot \sigma) \in A$ with an associated pair $(l\tau, G) \in N$ and a substitution μ from constant symbols to terms of S such that $\lfloor A \rfloor \cup known(S) \not\models_T known(\mathscr{T}(l\sigma))$ and $\lfloor A \rfloor \cup known(S) \models_T known(\mathscr{T}(G\mu))$. Let us consider the first time we add a closure $l \cdot \sigma$ such that $\lfloor A \rfloor \cup known(S) \not\models_T known(\mathscr{T}(l\sigma))$ to a truth assignment A' above A in Z. Since $\mathscr{T}(L) \subseteq S$, $l \cdot \sigma \notin L \cdot \varnothing$ and, as we have just shown, there is $(C\tau, G) \in N$ where τ maps free variables to constant symbols and a substitution μ' from constant symbols to terms such that $\tau\mu'|_{vars(C)}$ is σ and $\lfloor A' \rfloor \rhd_T G\mu'$. What is more, by definition of $l \cdot \sigma$, we have $\lfloor A \rfloor \cup known(S) \models_T known(\mathscr{T}(\lfloor A' \rfloor))$ and, since $\lfloor A' \rfloor \rhd_T G\mu'$, we have $\lfloor A \rfloor \cup known(S) \models_T known(\mathscr{T}(G\mu'))$.

Finally, we take a substitution μ from constant symbols to terms of S that is equal to μ' modulo $\lfloor A \rfloor$. Since $\lfloor A \rfloor \cup known(S) \models_T known(\mathscr{T}(\mu'))$, there is one. By definition of μ, $\lfloor A \rfloor \cup known(S) \models_T known(\mathscr{T}(G\mu))$ and $\lfloor A \rfloor \models_T G\mu \cup l\mu$. □ □

As $\lfloor A \rfloor \cup known(S) \not\models_T known(\mathscr{T}(\lfloor A \rfloor))$, there are a pair $(l, G) \in N$ and a substitution σ such that $\lfloor A \rfloor \cup known(S) \not\models_T known(\mathscr{T}(l\sigma))$, $\lfloor A \rfloor \cup known(S) \models_T known(\mathscr{T}(G\sigma))$, and $\lfloor A \rfloor \models_T G\sigma \cup l\sigma$ by Lemma 2.6. Let t be a term of l such that $\lfloor A \rfloor \cup known(S) \not\models_T known(t\sigma)$, since every element of σ appears in $\mathscr{T}(G\sigma)$, there must be such a term. Notice that, by definition of $\mathscr{N}_W$, every variable of I freely occurs in G. Also notice that, by definition of $\overline{S}$ and σ, $t\sigma \in \overline{S}$. It now remains to show that there is a finite partial instantiation tree Z starting from A such that, for every truth assignment A' at a orphan edge of Z, $\lfloor A' \rfloor \cup known(S) \models_T known(t\sigma)$.

Since W has a finite set of new terms $\bigcup_{\varphi \in W} \mathscr{N}_W(\varphi, \varnothing)$, no term of $\mathscr{T}(l)$ that contains a constant introduced by $\mathscr{N}_W$ can appear in the set of new terms of W. Since $\lfloor A \rfloor \cup known(S) \not\models_T known(t\sigma)$, the term t contains at least a constant introduced by $\mathscr{N}_W$. As a consequence, we have that $\textsc{Rch}^n_W(G \cup l) \cup known(\mathscr{T}(\textsc{Rch}^0_W(G))) \models_T known(t)$. Replacing constant symbols with terms using σ, we get $\textsc{Rch}^n_W(G \cup l)\sigma \cup known(\mathscr{T}(\textsc{Rch}^0_W(G)))\sigma \models_T known(t)\sigma$. By immediate induction on the construction of $\textsc{Rch}^n_W(G \cup l)$, we have both $\textsc{Rch}^n_W(G \cup l)\sigma \subseteq \textsc{Rch}^n_W(G\sigma \cup l\sigma)$ and $\mathscr{T}(\textsc{Rch}^0_W(G))\sigma \subseteq \mathscr{T}(\textsc{Rch}^0_W(G)\sigma) \subseteq \mathscr{T}(G\sigma) \cup S$. Since $\lfloor A \rfloor \cup known(S) \models_T known(\mathscr{T}(G\sigma))$, we have $\textsc{Rch}^n_W(G\sigma \cup l\sigma) \cup \lfloor A \rfloor \cup known(S) \models_T known(t\sigma)$.

By Lemma 2.5, since $\lfloor A \rfloor \models_T G\sigma \cup l\sigma$, we can continue the instantiation tree Z after A so that, for every satisfiable truth assignment A' labeling a orphan edge after A, we have $\lfloor A' \rfloor \models_T \textsc{Rch}^n_W(G\sigma \cup l\sigma)$ and, thus, $\lfloor A' \rfloor \cup known(S) \models_T known(t\sigma)$. □ □

Example 2.4. Let us consider the theory W_{array} of non-extensional arrays defined in Example 1.1. For any natural n in the definition of RCH, we have that $\bigcup_{\varphi \in W_{array}} \mathscr{N}_{W_{array}}(\varphi, \varnothing)$. For example, if

$n = 0$ then, for the first axiom, we have:

$$\begin{aligned}
&\mathscr{N}_{W_{array}}(\forall a, i, e.[set(a,i,e)]\ (get(set(a,i,e),i) \approx e), \varnothing)\\
&\quad = \mathscr{N}_{W_{array}}([set(a,i,e)]\ (get(set(a,i,e),i) \approx e), \{a \approx a, i \approx i, e \approx e\})\\
&\quad = \mathscr{N}_{W_{array}}((get(set(a,i,e),i) \approx e), \{set(a,i,e) \approx set(a,i,e), a \approx a, i \approx i, e \approx e\})\\
&\quad = \mathscr{T}\left(\left\{ \begin{array}{l} t \subseteq \mathscr{T}(get(set(a,i,e),i) \approx e)) \mid \\ \qquad get(set(a,i,e),i) \approx e \cup \mathscr{T}(\{set(a,i,e), a, i, e\}) \not\models_T known(t) \end{array} \right\}\right)\\
&\quad = \varnothing
\end{aligned}$$

If we remove the trigger from this axiom, then it can produce a new term $set(a,i,e)$ as we have:

$$\begin{aligned}
&\mathscr{N}_{W_{array}}(\forall a, i, e.\ (get(set(a,i,e),i) \approx e), \varnothing)\\
&\qquad = \mathscr{T}\left(\left\{ \begin{array}{l} t \subseteq \mathscr{T}(get(set(a,i,e),i) \approx e)) \mid \\ \qquad get(set(a,i,e),i) \approx e \cup \mathscr{T}(\{a, i, e\}) \not\models_T known(t) \end{array} \right\}\right)\\
&\qquad = \{set(a,i,e)\}
\end{aligned}$$

Example 2.5. The axiomatization W_{conv} presented in Example 2.3 does not have finitely many new terms. Indeed, there are terms that can be created by W_{conv} and that will not be equated to already existing terms.

Even if an axiomatization W does not have the above stated property, it is sometimes enough to go through the set of new terms from W that contain free variables to justify that only a finite number of them can be created. This is what we do for the proof of termination of the axiomatization for doubly-linked lists: We go through the new terms of the axiomatization that can be used to instantiate universally quantified formulas of W and we justify that there can only be a finite number of those.

Such a proof can be complex, in particular if quantification is done on interpreted sorts – which is not the case for doubly-linked lists. It can be simplified further if the triggers of the axiomatization only contain uninterpreted function symbols. Indeed, in this case, it is enough to show that there can only be a finite number of new terms starting with function symbols that appear in a trigger. Such a proof can sometimes be done modularly. The idea is to find a partition $S_0 \ldots S_n$ of the set of function symbols that appear in triggers in an axiomatization W such that, if W_i is the subset of formulas of W using symbols of S_i in triggers, W_i can only create new terms starting with symbols from $S_0 \cup \cdots \cup S_i$. We can then show separately that, for each set S_i, formulas of W_i cannot create an infinite number of new terms starting with function symbols in S_i.

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

2.3.2 Designing a Complete Axiomatization

The natural way to prove the completeness of an axiomatization W is to give a general method for completing a world L in which W is true into a model of the theory. This implies that terms

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

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

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

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

Example 2.6. Consider triggers containing the function symbol *equal_lists* in the theory of doubly-linked lists. We cannot deduce that $equal\_lists(co_1, co_2) \approx \mathtt{t}$ is true for two lists co_1 and co_2 if the term $equal\_lists(co_1, co_2)$ does not appear in the context, since every axiom involving a term starting with *equal_lists* has this term as a trigger.

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

IS_PREFIX_INV:

$$\begin{array}{l}\forall co_1, co_2 : list.[is\_prefix(co_1, co_2)]\ is\_prefix(co_1, co_2) \not\approx \mathtt{t} \rightarrow \\ \quad (\forall cu : cursor.[equal\_lists(co_1, left(co_2, cu))] \\ \quad\quad (has\_element(co_2, cu) \approx \mathtt{t} \vee cu \approx no\_element) \rightarrow \\ \quad\quad\quad equal\_lists(co_1, left(co_2, cu)) \not\approx \mathtt{t})\end{array}$$

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

A notable exception to this principle are defining axioms. Axioms used to provide an uninterpreted symbol f with its meaning can generally use f in their triggers even though we cannot deduce the value of a term starting with f in the axiomatization every time we can deduce it

in the theory. For example, the axiom EQUAL_LISTS_LENGTH has $equal\_lists(co_1, co_2)$ as a trigger and yet the axiomatization is complete. Indeed, in the theory of doubly-linked lists, we cannot deduce that two lists are equal if we do not know *a priori* that they have the same length. Thus, every world in which the axiomatization is true can be completed by assuming $equal\_lists(co_1, co_2) \not\approx \mathtt{t}$ whenever we do not have $length(co_1) \approx length(co_2)$.

EQUAL_LISTS_LENGTH:

$$\forall co_1, co_2 : list.[equal\_lists(co_1, co_2)] \\ equal\_lists(co_1, co_2) \approx \mathtt{t} \rightarrow length(co_1) \approx length(co_2)$$

2.3.3 An Automatable Debugger for Completeness

When designing an axiomatization W, it may be useful to have an automatable way to search for counter-examples to the completeness of W with respect to the first-order axiomatization W' which is W where triggers are translated into implications. More precisely, we look for sets of literals L such that $L \cup W'$ is unsatisfiable in first-order logic but there is a world in which $L \cup W$ is true. The algorithm below searches systematically for sets of literals L such that $L \cup W'$ is unsatisfiable in first-order logic. An implementation of the solver can then be used to decide if there is a world in which $L \cup W$ is true (see Definition 2.7).

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

Example 2.7. The two first-order axioms of the theory of arrays are converted into the two clauses $get(set(a,i,e),i) \approx e$ and $i \approx j \vee get(a,j) \approx get(set(a,i,e),j)$ where a, i, j, and e are free variables. We can then use the above algorithm to infer the counter-examples that we produced in Example 2.1. The first axiom coupled with itself gives:

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

So we deduce a potential counter-example $\{set(a_1,i,e_1) \approx set(a_2,i,e_2),\ e_1 \not\approx e_2\}$ which is indeed a counter-example for the axiom with a bad trigger in Example 2.1:

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

The second axiom contains two occurrences of *get*. Rewriting the equality both ways in each occurrence of *get* gives the four following clauses. The first and the fourth give the counter-examples for the other two cases in Example 2.1:

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

We see that this automatable approach manages to find all the counter-examples we used in Example 2.1.

2.4 Conclusion

We have given a framework for reasoning about soundness, completeness, and termination of a first-order axiomatization with triggers. We have seen in Section 2.2 that our semantics is close, but not identical, to the behavior of existing solvers. We should now implement an SMT solver that supports first-order logic with triggers and transforms a sound, complete, and terminating axiomatization of a theory T' into a decision procedure for satisfiability modulo T'.

3 A Black-Box Decision Procedure

Contents

3.1 Description 45
3.1.1 Preliminaries 45
3.1.2 Deduction Rules for First-Order Logic with Triggers 47
3.1.3 Soundness, Completeness, and Termination 48
3.2 Implementation 53
3.2.1 Description 53
3.2.2 Benchmarks 54
3.3 Conclusion 56

In Chapter 2 we introduced a semantics for first-order logic with triggers as well as notions of soundness, completeness, and termination of axiomatizations in this logic.

In this chapter, we present a theoretical way of extending a generic ground SMT solver so that it can turn an axiomatization with triggers into a decision procedure. It requires that the axiomatization is sound and complete in the framework defined in Chapter 2, and that it is *strongly terminating* as defined in 2.12.

3.1 Description

In this section, we define a wrapper over a generic SMT solver for ground formulas that accepts a theory written as a set of formulas with triggers. This solver is a theoretical model and it is not meant to be efficient. We prove it sound with respect to our framework.

3.1.1 Preliminaries

We extend an existing solver **S** that decides satisfiability of ground formulas modulo T. To be able to reason about it, we make a few assumptions about the interface of the ground solver **S**:

- **S** returns $\mathit{Unsat}(U)$ when called on an unsatisfiable set of ground formulas G where U is an unsatisfiable core of G, that is, a subset of G such that $U \vDash_T \bot$.
- **S** returns $\mathit{Sat}(L)$ when called on a satisfiable set of ground formulas G. We assume that there is a model I of G in T such that L is exactly the set of literals of G that are true in I.

We describe a solver $\mathscr{L}ift(\mathbf{S})$ 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 procedure, we Skolemize and clausify the axioms in Ax, producing a set of pseudo-clauses W.

To use $\mathbf{S}$ to reason about formulas with triggers and witnesses, we need to encode theory clauses coming from the axiomatization into regular clauses.

Definition 3.1. To each pseudo-literal φ in $W \cup G$, we associate an atomic formula $B_\varphi = P_\varphi(\bar{x})$, where P_φ is a fresh predicate symbol and $\bar{x}$ are the free variables of φ. We define an encoding of closures into literals in the following way:

$$\begin{aligned} [\![l \cdot \sigma]\!] &\triangleq l\sigma \\ [\![\varphi \cdot \sigma]\!] &\triangleq B_\varphi \sigma \qquad\qquad \text{if } \varphi \text{ is not a literal} \end{aligned}$$

A literal $B_\varphi\sigma$ is called an *opaque* literal.

To model the trigger mechanism, we need a way to protect a theory 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 theory clause. We extend our encoding to guarded clauses: $[\![H \to C]\!] \triangleq \bigvee_{\varphi\cdot\sigma \in H} \neg [\![\varphi \cdot \sigma]\!] \vee \bigvee_{\varphi\cdot\sigma \in C} [\![\varphi \cdot \sigma]\!]$.

The solver $\mathscr{L}ift(\mathbf{S})$ for the theory T' maintains a set of guarded clauses R. It starts with $R = (G \cup W) \cdot \varnothing$. It repeatedly launches the solver $\mathbf{S}$ on $[\![R]\!]$ and then, if $\mathbf{S}$ returns $Sat(L)$, it uses the set of literals L to augment R with new guarded clauses by reasoning on formulas with triggers and witnesses. If $\mathbf{S}$ returns $Unsat(U)$, $\mathscr{L}ift(\mathbf{S})$ returns $Unsat$.

To deduce new guarded clauses that should be added to R, $\mathscr{L}ift(\mathbf{S})$ interprets ground models of $[\![R]\!]$ returned by $\mathbf{S}$ as sets of closures. $\mathscr{L}ift(\mathbf{S})$ must prune the set of ground literals L returned by $\mathbf{S}$ in $Sat(L)$ to only consider relevant ones:

Definition 3.2. Let L be a world and R be a set of guarded clauses. We define the set of closures that are *relevant* in L, written $L|_R$, to be the limit of a sequence of sets of closures $\{M_i\}$ such that $M_0 = \varnothing$ and M_{n+1} is the biggest superset of M_n such that, for every closure $\varphi \cdot \sigma \in M_{n+1}$:

- $[\![\varphi \cdot \sigma]\!] \in L$, and
- there is a guarded clause $H \to C \in R$ such that $\varphi \cdot \sigma \in C$ and $H \subseteq M_n$.

We say that a guarded clause $H \to C$ is *relevant* in L if $H \subseteq L|_R$.

To comply with the semantics of triggers, when the solver $\mathbf{S}$ returns $Sat(L)$ on a set of ground clauses $[\![R]\!]$, we need to be able to decide whether $L|_R$ should allow to deduce $C \cdot \sigma$ from a trigger $[l]C \cdot \sigma$. We use $\mathbf{S}$ to compute a subset M' of $L|_R$ such that $\lfloor M' \rfloor \rhd l$ if any. First, we launch $\mathbf{S}$ on $known(\mathscr{T}(\lfloor L|_R \rfloor)) \cup \lfloor L|_R \rfloor \cup \{\neg l \vee \neg known(\mathscr{T}(l))\}$. Since $\lfloor L|_R \rfloor$ is a subset of L and therefore is satisfiable, $\mathbf{S}$ can either return $Sat(L')$ or $Unsat(U \cup \{\neg l \vee \neg known(\mathscr{T}(l))\})$. If it returns $Sat(L')$, then $\lfloor L|_R \rfloor \not\rhd l$. Otherwise, we compute a subset M' of $L|_R$ such that, for every literal $known(t)$ in U, there is a closure $l \cdot \sigma \in M'$ such that $t \in \mathscr{T}(l\sigma)$, and, for every other literal l in U, there is a closure $l' \cdot \sigma \in M'$ such that $l = l'\sigma$.

$$\textsc{Pos Unfold}\quad \frac{\langle l\rangle C\cdot\sigma \in L|_R}{\langle l\rangle C\cdot\sigma \to C\cdot\sigma \qquad \langle l\rangle C\cdot\sigma \to l\cdot\sigma} \qquad \textsc{Neg Unfold}\quad \frac{[l]C\cdot\sigma \in L|_R \qquad M' \subseteq L|_R \qquad \lfloor M'\rfloor \rhd l\sigma}{[l]C\cdot\sigma \wedge \bigwedge M' \to C\cdot\sigma}$$

$$\textsc{Inst}\quad \frac{\begin{array}{c}\forall x.C\cdot\sigma \in L|_R \qquad l\cdot\sigma' \in L|_R \\ t \in \mathscr{T}(l\sigma') \qquad \lfloor L|_R\rfloor \cup \{B_\varphi\sigma \mid \varphi\cdot\sigma \in L|_R\} \cup \{\neg B_\varphi(\sigma \cup [x\mapsto t]) \mid \varphi \in C\} \not\models_T \bot\end{array}}{\forall x.C\cdot\sigma \wedge l\cdot\sigma' \to C\cdot(\sigma \cup [x\mapsto t])}$$

Figure 3.1: Deduction rules used to compute new guarded clauses from the set of relevant closures $L|_R$ in a model L of $[\![R]\!]$.

3.1.2 Deduction Rules for First-Order Logic with Triggers

The algorithm for $\mathscr{L}ift(\mathbf{S})$ is as follows:

1. Initialize R to $(W \cup G)\cdot\varnothing$.
2. Call $\mathbf{S}$ on $[\![R]\!]$.
3. If $\mathbf{S}$ returns $Unsat(U)$ then return *Unsat*.
4. If $\mathbf{S}$ returns $Sat(L)$:
   - If no new guarded clause can be inferred from L using deduction rules from Figure 3.1 then return *Sat*.
   - If at least one new clause can be inferred from L using Pos Unfold or Neg Unfold, let R' be a non-empty set of new clause inferred from L using these rules.
   - Otherwise, let R' be a non-empty set of new clauses inferred from L using Inst.
   - Go to step 2 with $R := R \cup R'$.

Deduction rules in Figure 3.1 are used to add to the set of closures $L|_R$ elements that can only be deduced from it using first-order logic with triggers. Still, since elements of $L|_R$ are not implied by R, every deduced formula needs to be guarded so that it is a tautology in first-order logic with triggers, and the satisfiability of R does not change.

The rule Pos Unfold states that a witness $\langle l\rangle C$ implies both the literal l and the pseudo-clause C.

In the rule Neg Unfold, we remove the trigger blocking a pseudo-clause. For this rule to produce a tautology, we need to guard it with a set of closures M' that imply the validity of the trigger in first-order logic with triggers, namely $\lfloor M'\rfloor \rhd l$. The set M' is computed using $\mathbf{S}$ as explained in the previous section.

The rule Inst instantiates a universally quantified formula with a term of $\lfloor L|_R\rfloor$. To ensure termination, we need to be careful not to produce redundant instances, following the Definition 2.10 of the previous chapter. For this check we also use $\mathbf{S}$. We only produce an instance of

$\forall x.C \cdot \sigma$ with a term t if $\lfloor L|_R \rfloor \cup \{B_\varphi\sigma \mid \varphi \cdot \sigma \in L|_R\}$ is not a model of $\bigvee_{\varphi \in C} B_\varphi(\sigma \cup [x \mapsto t])$. In particular, if R contains a guarded clause $H \to C \cdot \mu$ relevant in L such that $\lfloor L|_R \rfloor \models_T \sigma \cup [x \mapsto t] \mathrel{\tilde{\approx}} \mu$ then $L|_R$ contains $\varphi \cdot \mu$ for some $\varphi \in C$ and $\lfloor L|_R \rfloor \cup \{B_\varphi\sigma \mid \varphi \cdot \sigma \in L|_R\}$ is a model of $\bigvee_{\varphi \in C} B_\varphi(\sigma \cup [x \mapsto t])$. Notice that we convert every closure in $L|_R$ and $C \cdot (\sigma \cup [x \mapsto t])$, even closures of literals, into an opaque literal. Indeed, to ensure completeness, our redundancy check must not block instantiations that may bring in new terms even if they are redundant from the logical point of view.

Example 3.1. Here is a possible execution of the solver $\mathscr{L}ift(\mathbf{S})$ on the set of ground formulas G and an axiomatization W. We assume that $\mathbf{S}$ is a decision procedure for the background theory T combining linear integer arithmetic and Equality with Uninterpreted Functions:

$$G = \{f(0) \approx 0,\ f(1) \not\approx 1\}$$

$$W = \{\forall x.[f(x+1)]\ f(x+1) \approx f(x)+1\}$$

Let us show how the solver $\mathscr{L}ift(\mathbf{S})$ can deduce that

$$R_0 = \left\{ \begin{array}{l} f(0) \approx 0 \cdot \varnothing,\ f(1) \not\approx 1 \cdot \varnothing, \\ \forall x.[f(x+1)]\ f(x+1) \approx f(x)+1 \cdot \varnothing \end{array} \right\}$$

is unsatisfiable.

1. The ground solver returns $Sat(L_0)$ on $[\![R_0]\!]$, where L_0 is $[\![R_0]\!]$ itself. Thus, we compute $L_0|_{R_0}$ to be $(G \cup W) \cdot \varnothing$. We have $f(0) \approx 0 \cdot \varnothing \in L_0|_{R_0}$ and $0 \in \mathscr{T}(f(0) \approx 0)$. As a consequence, the rule INST can instantiate x with 0 in the universal formula:

$$R_1 = R_0 \cup \left\{ \begin{array}{l} \forall x.\ [f(x+1)]\ f(x+1) \approx f(x)+1 \cdot \varnothing \wedge \\ f(0) \approx 0 \cdot \varnothing \to \\ \qquad [f(x+1)]\ f(x+1) \approx f(x)+1 \cdot [x \mapsto 0] \end{array} \right\}$$

2. $\mathbf{S}$ returns $Sat(L_1)$ on $[\![R_1]\!]$ with $L_1 = L_0 \cup \{[\![[f(x+1)]\ f(x+1) \approx f(x)+1 \cdot [x \mapsto 0]]\!]\}$. We compute $L_1|_{R_1}$ to be $L_0|_{R_0} \cup \{[f(x+1)]\ f(x+1) \approx f(x)+1 \cdot [x \mapsto 0]\}$. Based on results from the theory of arithmetics, the ground solver can deduce that we have $\{f(0) \approx 0,\ f(1) \not\approx 1\} \rhd f(0+1) \approx f(0+1)$. Thus, the rule NEG UNFOLD can add another formula to R_1:

$$R_2 = R_1 \cup \left\{ \begin{array}{l} [f(x+1)]\ f(x+1) \approx f(x)+1 \cdot [x \mapsto 0] \wedge \\ f(0) \approx 0 \cdot \varnothing \wedge f(1) \not\approx 1 \cdot \varnothing \to \\ \qquad f(x+1) \approx f(x)+1 \cdot [x \mapsto 0] \end{array} \right\}$$

3. The ground solver returns *Unsat* on $[\![R_2]\!]$. Indeed, any model of $[\![R_2]\!]$ would satisfy $f(0+1) \approx f(0)+1$, $f(0) \approx 0$ and $f(1) \not\approx 1$. Thus, the initial set G is unsatisfiable modulo W.

3.1.3 Soundness, Completeness, and Termination

In this section, we prove that our solver can turn a sound, complete, and strongly terminating axiomatization Ax of a theory T' into a decision procedure for T'.

3.1.3.1 Soundness and Completeness

Lemma 3.1. *Let $H \to C$ be a new guarded clause that can appear as the conclusion of a deduction rule in Figure 3.1. For world L, if $L \rhd \bigwedge_{\varphi\cdot\sigma\in H} \varphi\sigma$ then $L \rhd \bigvee_{\varphi\cdot\sigma\in C} \varphi\sigma$.*

Proof. The guarded clauses $\langle l\rangle C\cdot\sigma \to C\cdot\sigma$ and $\langle l\rangle C\cdot\sigma \to l\cdot\sigma$ can be deduced by POS UNFOLD. If $L \rhd \langle l\rangle C\sigma$ then $L \rhd C\sigma$ and $L \rhd l\sigma$.

The ground clause $[l]C\cdot\sigma \wedge \bigwedge M' \to C\cdot\sigma$ can be deduced by NEG UNFOLD if $\lfloor M'\rfloor \rhd l\sigma$. If $L \rhd \{\varphi\sigma' \mid \varphi\cdot\sigma' \in M'\}$ then $L \rhd l\sigma$. Thus, if $L \rhd [l]C\sigma$ and $L \rhd \{\varphi\sigma' \mid \varphi\cdot\sigma' \in M'\}$ then $L \rhd C\sigma$.

The clause $\forall x.C\cdot\sigma \wedge l\cdot\sigma' \to C\cdot(\sigma\cup[x\mapsto t])$ can be deduced by INST if $t \in \mathscr{T}(l\sigma')$. If $L \rhd l\sigma'$ then $L \cup known(\mathscr{T}(L)) \vDash_T known(\mathscr{T}(t))$. Thus, if $L \rhd \forall x.C\sigma$ and $L \rhd l\sigma'$ then there is $t' \in known(\mathscr{T}(L))$ such that $L \rhd C(\sigma\cup[x\mapsto t'])$ and $L \vDash_T t \approx t'$. By immediate induction on the structure of elements of C, $L \rhd C(\sigma\cup[x\mapsto t])$. □

Lemma 3.2. *Let $W \cup G$ a set of pseudo-clauses. Let R be the set of theory clauses that have been computed by our algorithm after some steps starting from $(W\cup G)\cdot\varnothing$ and let L be a set of ground literals returned by* **S** *on $[\![R]\!]$ in* Sat(L). *For every closure $\varphi\cdot\sigma \in L|_R$, $\mathscr{T}(\sigma) \subseteq \mathscr{T}(\lfloor L|_R\rfloor)$.*

Proof. We do the proof by induction over the construction of the set of relevant literals. For every $\varphi\cdot\sigma \in L|_R$ there is a guarded clause $H \to C \in R$ such that $\varphi\cdot\sigma \in C$ and, for every closure $\varphi'\cdot\sigma' \in H$, $\varphi'\cdot\sigma'$ was added to $L|_R$ before $\varphi\cdot\sigma$. As a consequence, it is enough to show that, for every clause $H \to C \in R$, if, for every closure $\varphi'\cdot\sigma' \in H$, $\mathscr{T}(\sigma') \subseteq \mathscr{T}(\lfloor L|_R\rfloor)$ then, for every positive closure $\varphi\cdot\sigma \in C$, $\mathscr{T}(\sigma) \subseteq \mathscr{T}(\lfloor L|_R\rfloor)$. Since we start with $(W\cup G)\cdot\varnothing$, guarded clauses with non-empty substitutions in closures may only be added via deduction rules in Figure 3.1.

The only interesting rule is INST, since it introduces a new term t in the substitution σ of its positive literals. Since t has to be in $\mathscr{T}(\lfloor L|_R\rfloor)$ for INST to be applied, we have $\mathscr{T}(\sigma\cup[x\mapsto t]) \subseteq \mathscr{T}(\lfloor L|_R\rfloor)$. □

Theorem 3.1 (Soundness)**.** *If $\mathscr{L}ift(\mathbf{S})$ returns* Unsat *on a set of ground formulas G and a sound axiomatization Ax of T' then G is unsatisfiable modulo 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 $W\cup G'$.

If $\mathscr{L}ift(\mathbf{S})$ returns *Unsat* on G, then there is a set of guarded clauses R that have been produced by the deduction rules such that $[\![(G\cup W)\cdot\varnothing\cup R]\!]$ is T-unsatisfiable. By contradiction, assume that G is T'-satisfiable. Then there is a subset G' of literals of G such that G' is T'-satisfiable and $G' \vDash_T G$. Since Ax is sound, there is a model L of $W\cup G'$. Consider $L' = L \cup \{[\![\varphi\cdot\sigma]\!] \mid \varphi\cdot\sigma \in (G\cup W)\cdot\varnothing\cup R \text{ and } L \rhd \varphi\sigma\} \cup \{\neg[\![\varphi\cdot\sigma]\!] \mid \varphi\cdot\sigma \in (G\cup W)\cdot\varnothing\cup R \text{ and } L \not\rhd \varphi\sigma\}$. Since L is complete, for every closure $l\cdot\sigma$, if $L \not\rhd l\sigma$ then $L \nvDash_T l\sigma$ and, for every closure $\varphi\cdot\sigma$ and every substitution σ' such that $L \vDash_T \sigma \mathrel{\dot\approx} \sigma'$, if $L \rhd_T \varphi\sigma$ then $L \rhd_T \varphi\sigma'$. Therefore, L' is satisfiable. By construction, $L' \vDash_T [\![(G\cup W)\cdot\varnothing]\!]$. By Lemma 3.1, we also have $L' \vDash_T [\![R]\!]$. Since $[\![(G\cup W)\cdot\varnothing\cup R]\!]$ is unsatisfiable, we reach a contradiction. □

Theorem 3.2 (Completeness)**.** *If $\mathscr{L}ift(\mathbf{S})$ returns* Sat *on a set of ground formulas G and a complete axiomatization Ax of T' then G has a model modulo T'.*

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 say that a world L propositionally satisfies a set of ground clauses G whenever, for every clause C in G, there is a literal $l \in C$ such that $l \in L$.

If $\mathscr{L}ift(\mathbf{S})$ returns *Sat* on G, then there is a set of guarded clauses R that have been produced by the deduction rules and a set of literals L that propositionally satisfies $[\![R]\!]$ and such that nothing more can be deduced from $L|_R$. By definition of $L|_R$, for every guarded clause $H \to C \in R$ such that $H \subseteq L|_R$, the set of ground literals $[\![L|_R]\!]$ propositionally satisfies $[\![C]\!]$. In particular, it propositionally satisfies $[\![(G \cup W) \cdot \varnothing]\!]$. Consider the set of literals $\lfloor L|_R \rfloor$. We show that $\lfloor L|_R \rfloor \rhd W \cup G$. Since Ax is complete and $L|_R \rfloor \rhd W \cup L|_R \rfloor$, $\lfloor L|_R \rfloor$ is T'-satisfiable. Since $\lfloor L|_R \rfloor \vdash_T G$ so is G.

We show that, for every closure $\varphi \cdot \sigma \in L|_R$, $\lfloor L|_R \rfloor \rhd \varphi\sigma$ by structural induction over φ. Since $[\![L|_R]\!]$ propositionally satisfies $[\![(G \cup W) \cdot \varnothing]\!]$, it is enough to conclude.

- $l \cdot \sigma \in L|_R$. By definition of $\lfloor L|_R \rfloor$, we have $l\sigma \in \lfloor L|_R \rfloor$.
- $\langle l \rangle C \cdot \sigma \in L|_R$. Since nothing new can be deduced from L, then the two guarded clauses $\langle l \rangle C \cdot \sigma \to l \cdot \sigma$ and $\langle l \rangle C \cdot \sigma \to C \cdot \sigma$ are in R. Since $\langle l \rangle C \cdot \sigma \in L|_R$, $[\![L|_R]\!]$ satisfies the encoding of these two formulas. Thus, $l \cdot \sigma \in L|_R$ and there is $\varphi \in C$ such that $\varphi \cdot \sigma \in L|_R$. By induction hypothesis, $\lfloor L|_R \rfloor \rhd l\sigma$ and $\lfloor L|_R \rfloor \rhd \varphi \cdot \sigma$, and, therefore, $\lfloor L|_R \rfloor \rhd \langle l \rangle C\sigma$.
- $[l]C \cdot \sigma \in L|_R$. If $\lfloor L|_R \rfloor \rhd l\sigma$, since nothing new can be deduced from L, there is $U \subseteq L|_R$ such that $[l]C \cdot \sigma \wedge \bigwedge U \to C \cdot \sigma \in R$. Since $[l]C \cdot \sigma \in L|_R$ and $U \subseteq L|_R$, $[\![L|_R]\!]$ satisfies the encoding of this formula. Thus, there is $\varphi \in C$ such that $\varphi \cdot \sigma \in L|_R$ and, by induction hypothesis, $\lfloor L|_R \rfloor \rhd \varphi\sigma$. As a consequence, $\lfloor L|_R \rfloor \rhd [l]C\sigma$.
- $\forall x.C \cdot \sigma \in L|_R$. Let t be a term of $\mathscr{T}(\lfloor L|_R \rfloor)$. Since nothing new can be deduced from L, we have $\lfloor L|_R \rfloor \cup \{B_\varphi\sigma \mid \varphi \cdot \sigma \in L|_R\} \cup \{\neg B_\varphi(\sigma \cup [x \mapsto t]) \mid \varphi \in C\} \not\models_T \bot$. Thus, there is $\varphi \in C$ and a substitution μ such that $\varphi \cdot \mu \in L|_R$ and $\lfloor L|_R \rfloor \models_T \mu \mathrel{\dot{\approx}} \sigma \cup [x \leftarrow t]$. What is more, by Lemma 3.2, $\mathscr{T}(\sigma) \subseteq \mathscr{T}(\lfloor L|_R \rfloor)$. Thus, by immediate induction over φ, $\lfloor L|_R \rfloor \rhd \varphi(\sigma \cup [x \mapsto t])$ and $\lfloor L|_R \rfloor \rhd C(\sigma \cup [x \mapsto t])$. Therefore, $\lfloor L|_R \rfloor \rhd \forall x.C\sigma$.

□

3.1.3.2 Motivating Example

To ensure termination, on a weakly terminating axiomatization, we need to ensure fairness in the choice of instances. Unfortunately, this cannot be done easily. We show on an example that restricting instantiation based on a notion of instantiation level is not enough to ensure fairness.

We define the instantiation level of a formula to be the minimal number of instances needed to obtain the term used for the instance. We consider the following axiomatization W:

$$W = \left\{ \begin{array}{c} \forall x.c \approx c, \forall x.\neg P(x), \forall x.[x \approx c]P(f(x)), \\ \forall x.Q(f(x)), \forall x.x \approx c \vee \top \end{array} \right\}$$

The axiomatization W is weakly terminating. Indeed, the first line can produce an unsatisfiable set of literals in every truth assignment. It is enough to instantiate the first formula with any known term ω to produce the term c, then to use c to instantiate the third formula and get $P(f(c))$, and finally to instantiate the second formula with $f(c)$ and thus obtain $\neg P(f(c))$.

The second line of axioms is designed to induce $\mathscr{L}ift(\mathbf{S})$ to loop on this example. The first formula, $\forall x.Q(f(x))$, has the potential to create an infinite sequence of new terms $f(\omega)$, $f(f(\omega))$... The last formula, $\forall x.x \approx c \vee \top$, can prevent the instantiation of $\forall x.[x \approx c]P(f(x))$ with c by equating it with a previously available term.

The solver $\mathscr{L}ift(\mathbf{S})$ can run forever on $W \cup \{\omega \approx \omega\}$, and so, even if we require that instances are only done if there is no other possible instance of a smaller instantiation level and that all the possible instances of the smallest instantiation level are done at once.

1. $\mathscr{L}ift(\mathbf{S})$ launches the ground solver $\mathbf{S}$ on $[\![R_0]\!] = [\![(W \cup \{\omega \approx \omega\}) \cdot \varnothing]\!]$. It returns $Sat(L_0)$ with $L_0 = [\![R_0]\!]$. The rules NEG UNFOLD and POS UNFOLD cannot be applied. Thus, $\mathscr{L}ift(\mathbf{S})$ generates every possible instance of level 1 in $L_0|_{R_0}$:

$$R_1 = R_0 \cup \left\{ \begin{array}{c} \forall x.c \approx c \cdot \varnothing \wedge \omega \approx \omega \cdot \varnothing \rightarrow c \approx c \cdot \varnothing, \\ \forall x.\neg P(x) \cdot \varnothing \wedge \omega \approx \omega \cdot \varnothing \rightarrow \neg P(x) \cdot [x \mapsto \omega], \\ \forall x.[x \approx c]P(f(x)) \cdot \varnothing \wedge \omega \approx \omega \cdot \varnothing \rightarrow [x \approx c]P(f(x)) \cdot [x \mapsto \omega], \\ \forall x.Q(f(x)) \cdot \varnothing \wedge \omega \approx \omega \cdot \varnothing \rightarrow Q(f(x)) \cdot [x \mapsto \omega], \\ \forall x.x \approx c \vee \top \cdot \varnothing \wedge \omega \approx \omega \cdot \varnothing \rightarrow x \approx c \vee \top \cdot [x \mapsto \omega] \end{array} \right\}$$

2. The ground solver $\mathbf{S}$ returns $Sat(L_1)$ on $[\![R_1]\!]$ with L_1 such that $L_1|_{R_1}$ contains $x \approx c \cdot [x \mapsto \omega]$. $\mathscr{L}ift(\mathbf{S})$ applies NEG UNFOLD to unfold the trigger on $[x \approx c]\ P(f(x)) \cdot [x \mapsto \omega]$:

$$R_2 = R_1 \cup \{[x \approx c]\ P(f(x)) \cdot [x \mapsto \omega] \wedge x \approx c \cdot [x \mapsto \omega] \rightarrow P(f(x)) \cdot [x \mapsto \omega]\}$$

3. The ground solver $\mathbf{S}$ returns $Sat(L_2)$ on $[\![R_2]\!]$ with $L_2 = L_1 \cup P(f(\omega))$. Since $\lfloor L_2|_{R_2} \rfloor \vDash_T \omega \approx c$, every instance with the term c is redundant. $\mathscr{L}ift(\mathbf{S})$ generates every possible instance of level 2 in $L_2|_{R_2}$:

$$R_3 = R_2 \cup \left\{ \begin{array}{c} \forall x.\neg P(x) \cdot \varnothing \wedge Q(f(x)) \cdot [x \mapsto \omega] \rightarrow \neg P(x) \cdot [x \mapsto f(\omega)], \\ \forall x.[x \approx c]P(f(x)) \cdot \varnothing \wedge Q(f(x)) \cdot [x \mapsto \omega] \rightarrow [x \approx c]P(f(x)) \cdot [x \mapsto f(\omega)], \\ \forall x.Q(f(x)) \cdot \varnothing \wedge Q(f(x)) \cdot [x \mapsto \omega] \rightarrow Q(f(x)) \cdot [x \mapsto f(\omega)], \\ \forall x.x \approx c \vee \top \cdot \varnothing \wedge Q(f(x)) \cdot [x \mapsto \omega] \rightarrow x \approx c \vee \top \cdot [x \mapsto f(\omega)] \end{array} \right\}$$

4. The ground solver $\mathbf{S}$ returns $Sat(L_3)$ on $[\![R_3]\!]$ with L_3 such that $L_3|_{R_3}$ does not contain $x \approx c \cdot [x \mapsto \omega]$ nor $P(f(x)) \cdot [x \mapsto \omega]$ anymore, but contains $x \approx c \cdot [x \mapsto f(\omega)]$. Like in step 2, $\mathscr{L}ift(\mathbf{S})$ applies NEG UNFOLD:

$$R_4 = R_3 \cup \{[x \approx c]\ P(f(x)) \cdot [x \mapsto f(\omega)] \wedge x \approx c \cdot [x \mapsto \omega] \rightarrow P(f(x)) \cdot [x \mapsto f(\omega)]\}$$

5. The ground solver **S** returns $Sat(L_4)$ on $[\![R_4]\!]$ with $L_4 = L_3 \cup P(f(f(\omega)))$. Since $\lfloor L_4|_{R_4} \rfloor \vDash_T f(\omega) \approx c$, every instance with the term c is still redundant. We can generate every possible instance of level 3.

   During this run, the solver $\mathscr{L}ift(\mathbf{S})$ will instantiate the third axiom with the terms ω, $f(\omega)$, $f(f(\omega))$... and never with c that would have allowed it to terminate.

3.1.3.3 Termination

As we have seen on the preceding example, enforcing fairness in application of INST is not enough to ensure the termination of $\mathscr{L}ift(\mathbf{S})$ on a weakly terminating axiomatization. Thus, we concentrate here on strongly terminating axiomatizations:

Theorem 3.3 (Termination)**.** *Let W be the Skolemization of an axiomatization Ax. If W is strongly terminating then the solver $\mathscr{L}ift(\mathbf{S})$ will terminate on any set of ground clauses G along with the axiomatization W.*

Proof. Let R be a set of guarded clauses deduced from $(W \cup G) \cdot \varnothing$. Let M be a set of closures. We define the set of *available theory clauses* of R from M, written V_M, to be the set of theory clauses $\{C \mid H \to C \in R$ and, for every $\varphi \cdot \sigma \in H$, $\varphi \cdot \sigma \in M\}$.

We say that a truth assignment A is *coherent* if, for every closure $\varphi \cdot \sigma \in A$, for every theory clause $C \in A$, and for every closure $\varphi \cdot \sigma' \in C$, if $\lfloor A \rfloor \vDash_T \sigma \mathrel{\dot{\approx}} \sigma'$ then $\varphi \cdot \sigma' \in A$.

We define a truth assignment of R to be the limit of the sequence A_i of truth assignments such that $A_0 = \varnothing$ and A_{n+1} is a truth assignment of $V_{A_n} \cup A_n$. Such a limit exists since $A_n \subseteq A_{n+1}$ and there is only a finite number of truth assignments of R since A only contains sub-formulas of elements of R.

We then consider the maximal depth d_A of any instantiation tree of A for every truth assignment A of R. We call S_R the multi-set $\{d_A \mid A$ is a coherent truth assignment of $R\}$. Theoretically, S_R may contain ∞ if there is a coherent truth assignment A of R that has an infinite instantiation tree. In practice, we start with a set of natural numbers since, by definition of strong termination, every truth assignment of $(W \cup G) \cdot \varnothing$ only admits finite instantiation trees and then we show that S_R decreases throughout the derivation according to the standard order on multisets.

Let V be a set of theory clauses and let $C \cdot \sigma$ be a theory clause and $H \subseteq V$ be a set of closures such that $H \to C \cdot \sigma$ can appear as the conclusion of POS UNFOLD, NEG UNFOLD, or INST. We show that $\{d_A \mid A$ is a coherent truth assignment of $V \cup C \cdot \sigma\}$ is smaller than $\{d_A \mid A$ is a coherent truth assignment of $V\}$.

If the clause $H \to C \cdot \sigma$ can be deduced by POS UNFOLD or NEG UNFOLD then the truth assignments of $V \cup C \cdot \sigma$ are exactly the truth assignments of V. Indeed, since the guards H are in V, the rules of Definition 2.10 are applicable. Thus $\{d_A \mid A$ is a coherent truth assignment of $V \cup C \cdot \sigma\}$ is equal to $\{d_A \mid A$ is a coherent truth assignment of $V\}$.

If the clause $H \to C \cdot \sigma$ can be deduced by INST then there is exactly one coherent truth assignment A' of $A \cup C \cdot \sigma$ per coherent truth assignment A of V such that $C \cdot \sigma$ is redundant in A for which we have $d_A = d_{A'}$ plus a bunch of coherent truth assignments A' of $A \cup C \cdot \sigma$ for each coherent truth assignment A of V such that $C \cdot \sigma$ is not redundant in A. For these additional truth assignments, we have $d_A > d_{A'}$. Indeed every instantiation tree for A' can be inserted in a strictly

greater instantiation tree for A that starts with the instance $C \cdot \sigma$. Thus, $\{d_A \mid A$ is a coherent truth assignment of $V \cup C \cdot \sigma\}$ is smaller or equal to $\{d_A \mid A$ is a coherent truth assignment of $V\}$. What is more, if there is a truth assignment A of V such that $C \cdot \sigma$ is not redundant in A then $\{d_A \mid A$ is a coherent truth assignment of $V \cup C \cdot \sigma\}$ is strictly smaller than $\{d_A \mid A$ is a coherent truth assignment of $V\}$.

Let us consider a deduction step from R to R' and let us compare S_R and $S_{R'}$. The set of coherent truth assignments of R' can be constructed by taking every truth assignment A of R and then adding iteratively the clauses of the set of available clauses of R' from A. As we have just shown, this can only decrease S_R.

What is more, if the deduction step from R to R' involves INST steps then no other deduction rule can be applied on $L|_R$. Thus $L|_R \cup V_{L|_R}$ is a truth assignment of R. Moreover, it is coherent since, by hypothesis on the ground solver $\mathbf{S}$, the model L returned by $\mathbf{S}$ in $Sat(L)$ contains every literal of $[\![R]\!]$ that is implied by L. Since $\lfloor L|_R \rfloor \cup \{B_\varphi \sigma \mid \varphi \cdot \sigma \in L|_R\} \not\models_T \bigvee_{\varphi \in C} B_\varphi(\sigma \cup [x \mapsto t])$, the theory clause $C \cdot (\sigma \cup [x \mapsto t])$ is not redundant in $L|_R \cup V_{L|_R}$. Indeed, since L propositionally satisfies $[\![R]\!]$, $L|_R$ propositionally satisfies $V_{L|_R}$ and, if $C \cdot (\sigma \cup [x \mapsto t])$ was redundant in $L|_R \cup V_{L|_R}$, $\bigvee_{\varphi \in C} B_\varphi(\sigma \cup [x \mapsto t])$ would be satisfied by $\lfloor L|_R \rfloor \cup \{B_\varphi \sigma \mid \varphi \cdot \sigma \in L|_R\}$. Thus, $S_{R'}$ must be strictly smaller than S_R.

Thus, if $\mathscr{L}ift(\mathbf{S})$ can run forever on $W \cup G$ then there is an infinite sequence of applications of POS UNFOLD and NEG UNFOLD on a finite set of clauses R. It is impossible because these three rules only deduce sub-formulas of R. □

3.2 Implementation

The implementation was done in OCaml using the OCaml API of Z3 3.2. It represents approximatively 1800 lines of code.

3.2.1 Description

The solver's interface uses a simplified version of SMT-LIB v2 syntax. Sorts as well as functions can be declared. Booleans and integers are built-in, along with some usual operations on them (conjunction, disjunction, and negation for booleans and addition, subtraction, multiplication, and comparison for integers). Boolean formulas may also contain universally quantified formulas with triggers with a positive polarity, that is, universal quantifiers that appear under an even number of negations. Existentially quantified formulas (or universally quantified formulas with a negative polarity) have to be Skolemized. Once a set of boolean formulas has been asserted, the solver $\mathscr{L}ift(\mathbf{S})$ can be launched. It either returns *Sat* or *Unsat*.

The core of the solver is made of two parts. The first part, that we call the wrapper, is in charge of interacting with the ground solver $\mathbf{S}$ (here Z3 3.2) using the SMT-LIB v2 syntax. The second part, called the generator, applies the deduction rules.

The wrapper uses an instance of Z3. It contains the set of ground formulas that is iteratively incremented by the generator. It is also used to assume the model returned by the first solver and use it during the application of the deduction rules. The wrapper takes care of the encoding

of pseudo-literals into opaque literals. It also translates back the relevant closures of the ground models.

The generator repeatedly uses the wrapper to check the validity of the current set R of ground formulas. If the wrapper returns *Sat* with a model L, it goes through the relevant closures of L to apply the deduction rules. For every trigger $[l]C \cdot \sigma$, no matter if it has already been unfolded before, the generator checks whether $\lfloor L|_R \rfloor \rhd l\sigma$. Indeed, it is faster than going through the previous deductions $[\![[l]C \cdot \sigma \wedge \bigwedge M' \to C \cdot \sigma]\!]$ of NEG UNFOLD to check whether $M' \subseteq L|_R$. In the same way, for every universally quantified formula $\forall x.C \cdot \sigma \in L|_R$ and every term $t \in \mathscr{T}(L|_R)$, we check whether $[\![L|_R]\!] \cup \lfloor L|_R \rfloor \vDash_T C \cdot \sigma$ whenever there is $l \cdot \sigma' \in L|_R$ such that $t \in \mathscr{T}(l \cdot \sigma')$ and the ground formula $[\![\forall x.C \cdot \sigma \wedge l \cdot \sigma' \to C \cdot (\sigma \cup [x \mapsto t])]\!]$ has not been deduced yet.

3.2.2 Benchmarks

We test our implementation on several examples using either the theory of arrays or our theory of imperative doubly-linked lists.

On the theory of arrays, we use two families of tests parametrized by a natural n. The first family, named `scoped_updates`, expresses that the value stored in an array at a position j is preserved through n updates at positions different from j:

$$\forall a, i_1, \ldots, i_n, j, v.\, (i_1 \not\approx j \wedge \cdots \wedge i_n \not\approx j) \to get(set(\ldots set(a, i_1, v) \ldots, i_n, v), j) \approx get(a, j)$$

The second family, named `equal_updates`, expresses that the values stored at an index j in a_1 and a_{n+1} are equal if there are indexes $i_1 ... i_n$ different from j such that a_k updated at the index i_k is a_{k+1} updated at the index i_k:

$$\begin{aligned}
&\forall a_1, \ldots, a_{n+1}, i_1, \ldots, i_n, j, v. \\
&\quad (i_1 \not\approx j \wedge \cdots \wedge i_n \not\approx j \wedge \\
&\qquad set(a_1, i_1, v) \approx set(a_2, i_1, v) \wedge \cdots \wedge set(a_n, i_n, v) \approx set(a_{n+1}, i_n, v)) \to \\
&\qquad\quad get(a_1, j) \approx get(a_{n+1}, j)
\end{aligned}$$

The results of these tests are presented in Figure 3.2. The errors come from the fact that Z3 stops after some time returning `Unknown`.

`scoped_updates:`	$n = 1$	0.89
	$n = 2$	6.47
	$n = 3$	27.71
	$n = 4$	Error
`equal_updates:`	$n = 1$	3.14
	$n = 2$	34.60
	$n = 3$	Error

Figure 3.2: Time (in seconds) needed to solve tests for arrays with our implementation

On the theory of list, we have launched a specific test and two families. The specific test, called `replace`, replaces an element in the list and then checks that the elements and the cursors are as expected:

$$\begin{array}{l}\forall co_1, co_2 : list, cu_1, cu_2 : cursor, e : element\_type.\\ \quad (has\_element(co_1, cu_1) \approx \mathtt{t} \wedge has\_element(co_1, cu_2) \approx \mathtt{t} \wedge cu_1 \not\approx c_2 \wedge\\ \quad\ replace\_element(co_1, cu_1, e, co_2) \approx \mathtt{t}) \rightarrow\\ \qquad position(co_2, cu_1) \approx position(co_1, cu_1) \wedge position(co_2, cu_2) \approx position(co_1, cu_2) \wedge\\ \qquad element(co_2, cu_1) \approx e \wedge element(co_2, cu_2) \approx element(co_1, cu_2)\end{array}$$

The first test family, called `delete_length`, deletes n cursors in a list and checks that the length is decreased by n:

$$\begin{array}{l}\forall co_0, co_1, \ldots, co_n : list, cu_1, \ldots, cu_n : cursor.\\ \quad (has\_element(co_0, cu_1) \approx \mathtt{t} \wedge \cdots \wedge has\_element(co_0, cu_n) \approx \mathtt{t} \wedge \bigwedge_{1 \leq i < j \leq n} cu_i \not\approx cu_j \wedge\\ \quad\ delete(co_0, cu_1, co_1) \approx \mathtt{t} \wedge \cdots \wedge delete(co_{n-1}, cu_n, co_n) \approx \mathtt{t}) \rightarrow\\ \qquad length(co_0) \approx length(co_n) + n\end{array}$$

The second test family, called `delete_last`, inserts an element at the beginning of a list of length 4 and then deletes the n last elements ($n < 4$). It checks that the inserted cursor is preserved:

$$\begin{array}{l}\forall co_0, co_1, \ldots, co_{n+1} : list, e : element\_type.\\ \quad (length(co_0) \approx 4 \wedge insert(co_0, first(co_0), e, co_1) \approx \mathtt{t} \wedge\\ \quad\ delete(co_1, last(co_1), co_2) \wedge \cdots \wedge delete(co_n, last(co_n), co_{n+1}) \rightarrow\\ \qquad has\_element(co_{n+1}, first(co_1))\end{array}$$

The results of these tests are presented in Figure 3.3.

Like previously, error denote the fact that Z3 stops after some time returning `Unknown`.

All these tests should be trivial (Z3 solves all of them in less than 0.1 second). The black-box approach gives poor benchmarks because:

- We need to call the solver **S** on every universally quantified formula in $L|_R$ and every term of $\mathscr{T}(\lfloor L|_R \rfloor)$ for which we want to do an instance and every trigger in $L|_R$ that we want to unfold.
- We do not use E-matching instantiation techniques so we generate an instance per couple of a term and a universally quantified formula (and maybe more if there are several literals with the same term).

`replace`:		13.97
`delete_length`:	$n = 0$	0.20
	$n = 1$	0.80
	$n = 2$	3.10
	$n = 3$	11.24
	$n = 4$	35.94
	$n = 5$	Error
`delete_last`:	$n = 0$	2.06
	$n = 1$	27.69
	$n = 2$	Error

Figure 3.3: Time (in seconds) needed to solve tests for lists with our implementation

- Instances and unfolding of triggers have to be done several times because they need to be prefixed by the literals that have caused the unfolding.
- The ground solver **S** has to handle a great deal of formulas as well as numerous push and pop operations in the environment.

3.3 Conclusion

This implementation that relies on a black-box SMT solver conforms with our framework as it yields a decision procedure for every sound, complete, and strongly terminating axiomatization of a theory T'. Still, we have seen that it is not usable in practice as it requires too costly communication with the underlying SMT solver. Alleviating this problem requires a tighter integration of the handling of first-order axioms into an SMT solver's main loop. This is the subject of the next chapter.

4 A White-Box Decision Procedure

Contents

4.1 Description 58

4.1.1 Preliminaries 58

4.1.2 Description of DPLL(T) with triggers 63

4.1.3 Termination Related Constraints 66

4.1.4 Soundness and Completeness 68

4.1.5 Progress and Termination 70

4.2 Implementation 80

4.2.1 E-Matching on Uninterpreted Sub-Terms 80

4.2.2 Different Notions of Termination 82

4.2.3 Inclusion into the Theory Combination Mechanism 83

4.2.4 Comparison with Alt-Ergo's Built-In Quantifier Handling 83

4.3 Conclusion 85

In Chapter 2, we have introduced a semantics for first-order logic with triggers as well of some notions of soundness, completeness, and termination in this logic. We have presented in Chapter 3 a theoretical way to extend a generic ground SMT solver to our logic. We have seen that its implementation results in poor performances so that this impementation does not give a pratical way of extending a ground SMT solver with a new theory.

In this chapter, we try to come up with a more efficient implementation by tightly integrating our mechanism with the solver's ground reasoning. We choose to extend the well-known Abstract DPLL Modulo Theory framework, on which most of the state of the art SMT solvers are based. We then demonstrate 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 4.2, we present 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 described in Section 2.2, not only we obtain completeness and termination, but the eager instantiation allowed by the termination of quantifier handling on this axiomatization results in improved overall performance of the prover.

4.1 Description

In this section, we introduce an extension of abstract DPLL modulo theories [61] 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. For all this section, the background theory T is fixed.

4.1.1 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 2.1.4. Then we convert W into a set of theory clauses (disjunctions of closures) by coupling it with the empty substitution: $W \cdot \varnothing$. We run the procedure on $W \cdot \varnothing$ and G, with one of the three possible outcomes:

- the solver returns *Unsat*, meaning that the union $Ax \cup G$ is unsatisfiable—therefore, if Ax is sound with respect to T', set G is T'-unsatisfiable;
- the solver returns *Sat*, meaning that there exists a ground formula G' such that $G' \models_T G$ 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 do not 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 2.1.2.

To maintain this distinction, the solver works with two distinct kinds of clauses. The clauses coming from Ax are theory clauses: disjunctions of closures that accumulate ground substitutions into free variables. The clauses coming from G are the usual disjunctions of ground literals;

we call them *user clauses* to distinguish them from the clauses of the first kind. The empty clause $\bot$ is considered to be a user clause. A *super-clause* is either a theory clause or a user clause.

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

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

The latter kind appears when we backtrack a decision step over a closure. We extend the $\mathscr{T}$ operation (set of subterms) to 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 φ is a formula under a trigger, a witness, or a universal quantifier, are treated as opaque boxes so that the only terms we can learn from them are the ones brought by substitution σ. An anti-closure $\neg(\varphi \cdot \sigma)$ does not give us any new terms at all (and thus should not be confused with $(\neg\varphi) \cdot \sigma$). Indeed, if the solver at some moment decides to assume a given closure and later reverts this decision, it should not retain the terms learned from that closure.

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

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

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

Any more complex reasoning on guards is left to DPLL. 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 2.4 and 2.5 onto super-literals and guarded clauses.

Definition 4.1 (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 2.4 and 2.5, even though they have different meanings for ordinary literals; in this section, we follow Definition 4.1.

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

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

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

$\vDash^\star_T$ is a conservative extension of the usual first-order implication $\vDash_T$ onto super-literals and guarded clauses. More generally, we have:

Lemma 4.1. *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 σ' such that $L \vDash_T \sigma \mathrel{\dot{\approx}} \sigma'$, $L \blacktriangleright_T \varphi \cdot \sigma$ 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 $S \vDash^\star_T E$, 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. We use the set $\lfloor V \rfloor$ of ground literals readily available from a set of theory clauses V as defined in Definition 2.9:

Definition 4.3. Let F be a set of super-clauses and C a super-clause. We write $F \vdash^\star_T C$ if and only if one of the following conditions holds:

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

Remark that $\vdash^\star_T$ does not coincide with implication modulo T on unit user clauses. Indeed, $\vdash^\star_T$ is used in particular to decide that a clause is unnecessary for the proof and therefore can be forgotten or not generated. In the definition of truth assignment, we state that the solver should assume unit clauses eagerly while it is allowed to postpone deciding on the literals of non-unit clauses. Thus, even if a set of non-unit clauses implies a unit clause C, the solver cannot be allowed to forget C without compromising termination. For example, consider the set of axioms:

$$F = \{c \approx c, f(c) \approx f(c), f(c) \approx c, \forall x[f(c) \approx c].f(x) \approx x, \forall x.f(x) \approx f(x)\}$$

The set F 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 $F \setminus \{f(c) \approx c\} \cdot \varnothing \cup G \vdash^\star_T f(c) \approx c \cdot \varnothing$, and thus $f(c) \approx c$ can be removed from F. We have $f(c) \approx c \vee c \not\approx c \vDash_T f(c) \approx c$. Assume we can remove $f(c) \approx c$ from G. Then, the solver can produce an infinite number of terms from $F \setminus \{f(c) \approx c\}$. 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 4.3, 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 4.2. *Let C be a super-clause and F be a set of super-clauses such that $F \vdash^\star_T C$. For every world L such that $L \blacktriangleright F$, $L \blacktriangleright C$.*

Proof. We have four cases to consider. Assume that C is a unit user clause and $\textsc{Lit}(F) \cup \lfloor \textsc{Clo}(F) \rfloor \vDash_T C$. 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 $F \vdash^\star_T l\sigma$ and $F \cup \mathit{known}(\mathscr{T}(\textsc{Clo}(F))) \vdash^\star_T \mathit{known}(\mathscr{T}(l\sigma))$. 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 $C' \subseteq D$, $F \vdash^\star_T \sigma|_{\mathit{Dom}(\sigma')} \dot{\approx} \sigma'$, and $F \cup \mathit{known}(\mathscr{T}(\textsc{Clo}(F))) \vdash^\star_T \mathit{known}(\mathscr{T}(\sigma|_{\mathit{Dom}(\sigma')}))$. 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 σ into a free variable of φ is known from L, and since σ and σ' substitute the same terms modulo L and T into every free variable of φ, we have $L \rhd \varphi\sigma$. As a consequence, $L \blacktriangleright \varphi \cdot \sigma|_{\mathit{vars}(\varphi)}$ and $L \blacktriangleright C$.

□

Lemma 4.3. *Let C be a super-clause and F be a set of super-clauses such that $F \vdash^\star_T C$. We have $F \vDash^\star_T C$.*

Proof. Let L be a model of $[\![F]\!]$. We have four cases to consider. Assume that C is a unit user clause and $\textsc{Lit}(F) \cup \lfloor \textsc{Clo}(F) \rfloor \vDash_T C$. 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 $F \vdash^\star_T l\sigma$ and $F \cup \mathit{known}(\mathscr{T}(\textsc{Clo}(F))) \vdash^\star_T \mathit{known}(\mathscr{T}(l\sigma))$. 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 $C' \subseteq D$, $F \vdash^\star_T \sigma|_{\mathit{Dom}(\sigma')} \dot{\approx} \sigma'$, and $F \cup \mathit{known}(\mathscr{T}(\textsc{Clo}(F))) \vdash^\star_T \mathit{known}(\mathscr{T}(\sigma|_{\mathit{Dom}(\sigma')}))$. 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 4.4. *Let C be a super-clause and F_1 and F_2 be two sets of super-clauses. If $F_1 \vdash^\star_T F_2$ and $F_2 \vdash^\star_T C$, then $F_1 \vdash^\star_T C$.*

Proof. Assume that C is a unit user clause l and $\textsc{Lit}(F_2) \cup \lfloor \textsc{Clo}(F_2) \rfloor \vDash_T C$. Since $F_1 \vdash^\star_T F_2$, we have $F_1 \vdash^\star_T \textsc{Lit}(F_2)$ and $F_1 \vdash^\star_T \lfloor \textsc{Clo}(F_2) \rfloor$. By definition of $\vdash^\star_T$ 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, $F_1 \vdash^\star_T C$. 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 a literal $l \in D$ such that $F_2 \vdash^\star_T l\sigma$ and $F_2 \cup \mathit{known}(\mathscr{T}(\textsc{Clo}(F_2))) \vdash^\star_T \mathit{known}(\mathscr{T}(l\sigma))$. Like in the previous case, $F_1 \vdash^\star_T l\sigma$. Since $F_1 \vdash^\star_T F_2$, $F_1 \cup \mathit{known}(\mathscr{T}(\textsc{Clo}(F_1))) \vdash^\star_T \mathit{known}(\mathscr{T}(\varphi \cdot \sigma))$ for every closure $\varphi \cdot \sigma \in \textsc{Clo}(F_2)$. As a consequence, $\textsc{Lit}(F_1) \cup \lfloor \textsc{Clo}(F_1) \rfloor \cup \mathit{known}(\mathscr{T}(\textsc{Clo}(F_1))) \models_T \textsc{Lit}(F_2) \cup \lfloor \textsc{Clo}(F_2) \rfloor \cup \mathit{known}(\mathscr{T}(\textsc{Clo}(F_2)))$ and $F_1 \vdash^\star_T C$.

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

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

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

□

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

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

Lemma 4.5. *Let M be a set of super-literals and e a super-literal. If $\lceil M \rceil \vdash^\star_T e$ then $M \models^\star_T e$.*

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

4.1.2 Description of DPLL(T) with triggers

The method introduced below adapts the principles of abstract DPLL modulo theories (following [61]) to super-literals and guarded clauses. We call this adaptation DPLL$^\star(T)$ or DPLL* since T is fixed.

4.1.2.1 Deduction Rules of DPLL$^\star(T)$

The rules are given in Figures 4.1 and 4.2. 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`

`UnitPropagate`:

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

`Decide`:

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

`Fail`:

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

`Restart`:

$$M \parallel F \Longrightarrow \varnothing \parallel F$$

T`-Propagate`:

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

T`-Learn`:

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

T`-Forget`:

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

T`-Backjump`:

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

Figure 4.1: Transition rules of DPLL$^{\star}(T)$ on guarded clauses

`Instantiate`:

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

`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 4.2: Additional transition rules for Abstract DPLL$^{\star}(T)$ on guarded clauses

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 the rules 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 $F, H \models^\star_T C$, 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, $\text{AVB}(F,H) \vdash^\star_T C$. We show below that this distinction is necessary for termination.

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 that are described in Fig. 4.2. 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 $\text{AVB}(F,H) \vdash^\star_T C$.

4.1.2.2 Termination Criteria of DPLL$^\star(T)$

A solver implementing DPLL$^\star(T)$ attempts to construct a model of a set of guarded clauses by using the rules described in Figures 4.1 and 4.2 in an arbitrary way. We finally define when such a solver is allowed to stop, that is, 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 4.1. The solver can return *Unsat* on G if $\varnothing \parallel W \cdot \varnothing \cup G \Longrightarrow^\star \_ \parallel \mathit{fail}$.

Property 4.2. The solver can return *Sat* on G if $\varnothing \parallel W \cdot \varnothing \cup G \Longrightarrow^\star M \parallel F$ where:

(i) $M \vdash^\star_T \text{AVB}(F,M)$,

(ii) $M \not\vdash^\star_T \bot$, and

(iii) if $H \to C$ can be added by either `Instantiate`, `Witness-Unfold`, or `Trigger-Unfold` then $\text{AVB}(F,M) \vdash^\star_T C$.

Remark 4.1. 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 4.2. The relation $\models_T^\star$ 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 4.11 and Corollary 4.2.

Remark 4.3. In classical abstract DPLL modulo theories, conflict driven lemmas, namely formulas allowing to deduce the added element e' in M after an application $Me^{\mathsf{d}}N \parallel F \Longrightarrow Me' \parallel F$ of T`-Backjump`, can be added to F using T`-Learn`. In our framework, this is not the case if e' is an anti-closure since super-clauses cannot contain anti-closures. This restriction can be removed by allowing to deduce guarded clauses $H \to C$ such that $F, H \models_T^\star C$ where C may contain super-literals of all three kinds: literals, closures, and anti-closures. With this modification, if there is $D \subseteq M$ such that $F, D \models_T^\star e'$ and e' or $\neg e'$ occurs in $\textsc{Avb}(F, M) \cup \textsc{Lit}(Me^{\mathsf{d}}N)$, $\textsc{Clo}(M) \to \{\neg e \mid e \text{ is an anti-closure or a literal of } D\} \vee e'$ can be added to F using T`-Learn`.

4.1.3 Termination Related Constraints

In this section, we motivate the constraints on the rules T`-Propagate`, T`-Backjump`, T`-Learn`, T`-Forget`, and `Instantiate` using examples. These constraints are closely related to the definition of termination in Section 2.1.4. 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 $\textsc{Lit}(M) \cup \{l'\sigma' \mid l' \cdot \sigma' \in M\} \models_T l\sigma$ and $\textsc{Lit}(M) \cup \{l'\sigma' \mid l' \cdot \sigma' \in M\} \cup \mathit{known}(\mathscr{T}(M)) \models_T \mathit{known}(\mathscr{T}(l\sigma))$. Only requesting that $M \models_T^\star l\sigma$ would not have been enough. For example, consider the axiomatization $W_1 = \{\forall x.[f(x)]p(f(x)) \approx \mathtt{t}\}$. We can easily check that W_1 is terminating. Indeed, every sub-term of the form $f(t')$ of every truth assignment of $[f(x)]p(f(x)) \approx \mathtt{t} \cdot [x \mapsto t] \cup L \cdot \varnothing$ is either a sub-term of L or a sub-term of t. Still, $M \models_T^\star (f(x) \approx f(x)) \cdot [x \mapsto t]$ for every term $t \in \mathscr{T}(M)$.

As a consequence, for any term t in M, $p(f(x)) \approx \mathtt{t} \cdot [x \mapsto t]$ and then $p(f(x)) \approx \mathtt{t} \cdot [x \mapsto f(t)]$, $p(f(x)) \approx \mathtt{t} \cdot [x \mapsto f(f(t))]$... can be added to M.

In the rule T-`Backjump`, we require that e' or $\neg e'$ occurs in $\text{AVB}(F,M) \cup \text{LIT}(Me^{\mathsf{d}}N)$. Assume that e' or $\neg e'$ is allowed to appear in $Me^{\mathsf{d}}N$ and consider the axiomatization:

$$W_2 = \{\forall y.[p(y) \approx \mathtt{t}]\forall x.f(x,y) \approx x,\ \forall y.[p(y) \approx \mathtt{t}]\forall x.f(x,y) \approx f(x,y),\ c \approx c\}$$

This axiomatization is terminating because as long as we have some $p(t) \approx \mathtt{t}$ 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 $G_2 = \{p(a) \approx \mathtt{t},\ p(a) \not\approx \mathtt{t} \vee p(b) \approx \mathtt{t},\ p(c) \approx \mathtt{t} \vee a \approx a,\ p(a) \not\approx \mathtt{t} \vee a \approx c\}$. We can add $p(a) \approx \mathtt{t}$ 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 $p(b) \not\approx \mathtt{t}$. We now add $p(c) \approx \mathtt{t}$ 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 $p(b) \approx \mathtt{t}$, 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 $f(t,c)$, $f(f(t,c),c)\ldots$ Since we do not have $M \vDash_T p(c) \approx \mathtt{t}$, 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 $\text{AVB}(F,H) \cup \text{LIT}(M)$. Even asking that $\text{AVB}(F,H) \vdash_T^\star C$ 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 $p(a) \approx \mathtt{t}$ and $p(c) \approx \mathtt{t}$ 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 $([p(y) \approx \mathtt{t}]\forall x.f(x,y) \approx f(x,y)) \cdot [y \mapsto a] \wedge (p(y) \approx \mathtt{t}) \cdot [y \mapsto a] \to (\forall x.f(x,y) \approx f(x,y)) \cdot [y \mapsto a]$ 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 $([p(y) \approx \mathtt{t}]\forall x.f(x,y) \approx f(x,y)) \cdot [y \mapsto a] \wedge (p(y) \approx \mathtt{t}) \cdot [y \mapsto a] \to (\forall x.f(x,y) \approx f(x,y)) \cdot [y \mapsto c]$ could be added to F using T-`Learn`. Indeed, $G_3 \cup \{c \approx c \cdot \varnothing,\ (\forall x.f(x,y) \approx f(x,y)) \cdot [y \mapsto a]\} \subseteq \text{AVB}(F, \{([p(y) \approx \mathtt{t}]\forall x.f(x,y) \approx f(x,y)) \cdot [y \mapsto a],\ (p(y) \approx \mathtt{t}) \cdot [y \mapsto a]\})$ 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 $p(a) \approx \mathtt{t}$, $([p(y) \approx \mathtt{t}]\forall x.f(x,y) \approx f(x,y)) \cdot [y \mapsto a]$, $(p(y) \approx \mathtt{t}) \cdot [y \mapsto a]$ and finally $(\forall x.f(x,y) \approx f(x,y)) \cdot [y \mapsto c]$ to M. Because of this closure, we can produce an infinite number of terms $f(t,c)$, $f(f(t,c),c)\ldots$ Since we do not have $p(c) \approx \mathtt{t}$, 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 $\text{AVB}(F,H)$. 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 $\text{AVB}(F,H) \vdash_T^\star C$. Assume that we can forget $H \to C$ as soon as we have $F, H \models_T^\star C$. Consider the axiomatization:

$$W_4 = \{[p(a) \approx \mathtt{t}]\forall x.f(x,a) \approx x, [p(c) \approx \mathtt{t}]\forall x.f(x,c) \approx f(x,c), a \approx c, a \approx a, c \approx c\}$$

Like W_2, W_4 is terminating. We launch the solver on the set of user clauses $G_4 = \{p(a) \approx \mathtt{t}, p(c) \approx \mathtt{t}, p(a) \not\approx \mathtt{t} \vee a \approx c\}$. We can easily check that $W_4 \cdot \varnothing \setminus \{a \approx c \cdot \varnothing\} \cup G_4 \models_T^\star a \approx c \cdot \varnothing$, and therefore we forget it. We can add $p(c) \approx \mathtt{t}$ and the second axiom of W_4 to M using `UnitPropagate`. With `Trigger-Unfold` and then T-`Propagate` and `UnitPropagate` we can add $\forall x.f(x,c) \approx f(x,c)$ to M. Because of this closure, we can produce an infinite number of terms $f(t,c), f(f(t,c),c) \ldots$ Since we do not have $a \approx c$, they are not all equal to t in M.

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

4.1.4 Soundness and Completeness

We show that DPLL$^\star(T)$ is compliant with the semantics defined in Section 2.1.2.

Lemma 4.6. *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 $F_1, H \models_T^\star C$. Since L is complete, by Lemma 4.1, 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 $\blacktriangleright$, $L \cup \mathit{known}(\mathscr{T}(L)) \models_T \mathit{known}(\mathscr{T}(t))$ and so there is $t' \in \mathscr{T}(L)$ such that $L \models_T t \approx t'$. Therefore, $L \blacktriangleright C \cdot (\sigma \cup [x \mapsto t])$.

□

Lemma 4.7. *For every derivation $M_1 \parallel F_1 \Longrightarrow^\star M_2 \parallel F_2$, we have $F_2 \models_T^\star F_1$.*

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, $F \cup H \to C \models_T^\star F_1$. Since $F \vdash_T^\star C$, $F \models_T^\star C$ by Lemma 4.3. □

Theorem 4.1 (T-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 4.8. *If $\varnothing \parallel G \cup W \cdot \varnothing \Longrightarrow^\star M_0 e_1^{\mathsf{d}} M_1 \dots e_n^{\mathsf{d}} M_n \parallel F$, L is a model of $G \cup W \cdot \varnothing$ and $L \blacktriangleright e_1, \dots, e_i$ then $L \blacktriangleright M_i$ for every i in $0 \dots n$.*

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

First note that, by Lemma 4.6, $L \blacktriangleright F$. 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`, $M \models_T^\star e$ and, since L is complete, $L \blacktriangleright e$ by Lemma 4.1. The only remaining rule is T-`Backjump`. There is a subset D of M such that $F \cup D \models_T^\star e$. Since $L \blacktriangleright M$ and $L \blacktriangleright F$, since L is complete, $L \blacktriangleright e$ by Lemma 4.1. □

Thanks to the previous lemma, we can perform the proof of Theorem 4.1. 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 $G' \models_T G$. Since Ax is sound, $W \wedge G'$ has a model L. By Lemma 4.8, $L \blacktriangleright M$. What is more, by Lemma 4.6, $L \blacktriangleright F, H \to C$. With $H \wedge \neg C \subseteq M$, we get a contradiction. □

Theorem 4.2 (T-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 $L \models_T G$ and $W \cup L$ is feasible. Since Ax is complete, L is T'-satisfiable. Since $L \models_T G$, so is G.

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

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

(ii) $M \not\vdash^\star_T \bot$, and

(iii) if $H \to C$ can be added by either `Instantiate`, `Witness-Unfold`, or `Trigger-Unfold` then $\text{AVB}(F,M) \vdash^\star_T C$.

Consider $L = \text{LIT}(M) \cup \{l\sigma \mid l \cdot \sigma \in M\} \cup \{t \approx t \mid t \in \mathscr{T}(M)\}$. By (ii), L is T-satisfiable. We need to show that $L \vDash_T G$ and $L \rhd W$, which is the same as $L \blacktriangleright \text{AVB}(W \cdot \varnothing \cup G, \varnothing)$. It is sufficient to prove that $L \blacktriangleright \text{AVB}(F, \varnothing)$. Indeed, the only rule that can remove an element of $W \cdot \varnothing \cup G$ is T-`Forget` and, if $L \blacktriangleright \text{AVB}(F,\varnothing)$ and $\text{AVB}(F,\varnothing) \vdash^\star_T C$, by Lemma 4.2, $L \blacktriangleright C$.

Now, we only need to show that $L \blacktriangleright \text{CLO}(M)$. Indeed, $M \vdash^\star_T \text{AVB}(F,M)$ is the same as $\text{LIT}(M) \cup \text{CLO}(M) \vdash^\star_T \text{AVB}(F,M)$ and thus $L \blacktriangleright \text{CLO}(M)$ implies $L \blacktriangleright \text{AVB}(F,M)$ by Lemma 4.2. For every closure $\varphi \cdot \sigma \in M$, we prove that $L \rhd \varphi\sigma$ by induction over the size of the formula φ.

- $l \cdot \sigma \in M$. By definition of L, $L \rhd l\sigma$.

- $\forall x.C \cdot \sigma \in M$. Let t be a ground term of L. By definition of L, t is a ground term of M. By (iii), $\text{AVB}(F,M) \vdash^\star_T C \cdot (\sigma \cup [x \mapsto t])$. Since $M \vdash^\star_T \text{AVB}(F,M)$, by Lemma 4.4, $M \vdash^\star_T C \cdot (\sigma \cup [x \mapsto t])$. Therefore, there is $\varphi \in C$ such that either $\varphi \cdot \sigma' \in M$ and $L \vDash_T (\sigma \cup [x \mapsto t]) \doteq \sigma'$ or φ is a literal, $M \vdash^\star_T \varphi\sigma$, and $M \cup \mathit{known}(\mathscr{T}(\text{CLO}(M))) \vdash^\star_T \mathit{known}(\mathscr{T}(\varphi\sigma))$. In the first case, since φ is strictly smaller than $\forall x.C$, we have $L \rhd \varphi\sigma'$ by induction hypothesis and, hence, $L \rhd \varphi\sigma$. In the second case, $L \rhd \varphi\sigma$ by definition of L. By definition of $\rhd$ on universally quantified formulas, $L \rhd (\forall x.C)\sigma$.

- $\langle l \rangle C \cdot \sigma \in M$. By (iii), we have $\text{AVB}(F,M) \vdash^\star_T l \cdot \sigma$ and $\text{AVB}(F,M) \vdash^\star_T C \cdot \sigma$. Since $M \vdash^\star_T \text{AVB}(F,M)$, by Lemma 4.4, $M \vdash^\star_T l \cdot \sigma$ and $M \vdash^\star_T C \cdot \sigma$. Hence, there is $\varphi \in C$ such that either there is a substitution σ' such that $\varphi \cdot \sigma' \in M$ and $M \vdash^\star_T \sigma \doteq \sigma'$ or φ is a literal, $M \vdash^\star_T \varphi\sigma$ and, $M \cup \mathit{known}(\mathscr{T}(\text{CLO}(M))) \vdash^\star_T \mathit{known}(\mathscr{T}(\varphi\sigma))$. In both cases, $L \rhd \varphi\sigma$ with the same reasoning as for universal quantifiers. In the same way, $L \rhd l\sigma$. Therefore, $L \rhd (\langle l \rangle C)\sigma$.

- $[l]C \cdot \sigma \in M$. Assume $L \rhd l\sigma$. By definition of $\rhd$, we have both $M \vdash^\star_T l\sigma$ and $M \cup \mathit{known}(\mathscr{T}(M)) \vdash^\star_T \mathit{known}(\mathscr{T}(l\sigma))$. Thus $\lceil M \rceil \vdash^\star_T l \cdot \sigma$. By (iii), $\text{AVB}(F,M) \vdash^\star_T C \cdot \sigma$. As a consequence, $L \rhd C\sigma$ like in the two previous cases and, by definition of $\rhd$, $L \rhd ([l]C)\sigma$.

□

4.1.5 Progress and Termination

We have shown that DPLL$^\star(T)$ only allows derivations that are compliant with the semantics of Section 2.1. 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 $\text{level}_M(t)$ (resp.

$\text{level}_M(e)$) of a term t (resp. a super-literal e) is either a non-negative integer or a special element ∞. It is defined as the limit of the sequence level^i_M computed in the following manner:

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

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 $\text{level}(M) = \max\{\text{level}_M(e) \mid e \in M\}$. 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 2.1.4, has been reached, and
- every previously available instance of a smaller instantiation level has already been handled.

These two requirements are obtained by a restriction on derivations:

Definition 4.4 (Fairness)**.** We say that a derivation is *fair* if, for every step $\_ \parallel F \Longrightarrow Me \parallel F$ where $\text{level}_M(e) > \text{level}(M)$, e has form $x \approx x \cdot [x \mapsto t]$ and `Instantiate` can be applied to some universal formula $\forall x.\varphi \cdot \sigma$ and the term t in $M \parallel F$. For every such step, if M' is the minimal prefix of M such that $t \subseteq \mathscr{T}(M')$, then there is a prefix N of M containing M' and $\forall x.\varphi \cdot \sigma$ such that:

(a) $N \not\vdash^\star_T \bot$,

(b) for every unit super-clause $e \in \text{AVB}(F, \varnothing)$, we have $\lceil N \rceil \vdash^\star_T e$,

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

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

(e) for every closure $\forall x.\varphi \cdot \sigma \in M'$ such that φ is a unit clause and for every term $t \in \mathscr{T}(M')$ such that $\text{level}_M(\varphi \cdot (\sigma \cup [x \mapsto t])) \leq \text{level}(M)$, we have $\lceil N \rceil \vdash^\star_T \varphi \cdot (\sigma \cup [x \mapsto t])$, and

(f) for every guarded clause $H \to C$ that can be added to the set F by applying `Instantiate`, `Witness-Unfold` or `Trigger-Unfold` on a closure of M', if $\text{level}_M(H) \leq \text{level}(M)$ then $\text{AVB}(F, M) \vdash^\star_T C$.

Remark 4.4. Note that, in a fair derivation, the current instantiation level of every partial model M is finite.

Remark 4.5. 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 4.3 (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 4.6. 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 applications of T`-Learn` and T`-Forget` as well as redundant applications of `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 $M_1 e_1^{\mathsf{d}} M_2 \dots M_n e_n^{\mathsf{d}} M_{n+1}$ where $e_1^{\mathsf{d}} \dots e_n^{\mathsf{d}}$ are the only decision super-literals in M. The order is defined as the lexicographic order on sequences $\sharp M_1 \dots \sharp M_{n+1}$ 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.

As a consequence, we only have to consider sub-derivations that only 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 4.7. 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 $\lceil M \rceil \not\vdash_T^\star l \cdot \varnothing$. 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^M_{i+1} is the biggest truth assignment of $A^M_i \cup C \cdot (\sigma \cup [x \mapsto t])$ such that $\lceil M \rceil \vdash^\star_T \textsc{Unit}(A^M_{i+1})$, if any. We call d^M the maximal i for which A^M_i exists and we define A^M as $A^M_{d^M}$.

For $i \in 0..d^M$, let n^M_i be the number of closures that are in A^M_i but not in A^M_{i-1}, 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 $n^M_i < n_{max}$ for every M and every $i \in 0..d^M$. We call n^M the integer $\sum_{i \in 0..d^M} (n^M_i + 1) \times (n_{max} + 1)^{(d_{max} - i)}$. Note that n^M models a lexicographic order on the finite sequence $n^M_0 \ldots n^M_{d^M}$.

Remark 4.8. 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 4.7, $M \subseteq Z_{m+1}$.

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 $\lceil M' \rceil \vdash^\star_T \textsc{Unit}(A^M)$.

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 4.8. 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) $N \not\vdash^\star_T \bot$,

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

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

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

(e) for every closure $\forall x.\varphi \cdot \sigma \in M'$ such that φ is a unit clause, and for every term $t \in \mathscr{T}(M')$ such that $\text{level}_M(\varphi \cdot (\sigma \cup [x \mapsto t])) \leq \text{level}(M)$, we have $\lceil N \rceil \vdash^\star_T \varphi \cdot (\sigma \cup [x \mapsto t])$, and

(f) for every guarded clause $H \to C$ that can be added to the set F by applying `Instantiate`, `Witness-Unfold` or `Trigger-Unfold` on a closure of M', if $\text{level}_M(H) \leq \text{level}(M)$ then $\text{AVB}(F,M) \vdash^\star_T C$.

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 $\lceil M'' \rceil \vdash^\star_T \text{UNIT}(A^M)$. We need two intermediate lemmas:

Lemma 4.9. *N contains a truth assignment of $A^M_0 \cup W \cdot \varnothing$.*

Proof. By definition of A^M_0, for every literal $l \cdot \varnothing \in \text{UNIT}(A^M_0)$, we have that $\lceil N \rceil \vdash^\star_T l \cdot \varnothing$. Since $\text{UNIT}(\text{AVB}(F,\varnothing)) \vdash^\star_T \text{UNIT}(W \cdot \varnothing)$, by (b), if $l \in W$ then $\lceil N \rceil \vdash^\star_T l \cdot \varnothing$. Moreover, for every closure $\varphi \cdot \varnothing \in \text{UNIT}(W \cdot \varnothing)$ such that φ is not a literal, $\varphi \cdot \varnothing \in \text{UNIT}(\text{AVB}(F,\varnothing))$. By (b), $\lceil N \rceil \vdash^\star_T \text{UNIT}(\text{AVB}(F,\varnothing))$. Therefore, for every closure $\varphi \cdot \varnothing \in \text{UNIT}(W \cdot \varnothing)$ such that φ is not a literal, $\lceil N \rceil \vdash^\star_T \varphi \cdot \varnothing$ and $\varphi \cdot \varnothing \in N$. What is more, we have:

- For every $\langle l \rangle C$ such that $N \vdash^\star_T \langle l \rangle C \cdot \varnothing$, $\lceil N \rceil \vdash^\star_T l \cdot \varnothing$ and, if C is a unit clause φ then $\lceil N \rceil \vdash^\star_T \varphi \cdot \varnothing$ by (c).
- For every $[l]\varphi \in \text{UNIT}(W)$ such that $N \vdash^\star_T [l]C \cdot \varnothing$ and $N \vdash^\star_T l \cdot \varnothing$, we have $\lceil N \rceil \vdash^\star_T \varphi \cdot \varnothing$ by (d).

□

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

Lemma 4.10. *For every closure $\varphi \cdot \sigma$ such that $\lceil N \rceil \vdash^\star_T \varphi \cdot \sigma$, there is a truth assignment A of $A^M \cup \varphi \cdot \sigma$ such that $\lceil N \rceil \vdash^\star_T \text{UNIT}(A)$.*

Proof. We do the proof by structural induction over φ.

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

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

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

Corollary 4.1. *For every guarded clause $H \to C \vee \varphi \cdot \sigma$ that can be obtained by applying either* `Witness-Unfold` *or* `Trigger-Unfold` *such that $\varphi \cdot \sigma \in N$, if $\text{UNIT}(A^M) \vdash^\star_T H$ then $\text{UNIT}(A^M) \vdash^\star_T \varphi \cdot \sigma$. If $\text{UNIT}(A^M)$ is final, then the same is true for any $H \to C \vee \varphi \cdot \sigma$ that can be obtained by applying* `Instantiate`*.*

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

If $H \to C \vee \varphi \cdot \sigma$ can be obtained by `Witness-Unfold`, H is $\langle l \rangle (C \vee \varphi) \cdot \mu$ such that σ is $\mu|_{\mathit{vars}(\varphi)}$. Thus, since $\text{UNIT}(A^M) \vdash^\star_T H$, there is $\langle l \rangle (C \vee \varphi) \cdot \mu' \in A^M$, such that $\text{UNIT}(A^M) \vdash^\star_T \mu \doteq \mu'$ and $\text{UNIT}(A^M) \cup \mathscr{T}(\text{UNIT}(A^M)) \vdash^\star_T \mathit{known}(\mathscr{T}(\mu))$. Since $\lceil N \rceil \vdash^\star_T \text{UNIT}(A^M)$, we have both $\lceil N \rceil \vdash^\star_T \mu \doteq \mu'$ and $\lceil N \rceil \cup \mathscr{T}(N) \vdash^\star_T \mathit{known}(\mathscr{T}(\mu))$. Hence, $\lceil N \rceil \vdash^\star_T \varphi \cdot \mu'|_{\mathit{vars}(\varphi)}$ and, by Lemma 4.10, there is a truth assignment A of $A^M \cup \varphi \cdot \mu'|_{\mathit{vars}(\varphi)}$ such that $\lceil N \rceil \vdash^\star_T \text{UNIT}(A)$. By construction, A is a truth assignment of A^M and, by maximality of A^M, $A = A^M$.

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

If $\text{UNIT}(A^M)$ is final and $H \to C \vee \varphi \cdot \sigma$ can be obtained by `Instantiate`, there is $\forall x.(C \vee \varphi) \cdot \mu$ and $x \approx x \cdot [x \mapsto t] \in H$ such that $\sigma = (\mu \cup [x \mapsto t])|_{\mathit{vars}(\varphi)}$. Since $\text{UNIT}(A^M) \vdash^\star_T H$, there is $\forall x.C \vee \varphi \cdot \mu' \in A^M$ and $t' \in \mathscr{T}(\text{UNIT}(A^M))$ such that $\text{UNIT}(A^M) \vdash^\star_T (\mu \cup [x \mapsto t]) \doteq (\mu' \cup [x \mapsto t'])$ and $\text{UNIT}(A^M) \cup \mathscr{T}(\text{UNIT}(A^M)) \vdash^\star_T \mathit{known}(\mathscr{T}(\mu \cup [x \mapsto t]))$. Since A^M is final, there is $C'' \vee \varphi \cdot \sigma'' \in A^M$ such that $\text{UNIT}(A^M) \vdash^\star_T (\mu' \cup [x \mapsto t']) \doteq \sigma''$. Since $\lceil N \rceil \vdash^\star_T \text{UNIT}(A^M)$, we have both $\lceil N \rceil \vdash^\star_T (\mu \cup [x \mapsto t])|_{\mathit{vars}(\varphi)} \doteq \sigma''$ and $\lceil N \rceil \cup \mathscr{T}(N) \vdash^\star_T \mathit{known}(\mathscr{T}(\mu \cup [x \mapsto t]))$. Thus, $\lceil N \rceil \vdash^\star_T \varphi \cdot \sigma''$ and, by Lemma 4.10, there is a truth assignment A of $A^M \cup \varphi \cdot \sigma''$ such that $\lceil N \rceil \vdash^\star_T \text{UNIT}(A)$. By construction, A is a truth assignment of A^M and, by maximality of A^M, $A = A^M$.

□

Since $N \not\vdash^\star_T \bot$ 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 $t \in \mathscr{T}(\text{UNIT}(A^M))$. If C is not a unit clause, $A^M \cup C \cdot (\sigma \cup [x \mapsto t])$ is a truth assignment of $A^M \cup C \cdot (\sigma \cup [x \mapsto t])$ such that $\lceil N \rceil \vdash^\star_T \text{UNIT}(A^M \cup C \cdot (\sigma \cup [x \mapsto t]))$ which contradicts the definition of A^M. Therefore, C is a unit clause. Since $\lceil M'' \rceil \vdash^\star_T \text{UNIT}(A^M)$, there is a substitution σ' and a term $t' \in \mathscr{T}(M'')$ such that $\forall x.C \cdot \sigma' \in M''$, $M'' \vdash^\star_T \sigma \doteq \sigma'$, $\mathit{known}(\mathscr{T}(M'')) \cup M'' \vdash^\star_T \mathit{known}(\mathscr{T}(\sigma)) \cup \mathit{known}(\mathscr{T}(t))$ and $M'' \vdash^\star_T t \approx t'$. Since $\forall x.C \cdot \sigma'$ and t' are in M'', this instance has an instantiation level smaller or equal to $n^M + 1$. By (e), $\lceil N \rceil \vdash^\star_T C \cdot (\sigma' \cup [x \mapsto t'])$. Since $M'' \subseteq N$, $\lceil N \rceil \vdash^\star_T C \cdot (\sigma \cup [x \mapsto t])$. By Lemma 4.10, there is a truth assignment A of $A^M \cup C \cdot (\sigma \cup [x \mapsto t])$ such that $\lceil N \rceil \vdash^\star_T \text{UNIT}(A)$ 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 $\text{AVB}(F, M) \vdash^\star_T C \cdot (\sigma \cup [x \mapsto t])$, which contradicts the non-redundancy condition of `Instantiate`.

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

that can be obtained by either `Witness-Unfold`, `Trigger-Unfold`, or `Instantiate` such that $\varphi \cdot \sigma \in C$ and $H \subseteq M^\circ$. If $\varphi \cdot \sigma \in C'$, for some $C' \in W \cdot \varnothing$, then $\varphi \cdot \sigma \in A^M$, by construction of A_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 4.1, $\textsc{Unit}(A^M) \vdash_T^\star \varphi \cdot \sigma$.

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 σ' 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 a theory clause $C \cdot \sigma'' \in A^M$ 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 $M'' \vdash_T^\star (\sigma'' \setminus [y \mapsto y\sigma'']) \mathrel{\dot\approx} \tau$, $M'' \vdash_T^\star y\sigma'' \approx s$, 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 4.11 (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 $M_1 e_1^{\mathsf{d}} \dots M_n e_n^{\mathsf{d}} M_{n+1}$ and M_i contain no decision super-literals. We write $\overline{M}_i$ for the prefix $\dots M_i$ of M.

Let us show that, for every $D \subseteq M$ such that $F \cup D \vDash_T^\star \bot$, there is an application $M \parallel F \Longrightarrow M_j \neg e_i \parallel F$ of T`-Backjump`. We do this proof by induction on position of the last and the before-last element of D in M. In other words, we can use the induction hypothesis on a set of super-literals D' if either there is an element of D that appears strictly after every element of D' in M or if the last element e of D in M is in D' and the before-last element of D in M appears strictly after every element of $D' \setminus e$ 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 $\text{AVB}(F, \overline{M}_j)$ (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 $\text{AVB}(F, \overline{M}_{j-1})$[1] then $F \cup (D \setminus e_i) \models^\star_T \neg e_i$ 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 $F \cup D' \models^\star_T e$. 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 $F \models^\star_T H \to C \vee e$ by Lemma 4.7. Thus, $F \cup H \cup \neg C \models^\star_T e$.
- If e was added to M' using T `-Propagate`, then $M' \models^\star_T e$ by Lemma 4.5. Let S be a minimal subset of M' such that $S \models^\star_T e$. We have $F \cup S \models^\star_T e$.
- If e was added to M' using T `-Backjump`, there is a set of super-literals $D \subseteq M'$ and a set of guarded clauses F' such that $F' \cup D \models^\star_T e$ and $F \models^\star_T F'$ by Lemma 4.7. Thus, $F \cup D \models^\star_T e$.

□

Remark 4.9. Compared to usual DPLL modulo theory, back-jumping is restricted by the requirement on T `-Backjump` that e' must appear in $\text{AVB}(F, M)$. This restriction is needed in general but it can be alleviated by allowing to add a subsequence of Me^dN to M using `UnitPropagate` and T `-Propagate` before e' is added with T `-Backjump`.

Corollary 4.2. *If here is a closure or a literal e such that $\neg e \in M$ and $\lceil M \rceil \vdash^\star_T e$, then a conflict clause can be learned so that either* `Fail` *or T* `-Backjump` *can be applied.*

Proof. Since $\lceil M \rceil \vdash^\star_T e$, there is a set of closures $S \subseteq \lceil M \rceil$ such that $S \vdash^\star_T e$. We construct a guarded clause $H \to e$ that can be added to F using T `-Learn`. If e is a literal, let H be S itself. Otherwise, since $\neg e \in M$ is an anti-closure, e occurs in $\text{AVB}(F, M)$. Indeed, a guarded clause $H \to C$ of F cannot be forgotten if there is a closure of C defined in M that does not occur in $\text{AVB}(F \setminus H \to C, H)$. Let $H \subseteq \lceil M \rceil$ be a superset of S such that e occurs in $\text{AVB}(F, H)$. Now, we can add $H \to e$ to F using T `-Learn`. By definition of $\lceil M \rceil$, closures of H either are already in M or can be propagated using T `-Propagate` without breaking the fairness property. As a consequence, $H \to e$ is a conflict clause and either `Fail` or, by Lemma 4.11, T `-Backjump` can be applied on $M \parallel F$. □

Theorem 4.4 (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*

[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.

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 4.10. 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 either `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 4.11, either `Fail` or T`-Backjump` can be applied. Otherwise, there is an undefined super-literal e that occurs in $\textsc{Avb}(F,M)$. Since $e \in \textsc{Avb}(F,M)$, $\text{level}_M(e) \leq \text{level}(M)$ 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 4.2, 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 the set F using either `Instantiate`, `Witness-Unfold`, or `Trigger-Unfold` on M such that either $\textsc{Avb}(F,H) \nvdash_T^\star C$ or we have $\textsc{Avb}(F,H) \vdash_T^\star C$ and $H \nsubseteq M$. If $\textsc{Avb}(F,H) \nvdash_T^\star C$, this 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 4.2. 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) $M \nvdash_T^\star \bot$,

(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 φ 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 φ is a unit clause and for every term $t \in \mathscr{T}(M)$ such that $\text{level}_M(\varphi \cdot (\sigma \cup [x \mapsto t])) \leq \text{level}(M)$, 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 the set F using either `Instantiate`, `Witness-Unfold` or `Trigger-Unfold` on M, if $\text{level}_M(H) \leq \text{level}(M)$, $\textsc{Avb}(F,M) \vdash_T^\star C$.

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

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

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

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

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

If $\text{AVB}(F, M) \vdash^{\star}_{T} \varphi \cdot (\sigma \cup [x \mapsto t])$ then we have $\text{UNIT}(\text{AVB}(F, M)) \vdash^{\star}_{T} \varphi \cdot (\sigma \cup [x \mapsto t])$. By Lemma 4.4, $\lceil M \rceil \not\vdash^{\star}_{T} \text{UNIT}(\text{AVB}(F, M))$ (otherwise, $\lceil M \rceil \vdash^{\star}_{T} \varphi \cdot (\sigma \cup [x \mapsto t])$). Then, there is a guarded clause $H \to e \in F$ such that $H \subseteq M$ and $M \not\vdash^{\star}_{T} e$. Since $H \subseteq M$, e has an instantiation level smaller than the current instantiation level in M and can be added to M using `UnitPropagate`.

Otherwise, the property (f) is false and `Trigger-Unfold`, `Witness-Unfold`, or `Instantiate` can be applied with $\text{level}_M(H) \leq \text{level}(M)$ so that either $\text{AVB}(F, H) \not\vdash^{\star}_{T} C$ or $\text{AVB}(F, H) \vdash^{\star}_{T} C$ and $H \not\subseteq M$. If $\text{AVB}(F, H) \not\vdash^{\star}_{T} C$ then the application of `Trigger-Unfold`, `Witness-Unfold`, or `Instantiate` is non-redundant in F. Otherwise, $\lceil M \rceil \vdash^{\star}_{T} H$ 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 4.2. Otherwise, T`-Propagate` can be applied to $l \cdot \sigma$. Indeed, since $\text{level}_M(H) \leq \text{level}(M)$, it is not forbidden by fairness. □

4.2 Implementation

In this section, we present our implementation of the framework of Section 4.1.2 in the Alt-Ergo theorem prover. We discuss various specificities of this implementation and finally give some benchmarks using the theory of doubly-linked lists given in Section 2.2.1.

4.2.1 *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. First-order SMT solvers commonly use a powerful technique, called E-matching [29], to determine the set of instances for which we will be able to remove the trigger just behind the universal quantifier. In our implementation, we would like to use this technique as much as possible. 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, which we have chosen, is to apply E-matching on parts of triggers on which it is complete and then to check the remaining ones.

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

(i) every free variable (that is not in the domain of σ) in l_φ or k_φ is also in p_φ,

(ii) terms of p_φ only contain variables and uninterpreted symbols,

(iii) if τ is a mapping from free variables of p_φ to terms containing only variables that are in the domain of σ, then, for i in $1..n$:

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

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

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

Example 4.1. Assume that we have a trigger $f(g(z, 2 \times (y + h(z)))) \approx f(y)$ where f, g, and h are uninterpreted and z and y are universally quantified variables. We can compute the three sets:

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

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

4.2.2 Different Notions of Termination

The notion of termination in Section 2.1.4 may turn out to be too constraining for some axiomatization. Let us start with an example. Assume that we want to add to our theory in Section 2.2.1 a predicate symbol that states that a list is a palindrome $is\_palindrome(co : list) : bool$. In an axiomatization of this concept, we could have:

IS_PALINDROME_DEF:

$$\begin{array}{l}\forall co : list.[is\_palindrome(co)]is\_palindrome(co) \approx \mathtt{t} \rightarrow \\ \quad (\forall cu_1 : cursor.[element(co,cu_1)]position(co,cu_1) > 0 \rightarrow \\ \quad\quad \exists cu_2 : cursor.\langle position(co,cu_2) = length(co) - position(co,cu_1) + 1\rangle \\ \quad\quad\quad equivalent\_elements(element(co,cu_1), element(co,cu_2)) \approx \mathtt{t})\end{array}$$

Unfortunately, such an axiom would introduce a loop. If the input set of literals includes the literal $is\_palindrome(co) \approx \mathtt{t}$ and the term $element(co,cu)$ is known for some cu, there is a branch deducing the term $element(co, sko(co,cu))$ which itself allows the deduction of the term $element(co, sko(co, sko(co,cu)))$ and so on. We can see that the term $sko(co, sko(co,cu))$ 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 2.1.4 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 4.1.5. 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 2.1.4, assignments are allowed to contain none). In practice, this amounts to enforcing a lazy instantiation approach, that is to say that new instances can only be generated when enough literals have been assigned a truth value by the model to 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 φ is added to assignments containing a witness $\langle l\rangle\varphi$ or a trigger $[l]\varphi$ with l true. The rules `Witness-Unfold` and `Trigger-Unfold` do no longer have to be applied eagerly (before new instances are made).

The first two alternatives would allow the proof of termination of the *is_palindrome* example to go through. The first one has the drawback of forbidding 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. Termination becomes difficult to achieve with this modification since any universal quantifier can be turned into an existential one and then cause the creation of a new term. To alleviate it, we do not allow negations of closures to

be added for the last, or the unique, element of a disjunction. This restriction is motivated by the fact that axioms are often written as implications, the guards being in general simple enough to avoid causing too many new instances.

In general, this gives a more constraining version of termination. Still, if every pseudo-clause occurring in an axiomatization can be written $l_1 \to \ldots l_n \to \varphi$, which is the case for the theory of doubly-linked lists, then only anti-closures of literals can be added to truth assignment. Such anti-closures can only shorten branches in which they are introduced and, therefore, cannot compromise termination. As compensation for this generally more constraining version of termination, new rules can be added to DPLL$^\star(T)$ to handle anti-closures. They have to work in a compatible way with the semantics of negations of closures defined for the proofs of Section 4.1.4. 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.

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

4.2.3 Inclusion into the Theory Combination Mechanism

The notion of satisfiability in Section 2.1.2 can also turn out to be too constraining. For example, in the proof of completeness of Section 2.2.3, we need to show that adding equalities on known terms of integer type does 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 $i \in 1..n$. Note that we need to modify the notion of termination accordingly. In the definition of truth assignment, we need to also consider the disjunctions coming from non-convex theories. Our implementation in Alt-Ergo complies with this second definition by using the disjunctions generated by the theory combination mechanism.

4.2.4 Comparison with Alt-Ergo's Built-In Quantifier Handling

We use the Why3 VC generator version 0.80 and the Alt-Ergo theorem prover version 0.95.2. The implementation instantiates every universally quantified formula of the theory before deciding on literals.

	Separated Axioms		Grouped Axioms	
	Alt-Ergo*	Alt-Ergo	Alt-Ergo*	Alt-Ergo
`test_delete`	1.36	2.95	**1.30**	1.96
`test_insert`	47.25	175.73	**41.39**	500.20
`double_size`	22.79	**13.85**	16.02	47.30
`filter`	21.29	19.52	20.30	20.70
`my_contain`	0.65	2.66	**0.36**	1.70
`my_find`	12.29	104.06	**8.64**	189.76
`map_f`	15.53	13.49	**9.90**	25.14

Figure 4.3: 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*).

Figure 4.3 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 use two slightly different versions of the axiomatization. The first one, named *Separate Axioms* is the one presented in Section 2.2. In the second one, we have grouped the axioms for each function symbol into one axiom. For example, for *delete*, we now have only one axiom:

DELETE_DEF:

$$\begin{array}{l}
\forall co_1 co_2 : list, cu : cursor.[delete(co_1, cu, co_2)]\ delete(co_1, cu, co_2) \approx \mathtt{t} \rightarrow \\
\quad position(co_1, cu) > 0 \wedge length(co_2) + 1 \approx length(co_1) \wedge \langle next(co_1, cu)\rangle \mathtt{t} \wedge \\
\quad (\forall cu_2 : cursor.[position(co_1, cu_2)] \\
\qquad (position(co_1, cu_2) < position(co_1, cu) \rightarrow \\
\qquad\qquad position(co_1, cu_2) \approx position(co_2, cu_2)) \wedge \\
\qquad (position(co_1, cu_2) > position(co_1, cu) \rightarrow \\
\qquad\qquad position(co_1, cu_2) \approx position(co_2, cu_2) + 1)) \wedge \\
\quad (\forall cu_2 : cursor.[position(co_2, cu_2)] \\
\qquad (0 < position(co_2, cu_2) < position(co_1, cu) \rightarrow \\
\qquad\qquad position(co_1, cu_2) \approx position(co_2, cu_2)) \wedge \\
\qquad (position(co_2, cu_2) \geq position(co_1, cu) \rightarrow \\
\qquad\qquad position(co_1, cu_2) \approx position(co_2, cu_2) + 1)) \wedge \\
\quad (\forall cu_2 : cursor.[element(co_1, cu_2)] \\
\qquad position(co_2, cu_2) > 0 \rightarrow element(co_1, cu_2) = element(co_2, cu_2))
\end{array}$$

Indeed, Alt-Ergo, on these examples, works better with separated axioms. It uses a lazy instantiation technique that matches every available trigger once in a while. With the separated axioms, it is the case that it often can instantiate every needed axiom at once while, with the grouped version, it need several matching rounds. On the other hand, the theory mechanism works better with the grouped version as it has less triggers to match.

We see that, on average, our implementation gives better results than the usual instantiation mechanism of Alt-Ergo. This is mainly due to the fact that instances are now generated in an eager way. Note that this is also a drawback in some cases where a lot of work is done even

if it was not necessary. This is all the more the case when there are things to prove that do not require theory handling and when the amount of work required by the theory is important.

4.3 Conclusion

The formalization of this white box implementation inside DPLL(T) is significantly more complex than the formalization of the black box implementation in Chapter 3. It is also much more satisfactory in practice as our implementation in the SMT solver Alt-Ergo needs a comparable amount of time to discharge VCs on doubly-linked list as Alt-Ergo's built-in quantifier handling.

Missing in the current implementation is the ability to handle literals as triggers. Like other SMT solvers, Alt-Ergo only accepts triggers that are terms and it would require significant modifications throughout the source to enable literal triggers. Extending Alt-Ergo to literal triggers would be interesting, for example, to handle the extensionality axiom of the theory of arrays. A first approach to handle such triggers would be to match every subterm of such a trigger modulo equality and then to check if the trigger holds in the current partial model. This implementation would not be efficient, in particular for literals that do not have non-variable subterms like $x \not\approx y$. Coming up with an efficient handling of matching on these triggers would require a tight integration with the theory solvers in order to be able to compute efficiently the set of ground literals true in the current partial model which correspond to a given pattern. This is one direction that we consider for future work.

We are now ready to experiment on more complex case studies.

5 Case Study: Set Theory of Why3

Contents

5.1 Context: the B Method 87
5.2 A Decision Procedure for Why3's Sets 88
5.2.1 Presentation of the Theory 88
5.2.2 Description of the Axiomatization 89
5.2.3 Proofs of Soundness, Completeness, and Termination 90
5.3 Benchmarks 92
5.3.1 Assessment of the Adequacy Between our Framework and Usual E-matching Techniques 93
5.3.2 Comparison Between our Implementation and the Built-in Quantifiers Handling of Alt-Ergo 96
5.4 Conclusion 96

In Chapter 4, we explained how DPLL(T) can be extended to support first-order logic with triggers. We have also shown that this extension yields a decision procedure when given a sound, complete, and terminating axiomatization as defined in Chapter 4. In this chapter, we apply this approach on the theory of sets provided in the Why3 standard library. This theory is a basis of a larger library of Why3 theories used to formalize the set theory of the B method, which is used in the ANR project BWare[1]. It is an industrial research project that aims to provide a framework allowing to automatically discharge verification conditions coming from the verification of safety critical industrial applications using the B method. A generic verification platform is designed that uses several solvers as a back-end. As part of this project, verification conditions coming from the B method are translated into the WhyML language using an axiomatization for polymorphic sets. They can then be discharged by SMT solvers like Alt-Ergo. A benchmark of more than 10,000 verification conditions is provided in this project.

5.1 Context: the B Method

The B method [18, 67] is a state-based method for designing formally verified software. It starts from a very abstract model of the system that can then be refined until code generation. Properties over the models' data are specified as a mathematical invariant using Zermelo-Fraenkel set

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

theory with the axiom of choice. Mathematical formulas are then generated to ensure both the consistency of the model and the preservation of the invariant through every operations of the model. At each refinement, the invariant becomes more precise as it deals with a more concrete model. This refinement mechanism allows to split the complexity of proof of the final, concrete model, into smaller, simpler proofs.

Verification conditions generated by the B Method are expressed in first-order logic with notations from the Zermelo-Fraenkel set theory with the axiom of choice. This theory includes:

- membership of an element into a set,
- usual set operators like union and intersection between sets,
- the power set of a set s, that is the set containing every subset of s,
- cartesian product of two sets s_1 and s_2, that is the set of all the pairs of an element of s_1 and an element of s_2,
- set comprehension, that is the set of the elements of a set that abide by a given property,
- relations between elements of two sets s_1 and s_2, represented as sets of pairs (x_1, x_2) where x_1 is an element of s_1, x_2 is an element of s_2, and x_1 and x_2 are in relation,
- partial functions, that are relations such that there is at most one pair $x_1, \_$ in the relation for each x_1, and
- total functions, that are partial functions such that there is exactly a pair $x_1, \_$ in the relation for each x_1.

5.2 A Decision Procedure for Why3's Sets

In this section, we present a minimal set theory which we then axiomatize using first-order logic with triggers. This axiomatization is then used to verify verification conditions coming from the BWare project.

5.2.1 Presentation of the Theory

The Why3 platform provides several theories for polymorphic sets. The simplest one, called generic sets, is based on the notion of membership. Sets are described using the function symbol $mem(x : \alpha, s : set\,\alpha) : bool$ which returns true if and only if the element x is in the set s. The function $subset(s_1 : set\,\alpha, s_2 : set\,\alpha) : bool$ returns true if and only if the set s_1 is the subset of a set s_2, that is, every element of s_1 i a member of s_2. Two sets are equal if and only if they contain the same elements.

The theory describes several functions over sets. The function $add(x : \alpha, s : set\,\alpha) : set\,\alpha$ returns the set containing both the element x and the elements of the set s, $remove(x : \alpha, s : set\,\alpha : set\,\alpha)$ returns the set containing every element of s except x, and $union(s_1 : set\,\alpha, s_2 : set\,\alpha) : set\,\alpha$, $inter(s_1 : set\,\alpha, s_2 : set\,\alpha) : set\,\alpha$, and $diff(s_1 : set\,\alpha, s_2 : set\,\alpha) : set\,\alpha$ return respectively the union, the intersection, and the difference of two sets s_1 and s_2.

The function $is\_empty(s : set\,\alpha) : bool$ returns true if the set s is equal to the empty set $empty$. An unspecified element contained in a non-empty set s can be retrieved using $choose(s : set\,\alpha) : \alpha$.

5.2.2 Description of the Axiomatization

We only describe some axioms in this section. The whole axiomatization is available in Appendix B.1. Equality between sets is modeled by a the function symbol $equal\_sets(s_1 : set\,\alpha, s_2 : set\,\alpha) : bool$. For two sets s_1 and s_2, $equal\_sets(s_1, s_2)$ is true if and only if s_1 and s_2 contain the same elements. The function $equal\_sets$ is then related to classical equality by an extensionality axiom. It states that, if $equal\_sets(s_1, s_2)$ is true then s_1 and s_2 must be equal. Note that since $equal\_sets(s_1, s_2) \approx \mathtt{t}$ can be implied by a set of ground literals L without the term $equal\_sets(s_1, s_2)$ to appear in L, we do not put $equal\_sets(s_1, s_2)$ as a trigger for EXTENSIONALITY. We rather use $s_1 \not\approx s_2$.

EQ_DEF :

$$\begin{aligned}\forall s_1, s_2 : set\,\alpha.[equal\_sets(s_1, s_2)]equal\_sets(s_1, s_2) \approx \mathtt{t} \rightarrow \\ (\forall x : \alpha.[mem(x, s_1)]mem(x, s_1) \approx \mathtt{t} \rightarrow mem(x, s_2) \approx \mathtt{t}) \wedge \\ (\forall x : \alpha.[mem(x, s_2)]mem(x, s_2) \approx \mathtt{t} \rightarrow mem(x, s_1) \approx \mathtt{t})\end{aligned}$$

EQ_INV :

$$\begin{aligned}\forall s_1, s_2 : set\,\alpha.[equal\_sets(s_1, s_2)]equal\_sets(s_1, s_2) \not\approx \mathtt{t} \rightarrow \\ (\exists x : \alpha.mem(x, s_1) \approx \mathtt{t} \wedge mem(x, s_2) \not\approx \mathtt{t} \vee mem(x, s_2) \approx \mathtt{t} \wedge mem(x, s_1) \not\approx \mathtt{t}))\end{aligned}$$

EXTENSIONALITY :

$$\forall s_1, s_2 : set\,\alpha.[s_1 \not\approx s_2]equal\_sets(s_1, s_2) \approx \mathtt{t} \rightarrow s_1 \approx s_2$$

For *union*, *inter*, and *diff*, we describe the relationship between members of the set arguments and members of the result. We put triggers so that the axioms can only be applied to deduce new memberships, that is, positive literals starting with the function symbol *mem*. Indeed, every element that is not known to be in a set can safely be assumed not to be in it.

UNION_DEF :

$$\begin{aligned}\forall s_1, s_2 : set\,\alpha, x : \alpha.[mem(x, union(s_1, s_2))] \\ mem(x, union(s_1, s_2)) \approx \mathtt{t} \rightarrow mem(x, s_1) \approx \mathtt{t} \vee mem(x, s_2) \approx \mathtt{t}\end{aligned}$$

UNION_INV1 :

$$\forall s_1, s_2 : set\,\alpha, x : \alpha.[union(s_1, s_2), mem(x, s_1)]mem(x, s_1) \approx \mathtt{t} \rightarrow mem(x, union(s_1, s_2)) \approx \mathtt{t}$$

UNION_INV2 :

$$\forall s_1, s_2 : set\,\alpha, x : \alpha.[union(s_1, s_2), mem(x, s_2)]mem(x, s_2) \approx \mathtt{t} \rightarrow mem(x, union(s_1, s_2)) \approx \mathtt{t}$$

For *add* and *remove*, we also describe the result of the modification using the *mem* function symbol. As for *union*, we choose triggers so that the axioms can only be used to deduce new memberships. We also need a special case so that $mem(y, add(y,s)) \approx \mathtt{t}$ can be deduced whenever $add(y,s)$ is known. Indeed, otherwise, *mem* would become incompletely generated as we need to deduce that $mem(y, add(y,s)) \approx \mathtt{t}$ is true in the theory even if the term $mem(y, add(y,s))$ is not known. For example, to deduce that the set of literals $\{add(x, empty) \approx add(y, empty), x \not\approx y\}$ is unsatisfiable we need to know that $mem(y, add(y, empty)) \approx \mathtt{t}$.

ADD_DEF :

$$\forall x, y : \alpha, s : set\,\alpha.[mem(x, add(y,s))]mem(x, add(y,s)) \approx \mathtt{t} \rightarrow x \approx y \vee mem(x,s) \approx \mathtt{t}$$

ADD_INV1:

$$\forall x, y : \alpha, s : set\,\alpha.[add(y,s), mem(x,s)]mem(x,s) \approx \mathtt{t} \rightarrow mem(x, add(y,s)) \approx \mathtt{t}$$

ADD_INV2 :

$$\forall y : \alpha, s : set\,\alpha.[add(y,s)]mem(y, add(y,s)) \approx \mathtt{t}$$

If a set is empty then it does not contains any element. The function *choose* returns an element contained in a non-empty set. If a set s is non-empty then it contains $choose(s)$.

IS_EMPTY_DEF :

$$\forall s : set\,\alpha.[is\_empty(s)]is\_empty(s) \approx \mathtt{t} \rightarrow (\forall x : \alpha.[mem(x,s)]mem(x,s) \not\approx \mathtt{t})$$

CHOOSE_DEF:

$$\forall s : set\,\alpha.[is\_empty(s)]is\_empty(s) \not\approx \mathtt{t} \rightarrow mem(choose(s), s) \approx \mathtt{t}$$
$$\forall s : set\,\alpha.[choose(s)]is\_empty(s) \not\approx \mathtt{t} \rightarrow mem(choose(s), s) \approx \mathtt{t}$$

Remark that we use the version of CHOOSE_DEF with $is\_empty(s)$ as a trigger to express the opposite direction of the implication in the definition of *is_empty*.

5.2.3 Proofs of Soundness, Completeness, and Termination

In this section, we show that we can use the white-box implementation defined in Chapter 4 on the axiomatization of sets as a decision procedure for the axiomatization of sets where triggers are replaced by implications.

Theorem 5.1. *The axiomatization in Section 5.2.2 is terminating, sound and complete with respect to the same axiomatization where triggers are replaced by implications.*

5.2.3.1 Proof of Termination

In the axiomatization, quantified variables are either of type α or of type $set\,\alpha$. The axiomatization does not create any term of type $set\,\alpha$. The axioms EQ_INV, SUBSET_INV, and CHOOSE_DEF can all create a new term of type α. These three axioms only quantify over variables of type $set\,\alpha$. Since this type is uninterpreted by the underlying theory T of the SMT solver, no term of type $set\,\alpha$ can be created by T. As a consequence, the axiomatization can create a finite number of new terms of type α.

5.2.3.2 Proof of Soundness

We show that, if a set of literals G has a model in the axiomatization of sets where triggers are replaced by implications then there is a total model of G and the axiomatization. If I is a model of a set of literals G with the axiomatization of sets where triggers are replaced by implications, we define $L = \{l \mid I(l) = \top\}$. By construction of L, L is a total model of G. Since L is total, for every axiom φ of the axiomatization, $L \rhd_T \varphi$.

5.2.3.3 Proof of Completeness

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 axiomatization of sets where triggers are replaced by implications. Since $L \rhd_T G$, it is also a model of G.

First of all, we give a value to $mem(x,s)$ for each known term s of type set and each known term x of type α in L. For every s and x such that $L \not\models_T mem(x,s) \approx \mathtt{t}$, we add the literal $mem(x,s) \not\approx \mathtt{t}$ to L. Since mem is uninterpreted, the set of literals L is still satisfiable in T.

What is more, the axiomatization is still true in L. Indeed, every clause in the axiomatization protected by a trigger containing $mem(x,s)$ contains a negative occurrence of $mem(x,s) \approx \mathtt{t}$. As a consequence, they are all automatically satisfied when assuming $mem(x,s) \not\approx \mathtt{t}$.

Then, we need to handle extensionality. For every known terms s_1 and s_2 of type set in L such that, for every known term x of type α, we have $L \models_T mem(x,s_1) \approx \mathtt{t}$ if and only if we have $L \models_T mem(x,s_2) \approx \mathtt{t}$, we add $s_1 \approx s_2$ to L. The set of literals L is still satisfiable. Indeed, since the type set is uninterpreted in the background theory T, if the set L implies a finite clause $s_1 \not\approx s'_1 \vee \cdots \vee s_n \not\approx s'_n$ modulo T then there is i in $1..n$ such that $s_i \not\approx s'_i$. Thus by EXTENSIONALITY and EQ_INV, there is a term x of type α such that $L \models_T mem(x,s_i) \not\approx mem(x,s'_i)$.

What is more, the axiomatization is still true in L. There can be no trigger $l[s_1]$ such $L \not\models_T l[s_1]$ and $L \models_T l[s_2]$ if, for every known term x of type α, $L \models_T mem(x,s_1) \approx \mathtt{t}$ if and only if $L \models_T mem(x,s_2) \approx \mathtt{t}$. Indeed, such a case can only happen if either there is a trigger containing twice the same set variable, in which case a new equality may allow this trigger to match, or if there are two nested triggers with the same set variable. The first case never occurs in the axiomatization. As stated previously, using $mem(add(y,s),y)$ as a trigger for ADD_INV2 would have cause such a case to occur and the axiomatization to be incomplete as it could not have found the unsatisfiability in $\{x \not\approx y, add(x,empty) \approx add(y,empty)\}$. The second case though, occurs in several axioms, but, each time, all the nested triggers but one are applications of mem. Since every term $mem(x,s)$ is known in L for s and x known in L, these instances are redundant.

Let us now handle *choose* and *is_empty*. For every known term s of type set in L, if s is not equal to *empty* then add $is\_empty(s) \not\approx \mathtt{t}$ to L. If there is no term x in L such that $L \models_T mem(x,s) \approx \mathtt{t}$ then we already have $L \models_T empty \approx s$. In every other case, if the term $choose(s)$ is not known in s, then pick a term x such that $L \models_T mem(x,s) \approx \mathtt{t}$ and add $choose(s) \approx x$ to L. The set L is still satisfiable. Since, when assuming a new term $is\_empty(s)$, we also ensure $mem(choose(s),s) \approx \mathtt{t}$, the axiomatization is still true in L.

Finally, we simply add appropriate values to membership of missing terms starting with *subset*, *add*, *remove*, *inter*, *union*, and *diff*.

5.3 Benchmarks

In this section, we compare the efficiency of usual E-matching techniques of several SMT solvers on our axiomatization with and without user provided triggers. We also compare Alt-Ergo built-in, lazy, instantiation technique with our, eager, implementation.

From the axiomatization in Section 5.2.2, we remove the extensionality axiom and we leave it as a first-order formula without triggers. Indeed, non-tautological triggers are neither handled by off-the-shelf SMT solvers nor supported by our implementation in Alt-Ergo.

The set theory used by the B method supports constructs that are not part of the Why3 set theory. They can be axiomatized with appropriate first order axioms. That is what was done for $power(s)$, that returns the power set of a set s, and for $interval(i,j)$ that represents the set of integers between i and j.

MEM_POWER:

$$\forall s_1, s_2 : set\,\alpha.[mem(s_1, power(s_2))]\ mem(s_1, power(s_2)) \approx \mathtt{t} \leftrightarrow subset(s_1, s_2) \approx \mathtt{t}$$
$$\forall s_1, s_2 : set\,\alpha.[power(s_2), subset(s_1, s_2)]\ mem(s_1, power(s_2)) \approx \mathtt{t} \leftrightarrow subset(s_1, s_2) \approx \mathtt{t}$$

MEM_INTERVAL:

$$\forall x, i, j : int.[mem(x, interval(i, j))]\ mem(x, interval(i, j)) \approx \mathtt{t} \leftrightarrow i \leq x \leq j$$

These axioms are terminating and therefore can be added to the theory of sets without compromising termination of the solver. Indeed, none of these axioms can create any new term. Remark that they are not complete though. The sets of literals $\{is\_empty(interval(2,4)) \approx \mathtt{t}\}$ and $\{is\_empty(power(s)) \approx \mathtt{t}\}$ are unsatisfiable in the theory of sets extended with *power* and *interval* but this cannot be deduced by our axiomatization. For these axioms, we have not tried to achieve completeness. Indeed, it would have required generating too many instances, hindering efficiency (one per integer between i and j for $interval(i,j)$ and one per term of type $set\,\alpha$ for *power*.

Among the additional first-order axioms that are used by Why3 to model the set theory used by the B method, are a couple of axioms that are specializations of the extensionality axiom:

ADD_REMOVE:

$$\forall x : \alpha, s : set\,\alpha.[add(x, remove(x, s))] mem(x, s) \approx \mathtt{t} \rightarrow add(x, remove(x, s)) \approx s$$

UNION_SINGLETON_ADD:

$$\forall x : \alpha, s : set\,\alpha.[union(s, add(x, empty))] union(s, add(x, empty)) \approx add(x, s)$$

Those axioms are theoretically redundant but not necessarily useless as properly choosing when to apply the extensionality axiom is difficult. We have done two runs of the BWare bechmarks containing a total of 10572 verification conditions with and without these axioms.

	No lemma		With lemmas	
	No trigger	With triggers	No trigger	With triggers
Z3 4.3.1	8994	8586	8495	8649
CVC4 1.1	8763	8768	8774	8781
Alt-Ergo	9726	9761	9381	9662

Figure 5.1: Number of verification conditions from the BWare benchmarks discharged by Alt-Ergo, Z3 4.3.2, and CVC4 1.1 with a timeout of 60s with and without the triggers.

5.3.1 Assessment of the Adequacy Between our Framework and Usual E-matching Techniques

In Figure 5.1, we compare the number of verification conditions discharged by Alt-Ergo 0.95.2, Z3 4.3.2, and CVC4 1.1 with and without user provided triggers in the axiomatization. There are two versions of the benchmarks, one includes additional lemmas for extensionality and the other does not. Figures 5.3 and 5.2 represents the time needed to solve the n fastest verification conditions with each prover, depending on n.

We see that the effect of the additional lemmas depends on the solver. While they seem to hinder Alt-Ergo a lot, they allow CVC4 to discharge a handful more of verification conditions. We can also see that, when the additional lemmas are present, manually written triggers are beneficial for both Alt-Ergo and Z3. Indeed, an SMT solver can make an unfortunate choice for the triggers of additional lemmas for extensionality like ADD_REMOVE and use the term $mem(x,s)$. Such a choice leads to useless instances thus hindering the solver.

The manually written triggers for the set theory do not seem to have a major impact on the efficiency of solvers. A notable exception to this remark is Z3. We can see on Figure 5.2 that the two runs of Z3 on our axiomatization without the additional axioms are close enough until a limit of 10s is reached. Then the curve without the triggers has a bend, as Z3 discharges more verification conditions without the triggers than with them. This bend may be caused for example by a change in the instantiation heuristic triggered by either a big number of instances or a time limit.

To explain this feeble impact of user provided triggers without the lemmas for extensionality, we need to consider the heuristics used inside SMT solvers for choosing instantiation patterns for a quantified formula $\forall x_1 \ldots \forall x_n . \varphi$. The most usual one consists in picking a subterm of φ whose free variables are exactly $x_1 \ldots x_n$ if any. We remark that, on our axiomatization, this heuristic always hands out triggers that are restrictive enough to enforce termination even though they do not ensure completeness.

The fact that we lose the completeness of our triggers will not necessarily lead to poorer performances, especially for Z3 and CVC4 which have other heuristics for instantiating triggers than E-matching. Indeed, while some instances are useful in general for completeness they may very well be useless on our benchmarks and therefore slow down the solver.

Note that a possible reason for explaining why CVC4 and Z3 are less efficient than Alt-Ergo on these verification conditions is that we use the SMT-LIB format for them which does not support polymorphism. An instance of each lemma is therefore generated for each set type in

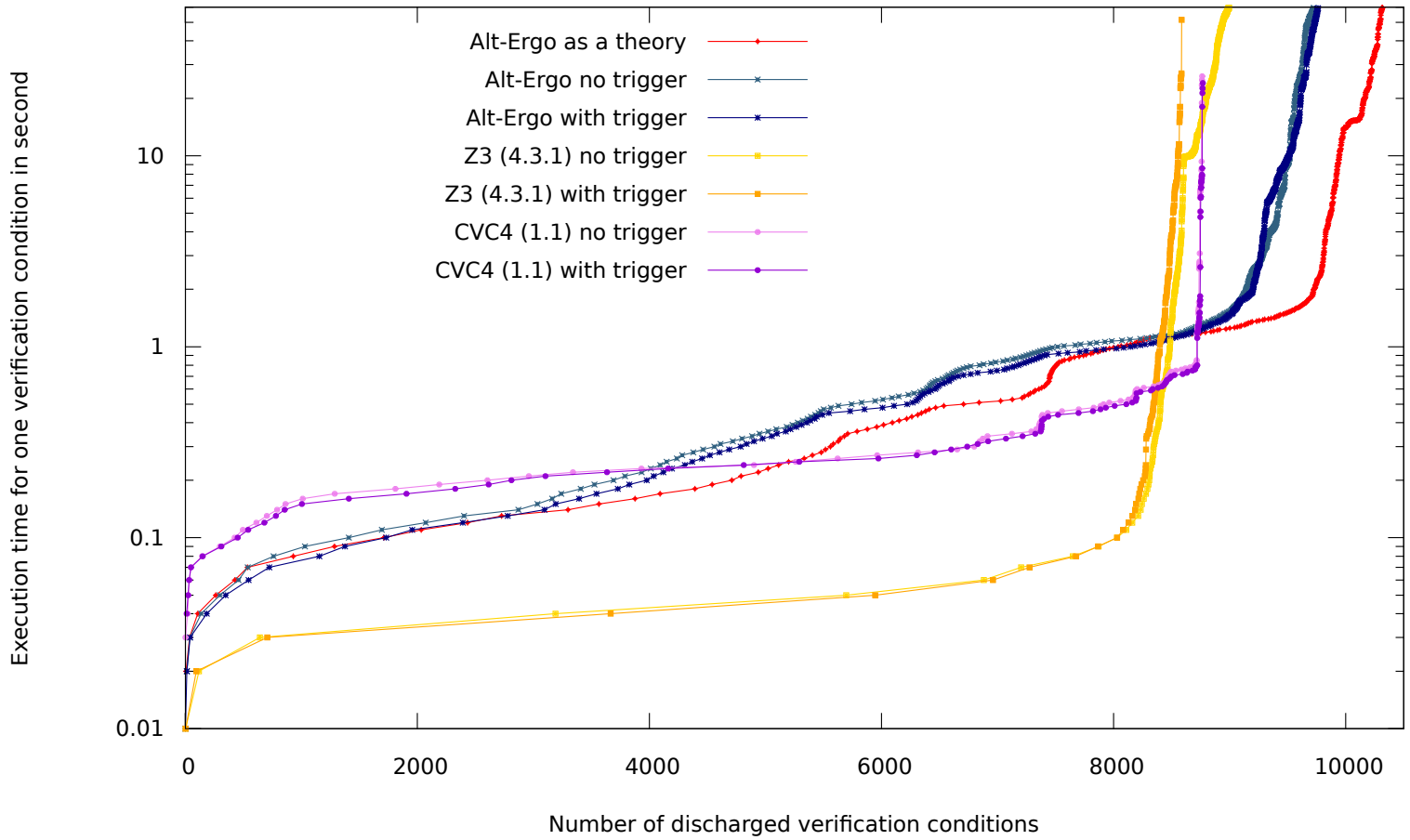


Figure 5.2: Evolution of the number of verification conditions from the BWare benchmarks without additional extensionality lemmas discharged by Alt-Ergo, Z3 4.3.2, and CVC4 1.1 with a timeout of 60s with and without the triggers and by Alt-Ergo through the theory mechanism.

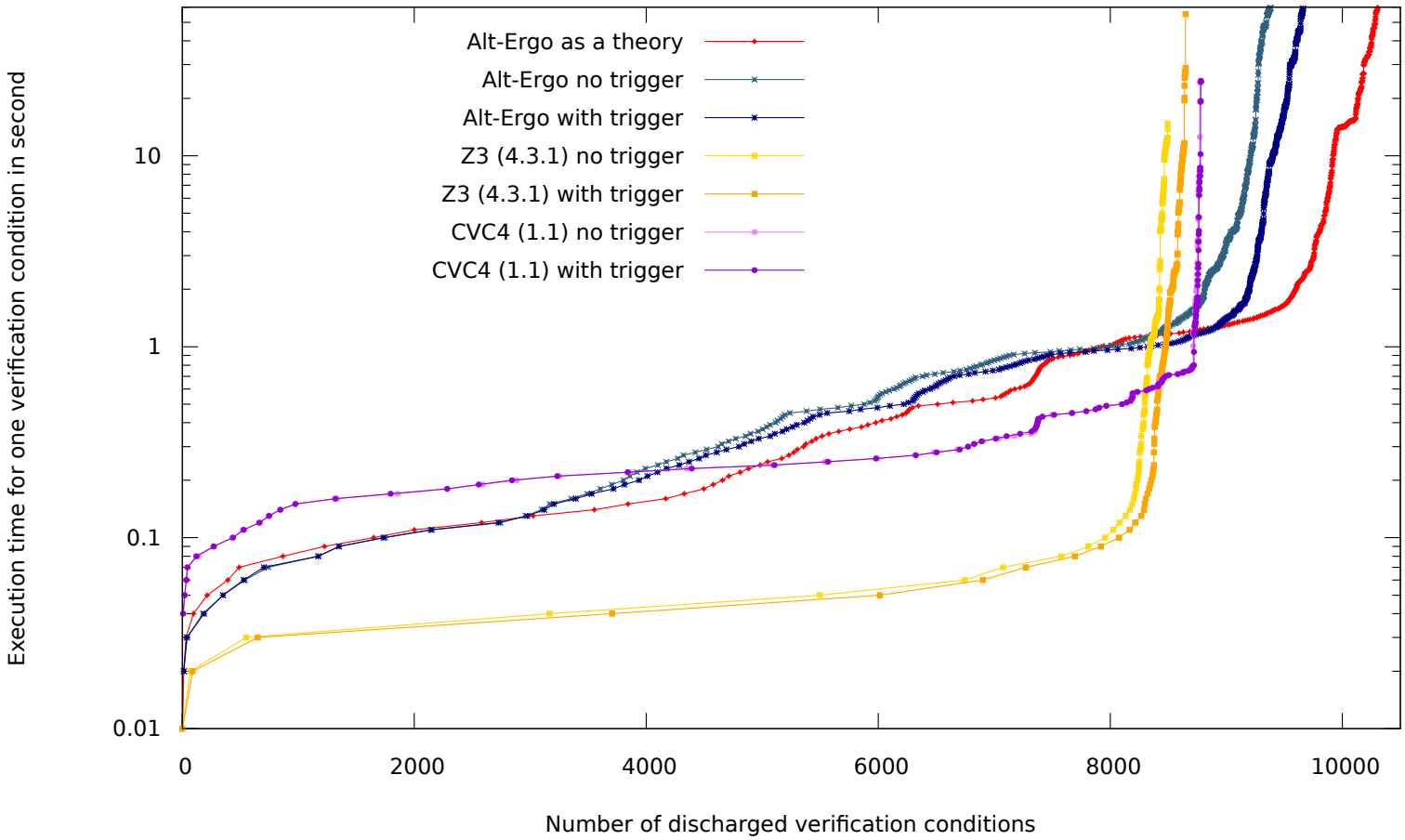


Figure 5.3: Evolution of the number of verification conditions from the BWare benchmarks with additional extensionality lemmas discharged by Alt-Ergo 0.95.2, Z3 4.3.2, and CVC4 1.1 with a timeout of 60s with and without the triggers and by Alt-Ergo through the theory mechanism.

Alt-Ergo \ Alt-Ergo*	Discharged	Undischarged	Total
Discharged	9690	71	**9761**
Undischarged	626	185	811
Total	**10316**	256	10572

Figure 5.4: Number of verification conditions from the BWare benchmarks discharged by our theory mechanism's implementation in Alt-Ergo, named Alt-Ergo*, and Alt-Ergo's built-in quantifiers handling.

the input problem, of which there may be many.

5.3.2 Comparison Between our Implementation and the Built-in Quantifiers Handling of Alt-Ergo

The Figure 5.4 compares the time needed to discharge verification conditions with our theory mechanism's implementation in Alt-Ergo and with Alt-Ergo's built-in quantifiers handling on the BWare benchmarks without additional lemmas and with user provided triggers. We see that eager instantiation ends up being rather efficient on these tests.

Remark that if MEM_POWER and MEM_INTERVAL are treated by the built-in quantifier handling of Alt-Ergo, the performances of the theory mechanism are poorer as the solver will do eagerly a lot of (unnecessary) set reasoning before instantiating these axioms. We see that choosing a good subset of axioms for eager instantiation is crucial for efficiency. It is even more appearent when the theory handling is time consuming, in particular when its axiomatization leads to numerous instances. Indeed, instances of axioms outside the theory will be sparser, delaying reasoning that requires them.

The inclusion/exclusion of lemmas for extensionality in the theory does not impact our implementation much. We do not even get the important slow-down when we do not put triggers for them. Indeed, those are not eagerly instantiated while other useful axioms are and, therefore, their ill effect is less noticeable.

Note that there are also several verification conditions that are discharged by Alt-Ergo's built-in quantifiers handling but not by our implementation. That can be because there are still a great deal of axioms describing additional constructs such as relations, functions, restrictions, finite sets... that we do not include in the theory.

5.4 Conclusion

On this case study, eager instantiation of universal quantifiers gives good results. Indeed, our implementation in Alt-Ergo discharges more verification conditions than Alt-Ergo's built-in handling of quantifiers. We have also seen that, although manually provided triggers are not always profitable to off-the-shelf SMT solvers, well-chosen triggers can prevent the slow-down due to unnecessary quantifier instantiations.

6 Case Study: Formal Bounded Vectors of SPARK 2014

Contents

6.1 Context: the SPARK 2014 Tool 97
6.2 Ada's Formal Vectors Package 99
6.2.1 Description of the Ada Package 99
6.2.2 Translation into WhyML 101
6.3 A Decision Procedure for Formal Vectors 101
6.3.1 Presentation of the Theory 102
6.3.2 Description of the Axiomatization 103
6.3.3 Proofs of Soundness, Completeness, and Termination 107
6.4 Benchmarks 111
6.4.1 Assessment of the Adequacy Between our Framework and Usual E-Matching Techniques 112
6.4.2 Comparison Between our Implementation and the Built-in Quantifiers Handling of Alt-Ergo 112
6.5 Conclusion 112

In Chapter 4, we explained how DPLL(T) can be extended to support first-order logic with triggers. We have also shown that this extension yields a decision procedure when given a sound, complete, and terminating axiomatization as defined in Chapter 4.

In this chapter, we explain how the mechanism described in Chapter 4 can be used during the verification of Ada programs dealing with vectors using the SPARK 2014 tool.

6.1 Context: the SPARK 2014 Tool

The translation from SPARK 2014 into WhyML does not preserve the structure of the SPARK 2014 code but rather the mathematical formulas that can be extracted from them. Entities in SPARK 2014 are translated into modules in WhyML. For a type in Ada, this WhyML module contains a type and elements needed to axiomatize it. For example, discrete types are modeled using conversion functions into Why3 mathematical integers as well as some bounds.

Here is the WhyML translation of the Ada type for 8 bits integers:

```
module Standard__short_short_integer
 use import "int".Int

 type short_short_integer

 function first : int = -128
 function last : int = 127
 predicate in_range (x : int) = first <= x <= last

 function to_int short_short_integer : int
 function of_int int : short_short_integer

 axiom range_axiom:
    forall x : short_short_integer. in_range (to_int x)

 axiom inversion_axiom:
    forall x : short_short_integer [to_int x].
    of_int (to_int (x)) = x
  ...
end
```

For a subprogram, the WhyML module contains a logic function – that can be called in assertions – a program function with contracts – that can be called in programs – and a procedure used to verify absence of runtime errors and conformance to contracts.

Consider the function `G` that calls the function `F`. We assume that `F_Pre` and `F_Post` are the precondition and postcondition of `F` and `R_Inv` is an invariant over elements of type `R` that happens to be preserved by `F`:

```
function G (X : R) return R is
  (F (X))
with Pre  => R_Inv (X) and then F_Pre (X),
     Post => R_Inv (X) and then F_Post (G'Result, X);
```

Here is the translation of `G` into WhyML, a bit simplified for readability:

```
(* Logic and program declarations for G *)
function g (x : r) : r
val g (x : r) : r
  requires { r_inv x = True /\ f_pre x = True }
  ensures  { result = g x /\ r_inv x = True /\ f_post result x = True }

(* Defining axiom since G is an expression function *)
axiom def_axiom :
  forall x : r [g x]. g x = f x

(* Global variables for the argument and the result of G *)
function x : r
val result : ref r
```

```
(* Program function to check G's body and contract *)
let def (void_param : unit)
 ensures { r_inv x = True /\ f_post result x = True }
  =
  (* Execute G's precondition to check there is no runtime error.
     Forget all about it afterward. *)
  abstract (_ignore (r_inv x && f_pre x));
  (* Assume the precondition *)
  any unit ensures  { r_inv x = True /\ f_pre x = True };
  (* Translate G's body, use exception to handle return statement *)
  try
   (result := f x;
    raise Return__exc)
  with
   Return__exc ->
    (* Execute G's postcondition to check there is no runtime error.
       Forget all about it afterward. *)
    abstract (_ignore (r_inv x && f_post !result x));
    (* Return result *)
    !result
  end
```

The WhyML program `def` has no parameter. Indeed, all SPARK parameters and local variables of the subprograms are declared as top-level global variables in WhyML. This is done to accommodate cross references between SPARK declarations. `def` also has no precondition since it needs to produce the mathematical formulas to check that the precondition of the subprogram cannot raise errors at runtime and this cannot be done in the context of the precondition. The precondition is assumed once these mathematical formulas are produced.

6.2 Ada's Formal Vectors Package

SPARK 2014 excludes data structures based on pointers. To work around this restriction, the content of a datastructure can be hidden using private types and ghost functions – functions that are allowed only in annotations – can be used to express the model of the structure. Another possibility is to use the library of formal containers provided with SPARK 2014. We have specifically designed and annotated these generic containers (vectors, lists, maps, and sets) to facilitate the proof of programs using them [36].

6.2.1 Description of the Ada Package

The Ada formal vectors package is a generic package. It can be instantiated with a type for indexes, which must be a discrete type, a type for elements, and an equality function over elements that will be used to search for elements in the vector:

```
generic
```

```
   type Index_Type is range <>;
   type Element_Type is private;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
```

Two private types are defined, one for vectors and one for cursors. Formal vectors are resizable arrays which are bounded to avoid dynamic memory management, that is, their length cannot be greater than a given bound, named capacity. They are implemented as arrays of elements of size the capacity of the vector, along with an index to give the position of the last valid element in the array. To iterate over vectors, the easiest way is to use indexes. Still, for conformance with other containers' packages, cursors are also provided. Even if they are implemented using indexes, the Ada reference manual only requires two cursors to be equal if they designate the same element in the same container. As a consequence, in the formalization, we cannot assume in general that two cursors at the same index in different vectors are equal.

```
type Elements_Array is array (Count_Type range <>) of Element_Type;

type Vector (Capacity : Capacity_Range) is record
  Elements : Elements_Array (1 .. Capacity);
  Last     : Extended_Index := No_Index;
end record;

type Cursor is record
  Valid : Boolean    := True;
  Index : Index_Type := Index_Type'First;
end record;
```

Most of the subprograms defined over vectors are procedures that modify their vector argument. Among them, there are twelve subprograms for insertion, four for deletion, two to swap elements and one to reverse the elements of the vector argument. According to the reference manual, these subprograms preserve cursors except when they are *ambiguous*, that is when elements have been slided over it by insertion or deletion or when they have been reordered.

Among the four procedures for deletion, the most generic one takes a vector `Container`, an index `Index`, and a natural `Count`:

```
procedure Delete (Container : in out Vector;
                  Index     : in     Extended_Index;
                  Count     : in     Count_Type := 1)
with
  Pre => First_Index (Container) <= Index
    and then Index <= Last_Index (Container) + 1;
```

If `Index` is not between `First_Index (Container)` and Last_Index (Container) + 1, `Delete` propagates `Constraint_Error`. If `Count` is 0, `Delete` has no effect. Otherwise, `Delete` slides the elements starting at position `Index + Count` down to `Index`. Any exception raised during element assignment is propagated.

The formal vectors API also contains a few functions returning vectors. There are four concatenation functions and a copy function for which the capacity of the copy can be specified:

```
function Copy (Source : Vector; Capacity : Count_Type := 0) return Vector
with
  Pre => Length (Source) <= Capacity and then
    Capacity in Capacity_Range;
```

`Copy` returns a vector whose elements and cursors are initialized from the corresponding elements and cursors of `Source`. If `Capacity` is 0, then the vector capacity is the length of `Source`; if `Capacity` is equal to or greater than the length of `Source`, the vector capacity is at least the specified value. Otherwise, the operation propagates `Capacity_Error`.

The API also contains other helpful subprograms, in particular equality over vectors, functions for searching elements in a vector and subprograms for iterating over cursors:

```
function Previous (Container : Vector; Position : Cursor) return Cursor
with
  Pre => Has_Element (Container, Position) or else
    Position = No_Element;
```

If the cursor `Position` equals `No_Element` or designates the first element of `Container`, then `Previous` returns the value `No_Element`. Otherwise, it returns a cursor that designates the element with index `To_Index (Position) - 1` in `Container`.

6.2.2 Translation into WhyML

Formal containers are translated through a special mechanism that we have implemented inside the SPARK 2014 tool. It allows to directly provide a manually written Why3 translation for potentially generic SPARK 2014 packages annotated with a dedicated pragma. This Why3 file must mimic the result of the translation on the Ada package so that SPARK 2014 can use its declarations.

This mechanism can be useful for several reasons. First of all, there are some SPARK 2014 subprograms that cannot be specified using only contracts. It is the case for example, for subprograms that require higher order reasoning, like a function that sums the elements of an array of integers, or programs that need to be linked to an existing WhyML library, like numeric elementary functions – square-root, logarithm... This mechanism can also be used to customize the translation of complex packages so that it allows the proof of user code to go through more easily, for example by providing triggers and splitting defining axioms. Finally, this mechanism can be used as a work-around for current SPARK 2014 limitations. It allows in particular to give defining axioms to regular functions and to quantify over non-integer values, for example the cursors of a container. For a mix of all these reasons, this mechanism is used in SPARK 2014 for formal containers, using manual translations that we have provided.

6.3 A Decision Procedure for Formal Vectors

In this section, we define a theory that models some of the Ada formal vectors package's aspects. This theory is then axiomatized so that it can be used to verify SPARK 2014 programs using Formal Vectors.

6.3.1 Presentation of the Theory

Vectors are resizable arrays of elements on which an equivalence is defined using the function $equal\_elements(e_1 : element\_type, e_2 : element\_type) : bool$. They are indexed by a range of integers bounded by two integer constants $index\_type\_\_first$ and $index\_type\_\_last$. A third constant, $index\_type\_\_zero$, is used to refer to $index\_type\_\_first - 1$. The size of a vector co can be retrieved using a function $length(co : vector) : int$. Vectors have a capacity, modeled by a function $capacity(co : vector) : int$, which is the maximal size they can reach. The capacity of a vector is such that the vector cannot exceed the index range. For every vector co, the first index of co is $index\_type\_\_first$ and its last index is $length(co) + index\_type\_\_zero$.

In a vector co, there is a cursor associated to each index i between the first and the last index of co. It can be retrieved using a function $to\_cursor(co : vector, i : int) : cursor$. A cursor constant, named $no\_element$, is associated to every integer between $index\_type\_\_zero$ and $index\_type\_\_last$ that is not between the first and the last index of co. A cursor cu is valid in a vector co if and only if there is an index i between the first and the last index of co such that $cu \approx to\_cursor(co, i)$. This property can be expressed using the function $has\_element(co : vector, cu : cursor) : bool$. A cursor can be valid in several vectors. However, it must always refer to the same index in all vectors it is valid in. The index referred by a cursor cu can be accessed using $to\_index(cu : cursor) : int$. The cursor $no\_element$ cannot be valid in any vector.

Iteration over cursors is supported thanks to dedicated functions. The function $first(co : vector) : cursor$ (resp. $last(co : vector) : cursor$) yields the cursor associated to the first (resp. the last) index of the vector co. In the same way, if cu is a valid cursor in the vector co, $previous(co : vector, cu : cursor) : cursor$ (resp. $next(co : vector, cu : cursor) : cursor$) yields the cursor associated to $to\_index(cu) - 1$ (resp. $to\_index(cu) + 1$) in co. On $no\_element$, $previous$ and $next$ return $no\_element$. In all other cases, $previous$ and $next$ are not specified.

The element stored in a vector co at an index i between the first and the last index of co can be retrieved using $element(co : vector, i : int)$. For every vector co, every element e, and every integer i, $find\_index(co : vector, e : element\_type, i : int) : int$ (resp. $reverse\_find\_index(co : vector, e : element\_type, i : int) : int$) returns the first index j greater than i (resp. the last index j less than i) such that $equal\_elements(element(co, j), e)$ returns true, if any. If there is no such index between i and the last index of co (resp. between the first index of co and i), the function $find\_index$ (resp. $reverse\_find\_index$) returns $index\_type\_\_zero$.

The theory also includes several predicate symbols used to describe how results of procedures that modify vectors in the API are related to their inputs:

- If $replace\_element(co : vector, i : int, e : element\_type, r : vector) : bool$ is true then the index i is between the first and the last index of the vector co and the vector r is co where the element at index i was replaced by the element e. The cursors are preserved.

- If $insert(co_1 : vector, i : int, co_2 : vector, r : vector) : bool$ is true then the index i is between the first and the last index of the vector co_1 plus one and the vector r is co_1 where the vector co_2 has been inserted just before i. The cursors associated to indexes strictly smaller than i in co_1 are preserved.

- If $swap(co : vector, i_1 : int, i_2 : int, r : vector) : bool$ is true then the indexes i_1 and i_2 are

between the first and the last index of the vector *co* and *r* is *co* where the elements at index i_1 and i_2 have been swapped. The cursors are preserved.

- If $delete(co : vector, i : int, l : int, r : vector) : bool$ is true then the index i is between the first and the last index of the vector *co*, the non-negative integer l is smaller than the length of *co*, and the vector *r* is *co* where the elements at the l consecutive indexes starting at i have been removed. The cursors associated to indexes strictly smaller than i in *co* are preserved.

- If $delete\_end(co : vector, i : int, r : vector) : bool$ is true then the index i is between the first and the last index of the vector *co* and the vector *r* is *co* where the last elements, starting at index i, have been removed. The cursors associated to indexes strictly smaller than i in *co* are preserved.

- If $reverse\_elements(co : vector, r : vector) : bool$ is true then the vector *r* contains the elements of the vector *co* in reverse order and *r* and *co* have the same capacity. Cursors are not preserved.

The function $copy(co : vector, new\_capacity : int) : vector$ returns a vector that has the same length and contains the same elements and cursors as the vector *co*. If the natural $new\_capacity$ is $index\_type\_\_zero$ then the vectors $copy(co, new\_capacity)$ and *co* have the same capacity. If $new\_capacity$ is between $length(co)$ and $index\_type\_\_last - index\_type\_\_zero$ then the capacity of the vector returned by *copy* is $new\_capacity$. Otherwise, it is not specified.

Like in Ada, there are four concatenation functions: one to concatenate two vectors, named $concat\_\_1(co_1 : vector, co_2 : vector) : vector$, two to concatenate a vector and an element, named $concat\_\_2(co : vector, e : element\_type) : vector$ and $concat\_\_3(e : element\_type, co : vector) : vector$, and one to concatenate two elements, $concat\_\_4(e_1 : element\_type, e_2 : element\_type) : vector$. The capacity of the result of these functions is not specified. Neither are the values of its cursors.

6.3.2 Description of the Axiomatization

We only describe some axioms in this section. The whole axiomatization is available in Appendix C.1. We first define a logic type for vectors, along with logic functions for *element*, *length*, and *capacity*. We use two axioms to give bounds for the value returned by these functions:

CAPACITY_RANGE:

$$\forall co : vector.[capacity(co)]$$
$$0 \leq capacity(co) \leq index\_type\_\_last - index\_type\_\_zero$$

LENGTH_RANGE:

$$\forall co : vector.[length(co)]\ 0 \leq length(co) \leq capacity(co)$$

The functions used for translation between cursors and indexes are defined as expected. We add a redundant axiom, HAS_ELEMENT__TO_CURSOR, which is not required for termination nor completeness of the axiomatization. It states that $has\_element(co, to\_cursor(co,i))$ is true for i such that $index\_type\_\_zero < i \leq length(co) + index\_type\_\_zero$, which can be deduced from the axioms HAS_ELEMENT__DEF and TO_INDEX_TO_CURSOR as soon as the term $has\_element(co, to\_cursor(co,i))$ is known. It is not required for completeness since the function symbol *has_element* does not appear in other triggers of the theory. Still, as verification conditions arising from Ada programs usually contain quantifiers, we may need to deduce that we have $has\_element(co, to\_cursor(co,i)) \approx \mathtt{t}$ when only $to\_cursor(co,i)$ is known.

TO_INDEX_RANGE:

$$\begin{array}{l}\forall cu : cursor.[to\_index(cu)] \\ \quad index\_type\_\_zero \leq to\_index(cu) \leq index\_type\_\_last\end{array}$$

TO_INDEX_NO_ELEMENT:

$$to\_index(no\_element) \approx 0$$

TO_INDEX_TO_CURSOR:

$$\begin{array}{l}\forall i : int, co : vector.[to\_cursor(co,i)] \\ \quad (length(co) + index\_type\_\_zero \geq i > index\_type\_\_zero \rightarrow \\ \qquad to\_index(to\_cursor(co,i)) \approx i) \wedge \\ \quad (length(co) + index\_type\_\_zero < i \vee i \approx index\_type\_\_zero \rightarrow \\ \qquad to\_cursor(co,i) \approx no\_element)\end{array}$$

HAS_ELEMENT__DEF:

$$\begin{array}{l}\forall cu : cursor, co : vector.[has\_element(co,cu)] \\ \quad has\_element(co,cu) \approx \mathtt{t} \leftrightarrow \\ \qquad (to\_index(cu) > 0 \wedge to\_cursor(co, to\_index(cu)) \approx cu)\end{array}$$

HAS_ELEMENT__TO_CURSOR:

$$\begin{array}{l}\forall i : int, co : vector.[to\_cursor(co,i)] \\ \quad length(co) + index\_type\_\_zero \geq i > index\_type\_\_zero \rightarrow \\ \qquad has\_element(co, to\_cursor(co,i)) \approx \mathtt{t}\end{array}$$

To describe *previous*, *next*, *first*, and *last*, we use *to_cursor*. Note that these functions are total but are only described on their domain of definition so that nothing can be deduced about illegal applications.

PREVIOUS_IN:

$$\begin{array}{l}\forall co : vector, cu : cursor.[previous(co,cu)] \\ \quad (length(co) + index\_type\_\_zero \geq to\_index(cu) > index\_type\_\_first \vee \\ \quad length(co) + index\_type\_\_zero > to\_index(previous(co,cu)) > index\_type\_\_zero) \rightarrow \\ \quad\quad to\_cursor(co, to\_index(cu) - 1) \approx previous(co,cu)\end{array}$$

PREVIOUS_EXT:

$$\begin{array}{l}\forall co : vector, cu : cursor.[previous(co,cu)] \\ \quad (to\_index(cu) = index\_type\_\_first \vee cu \approx no\_element) \rightarrow \\ \quad\quad previous(co,cu) \approx no\_element\end{array}$$

The predicates describing a modification of vectors are only relevant if they are known to be true. Here are axioms describing how the value of a vector after a deletion is related to its previous value. They express the links between vectors using functions *length*, *element*, and *to_cursor*. We take care to only specify the value of partial functions such that *element* and *to_cursor* on domains where they are defined. To allow the proof of completeness to go through, we allow equalities between elements and cursors to be usable from both sides. It sometimes requires some axioms to be split in two, like DELETE_ELEMENT_1 and DELETE_CURSORS. Remark that, like in DELETE_ELEMENT_2, we avoid having interpreted symbols in our triggers, $element(co_2, j-l)$ for example, since this makes both reasoning about the axiomatization more complex and the implementation less efficient.

DELETE_RANGE:

$$\begin{array}{l}\forall co_1, co_2 : vector, i, l : int.[delete(co_1, i, l, co_2)] \\ \quad delete(co_1, i, l, co_2) \approx \mathtt{t} \rightarrow index\_type\_\_zero < i \leq length(co_1) + index\_type\_\_zero\end{array}$$

DELETE_LENGTH:

$$\begin{array}{l}\forall co_1, co_2 : vector, i, l : int.[delete(co_1, i, l, co_2)] \\ \quad delete(co_1, i, l, co_2) \approx \mathtt{t} \rightarrow length(co_1) + index\_type\_\_zero \approx length(co_2) + index\_type\_\_zero + l\end{array}$$

DELETE_CAPACITY:

$$\begin{array}{l}\forall co_1, co_2 : vector, i, l : int.[delete(co_1, i, l, co_2)] \\ \quad delete(co_1, i, l, co_2) \approx \mathtt{t} \rightarrow capacity(co_1) \approx capacity(co_2)\end{array}$$

DELETE_ELEMENT_1:

$$\begin{array}{l}\forall co_1, co_2 : vector, i, l, j : int.[delete(co_1, i, l, co_2), element(co_1, j)] \\ \quad delete(co_1, i, l, co_2) \approx \mathtt{t} \rightarrow index\_type\_\_zero < j < i \rightarrow \\ \quad\quad element(co_1, j) \approx element(co_2, j) \\ \forall co_1, co_2 : vector, i, l, j : int.[delete(co_1, i, l, co_2), element(co_2, j)] \\ \quad delete(co_1, i, l, co_2) \approx \mathtt{t} \rightarrow index\_type\_\_zero < j < i \rightarrow \\ \quad\quad element(co_1, j) \approx element(co_2, j)\end{array}$$

DELETE_ELEMENT_2 :

$$\begin{array}{l}\forall co_1, co_2 : vector, i, l, j : int.[delete(co_1, i, l, co_2), element(co_1, j)] \\ \quad delete(co_1, i, l, co_2) \approx \mathtt{t} \rightarrow i + l \leq j \leq length(co_1) + index\_type\_\_zero \rightarrow \\ \quad\quad \langle i + l \rangle\, element(co_1, j) \approx element(co_2, j - l) \\ \forall co_1, co_2 : vector, i, l, j : int.[delete(co_1, i, l, co_2), element(co_2, j)] \\ \quad delete(co_1, i, l, co_2) \approx \mathtt{t} \rightarrow i \leq j \leq length(co_2) + index\_type\_\_zero \rightarrow \\ \quad\quad element(co_1, j + l) \approx element(co_2, j)\end{array}$$

DELETE_CURSORS:

$$\begin{array}{l}\forall co_1, co_2 : vector, i, l, j : int.[delete(co_1, i, l, co_2), to\_cursor(co_1, j)] \\ \quad delete(co_1, i, l, co_2) \approx \mathtt{t} \rightarrow index\_type\_\_zero < j < i \rightarrow \\ \quad\quad to\_cursor(co_1, j) \approx to\_cursor(co_2, j) \\ \forall co_1, co_2 : vector, i, l, j : int.[delete(co_1, i, l, co_2), to\_cursor(co_2, j)] \\ \quad delete(co_1, i, l, co_2) \approx \mathtt{t} \rightarrow index\_type\_\_zero < j < i \rightarrow \\ \quad\quad to\_cursor(co_1, j) \approx to\_cursor(co_2, j)\end{array}$$

Remark 6.1. In DELETE_ELEMENT_2, we use a witness to make sure that, when the literal $element(co_1, j) \approx element(co_2, j - l)$ is assumed by the solver, the terms of the guard $i + l \leq j \leq length(co_1) + index\_type\_\_zero$ are known in the current partial assignment. This is required so that, whenever $element(co_1, j) \approx element(co_2, j - l)$ is assumed, we are sure that the guard $i + l \leq j \leq length(co_1) + index\_type\_\_zero$ is true. Indeed, in the variant of termination described in the last paragraph of Section 4.2.2, for $element(co_1, j) \approx element(co_2, j - l)$ to appear in a truth assignment A, $\neg(i + l \leq j \leq length(co_1) + index\_type\_\_zero)$ must be false in A. Using the semantics of worlds, we can only deduce $i + l \leq j \leq length(co_1) + index\_type\_\_zero$ if the terms of $i + l \leq j \leq length(co_1) + index\_type\_\_zero$ are known in A.

For *copy*, and concatenation functions, we introduce a predicate symbol to describe the result of the call. For concatenation, $concat(co_1, co_2, r)$ states that r is the result of the concatenation of co_1 and co_2. The predicate symbol *equal_vectors* describes the result of *copy*:

EQUAL_VECTORS__DEF:

$$\begin{array}{l}\forall co_1, co_2 : vector.[equal\_vectors(co_1, co_2)] \\ \quad equal\_vectors(co_1, co_2) \approx \mathtt{t} \rightarrow (length(co_1) \approx length(co_2) \wedge \\ \quad\quad (\forall i : int.[element(co_1, i)] \\ \quad\quad\quad index\_type\_\_zero < i \leq length(co_1) + index\_type\_\_zero \rightarrow \\ \quad\quad\quad\quad element(co_1, i) \approx element(co_2, i)) \wedge \\ \quad\quad (\forall i : int.[element(co_2, i)] \\ \quad\quad\quad index\_type\_\_zero < i \leq length(co_1) + index\_type\_\_zero \rightarrow \\ \quad\quad\quad\quad element(co_1, i) \approx element(co_2, i)) \wedge \\ \quad\quad (\forall i : int.[to\_cursor(co_1, i)]\, to\_cursor(co_1, i) \approx to\_cursor(co_2, i)) \wedge \\ \quad\quad (\forall i : int.[to\_cursor(co_2, i)]\, to\_cursor(co_1, i) \approx to\_cursor(co_2, i)))\end{array}$$

COPY__DEF:

$$\begin{aligned}&\forall co : vector, cap : int.[copy(co, cap)]\, equal\_vectors(co, copy(co, cap)) \approx \mathtt{t} \wedge \\ &\quad (cap \approx 0 \rightarrow capacity(co) \approx capacity(copy(co, cap))) \wedge \\ &\quad (length(co) \leq cap \leq index\_type\_\_last - index\_type\_\_zero \rightarrow \\ &\qquad capacity(copy(co, cap)) \approx cap)\end{aligned}$$

6.3.3 Proofs of Soundness, Completeness, and Termination

First, we can note that the axiomatization is not complete in general. Indeed, consider the following set of ground literals:

$$G = \left\{ \begin{array}{l} length(u) \approx 1, element(u, index\_type\_\_first) \approx e, \\ insert(co_1, 1, u, co_2) \approx \mathtt{t}, delete(co_2, length(co_2), co_1) \approx \mathtt{t}, \\ index\_type\_\_zero < i \leq length(co_1), element(co_1, i) \not\approx e \end{array} \right\}$$

The set G is unsatisfiable in the axiomatization of vectors without triggers and witnesses. Indeed, every element at index $index\_type\_\_zero < i \leq length(co_1)$ in co_1 is equal to the element at index i in co_2 since $delete(co_2, length(co_2), co_1) \approx \mathtt{t}$. But it is also equal to the element at index $i+1$ in co_2 since $insert(co_1, 1, u, co_2) \approx \mathtt{t}$. As a consequence every element of co_1 and co_2 are equal to e. Such a reasoning needs induction and, therefore, is not in the reach of an SMT solver.

Fortunately, this kind of example cannot arise from the verification of an Ada program. Indeed, verification conditions coming from such a program cannot contain equalities between vectors and modification predicates such as *insert* and *delete* only appear as the postcondition of an Ada function on its result, which is a fresh vector variable.

Note that we would have liked to include an equivalence relation *equivalent_vectors* on vectors in our theory to model Ada equality on vectors. Since this would have lead to incompleteness and non termination, we are forced to leave axioms dealing with it to other quantifier handling mechanisms:

EQUIVALENT_VECTORS__DEF:

$$\begin{aligned}&\forall co_1, co_2 : vector.[equivalent\_vectors(co_1, co_2)]\, equivalent\_vectors(co_1, co_2) \approx \mathtt{t} \leftrightarrow \\ &\quad length(co_1) \approx length(co_2) \wedge \\ &\quad (\forall i : int.[element(co_1, i)] \\ &\qquad index\_type\_\_zero < i \leq length(co_1) + index\_type\_\_zero \rightarrow \\ &\qquad equal\_elements(element(co_1, i), element(co_2, i)) \approx \mathtt{t}) \wedge \\ &\quad (\forall i : int.[element(co_2, i)] \\ &\qquad index\_type\_\_zero < i \leq length(co_1) + index\_type\_\_zero \rightarrow \\ &\qquad equal\_elements(element(co_1, i), element(co_2, i)) \approx \mathtt{t})\end{aligned}$$

Theorem 6.1. *The axiomatization in Section 6.3.2 is terminating, sound and complete with respect to the same axiomatization without triggers and witnesses on verification conditions coming from Ada programs.*

6.3.3.1 Proof of Termination

We show that, on verification conditions coming from Ada programs using vectors, our axiomatization is terminating.

Since we have quantification over integers, we cannot, like for lists, show that there can only be a finite number of new terms of each type. Instead, we use the fact that triggers only contain uninterpreted function symbols. We only need to show that there can only be a finite number of terms starting with every uninterpreted function symbol. Indeed, the underlying theory T cannot create a new term starting with such a function symbol and, therefore, if there can only be a finite number of terms starting with every uninterpreted function symbol, there can only be a finite number of new instances.

We use modularity as much as possible. We first consider the axioms that use *to_index* or *capacity* as a trigger. There is only one for each, and they do not create any new term except constants. The function symbol *length* is used as a trigger for only one axiom, that is LENGTH_RANGE. This axiom can create new terms, but only new terms starting with *capacity*. We already know that such terms cannot lead to an infinite number of new terms.

Most of the remaining function symbols (*find*, *concat*, *delete*, *insert*...) cannot be created by the axiomatization. Therefore, there can only be a finite number of terms starting with them. In the same way, there can be one term starting with *equal_vectors* created for each *copy* term and one *concat* term for each *concat*, *concat__2*, or *concat__3* term.

The function symbol *has_element* can be used to create terms starting with *to_index*, which cannot lead to an infinite number of new terms, as well as a term starting with *to_cursor* through HAS_ELEMENT__DEF. The only way to create a term starting with *has_element* is with the axiom HAS_ELEMENT__TO_CURSOR. It has the term *to_cursor(co,i)* as a trigger. Proving that these two axioms cannot get into a loop requires a finer grain reasoning. For each term *has_element(co,cu)*, HAS_ELEMENT__DEF can create a term *to_cursor(co,to_index(cu))* by assuming *to_cursor(co,to_index(cu))*$\approx$*cu*. The axiom HAS_ELEMENT__TO_CURSOR can then create a term *has_element(co,to_cursor(co,to_index(cu))*. This will not lead to any new instance from HAS_ELEMENT__DEF since *has_element(co,cu)* is known and *to_cursor (co, to_index (cu))* $\approx$ *cu*.

Let us now consider the two remaining function symbols, *element* and *to_cursor*. They appear in triggers all over the axiomatization but a quick survey of the axioms shows that they either appear as triggers in the first level of quantifiers with a function symbol that can only be created a finite number of times in the axiomatization or as unique trigger directly below a quantifier with a trigger containing only a function symbol that can only be created a finite number of times in the axiomatization. In both cases, we have that, from a given set of ground literals, there can only be a finite number of distinct quantifier instantiations with terms starting with either *element* or *to_cursor* as a trigger.

Quantifiers that have the term *to_cursor(co,i)* as a trigger can only create terms of the form *to_cursor(co',i)* where co' is a known term of type vector. Since there can only be a finite number of those, the axiomatization cannot create an infinite number of terms starting with *to_cursor*.

Axioms for *find* cannot create an infinite number of terms starting with *element*. For the other axioms, we use the fact that proof obligations are coming from Ada programs. As a consequence, equivalence classes modulo L of known vector terms can be linked in a directed

acyclic graph where there is an edge from r to co if r is the result of a vector modification involving co.

We show that, from a set of ground literals L, the axiomatization can only create a finite number a new terms starting with $element$. Let $element(co,i)$ be a term of L. For every known vector term co', we define iteratively a set $eq^L_{(i,co)}(co')$ of pairs of an integer $index$ and a set of literals $guards$ such that $L \cup guards \vDash_T element(co,i) \approx element(co',index)$. More precisely, we start with $eq^L_{(i,co)}(co) = \{(i,\varnothing)\}$ and $eq^L_{(i,co)}(co') = \varnothing$ for every term co' different from co. Then, for every node co', we define $eq^L_{(i,co)}(co')$ as the union of its previous value and the values coming from its adjacent nodes. For example, if a node is labeled by co_1 and there is an edge from co_2 to co_1 labeled by $delete(co_1,j,l,co_2)$ then $eq^L_{(i,co)}(co_2)$ is augmented by:

$$\{(index, guards \cup \{index\_type\_\_zero < index < j\} \mid$$
$$(index, guards) \in eq^L_{(i,co)}(co_1) \text{ and } L \cup guards \cup \{index\_type\_\_zero < index < j\} \nvDash_T \bot\} \cup$$
$$\{(index - l, guards \cup \{j + l \leq index \leq length(co_1) + index\_type\_\_zero\} \mid$$
$$(index, guards) \in eq(co_1) \text{ and}$$
$$L \cup guards \cup \{j + l \leq index \leq length(co_1) + index\_type\_\_zero\} \nvDash_T \bot\}$$

This construction will terminate. We can do this proof by induction over the number of nodes in the graph. If there is only one node then there is no propagation. Otherwise, let us choose a node that has no entering edge. If it is the initial node co then it will down propagate its initial information. From there, this node will interfere no more with the propagation in its subgraphs as it can neither propagate back on the same edge what it has just received nor propagate what it has received from one one edge onto the other as guards on leaving edges are exclusive.

Remark 6.2. If L' is a superset of L such that $L' \setminus L$ only contains inequalities between integers, then, for every vector terms co_1 and co_2 known in L and every pair $(index, guards) \in eq^L_{(i,co)}(co_1)$ such that $L \vDash_T guards$, we have $eq^{L'}_{(index,co_1)}(co_2) \subseteq eq^L_{(i,co)}(co_2)$.

The sets of terms starting with $element$ that can be created from a term $element(co,i) \in \mathscr{T}(L)$ is bounded by the set $\{element(co',index) \mid co' \in \mathscr{T}(L) \text{ and } (index, guards) \in eq^L_{(i,co)}(co')\}$. The proof relies on the fact that, for every modification term in L, we can only produce a new term $element(co',i')$ if we already know that the guard is true. This comes from the fact that our implementation uses a rather permissive notion of termination by requiring decisions to be done on the first element of a clause, as explained in Section 4.2.2. Thanks to this property, we have that we can only produce a term $element(co',i')$ using our axiomatization if there is a set of literals $guards$ such that $(i', guards) \in eq^L_{(i,co)}(co')$ and $L \vDash_T guards$. The complete bound then comes from Remark 6.2. □

6.3.3.2 Proof of Soundness

We show that, if a set of literals G has a model in the axiomatization without triggers and witnesses then there is a total model of G and the axiomatization. If I is a model of a set of literals G in the axiomatization without triggers and witnesses, we define $L = \{l \mid I(l) = \top\}$. By construction of L, L is a total model of G. Since L is total, for every axiom φ of the axiomatization, $L \rhd_T \varphi$. □

6.3.3.3 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 6.1. *If the axiomatization is true in a world L, $t_1, t_2 \in \mathscr{T}(L)$ have type integer and $L \not\vDash_T t_1 \not\approx t_2$ then the axiomatization is also true in $L \cup t_1 \approx t_2$.*

Proof. The proof is the same proof we did for Lemma 2.4 □

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 axiomatization without triggers and witnesses. 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 $L \cup \{i \approx i \mid i \text{ is an integer constant}\}$. For every term $t \in \mathscr{T}(L)$ of the form $length(co)$, $to\_index(cu)$, or $capacity(co)$, we add $t \approx I_T(t)$ to L. By Lemma 6.1, the axiomatization is still true in L.

For every term co of type vector in L, if $length(co)$ (resp. $capacity(co)$) is not in $\mathscr{T}(L)$ modulo T, we add $length(co) \approx 0$ (reps. $capacity(co) \approx 0$ to L). This amounts to deciding that vectors that are not forced to be non-empty are empty. The axiomatization is still true in L. Indeed, the axiom LENGTH_RANGE creates a term $capacity(co)$ for each term $length(co)$ in L and TO_INDEX_TO_CURSOR creates a term $length(co)$ for each term $to\_cursor(co,i)$ that is not $no\_element$. In the same way, for each term cu of type cursor in L, if $to\_index(cu)$ is not in $\mathscr{T}(L)$ we can add $to\_index(cu) \approx 0$ to L.

We now need to associate a cursor to every position of every non-empty vector. For this, like for doubly-linked lists, we consider *zones* of vectors. For vectors it is simpler since cursors are not preserved when elements are shifted. We define a zone of a term co of type vector in L to be a pair of a vector and an index (co,i), with i in $index\_type\_\_zero..length(co) + index\_type\_\_zero$.

For every zone z of a vector, 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. For every $(co,i) \in eq(z)$:

$$
\begin{array}{lll}
L \vDash_T insert(co_1, j, co_2, co) \approx \mathtt{t} \text{ and } L \vDash_T index\_type\_\_zero < i < j & \rightarrow & (co_1, i) \in eq(z) \\
L \vDash_T insert(co, j, co_2, r) \approx \mathtt{t} \text{ and } L \vDash_T index\_type\_\_zero < i < j & \rightarrow & (r, i) \in eq(z) \\
L \vDash_T delete(co_1, j, l, co) \approx \mathtt{t} \text{ and } L \vDash_T index\_type\_\_zero < i < j & \rightarrow & (co_1, i) \in eq(z) \\
L \vDash_T delete(co, j, l, r) \approx \mathtt{t} \text{ and } L \vDash_T index\_type\_\_zero < i < j & \rightarrow & (r, i) \in eq(z) \\
L \vDash_T delete\_end(co_1, j, co) \approx \mathtt{t} \text{ and } L \vDash_T index\_type\_\_zero < i < j & \rightarrow & (co_1, i) \in eq(z) \\
L \vDash_T delete\_end(co, j, r) \approx \mathtt{t} \text{ and } L \vDash_T index\_type\_\_zero < i < j & \rightarrow & (r, i) \in eq(z) \\
L \vDash_T replace\_element(co_1, j, e, co) \approx \mathtt{t} & \rightarrow & (co_1, i) \in eq(z) \\
L \vDash_T replace\_element(co, j, e, r) \approx \mathtt{t} & \rightarrow & (r, i) \in eq(z) \\
L \vDash_T swap(co_1, i_1, i_2, co) \approx \mathtt{t} & \rightarrow & (co_1, i) \in eq(z) \\
L \vDash_T swap(co, i_1, i_2, r) \approx \mathtt{t} & \rightarrow & (r, i) \in eq(z) \\
L \vDash_T equal\_vectors(co_1, co) \approx \mathtt{t} & \rightarrow & (co_1, i) \in eq(z) \\
L \vDash_T equal\_vectors(co, r) \approx \mathtt{t} & \rightarrow & (r, i) \in eq(z)
\end{array}
$$

It is straightforward to check that, if $to\_cursor(co,i)$ is not known in L then, for every pair $(co',i) \in eq(co,i)$, the term $to\_cursor(co',i')$ is not known in L. If, for a vector co and an index $index\_type\_\_zero < i \leq length(co) + index\_type\_\_zero$, the term $to\_cursor(co,i)$ is not known in L, for each zone $(co',i) \in eq(co,i)$, we add the literals $to\_cursor(co,i) \approx to\_cursor(co',i)$, $has\_element(co',to\_cursor(co',i)) \approx \mathtt{t}$, and $to\_index(to\_cursor(co,i)) \approx i$ to L. This cannot introduce an inconsistency since $to\_cursor(co',i)$, and thus $has\_element(co',to\_cursor(co',i))$, are not known in L. What is more, a quick survey of the axiomatization shows that the axiomatization is still true in L. After this step, there is a cursor per index in every vector in L, and the value of *next*, *previous*, *first*, and *last* can be set without introducing new terms nor contradictions.

Let us now consider the elements contained by the vector. Let e be a fresh term of type element. We map $element(co,cu)$ to e for any term co of type vector and any term cu of type cursor in L such that $element(co,cu)$ is not in L modulo T. Each axiom with $element(co,cu)$ 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. □

6.4 Benchmarks

We have written solutions in SPARK 2014 for problems given in the verification competition held at the VSTTE conference (Verified Software: Theories, Tools, and Experiments)[1]. We try our mechanism on some of these solutions involving vectors:

- a two-way sorting algorithm for vectors of Booleans from VSTTE 2012. It is implemented using swap. Our implementation only solves partially the problem as its postcondition only states that the result of the function is sorted, not that it is a permutation of the input vector.
- an implementation that solves the n queens problem from VSTTE 2010 using a vector. The vector stores the positions of the queens on the board. We show both that, if a solution is found then it is indeed a solution and that, if no solution is found then there is no solution. This last property is expressed using an additional, unspecified vector argument to the procedure to model universal quantification over vectors.
- a ring buffer implementation in a vector of fixed size, from VSTTE 2012. This implementation is specified thanks to a model of the content of the ring buffer represented as a vector.
- a functional amortized queue implementation made of two vectors, from VSTTE 2010. The elements are dequeued at the end of the first vector and enqueued at the end of the second. If the second part becomes bigger than the first, then its elements are reversed and inserted at the beginning of the first vector. This implementation is also specified thanks to a model of the content of the queue represented as a vector.

[1] `http://vscomp.org/`

The SPARK 2014 code for these examples is given in Appendix C.2.

6.4.1 Assessment of the Adequacy Between our Framework and Usual E-Matching Techniques

In Figure 6.1, we compare the results of the tests with and without the manually written triggers using both Alt-Ergo 0.95, CVC4 1.2, and Z3 3.2. The results are heterogeneous. First, we can see that, without the triggers, we are more incomplete. Alt-Ergo returns *Unknown* on several tests. Some tests take a lot more time to complete without the triggers, like the postcondition of the function returning the model of an amortized queue. We can also see that, in some cases, for example the postconditions of the Push function on ring buffers, Alt-Ergo does better without the triggers. It can be that it has less things to do because it is incomplete.

6.4.2 Comparison Between our Implementation and the Built-in Quantifiers Handling of Alt-Ergo

As in Section 4.2, we have two versions of our axiomatization of bounded vectors, one with separated and one with grouped axioms.

The theory mechanism is useful in particular when the vector API is used a lot. It is the case for the ring buffer and the amortized queue. We can see this speedup in the postcondition of the function returning the model of the structure in particular. For the postcondition of the API function, this is less obvious since the results are polluted by the fact that the definition of the function returning the model is always handled outside the theory.

There are always a great deal of axioms with quantifiers that do not belong to the theory when verifying a SPARK program. Still, if these axioms do not break termination, which is the case for the definition of the two model functions, they can be included in the theory. For the last two axiomatizations, we redo the tests by adding a new axiom, specific to the use case, to describe how to get a model for the content of the structure.

As we see in Figure 6.3, this new axiom is helpful both for Alt-Ergo and Alt-Ergo*. Indeed, the manually added triggers are also useful for the general quantifier handling of Alt-Ergo. Still, it appears that eager instantiation gives better results. Indeed, the proofs of the postconditions of the API function now mostly use axioms from the theory.

6.5 Conclusion

Although eager instantiation results in a significant speedup for verification conditions requiring a lot of theory reasoning, results are overall less impressive than for the BWare verification conditions. Indeed, verification conditions coming from the verification of our SPARK 2014 programs involve a lot of first-order axioms that are not part of the theory, coming from the translation of integer ranges like in Section 6.1 for example, and require relatively little vector specific reasoning.

With Triggers	Alt-Ergo		Z3 3.2		CVC4 1.2	
	Yes	No	Yes	No	Yes	No
two way sort:						
loop-invariant	4.50	**3.63**	0.04	0.05	MO	MO
postcondition	**54.06**	72.33	UK	UK	MO	MO
n queens:						
loop-invariant in Copy_Until	**0.29**	UK	0.03	0.03	TO	TO
loop-invariant in Add_Next	**0.32**	UK	0.21	0.20	TO	TO
Try_Row loop-invariant	2.13	**1.79**	0.03	0.03	TO	TO
1st call to Add_Next in Try_Row	4.22	**3.72**	0.03	0.03	TO	TO
2nd call to Add_Next in Try_Row	107.33	108.67	0.03	0.03	TO	TO
call to Copy_Until in Try_Row	0.02	0.02	0.03	0.03	**0.41**	TO
post of Add_Next	**6.37**	7.52	0.04	0.03	0.92	0.92
post of Copy_Until	**0.37**	UK	1.13	1.12	TO	TO
post of Try_Row	18.24	18.30	0.03	0.03	TO	TO
ring buffer:						
call to Append in Model	545.06	**72.08**	0.03	0.04	0.23	0.56
post of Model	TO	MO	0.04	0.04	MO	TO
post of Push	762.44	**364.29**	0.03	0.05	TO	TO
call to Is_Model in post of Push	0.04	0.11	0.03	0.03	**0.69**	TO
call to & in post of Push	**0.06**	202.72	0.03	0.03	0.50	0.75
post of Pop	6.18	**4.66**	**0.23**	0.48	TO	TO
amortized queue:						
call to & in Model	0.11	0.35	0.03	0.03	**0.37**	TO
call to Insert in Tail	**1.71**	3.71	0.04	0.04	**0.60**	TO
discr check in Tail	**2.55**	3.42	0.04	0.04	0.64	0.64
call to Append in Enqueue	0.20	0.11	0.03	0.03	**0.49**	TO
call to Insert in Enqueue	**2.20**	4.29	0.03	0.03	**0.55**	TO
discr check in Enqueue	**1.05**	4.72	0.03	0.03	0.57	0.56
post of Model	**5.97**	1664.58	**0.04**	0.10	TO	TO
post of Is_Model	0.06	0.06	0.03	0.03	TO	TO
post of Tail	**865.03**	TO	**3.56**	381.35	error	error
call to & in post of Tail	**1.06**	4.77	0.04	0.04	**0.24**	336.56
post of Enqueue	**2083.67**	TO	**2.04**	40.32	error	error
call to & in post of Enqueue	0.18	0.88	0.03	0.03	MO	MO

Figure 6.1: Time (in seconds) needed to solve all the tests with Alt-Ergo, Z3 3.2, and CVC4 1.2 with a timeout of 3600s with and without the triggers. TO stands for timeout, UK for unknown and MO for out of memory.

	Separated Axioms		Grouped Axioms	
	Alt-Ergo*	Alt-Ergo	Alt-Ergo*	Alt-Ergo
two way sort:				
loop-invariant	**1.44**	4.50	**1.44**	6.58
postcondition	125.77	54.06	**25.34**	85.26
n queens:				
loop-invariant in Add_Next	1.95	**0.90**	1.27	2.66
loop-invariant in Try_Row	**1.06**	2.13	1.24	1.89
1st call to Add_Next in Try_Row	10.30	**4.22**	10.30	4.89
2nd call to Add_Next in Try_Row	**53.01**	107.33	**53.11**	128.15
post of Add_Next	7.40	**6.37**	7.36	7.42
post of Copy_Until	0.94	**0.37**	0.89	1.47
post of Try_Row	**7.02**	18.24	**7.02**	18.32
ring buffer:				
call to Append in Model	**2.26**	545.06	**2.27**	7.09
post of Model	1090.18	TO	**713.06**	820.12
post of Push	1047.97	**762.44**	TO	**716.32**
post of Pop	342.23	**6.18**	23.56	16.40
amortized queue:				
Range check in Is_Model	**0.23**	5.28	**0.21**	5.27
call to Insert in Tail	**1.04**	1.71	**1.02**	1.84
discr check in Tail	**1.50**	2.55	2.14	2.37
call to Insert in Enqueue	**0.60**	2.20	**0.59**	2.32
discr check in Enqueue	1.61	**1.05**	1.62	**1.10**
post of Model	**0.49**	5.97	**0.44**	26.94
post of Tail	**201.28**	865.03	**200.23**	3033.20
call to & in post of Tail	1.36	**1.06**	1.43	**1.01**
post of Enqueue	2801.83	2083.67	**97.68**	TO
call to & in post of Enqueue	1.57	**0.18**	1.57	**0.16**

Figure 6.2: Time (in seconds) needed to solve all tests with Alt-Ergo with a timeout of 3600s giving the first-order axiomatization directly and Alt-Ergo through the theory mechanism (named Alt-Ergo*). Here we only list the verification conditions that are not immediately discharged by both provers.

	Separated Axioms		Grouped Axioms	
	Alt-Ergo*	Alt-Ergo	Alt-Ergo*	Alt-Ergo
ring buffer:				
call to Append in Model	**2.32**	645.49	**2.32**	7.11
post of Model	848.04	TO	**540.92**	836.25
post of Push	**6.55**	1299.29	8.66	1437.05
post of Pop	5.18	**3.14**	6.50	8.64
amortized queue:				
call to Insert in Tail	**1.03**	1.68	**1.02**	1.84
discr check in Tail	**1.50**	2.20	**1.46**	2.35
call to Insert in Enqueue	**0.58**	2.21	**0.57**	2.30
discr check in Enqueue	**1.58**	2.21	**1.58**	1.12
post of Model	0.27	8.47	**0.24**	42.86
post of Tail	160.03	226.44	**132.02**	674.79
call to & in post of Tail	0.34	**0.97**	0.33	**0.89**
post of Enqueue	**95.64**	217.53	110.67	1321.88
call to & in post of Enqueue	0.49	**0.16**	0.49	**0.15**

Figure 6.3: Time (in seconds) needed to solve the two last tests with Alt-Ergo with a timeout of 3600s giving the first-order axiomatization directly and Alt-Ergo through the theory mechanism (named Alt-Ergo*) using an additional, test specific axiom. Here we only list the verification conditions that are not immediately discharged by both provers.

7 Conclusion

7.1 Summary of the Contributions

We have introduced a semantics for first-order logic with triggers and witnesses. Thanks to this semantics, we have defined three properties of soundness, completeness, and termination for first-order axiomatizations with triggers. We have then presented two implementations of SMT solvers that accept a first-order axiomatization of a theory T' and yield a decision procedure for T' if the axiomatization is sound, complete, and terminating in our framework. The first one uses a ground SMT solver as a black box (implemented in practice using Z3). Though the formalization is rather simple, it appears to be unpractical as it requires too costly communications with the underlying solver. The second implementation is formalized using an extension of the abstract DPLL(T) framework and is integrated into the SMT solver Alt-Ergo's main algorithm. Since we have demonstrated that, in this formalization, quantifier handling terminates for a terminating axiomatization, we have designed our implementation to instantiate eagerly quantifiers coming from the axiomatization of the theory, that is, create new instances as soon as they become possible, without waiting for all the literals to be assigned in the current truth assignment of DPLL(T).

We have tested the white-box implementation on several case studies: hand-written Why3 programs using a theory for imperative doubly-linked lists with iterators, large benchmarks using a theory of sets coming from the verification of *B* programs, and SPARK programs inspired from the VSTTE competition using a library for bounded vectors. We have compared our implementation with the built-in quantifier handling of Alt-Ergo, which treats quantifiers lazily. As a result, it seems that eager instantiation can be really efficient when a lot of theory reasoning is needed. On the other hand, when the axiomatization of the theory is big, this eager instantiation becomes costly. It is all the more noticeable when the tests require a lot of instances of axioms outside the theory and relatively few theory reasoning, which is the case for our programs using SPARK vectors.

Since in off-the-shelf SMT solvers, triggers are a mere heuristic, they don't have a proper semantics and users are sometimes at a loss to find the proper triggers for their axiomatizations. We have compared the behavior of off-the-shelf SMT solvers that handle quantifiers on the axiomatizations used in our case studies with and without manually supplied triggers. The results show that our semantics can indeed help in the choice of these triggers. In particular, it can help understanding why an axiom may slow down a prover by inducing too many instances and what are the triggers that should be used to prevent this explosion. Still, it is not in general worthwhile

to achieve completeness or termination of axiomatizations in this context as most SMT solvers use other instantiation heuristics than triggers.

7.2 Related Work

Instantiation with Triggers: Triggers 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 [56]. We believe that our formalization along with the methods and advice in Section 2.3 give valuable guidance for this choice. 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 [34, 58], Z3 [29], and CVC3 [38].

Triggers can also be used in semi-complete first-order ATPs to guide the proof search and improve the prover's efficiency. For example, the Princess [66] prover that combines a complete calculus for first-order logic with a decision procedure for linear arithmetic can use triggers.

Other generic techniques for quantifier handling in SMT solvers: Model-based quantifier instantiation [39] is an other heuristics for generating instances. This instantiation mechanism generates new instances when the solver is about to return *Sat*, that is, when it has computed a *candidate model*. A candidate model is not always a model when the solver is not in its domain of completeness. It is the case in particular when quantifiers are involved. Model-based instantiation does not in general increase the solver's performance. On the other hand, it increases its accuracy since it allows to continue the search when otherwise the solver would have stopped with only a partial model. It goes through the candidate model M and searches for an instance that is not already implied by M. If one such instance is found, the search goes on.

A saturation process, close to the superposition calculus, has also been integrated into abstract DPLL(T) by de Moura and Bjørner [30]. The idea is to add elements to the set of clauses that are handled by DPLL by using the superposition calculus. If superposition steps use the current partial assignment of DPLL, the inferred clauses must be prefixed by a guard so that they can be reversed when the current partial assignment is backtracked.

Specialized complete instantiation techniques: In SMT solvers, the idea that a set of first-order formulas can be saturated with a finite set of ground instances has been explored previously.

For example, decision procedures for universally quantified properties of functional programs can be designed using local model reasoning [44]. This property states that, for every input problem, it is enough to instantiate universal quantifiers with terms from the problem. Like us, they work modulo an existing background theory and, like us, proofs of locality of axiomatizations are not in general automatic. They also describe some restricted forms of universally quantified formulas for which locality is automatic.

In the same way, Ge and de Moura [39] describe fragments of first-order logic for which one can automatically compute a finite set of ground instances that are enough to ensure complete-

ness. Like for [44], the automation of the termination checks are at the expense of the range of application of these fragments which is much smaller than what we propose in our framework.

McPeak et al. also designed a procedure to decide a particular form of first-order formulas [55]. It is dedicated to descriptions of shape properties of data structures, involving pointers and predicates on scalar fields. This procedure is always terminating but is only complete for a more restricted form of formulas. Outside this restriction, they do not provide ways to check that a particular axiomatization is complete.

Our work is in this respect closer to the work done on the axioms for the theory of arrays. For this theory, a specialized instantiation technique achieves both termination and completeness [17, 32]. The idea is to treat operations over arrays as uninterpreted functions but to supply additional information about them, that is, instances of array axioms, at appropriate times. In a more general way, some SMT solvers accept sets of rewriting rules as input. Such a set of rules can also produce a decision procedure in specific cases [45].

Instantiation-based theorem prover such as I-prover [46] are semi-complete procedures for first-order logic based on instantiation [47]. To choose appropriate instances, they use a heuristic based on models. A ground solver is launched on the set of ground facts contains in the input problem. If a model is found, they use it to generate instances of universally quantified formulas from the problem that contradict this model. Unlike in our framework, no specific proof is required to insure the completeness of this procedure. On the other hand, this procedure only works modulo theories for which there is an "answer-complete" ground solver, that is, a solver able to find a most general substitution for ground instances of a set of literals with free variables to be unsatisfiable [37]. For example, there can be no answer-complete solver for arithmetics.

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

Lynch et al. then extended this work by introducing automatic procedures for deciding the termination [53, 54]. In particular, *Schematic Saturation* is used to over-approximate the inferences that paramodulation can generate while solving the satisfiability problem for a certain theory. As opposed to our work, they do not reason modulo a background theory. To allow combination with other decision procedures in a Nelson-Oppen framework, they explain how schematic saturation can also be used to check stably-infiniteness and complete deduction in some cases.

Other ways of integrating decision procedures in an SMT solver: Apart from first-order axiomatizations, there are several ways that can be attempted to add a new decision procedure into a SMT solver. Nikolaj Bjørner describes how several non-native theories where successfully supported using the SMT solver Z3 [12]. A rather usual way to support a new theory is to encode it in an already supported domain. This paper also presents an API that can be used to add decision procedure to Z3 using callbacks. Both systems require a lot of thinking to get an efficient design as well as manual proofs of completeness and termination, exactly like our axiomatization framework. We believe that each of these different techniques can be better suited than the others to specific theories.

7.3 Perspectives

Try our Framework on Other Classical Theories: A lot of different theories have been axiomatized in first-order logic to alleviate the lack of a specific decision procedure. It is the case for example of algebraic data-types and monoïds. It would be interesting to see if it can be decided using our approach.

Use Literals as Triggers: As discussed in Section 4.3, in our white-box implementation in Alt-Ergo (like in most SMT solvers) we only handle terms as triggers. It would be worthwhile to support literal triggers, as described in our theoretical framework, in particular to handle extensionality axioms.

Automatic Checking for Completeness and Termination: In Section 2.3, we have presented two automatable techniques for checking termination and completeness of an axiomatization with triggers. It would be interesting to implement them and to improve their range of application, as we believe they would be a valuable help for developers of new axiomatizations. Such an implementation could also be used to check triggers automatically computed by SMT solvers as part of the E-matching heuristic or even to automatically compute triggers for a given axiomatization.

Automatic Generation of Complete Triggers: E-matching, that is, instantiation with triggers is the most popular heuristic to handle first-order logic in SMT solvers. The manual annotation of every universal quantifier with triggers is not usually required though. These solvers rather use heuristics to automatically choose triggers for quantified formula. One of the most common heuristic consists in choosing one or several terms from the formula. We could use our framework to characterize formulas for which a specific heuristic for choosing triggers always succeeds in yielding a complete axiomatization.

Learn Conflict Clauses in DPLL$^\star(T)$: The DPLL* framework described in Chapter 4 does not allow learning of conflict clauses. As the ability to keep clauses that have lead to previous conflicts is one of the keys for efficiency of modern SAT and SMT solvers, we would like to extend our framework to allow it. In particular, as we notice in Remark 4.3, that would require extending our logic of guarded clauses to allow negations of closures in theory clauses.

Adapt Triggers to Reasoning Modulo Theory: In our framework, to instantiate a formula protected by a trigger l, we need l to be true and all its subterms to be known. When l contains interpreted subterms, in particular integer constants or operations, this may not be the appropriate behavior. For example, if l is $x > 0$, we may not want to ask for 0 to be known for l to be unfolded. To improve our semantics, we could group some interpreted subterms together and not require the presence of interpreted subterms of interpreted terms of l.

Combine Axiomatized Theories: In the longer term, we would like to investigate the combination of several theories defined as first-order axiomatizations. Indeed, if we want to add several theories using our framework, then we cannot do the proofs of completeness and termination modularly. In particular, we will need to determine which requirements are needed to preserve completeness and termination of axiomatizations. In a first approximation, we expect that a combination of two terminating axiomatizations W_1 and W_2 is terminating whenever triggers from W_2 only contain uninterpreted function symbols that do not appear in W_1. In the same way, we expect that the combination of two complete axiomatizations W_1 and W_2 is complete if W_2 can only deduce equalities between terms whose head symbol does not appear in W_1. More fine-grain combinations would require more thought.

Use Triggers in Other Automatic Theorem Provers: We could reuse our mechanism for triggers in other solvers than SMT. We could for example extend resolution and paramodulation with triggers to restrict the search space. An idea could be to consider triggers as guards and to only allow resolution and paramodulation inside the clause if all the guards have been removed. In other words, we would have to paramodulate away guards before accessing the content of a clause. We could try to prove that this restriction preserves semi-completeness on a complete axiomatization following Definition 2.7.

Consider Theories not Efficiently Axiomatized: There are theories on which our approach cannot be used as they have no practical first-order axiomatization with triggers. First, too many instances may be required to ensure completeness. It is the case for example of the $interval(i, j)$ function of the theory of sets that returns the set containing every integer between i and j. To be complete, we need to produce $mem(k, interval(i, j)) \approx \mathtt{t}$ for every integer k between i and j. Then, for some theories, there can be no efficient decision procedure based on instantiation, like for linear arithmetics. Finally, in some cases, triggers are not good hints to restrict instantiation of a formula. For example, in axioms for associativity and commutativity, the only triggers that ensure completeness are those that allow every equivalent form of an expression to be deduced. Efficient approaches for handling these axioms are based on normalization of terms via flattening [23].

Improve Provability in the SPARK 2014 Tool: Our theory mechanism is not included in the SPARK 2014 tool for now. Indeed, as we have seen in Chapter 6, it is rarely the case that the verification of a SPARK 2014 program heavily relies on data-structure reasoning. Even when a program requires several updates to a structure, they are generally done using a loop and therefore only one update at a time is seen by the tool. Still, a user may benefit from our mechanism to discharge only the verification conditions requiring a lot of theory reasoning, as an alternative solver. We could also experiment on other theories to see if we have a better gain.

Unless explicitly specified otherwise, the SMT solver Alt-Ergo is the only solver used in the SPARK 2014 tool. While using the tool we have witnessed several bottlenecks. First, linear arithmetics in Alt-Ergo is handled by a Fourier-Motzkin based approach which is significantly less efficient when there are a lot of inequalities in the problem. As Ada programs generally use lots of specific integer types with a precise range, the SPARK 2014 tool often runs into this

problem. Not surprisingly, the other big issue is quantifiers. Indeed, even verification conditions coming from a simple SPARK 2014 program like the `Max_Array` function introduced in Section 1.1.1 include more than one hundred universal quantifiers (170 for the postcondition of `Max_Array`).

Bibliography

[1] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, W. Mostowski, and P. H. Schmitt. The KeY system: Integrating object-oriented design and formal methods. In *Fundamental Approaches to Software Engineering*, pages 327–330. Springer, 2002.

[2] A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, 183(2):140 – 164, 2003. 12th International Conference on Rewriting Techniques and Applications (RTA 2001).

[3] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In *Fundamental approaches to software engineering*, pages 363–366. Springer, 2000.

[4] R. Bardou. *Verification of Pointer Programs Using Regions and Permissions*. Phd thesis, Université Paris-Sud, 2011.

[5] J. Barnes. Rationale for Ada 2012: 1 contracts and aspects. *ADA USER*, 32(4):247, 2011.

[6] J. Barnes. *SPARK: The Proven Approach to High Integrity Software*. Altran Praxis, 2012.

[7] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and analysis of safe, secure, and interoperable smart devices*, pages 49–69. Springer, 2005.

[8] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer Berlin Heidelberg, 2011.

[9] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.

[10] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard version 2.0. *Technical report, University of Iowa*, december 2010.

[11] P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. METEOR: A successful application of B in a large project. In *FM'99—Formal Methods*, pages 369–387. Springer, 1999.

[12] N. Bjørner. Engineering theories with Z3. *Programming Languages and Systems*, pages 4–16, 2011.

[13] N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[14] F. Bobot, S. Conchon, E. Contejean, and S. Lescuyer. Implementing polymorphism in SMT solvers. In *SMT'08*, volume 367 of *ACM ICPS*, pages 1–5. ACM, 2008.

[15] F. Bobot, J.-C. Filliâtre, C. Marché, A. Paskevich, et al. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, 2011.

[16] T. Bouton, D. C. B. De Oliveira, D. Déharbe, and P. Fontaine. veriT: an open, trustable and efficient SMT-solver. In *Automated Deduction–CADE-22*, pages 151–156. Springer, 2009.

[17] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation*, pages 427–442. Springer, 2006.

[18] D. Cansell and D. Méry. Foundations of the B method. *Computing and informatics*, 22(3-4):221–256, 2012.

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

[20] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT press, 1999.

[21] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009.

[22] D. R. Cok and J. R. Kiniry. Esc/Java2: Uniting Esc/Java and JML. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 108–128. Springer, 2005.

[23] S. Conchon, E. Contejean, and M. Iguernelala. Canonized rewriting and ground AC completion modulo Shostak theories. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 45–59. Springer, 2011.

[24] S. Conchon, E. Contejean, J. Kanig, and S. Lescuyer. Cc(x): Semantic combination of congruence closure with solvable theories. *Electronic Notes in Theoretical Computer Science*, 198(2):51–69, May 2008.

[25] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[26] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C. In *Software Engineering and Formal Methods*, pages 233–247. Springer, 2012.

[27] D. Cyrluk, P. Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In *Automated Deduction—CADE-13*, pages 463–477. Springer, 1996.

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

[29] L. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In *CADE-21*, volume 4603 of *LNCS*, pages 183–198. Springer, 2007.

[30] L. de Moura and N. Bjørner. Engineering DPLL(T) + saturation. In *IJCAR 2008*, volume 5195 of *LNCS*, pages 475–490. Springer, 2008.

[31] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[32] L. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pages 45–52. IEEE, 2009.

[33] L. de Moura and N. Bjørner. Satisfiability modulo theories: An appetizer. In *Formal Methods: Foundations and Applications*, pages 23–36. Springer, 2009.

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

[35] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

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

[37] H. Ganzinger and K. Korovin. Theory instantiation. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 497–511. Springer, 2006.

[38] Y. Ge, C. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In *CADE-21*, volume 4603 of *LNCS*, pages 167–182. Springer, 2007.

[39] Y. Ge and L. De Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Computer Aided Verification*, volume 5643 of *LNCS*, pages 306–320. Springer, 2009.

[40] A. Goel, S. Krstić, and A. Fuchs. 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, pages 12–17. ACM, 2008.

[41] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *ACM SIGPLAN Notices*, volume 43, pages 339–348. ACM, 2008.

[42] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[43] B. Jacobs and F. Piessens. The VeriFast program verifier. *CW Reports*, 2008.

[44] S. Jacobs and V. Kuncak. Towards complete reasoning about axiomatic specifications. In *Proceedings of VMCAI-12*, volume 6538 of *LNCS*, pages 278–293. Springer, 2011.

[45] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In *Automation of Reasoning*, pages 342–376. Springer, 1983.

[46] K. Korovin. iProver–an instantiation-based theorem prover for first-order logic (system description). In *Automated Reasoning*, pages 292–298. Springer, 2008.

[47] K. Korovin. An invitation to instantiation-based reasoning: From theory to practice. *Volume in memoriam of Harald Ganzinger, Lecture Notes in Computer Science. Springer*, 2009.

[48] L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Fundamental Approaches to Software Engineering*, pages 470–485. Springer, 2009.

[49] D. Larraz, E. Rodríguez-Carbonell, and A. Rubio. SMT-based array invariant generation. In R. Giacobazzi, J. Berdine, and I. Mastroeni, editors, *VMCAI*, volume 7737 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2013.

[50] K. R. M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.

[51] K. R. M. Leino. This is Boogie 2. *Manuscript KRML*, 178, 2008.

[52] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370. Springer, 2010.

[53] C. Lynch and B. Morawska. Automatic decidability. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 7–16. IEEE, 2002.

[54] C. Lynch, S. Ranise, C. Ringeissen, and D.-K. Tran. Automatic decidability and combinability. volume 209, pages 1026–1047. Elsevier, 2011.

[55] S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *Computer Aided Verification*, pages 476–490. Springer, 2005.

[56] M. Moskal. Programming with triggers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, ACM ICPS, pages 20–29. ACM, 2009.

[57] Y. Moy and C. Marché. Modular inference of subprogram contracts for safety checking. *Journal of Symbolic Computation*, 45:1184–1211, 2010.

[58] G. Nelson. Techniques for program verification. *Technical Report CSL81-10, Xerox Palo Alto Research Center*, 1981.

[59] G. Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):517–561, 1989.

[60] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

[61] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. 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.

[62] I. O'Neill. SPARK – a language and tool-set for high-integrity software development. In J.-L. Boulanger, editor, *Industrial Use of Formal Methods: Formal Verification*. Wiley, 2012.

[63] S. Qadeer. Algorithmic verification of systems software using SMT solvers. In J. Palsberg and Z. Su, editors, *SAS*, volume 5673 of *Lecture Notes in Computer Science*, page 2. Springer, 2009.

[64] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. 2002.

[65] A. Riazanov and A. Voronkov. Vampire 1.1. In *Automated Reasoning*, pages 376–380. Springer, 2001.

[66] P. Rümmer. E-matching with free variables. In *Logic for Programming, Artificial Intelligence, and Reasoning: 18th International Conference, LPAR-18*, volume 7180 of *LNCS*, pages 359–374. Springer, 2012.

[67] S. Schneider. *The B-method: An introduction*, volume 200. Palgrave Oxford, 2001.

[68] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[69] R. E. Shostak. Deciding combinations of theories. In *6th Conference on Automated Deduction*, pages 209–222. Springer, 1982.

[70] J. Souyris, V. Wiels, D. Delmas, and H. Delseny. Formal verification of avionics software products. In *FM 2009: Formal Methods*, pages 532–546. Springer, 2009.

[71] N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying higher-order programs with the Dijkstra monad. In H.-J. Boehm and C. Flanagan, editors, *PLDI*, pages 387–398. ACM, 2013.

[72] A. Tafat. *Preuves par raffinement de programmes avec pointeurs*. Phd thesis, Université Paris-Sud, 2013.

[73] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topić. Spass version 2.0. In *Automated Deduction—CADE-18*, pages 275–279. Springer, 2002.

Index

A

Abstract interpretation 6
Ada 2012 2
Alt-Ergo SMT solver 10
Anti-closure 59
Automatic theorem prover (ATP) 8
Available super-clauses 59

B

B Method 87
Boogie intermediate language 8
Bounded model checking 6
BWare project 87

C

Closed formula 16
Closure 24
Complete world 18
Completeness of an axiomatization 21
Conflict clause in DPLL* framework 65
CVC4 SMT solver 10

D

Dafny programing language 7
Deductive verification 7
Defined super-literal in DPLL* framework . 63
DPLL algorithm 9
DPLL* algorithm 64

E

E-matching 118
ESC/Java verifier 7

F

Fair DPLL* derivation 71
Feasibility of a formula with triggers 18
Feasibility of a guarded clause 60
Final truth assignment 24
Frama-C verifier 6

G

Ground term, literal, formula 16
Guard 46, 59
Guarded clause 46, 59

H

HAVOC verifier 6
Hoare logic . . . . . *see* WP calculus

I

Inhabited world 18
Instantiation level in DPLL* derivations . . . 71
Instantiation pattern . . . . . *see* Trigger
Instantiation tree 24
Iprover automatic theorem prover 8

K

KeY verifier 7
KIV verifier 7
Known term modulo T 18

L

Local model reasoning 118

M

Model based quantifier instantiation 118
Model of a formula with triggers 18
Model of a guarded clause 60

N

Nelson-Oppen combination procedure 9

O

Opaque literal 46

P

Pseudo-clause 23
Pseudo-literal 23

R

Redundant guarded clause 65
Redundant instantiation 24
Relevant closures 46
Relevant guarded clause 46

S

Satisfiability modulo T 17
Satisfiability modulo theory (SMT) solver . . . 8
Satisfiability of a formula with triggers 18
Satisfiability of a guarded clause 60
Simplify SMT solver 10
Skolemization 23
Soundness of an axiomatization 21
SPARK 2014 Language 2
SPARK 2014 Tool 97
SPASS automatic theorem prover 8
Spec# specification language 7
State of DPLL* framework 63
Strong termination of an axiomatization . . . 25
Super-clause 59
Super-literal 59

T

Theory clause 24
Trigger 10, 17
Truth assignment 24
Truth value of a formula with triggers 18
Truth value of a guarded clause 59

U

User clause 58

V

Vampire automatic theorem prover 8
VCC verifier 6
Verifast verifier 7
Verification Condition 7
VeriT SMT solver 10

W

Weak termination of an axiomatization 25
Weakest Precondition (WP) calculus 7
Why3 intermediate language 8
Witness 17
World modulo T 18

Z

Z3 SMT solver 10

Glossary

$\mathscr{T}(t)$ set of all the subterms of a term t. 16

$L \rhd \varphi$ the formula with triggers φ is true in the world L. 18

$\|\varphi\|$ is the formula with triggers φ where triggers have been replaced with implications and witnesses with conjunctions. 19

$\lfloor V \rfloor$ is $\{l\sigma \mid l \cdot \sigma \in V\}$ for any set of theory clause V. 24

$L \blacktriangleright H \to C$ the guarded formula $H \to C$ is true in the world L. 60

$[\![H \to C]\!]$ is the encoding of the guarded formula $H \to C$ into a user clause, replacing triggers, witnesses, and universally quantified formulas with fresh predicate symbols. 46, 60

$\lceil M \rceil$ is $M \cup \{l \cdot \varnothing \mid l \in \textsc{Lit}(M)\}$ for any set of super-literals M. 63

$F \vdash^{\star}_{T} C$ the super-clause C is redundant in the set of super-clauses F, that is, the behavior of the solver will be the same with and without it.. 61

$S \models^{\star}_{T} E$ $[\![S]\!]$ entails $[\![E]\!]$ modulo the theory T, for any conjunctive set of super-literals and guarded clauses S and any super-literal of guarded clause E. 60

Anti-closure the negation of a closure $\neg(\varphi \cdot \sigma)$. 59

Closure pair of a pseudo-literal and a substitution $\varphi \cdot \sigma$. 24

Guarded clause a pair $H \to C$ of a conjunctive set of closures H and a super-clause C. 46, 59

Instantiation tree representation of any possible run of the solver using a particular instantiation strategy. 24

Pseudo-clause disjunctive set of pseudo-literals. 23

Pseudo-literal either a literal or a universally quantified formula, a trigger, or a witness over a pseudo-clause. 23

Super-clause either a user clause or a theory clause. 59

Super-literal either a literal, a closure, or an anti-closure. 59

Theory clause disjunctive set of closures. 24

Truth assignment set of further facts that a solver may assume from a set of theory clause using only propositional reasoning. 24

User clause a disjunctive set of literals. 59

A Imperative Doubly-Linked Lists

A.1 Axiomatization

LENGTH_GTE_ZERO:

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

IS_EMPTY:

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

EMPTY_IS_EMPTY:

$$is\_empty(empty)$$

EQUAL_ELEMENTS_REFL:

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

EQUAL_ELEMENTS_SYM:

$$\begin{array}{l}\forall e_1 e_2 : element\_type.[equal\_elements(e_1,e_2)] \\ \quad equal\_elements(e_1,e_2) \approx equal\_elements(e_2,e_1)\end{array}$$

EQUAL_ELEMENTS_TRANS:

$$\begin{array}{l}\forall e_1 e_2 e_3 : element\_type.[equal\_elements(e_1,e_2), equal\_elements(e_2,e_3)] \\ \quad equal\_elements(e_1,e_2) \approx \mathtt{t} \rightarrow equal\_elements(e_2,e_3) \approx \mathtt{t} \rightarrow \\ \quad\quad equal\_elements(e_1,e_3) \approx \mathtt{t} \\ \forall e_1 e_2 e_3 : element\_type.[equal\_elements(e_1,e_2), equal\_elements(e_1,e_3)] \\ \quad equal\_elements(e_1,e_2) \approx \mathtt{t} \rightarrow equal\_elements(e_2,e_3) \approx \mathtt{t} \rightarrow \\ \quad\quad equal\_elements(e_1,e_3) \approx \mathtt{t}\end{array}$$

POSITION_GTE_ZERO:

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

POSITIONNO_ELEMENT:

$$\forall co : list.[position(co, no\_element)]position(co, no\_element) \approx 0$$

POSITION_EQ:

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

PREVIOUS_IN:

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

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:

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

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:

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

FIND_FIRST_FIRST:

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

FIND_FIRST_ELEMENT:

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

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:

$$\forall co_1, co_2 : list, cu : cursor, e : element\_type.[replace\_element(co_1,cu,e,co_2)]$$
$$replace\_element(co_1,cu,e,co_2) \approx \mathtt{t} \rightarrow position(co_1,cu) > 0$$

REPLACE_ELEMENT_LENGTH:

$$\forall co_1, co_2 : list, cu : cursor, e : element\_type.[replace\_element(co_1,cu,e,co_2)]$$
$$replace\_element(co_1,cu,e,co_2) \approx \mathtt{t} \rightarrow length(co_1) \approx length(co_2)$$

REPLACE_ELEMENT_POSITION:

$$\forall co_1, co_2 : list, cu_1 cu_2 : cursor, e : element\_type.$$
$$[replace\_element(co_1,cu_1,e,co_2), position(co_1,cu_2)]$$
$$replace\_element(co_1,cu_1,e,co_2) \approx \mathtt{t} \rightarrow position(co_1,cu_2) \approx position(co_2,cu_2)$$
$$\forall co_1, co_2 : list, cu_1 cu_2 : cursor, e : element\_type.$$
$$[replace\_element(co_1,cu_1,e,co_2), position(co_2,cu_2)]$$
$$replace\_element(co_1,cu_1,e,co_2) \approx \mathtt{t} \rightarrow position(co_1,cu_2) \approx position(co_2,cu_2)$$

REPLACE_ELEMENT_ELEMENT_IN:

$$\forall co_1, co_2 : list, cu : cursor, e : element\_type.[replace\_element(co_1,cu,e,co_2)]$$
$$replace\_element(co_1,cu,e,co_2) \approx \mathtt{t} \rightarrow element(co_2,cu) \approx e$$

REPLACE_ELEMENT_ELEMENT_EXT:

$$\begin{array}{l}\forall co_1, co_2 : list, cu_1, cu_2 : cursor, e : element\_type. \\ \quad [replace\_element(co_1, cu_1, e, co_2), element(co_1, cu_2)] \\ \quad\quad (replace\_element(co_1, cu_1, e, co_2) \approx \mathtt{t} \wedge position(co_1, cu_2) > 0 \wedge cu_1 \not\approx cu_2) \rightarrow \\ \quad\quad\quad element(co_1, cu_2) \approx element(co_2, cu_2) \\ \forall co_1, co_2 : list, cu_1, cu_2 : cursor, e : element\_type. \\ \quad [replace\_element(co_1, cu_1, e, co_2), element(co_2, cu_2)] \\ \quad\quad (replace\_element(co_1, cu_1, e, co_2) \approx \mathtt{t} \wedge position(co_1, cu_2) > 0 \wedge cu_1 \not\approx cu_2) \rightarrow \\ \quad\quad\quad element(co_1, cu_2) \approx element(co_2, cu_2)\end{array}$$

INSERT_RANGE:

$$\begin{array}{l}\forall co_1, co_2 : list, cu : cursor, e : element\_type.[insert(co_1, cu, e, co_2)] \\ \quad\quad insert(co_1, cu, e, co_2) \approx \mathtt{t} \rightarrow cu \approx no\_element \vee position(co_1, cu) > 0\end{array}$$

INSERT_LENGTH:

$$\begin{array}{l}\forall co_1, co_2 : list, cu : cursor, e : element\_type.[insert(co_1, cu, e, co_2)] \\ \quad\quad insert(co_1, cu, e, co_2) \approx \mathtt{t} \rightarrow length(co_2) \approx length(co_1) + 1\end{array}$$

INSERT_NEW:

$$\begin{array}{l}\forall co_1, co_2 : list, cu : cursor, e : element\_type.[insert(co_1, cu, e, co_2)] \\ \quad (insert(co_1, cu, e, co_2) \approx \mathtt{t} \wedge position(co_1, cu) > 0) \rightarrow \\ \quad\quad position(co_1, previous(co_2, cu)) \approx 0 \wedge element(co_2, previous(co_2, cu)) \approx e\end{array}$$

INSERT_NEW_NO_ELEMENT:

$$\begin{array}{l}\forall co_1, co_2 : list, cu : cursor, e : element\_type.[insert(co_1, no\_element, e, co_2)] \\ \quad\quad insert(co_1, no\_element, e, co_2) \approx \mathtt{t} \rightarrow \\ \quad\quad\quad position(co_1, last(co_2)) \approx 0 \wedge element(co_2, last(co_2)) \approx e\end{array}$$

INSERT_POSITION_BEFORE:

$$\begin{array}{l}\forall co_1, co_2 : list, cu_1 cu_2 : cursor, e : element\_type.[insert(co_1, cu_1, e, co_2), position(co_1, cu_2)] \\ \quad (insert(co_1, cu_1, e, co_2) \approx \mathtt{t} \wedge 0 < position(co_1, cu_2) < position(co_1, cu_1)) \rightarrow \\ \quad\quad position(co_1, cu_2) \approx position(co_2, cu_2) \\ \forall co_1, co_2 : list, cu_1 cu_2 : cursor, e : element\_type.[insert(co_1, cu_1, e, co_2), position(co_2, cu_2)] \\ \quad (insert(co_1, cu_1, e, co_2) \approx \mathtt{t} \wedge position(co_2, cu_2) < position(co_1, cu_1)) \rightarrow \\ \quad\quad position(co_1, cu_2) \approx position(co_2, cu_2)\end{array}$$

INSERT_POSITION_AFTER:

$$\begin{array}{l}\forall co_1, co_2 : list, cu_1 cu_2 : cursor, e : element\_type.[insert(co_1, cu_1, e, co_2), position(co_1, cu_2)] \\ \quad (insert(co_1, cu_1, e, co_2) \approx \mathtt{t} \wedge position(co_1, cu_2) \geq position(co_1, cu_1) > 0) \rightarrow \\ \quad\quad position(co_1, cu_2) + 1 \approx position(co_2, cu_2) \\ \forall co_1, co_2 : list, cu_1 cu_2 : cursor, e : element\_type.[insert(co_1, cu_1, e, co_2), position(co_2, cu_2)] \\ \quad (insert(co_1, cu_1, e, co_2) \approx \mathtt{t} \wedge position(co_2, cu_2) > position(co_1, cu_1) > 0) \rightarrow \\ \quad\quad position(co_1, cu_2) + 1 \approx position(co_2, cu_2)\end{array}$$

INSERT_POSITION_NO_ELEMENT:

$$\begin{aligned}
&\forall co_1, co_2 : list, cu : cursor, e : element_type.[insert(co_1, no_element, e, co_2), position(co_1, cu)] \\
&\quad (insert(co_1, no_element, e, co_2) \approx \mathtt{t} \wedge position(co_1, cu) > 0) \rightarrow \\
&\qquad position(co_1, cu) \approx position(co_2, cu) \\
&\forall co_1, co_2 : list, cu : cursor, e : element_type.[insert(co_1, no_element, e, co_2), position(co_2, cu)] \\
&\quad (insert(co_1, no_element, e, co_2) \approx \mathtt{t} \wedge position(co_2, cu_2) < length(co_2)) \rightarrow \\
&\qquad position(co_1, cu) \approx position(co_2, cu)
\end{aligned}$$

INSERT_ELEMENT:

$$\begin{aligned}
&\forall co_1, co_2 : list, cu_1 cu_2 : cursor, e : element_type.[insert(co_1, cu_1, e, co_2), element(co_1, cu_2)] \\
&\quad (insert(co_1, cu_1, e, co_2) \approx \mathtt{t} \wedge position(co_1, cu_2) > 0) \rightarrow \\
&\qquad element(co_1, cu_2) \approx element(co_2, cu_2) \\
&\forall co_1, co_2 : list, cu_1 cu_2 : cursor, e : element_type.[insert(co_1, cu_1, e, co_2), element(co_2, cu_2)] \\
&\quad (insert(co_1, cu_1, e, co_2) \approx \mathtt{t} \wedge position(co_1, cu_2) > 0) \rightarrow \\
&\qquad element(co_1, cu_2) \approx element(co_2, cu_2)
\end{aligned}$$

DELETE_RANGE:

$$\begin{aligned}
&\forall co_1, co_2 : list, cu : cursor.[delete(co_1, cu, co_2)] \\
&\quad delete(co_1, cu, co_2) \approx \mathtt{t} \rightarrow position(co_1, cu) > 0
\end{aligned}$$

DELETE_LENGTH:

$$\begin{aligned}
&\forall co_1, co_2 : list, cu : cursor.[delete(co_1, cu, co_2)] \\
&\quad delete(co_1, cu, co_2) \approx \mathtt{t} \rightarrow length(co_2) + 1 \approx length(co_1)
\end{aligned}$$

DELETE_POSITION_BEFORE:

$$\begin{aligned}
&\forall co_1, co_2 : list, cu_1 cu_2 : cursor.[delete(co_1, cu_1, co_2), position(co_1, cu_2)] \\
&\quad (delete(co_1, cu_1, co_2) \approx \mathtt{t} \wedge position(co_1, cu_2) < position(co_1, cu_1)) \rightarrow \\
&\qquad position(co_1, cu_2) \approx position(co_2, cu_2) \\
&\forall co_1, co_2 : list, cu_1 cu_2 : cursor.[delete(co_1, cu_1, co_2), position(co_2, cu_2)] \\
&\quad (delete(co_1, cu_1, co_2) \approx \mathtt{t} \wedge 0 < position(co_2, cu_2) < position(co_1, cu_1)) \rightarrow \\
&\qquad position(co_1, cu_2) \approx position(co_2, cu_2)
\end{aligned}$$

DELETE_POSITION_AFTER:

$$\begin{aligned}
&\forall co_1, co_2 : list, cu_1 cu_2 : cursor.[delete(co_1, cu_1, co_2), position(co_1, cu_2)] \\
&\quad (delete(co_1, cu_1, co_2) \approx \mathtt{t} \wedge position(co_1, cu_2) > position(co_1, cu_1)) \rightarrow \\
&\qquad position(co_1, cu_2) \approx position(co_2, cu_2) + 1 \\
&\forall co_1, co_2 : list, cu_1 cu_2 : cursor.[delete(co_1, cu_1, co_2), position(co_2, cu_2)] \\
&\quad (delete(co_1, cu_1, co_2) \approx \mathtt{t} \wedge position(co_2, cu_2) \geq position(co_1, cu_1)) \rightarrow \\
&\qquad position(co_1, cu_2) \approx position(co_2, cu_2) + 1
\end{aligned}$$

DELETE_POSITION_NEXT:

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

DELETE_ELEMENT:

$$\begin{aligned}
&\forall co_1, co_2 : list, cu_1 cu_2 : cursor.[delete(co_1, cu_1, co_2), element(co_1, cu_2)] \\
&\quad (delete(co_1, cu, co_2) \approx \mathtt{t} \wedge position(co_2, cu_2) > 0) \rightarrow \\
&\qquad element(co_1, cu_2) = element(co_2, cu_2)
\end{aligned}$$

EQUAL_LISTS_POSITION:

$$\forall co_1, co_2 : list.[equal\_lists(co_1, co_2)]equal\_lists(co_1, co_2) \approx \mathtt{t} \rightarrow \\ (\forall cu : cursor.[position(co_1, cu)]position(co_1, cu) \approx position(co_2, cu)) \wedge \\ (\forall cu : cursor.[position(co_2, cu)]position(co_1, cu) \approx position(co_2, cu))$$

EQUAL_LISTS_ELEMENT:

$$\forall co_1, co_2 : list.[equal\_lists(co_1, co_2)]equal\_lists(co_1, co_2) \approx \mathtt{t} \rightarrow \\ (\forall cu : cursor.[element(co_1, cu)]position(co_1, cu) > 0 \rightarrow \\ element(co_1, cu) \approx element(co_2, cu)) \wedge \\ (\forall cu : cursor.[element(co_2, cu)]position(co_1, cu) > 0 \rightarrow \\ element(co_1, cu) \approx element(co_2, cu))$$

EQUAL_LISTS_INV:

$$\forall co_1, co_2 : list.[equal\_lists(co_1, co_2)]equal\_lists(co_1, co_2) \not\approx \mathtt{t} \rightarrow \\ (\exists cu : cursor.position(co_1, cu) > 0 \wedge \\ (position(co_1, cu) \approx position(co_2, cu) \rightarrow element(co_1, cu) \not\approx element(co_2, cu)))$$

EQUAL_LISTS_LENGTH:

$$\forall co_1, co_2 : list.[equal\_lists(co_1, co_2)]equal\_lists(co_1, co_2) \approx \mathtt{t} \rightarrow length(co_1) \approx length(co_2)$$

A.2 Tests in WhyML

A.2.1 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 }
```

A.2.2 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
```

B Why3 Sets

B.1 Axiomatization for Sets

EQ_DEF :

$$\begin{aligned}&\forall s_1, s_2 : set\,\alpha.[equal\_sets(s_1,s_2)]equal\_sets(s_1,s_2) \approx \mathtt{t} \rightarrow \\ &\quad (\forall x : \alpha.[mem(x,s_1)]mem(x,s_1) \approx \mathtt{t} \rightarrow mem(x,s_2) \approx \mathtt{t}) \wedge \\ &\quad (\forall x : \alpha.[mem(x,s_2)]mem(x,s_2) \approx \mathtt{t} \rightarrow mem(x,s_1) \approx \mathtt{t})\end{aligned}$$

EQ_INV :

$$\begin{aligned}&\forall s_1, s_2 : set\,\alpha.[equal\_sets(s_1,s_2)]equal\_sets(s_1,s_2) \not\approx \mathtt{t} \rightarrow \\ &\quad (\exists x : \alpha.mem(x,s_1) \approx \mathtt{t} \wedge mem(x,s_2) \not\approx \mathtt{t} \vee mem(x,s_2) \approx \mathtt{t} \wedge mem(x,s_1) \not\approx \mathtt{t}))\end{aligned}$$

EXTENSIONALITY :

$$\forall s_1, s_2 : set\,\alpha.[s_1 \not\approx s_2]equal\_sets(s_1,s_2) \approx \mathtt{t} \rightarrow s_1 \approx s_2$$

SUBSET_DEF :

$$\begin{aligned}&\forall s_1, s_2 : set\,\alpha.[subset(s_1,s_2))] \\ &\quad subset(s_1,s_2) \approx \mathtt{t} \rightarrow (\forall x : \alpha.[mem(x,s_1)]mem(x,s_1) \approx \mathtt{t} \rightarrow mem(x,s_2) \approx \mathtt{t})\end{aligned}$$

SUBSET_INV :

$$\begin{aligned}&\forall s_1, s_2 : set\,\alpha.[subset(s_1,s_2))] \\ &\quad subset(s_1,s_2) \not\approx \mathtt{t} \rightarrow (\exists x : \alpha.mem(x,s_1) \approx \mathtt{t} \wedge mem(x,s_2) \not\approx \mathtt{t})\end{aligned}$$

UNION_DEF :

$$\begin{aligned}&\forall s_1, s_2 : set\,\alpha, x : \alpha.[mem(x,union(s_1,s_2))] \\ &\quad mem(x,union(s_1,s_2)) \approx \mathtt{t} \rightarrow mem(x,s_1) \approx \mathtt{t} \vee mem(x,s_2) \approx \mathtt{t}\end{aligned}$$

UNION_INV1 :

$$\forall s_1, s_2 : set\,\alpha, x : \alpha.[union(s_1,s_2), mem(x,s_1)]mem(x,s_1) \approx \mathtt{t} \rightarrow mem(x,union(s_1,s_2)) \approx \mathtt{t}$$

UNION_INV2 :

$$\forall s_1, s_2 : set\,\alpha, x : \alpha.[union(s_1,s_2), mem(x,s_2)]mem(x,s_2) \approx \mathtt{t} \rightarrow mem(x,union(s_1,s_2)) \approx \mathtt{t}$$

INTER_DEF :

$$\forall s_1, s_2 : set\,\alpha, x : \alpha.[mem(x, inter(s_1, s_2))]$$
$$mem(x, inter(s_1, s_2)) \approx \mathtt{t} \rightarrow mem(x, s_1) \approx \mathtt{t} \wedge mem(x, s_2) \approx \mathtt{t}$$

INTER_INV :

$$\forall s_1, s_2 : set\,\alpha, x : \alpha.[inter(s_1, s_2), mem(x, s_1)]$$
$$mem(x, s_1) \approx \mathtt{t} \rightarrow mem(x, s_2) \approx \mathtt{t} \rightarrow mem(x, inter(s_1, s_2)) \approx \mathtt{t}$$
$$\forall s_1, s_2 : set\,\alpha, x : \alpha.[inter(s_1, s_2), mem(x, s_2)]$$
$$mem(x, s_1) \approx \mathtt{t} \rightarrow mem(x, s_2) \approx \mathtt{t} \rightarrow mem(x, inter(s_1, s_2)) \approx \mathtt{t}$$

DIFF_DEF1 :

$$\forall s_1, s_2 : set\,\alpha, x : \alpha.[mem(x, diff(s_1, s_2))]$$
$$mem(x, diff(s_1, s_2)) \approx \mathtt{t} \rightarrow mem(x, s_1) \approx \mathtt{t} \wedge mem(x, s_2) \not\approx \mathtt{t}$$

DIFF_DEF2 :

$$\forall s_1, s_2 : set\,\alpha, x : \alpha.[diff(s_1, s_2), mem(x, s_2)]mem(x, s_2) \approx \mathtt{t} \rightarrow mem(x, diff(s_1, s_2)) \not\approx \mathtt{t}$$

DIFF_INV :

$$\forall s_1, s_2 : set\,\alpha, x : \alpha.[diff(s_1, s_2), mem(x, s_1)]$$
$$mem(x, s_1) \approx \mathtt{t} \rightarrow mem(x, s_2) \not\approx \mathtt{t} \rightarrow mem(x, diff(s_1, s_2)) \approx \mathtt{t}$$

ADD_DEF :

$$\forall x, y : \alpha, s : set\,\alpha.[mem(x, add(y, s))]mem(x, add(y, s)) \approx \mathtt{t} \rightarrow x \approx y \vee mem(x, s) \approx \mathtt{t}$$

ADD_INV1:

$$\forall x, y : \alpha, s : set\,\alpha.[add(y, s), mem(x, s)]mem(x, s) \approx \mathtt{t} \rightarrow mem(x, add(y, s)) \approx \mathtt{t}$$

ADD_INV2 :

$$\forall y : \alpha, s : set\,\alpha.[add(y, s)]mem(y, add(y, s)) \approx \mathtt{t}$$

REMOVE_DEF :

$$\forall x, y : \alpha, s : set\,\alpha.[mem(x, remove(y, s))]mem(x, remove(y, s)) \approx \mathtt{t} \rightarrow x \not\approx y \wedge mem(x, s) \approx \mathtt{t}$$

REMOVE_INV:

$$\forall x, y : \alpha, s : set\,\alpha.[remove(y, s), mem(x, s)]mem(x, s) \approx \mathtt{t} \rightarrow x \not\approx y \rightarrow mem(x, remove(y, s)) \approx \mathtt{t}$$

IS_EMPTY_DEF :

$$\forall s : set\,\alpha.[is\_empty(s)]is\_empty(s) \approx \mathtt{t} \rightarrow (\forall x : \alpha.[mem(x, s)]mem(x, s) \not\approx \mathtt{t})$$

CHOOSE_DEF:

$$\forall s : set\,\alpha.[is\_empty(s)]is\_empty(s) \not\approx \mathtt{t} \rightarrow mem(choose(s), s) \approx \mathtt{t}$$
$$\forall s : set\,\alpha.[choose(s)]is\_empty(s) \not\approx \mathtt{t} \rightarrow mem(choose(s), s) \approx \mathtt{t}$$

C SPARK 2014 Vectors

C.1 Axiomatization for Formal Vectors

CAPACITY_RANGE:

$$\forall co : vector.[capacity(co)]$$
$$0 \leq capacity(co) \leq index\_type\__last - index\_type\__zero$$

LENGTH_RANGE:

$$\forall co : vector.[length(co)]\ 0 \leq length(co) \leq capacity(co)$$

TO_INDEX_RANGE:

$$\forall cu : cursor.[to\_index(cu)]$$
$$index\_type\__zero \leq to\_index(cu) \leq index\_type\__last$$

TO_INDEX_NO_ELEMENT:

$$to\_index(no\_element) \approx 0$$

TO_INDEX_TO_CURSOR:

$$\forall i : int, co : vector.[to\_cursor(co,i)]$$
$$(length(co) + index\_type\__zero \geq i > index\_type\__zero \rightarrow$$
$$to\_index(to\_cursor(co,i)) \approx i) \wedge$$
$$(length(co) + index\_type\__zero < i \vee i \approx index\_type\__zero \rightarrow$$
$$to\_cursor(co,i) \approx no\_element)$$

HAS_ELEMENT__DEF:

$$\forall cu : cursor, co : vector.[has\_element(co,cu)]$$
$$has\_element(co,cu) \approx \mathtt{t} \leftrightarrow$$
$$(to\_index(cu) > 0 \wedge to\_cursor(co, to\_index(cu)) \approx cu)$$

HAS_ELEMENT__TO_CURSOR:

$$\forall i : int, co : vector.[to\_cursor(co,i)]$$
$$length(co) + index\_type\_\_zero \geq i > index\_type\_\_zero \rightarrow$$
$$has\_element(co, to\_cursor(co,i)) \approx \mathtt{t}$$

PREVIOUS_IN:

$$\forall co : vector, cu : cursor.[previous(co,cu)]$$
$$(length(co) + index\_type\_\_zero \geq to\_index(cu) > index\_type\_\_first \vee$$
$$length(co) + index\_type\_\_zero \geq to\_index(previous(co,cu)) > index\_type\_\_zero) \rightarrow$$
$$to\_cursor(co, to\_index(cu) - 1) \approx previous(co,cu)$$

PREVIOUS_EXT:

$$\forall co : vector, cu : cursor.[previous(co,cu)]$$
$$(to\_index(cu) = index\_type\_\_first \vee cu \approx no\_element) \rightarrow$$
$$previous(co,cu) \approx no\_element$$

NEXT_IN:

$$\forall co : vector, cu : cursor.[next(co,cu)]$$
$$(length(co) + index\_type\_\_zero > to\_index(cu) > index\_type\_\_zero \vee$$
$$length(co) + index\_type\_\_zero \geq to\_index(next(co,cu)) > index\_type\_\_zero) \rightarrow$$
$$to\_cursor(co, to\_index(cu) + 1) \approx next(co,cu)$$

NEXT_EXT:

$$\forall co : vector, cu : cursor.[next(co,cu)]$$
$$(to\_index(cu) = length(co) + index\_type\_\_zero \vee cu \approx no\_element) \rightarrow$$
$$next(co,cu) \approx no\_element$$

LAST_GEN:

$$\forall co : vector.[last(co)]to\_cursor(co, length(co) + index\_type\_\_zero) \approx last(co)$$

FIRST_GEN:

$$\forall co : vector.[first(co)]to\_cursor(co, index\_type\_\_first) \approx first(co)$$

REPLACE_ELEMENT_RANGE:

$$\forall co_1, co_2 : vector, i : int, e : element\_type.[replace\_element(co_1, i, e, co_2)]$$
$$replace\_element(co_1, i, e, co_2) \approx \mathtt{t} \rightarrow index\_type\_\_zero < i \leq length(co_1) + index\_type\_\_zero$$

REPLACE_ELEMENT_LENGTH:

$$\forall co_1, co_2 : vector, i : int, e : element\_type.[replace\_element(co_1, i, e, co_2)]$$
$$replace\_element(co_1, e, l, co_2) \approx \mathtt{t} \rightarrow length(co_1) \approx length(co_2)$$

REPLACE_ELEMENT_CAPACITY:

$$\forall co_1, co_2 : vector, i : int, e : element\_type.[replace\_element(co_1, i, e, co_2)]$$
$$replace\_element(co_1, i, e, co_2) \approx \mathtt{t} \rightarrow capacity(co_1) \approx capacity(co_2)$$

REPLACE_ELEMENT_ELEMENT_1:

$$\forall co_1, co_2 : vector, i : int, e : element\_type.[replace\_element(co_1, i, e, co_2)]$$
$$replace\_element(co_1, i, e, co_2) \approx \mathtt{t} \rightarrow element(co_2, i) \approx e$$

REPLACE_ELEMENT_ELEMENT_2 :

$$\forall co_1, co_2 : vector, i, j : int, e : element\_type.[replace\_element(co_1, i, e, co_2), element(co_1, j)]$$
$$replace\_element(co_1, i, e, co_2) \approx \mathtt{t} \rightarrow$$
$$index\_type\_\_zero < j \leq length(co_1) + index\_type\_\_zero \wedge i \not\approx j \rightarrow$$
$$element(co_1, j) \approx element(co_2, j)$$
$$\forall co_1, co_2 : vector, i, j : int, e : element\_type.[replace\_element(co_1, i, e, co_2), element(co_2, j)]$$
$$replace\_element(co_1, i, e, co_2) \approx \mathtt{t} \rightarrow$$
$$index\_type\_\_zero < j \leq length(co_1) + index\_type\_\_zero \wedge i \not\approx j \rightarrow$$
$$element(co_1, j) \approx element(co_2, j)$$

REPLACE_ELEMENT_CURSORS:

$$\forall co_1, co_2 : vector, i, j : int, e : element\_type.[replace\_element(co_1, i, e, co_2), to\_cursor(co_1, j)]$$
$$replace\_element(co_1, i, e, co_2) \approx \mathtt{t} \rightarrow$$
$$index\_type\_\_zero < j \leq length(co_1) + index\_type\_\_zero \rightarrow$$
$$to\_cursor(co_1, j) \approx to\_cursor(co_2, j)$$
$$\forall co_1, co_2 : vector, i, j : int, e : element\_type.[replace\_element(co_1, i, e, co_2), to\_cursor(co_2, j)]$$
$$replace\_element(co_1, i, e, co_2) \approx \mathtt{t} \rightarrow$$
$$index\_type\_\_zero < j \leq length(co_1) + index\_type\_\_zero \rightarrow$$
$$to\_cursor(co_1, j) \approx to\_cursor(co_2, j)$$

INSERT_RANGE:

$$\forall co_1, co_2, r : vector, i : int.[insert(co_1, i, co_2, r)]$$
$$insert(co_1, i, co_2, r) \approx \mathtt{t} \rightarrow index\_type\_\_zero < i \leq length(co_1) + index\_type\_\_first$$

INSERT_LENGTH:

$$\forall co_1, co_2, r : vector, i : int.[insert(co_1, i, co_2, r)]$$
$$insert(co_1, i, co_2, r) \approx \mathtt{t} \rightarrow length(r) \approx length(co_2) + length(co_1)$$

INSERT_CAPACITY:

$$\forall co_1, co_2, r : vector, i : int.[insert(co_1, i, co_2, r)]$$
$$insert(co_1, i, co_2, r) \approx \mathtt{t} \rightarrow capacity(co_1) \approx capacity(r)$$

INSERT_ELEMENT_1:

$$\begin{aligned}
&\forall co_1, co_2, r : vector, i, j : int.[insert(co_1, i, co_2, r), element(co_1, j)] \\
&\quad insert(co_1, i, co_2, r) \approx \mathtt{t} \rightarrow index_type__zero < j < i \rightarrow \\
&\qquad element(co_1, j) \approx element(r, j) \\
&\forall co_1, co_2, r : vector, i, j : int.[insert(co_1, i, co_2, r), element(r, j)] \\
&\quad insert(co_1, i, co_2, r) \approx \mathtt{t} \rightarrow index_type__zero < j < i \rightarrow \\
&\qquad element(co_1, j) \approx element(r, j)
\end{aligned}$$

INSERT_ELEMENT_2 :

$$\begin{aligned}
&\forall co_1, co_2, r : vector, i, j : int.[insert(co_1, i, co_2, r), element(co_2, j)] \\
&\quad insert(co_1, i, co_2, r) \approx \mathtt{t} \rightarrow index_type__zero < j \leq length(co_2) + index_type__zero \rightarrow \\
&\qquad element(r, j + i - index_type__first) \approx element(co_2, j) \\
&\forall co_1, co_2, r : vector, i, j : int.[insert(co_1, i, co_2, r), element(r, j)] \\
&\quad insert(co_1, i, co_2, r) \approx \mathtt{t} \rightarrow i \leq j < i + length(co_2) \rightarrow \\
&\qquad element(r, j) \approx element(co_2, j - i + index_type__first)
\end{aligned}$$

INSERT_ELEMENT_3:

$$\begin{aligned}
&\forall co_1, co_2, r : vector, i, j : int.[insert(co_1, i, co_2, r), element(co_1, j)] \\
&\quad insert(co_1, i, co_2, r) \approx \mathtt{t} \rightarrow i \leq j \leq length(co_1) + index_type__zero \rightarrow \\
&\qquad element(co_1, j) \approx element(r, j + length(co_2)) \\
&\forall co_1, co_2, r : vector, i, j : int.[insert(co_1, i, co_2, r), element(r, j)] \\
&\quad insert(co_1, i, co_2, r) \approx \mathtt{t} \rightarrow i + length(co_2) \leq j \leq length(r) + index_type__zero \rightarrow \\
&\qquad element(co_1, j - length(co_2)) \approx element(r, j)
\end{aligned}$$

INSERT_CURSORS:

$$\begin{aligned}
&\forall co_1, co_2, r : vector, i, j : int.[insert(co_1, i, co_2, r), to_cursor(co_1, j)] \\
&\quad insert(co_1, i, co_2, r) \approx \mathtt{t} \rightarrow index_type__zero < j < i \rightarrow \\
&\qquad to_cursor(co_1, j) \approx to_cursor(r, j) \\
&\forall co_1, co_2, r : vector, i, j : int.[insert(co_1, i, co_2, r), to_cursor(r, j)] \\
&\quad insert(co_1, i, co_2, r) \approx \mathtt{t} \rightarrow index_type__zero < j < i \rightarrow \\
&\qquad to_cursor(co_1, j) \approx to_cursor(r, j)
\end{aligned}$$

SWAP_RANGE:

$$\begin{aligned}
&\forall co_1, co_2 : vector, i_1, i_2 : int.[swap(co_1, i_1, i_2, co_2)] swap(co_1, i_1, i_2, co_2) \approx \mathtt{t} \rightarrow \\
&\quad index_type__zero < i_1 \leq length(co_1) + index_type__zero \wedge \\
&\quad index_type__zero < i_2 \leq length(co_1) + index_type__zero
\end{aligned}$$

SWAP_LENGTH:

$$\begin{aligned}
&\forall co_1, co_2 : vector, i_1, i_2 : int.[swap(co_1, i_1, i_2, co_2)] \\
&\quad swap(co_1, i_1, i_2, co_2) \approx \mathtt{t} \rightarrow length(co_1) \approx length(co_2)
\end{aligned}$$

SWAP_CAPACITY:

$$\forall co_1, co_2 : vector, i_1, i_2 : int.[swap(co_1, i_1, i_2, co_2)]$$
$$swap(co_1, i_1, i_2, co_2) \approx \mathtt{t} \rightarrow capacity(co_1) \approx capacity(co_2)$$

SWAP_ELEMENT_1:

$$\forall co_1, co_2 : vector, i_1, i_2 : int.[swap(co_1, i_1, i_2, co_2)]\, swap(co_1, i_1, i_2, co_2) \approx \mathtt{t} \rightarrow$$
$$element(co_2, i_1) \approx element(co_1, i_2) \wedge element(co_2, i_2) \approx element(co_1, i_1)$$

SWAP_ELEMENT_2 :

$$\forall co_1, co_2 : vector, i_1, i_2, j : int.[swap(co_1, i_1, i_2, co_2), element(co_1, j)]$$
$$swap(co_1, i_1, i_2, co_2) \approx \mathtt{t} \rightarrow$$
$$index\_type\_\_zero < j \leq length(co_1) + index\_type\_\_zero \wedge i_1 \not\approx j \wedge i_2 \not\approx j \rightarrow$$
$$element(co_1, j) \approx element(co_2, j)$$
$$\forall co_1, co_2 : vector, i_1, i_2, j : int.[swap(co_1, i_1, i_2, co_2), element(co_2, j)]$$
$$swap(co_1, i_1, i_2, co_2) \approx \mathtt{t} \rightarrow$$
$$index\_type\_\_zero < j \leq length(co_1) + index\_type\_\_zero \wedge i_1 \not\approx j \wedge i_2 \not\approx j \rightarrow$$
$$element(co_1, j) \approx element(co_2, j)$$

SWAP_CURSORS:

$$\forall co_1, co_2 : vector, i_1, i_2, j : int.[swap(co_1, i_1, i_2, co_2), to\_cursor(co_1, j)]$$
$$swap(co_1, i_1, i_2, co_2) \approx \mathtt{t} \rightarrow$$
$$index\_type\_\_zero < j \leq length(co_1) + index\_type\_\_zero \rightarrow$$
$$to\_cursor(co_1, j) \approx to\_cursor(co_2, j)$$
$$\forall co_1, co_2 : vector, i_1, i_2, j : int.[swap(co_1, i_1, i_2, co_2), to\_cursor(co_2, j)]$$
$$swap(co_1, i_1, i_2, co_2) \approx \mathtt{t} \rightarrow$$
$$index\_type\_\_zero < j \leq length(co_1) + index\_type\_\_zero \rightarrow$$
$$to\_cursor(co_1, j) \approx to\_cursor(co_2, j)$$

DELETE_RANGE:

$$\forall co_1, co_2 : vector, i, l : int.[delete(co_1, i, l, co_2)]$$
$$delete(co_1, i, l, co_2) \approx \mathtt{t} \rightarrow index\_type\_\_zero < i \leq length(co_1) + index\_type\_\_zero$$

DELETE_LENGTH:

$$\forall co_1, co_2 : vector, i, l : int.[delete(co_1, i, l, co_2)]$$
$$delete(co_1, i, l, co_2) \approx \mathtt{t} \rightarrow length(co_1) \approx length(co_2) + l$$

DELETE_CAPACITY:

$$\forall co_1, co_2 : vector, i, l : int.[delete(co_1, i, l, co_2)]$$
$$delete(co_1, i, l, co_2) \approx \mathtt{t} \rightarrow capacity(co_1) \approx capacity(co_2)$$

DELETE_ELEMENT_1:

$$\forall co_1, co_2 : vector, i, l, j : int.[delete(co_1, i, l, co_2), element(co_1, j)]$$
$$delete(co_1, i, l, co_2) \approx \mathtt{t} \rightarrow index\_type\_\_zero < j < i \rightarrow$$
$$element(co_1, j) \approx element(co_2, j)$$
$$\forall co_1, co_2 : vector, i, l, j : int.[delete(co_1, i, l, co_2), element(co_2, j)]$$
$$delete(co_1, i, l, co_2) \approx \mathtt{t} \rightarrow index\_type\_\_zero < j < i \rightarrow$$
$$element(co_1, j) \approx element(co_2, j)$$

DELETE_ELEMENT_2 :

$$\forall co_1, co_2 : vector, i, l, j : int.[delete(co_1, i, l, co_2), element(co_1, j)]$$
$$delete(co_1, i, l, co_2) \approx \mathtt{t} \rightarrow i + l \leq j \leq length(co_1) + index\_type\_\_zero \rightarrow$$
$$\langle i + l \rangle\, element(co_1, j) \approx element(co_2, j - l)$$
$$\forall co_1, co_2 : vector, i, l, j : int.[delete(co_1, i, l, co_2), element(co_2, j)]$$
$$delete(co_1, i, l, co_2) \approx \mathtt{t} \rightarrow i \leq j \leq length(co_2) + index\_type\_\_zero \rightarrow$$
$$element(co_1, j + l) \approx element(co_2, j)$$

DELETE_CURSORS:

$$\forall co_1, co_2 : vector, i, l, j : int.[delete(co_1, i, l, co_2), to\_cursor(co_1, j)]$$
$$delete(co_1, i, l, co_2) \approx \mathtt{t} \rightarrow index\_type\_\_zero < j < i \rightarrow$$
$$to\_cursor(co_1, j) \approx to\_cursor(co_2, j)$$
$$\forall co_1, co_2 : vector, i, l, j : int.[delete(co_1, i, l, co_2), to\_cursor(co_2, j)]$$
$$delete(co_1, i, l, co_2) \approx \mathtt{t} \rightarrow index\_type\_\_zero < j < i \rightarrow$$
$$to\_cursor(co_1, j) \approx to\_cursor(co_2, j)$$

DELETE_END_RANGE:

$$\forall co_1, co_2 : vector, i : int.[delete\_end(co_1, i, co_2)]$$
$$delete\_end(co_1, i, co_2) \approx \mathtt{t} \rightarrow index\_type\_\_zero < i \leq length(co_1) + index\_type\_\_zero$$

DELETE_END_LENGTH:

$$\forall co_1, co_2 : vector, i : int.[delete\_end(co_1, i, co_2)]$$
$$delete\_end(co_1, i, co_2) \approx \mathtt{t} \rightarrow length(co_2) \approx i - index\_type\_\_first$$

DELETE_END_CAPACITY:

$$\forall co_1, co_2 : vector, i : int.[delete\_end(co_1, i, co_2)]$$
$$delete\_end(co_1, i, co_2) \approx \mathtt{t} \rightarrow capacity(co_1) \approx capacity(co_2)$$

DELETE_END_ELEMENT:

$$\forall co_1, co_2 : vector, ij : int.[delete\_end(co_1, i, co_2), element(co_1, j)]$$
$$delete\_end(co_1, i, co_2) \approx \mathtt{t} \rightarrow index\_type\_\_zero < j < i \rightarrow element(co_1, j) \approx element(co_2, j)$$
$$\forall co_1, co_2 : vector, ij : int.[delete\_end(co_1, i, co_2), element(co_2, j)]$$
$$delete\_end(co_1, i, co_2) \approx \mathtt{t} \rightarrow index\_type\_\_zero < j < i \rightarrow element(co_1, j) \approx element(co_2, j)$$

DELETE_END_CURS:

$$\begin{aligned}
&\forall co_1, co_2 : vector, i\,j : int.[delete_end(co_1, i, co_2), to_cursor(co_2, j)] \\
&\quad delete_end(co_1, i, co_2) \approx \mathtt{t} \rightarrow index_type__zero < j < i \rightarrow \\
&\qquad to_cursor(co_1, j) \approx to_cursor(co_2, j) \\
&\forall co_1, co_2 : vector, i\,j : int.[delete_end(co_1, i, co_2), to_cursor(co_1, j)] \\
&\quad delete_end(co_1, i, co_2) \approx \mathtt{t} \rightarrow index_type__zero < j < i \rightarrow \\
&\qquad to_cursor(co_1, j) \approx to_cursor(co_2, j)
\end{aligned}$$

REVERSE_ELEMENTS_LENGTH:

$$\begin{aligned}
&\forall co_1, co_2 : vector.[reverse_elements(co_1, co_2)] \\
&\quad reverse_elements(co_1, co_2) \approx \mathtt{t} \rightarrow length(co_1) \approx length(co_2)
\end{aligned}$$

REVERSE_ELEMENTS_CAPACITY:

$$\begin{aligned}
&\forall co_1, co_2 : vector.[reverse_elements(co_1, co_2)] \\
&\quad reverse_elements(co_1, co_2) \approx \mathtt{t} \rightarrow capacity(co_1) \approx capacity(co_2)
\end{aligned}$$

REVERSE_ELEMENTS_ELEMENT:

$$\begin{aligned}
&\forall co_1, co_2 : vector, i : int.[reverse_elements(co_1, co_2), element(co_1, i)] \\
&\quad reverse_elements(co_1, co_2) \approx \mathtt{t} \rightarrow index_type__zero < i \leq length(co_1) + index_type__zero \rightarrow \\
&\qquad element(co_2, length(co_1) - i + index_type__first) \approx element(co_1, i) \\
&\forall co_1, co_2, r : vector, i : int.[reverse_elements(co_1, co_2), element(co_2, i)] \\
&\quad reverse_elements(co_1, co_2) \approx \mathtt{t} \rightarrow index_type__zero < i \leq length(co_1) + index_type__zero \rightarrow \\
&\qquad element(co_2, i) \approx element(co_1, length(co_1) - i + index_type__first)
\end{aligned}$$

EQUAL_VECTORS__DEF:

$$\begin{aligned}
&\forall co_1, co_2 : vector.[equal_vectors(co_1, co_2)] \\
&\quad equal_vectors(co_1, co_2) \approx \mathtt{t} \rightarrow (length(co_1) \approx length(co_2) \wedge \\
&\qquad (\forall i : int.[element(co_1, i)] \\
&\qquad\quad index_type__zero < i \leq length(co_1) + index_type__zero \rightarrow \\
&\qquad\qquad element(co_1, i) \approx element(co_2, i)) \wedge \\
&\qquad (\forall i : int.[element(co_2, i)] \\
&\qquad\quad index_type__zero < i \leq length(co_1) + index_type__zero \rightarrow \\
&\qquad\qquad element(co_1, i) \approx element(co_2, i)) \wedge \\
&\qquad (\forall i : int.[to_cursor(co_1, i)]\, to_cursor(co_1, i) \approx to_cursor(co_2, i)) \wedge \\
&\qquad (\forall i : int.[to_cursor(co_2, i)]\, to_cursor(co_1, i) \approx to_cursor(co_2, i)))
\end{aligned}$$

COPY__DEF:

$$\begin{aligned}
&\forall co : vector, cap : int.[copy(co, cap)]\, equal_vectors(co, copy(co, cap)) \approx \mathtt{t} \wedge \\
&\quad (cap \approx 0 -> capacity(co) \approx capacity(copy(co, cap))) \wedge \\
&\quad (length(co) \leq cap \leq index_type__last - index_type__zero \rightarrow \\
&\qquad capacity(copy(co, cap)) \approx cap)
\end{aligned}$$

CONCAT_LENGTH:

$$\forall co_1, co_2, r : vector.[concat(co_1, co_2, r)]$$
$$concat(co_1, co_2, r) \approx \mathtt{t} \rightarrow length(r) \approx length(co_1) + length(co_2)$$

CONCAT_ELEMENT_1:

$$\forall co_1, co_2, r : vector, i : int.[concat(co_1, co_2, r), element(r, i)]$$
$$concat(co_1, co_2, r) \approx \mathtt{t} \rightarrow index\_type\_\_zero < i \leq length(co_1) + index\_type\_\_zero \rightarrow$$
$$element(r, i) \approx element(co_1, i)$$
$$\forall co_1, co_2, r : vector, i : int.[concat(co_1, co_2, r), element(co_1, i)]$$
$$concat(co_1, co_2, r) \approx \mathtt{t} \rightarrow index\_type\_\_zero < i \leq length(co_1) + index\_type\_\_zero \rightarrow$$
$$element(r, i) \approx element(co_1, i)$$

CONCAT_ELEMENT_2:

$$\forall co_1, co_2, r : vector, i : int.[concat(co_1, co_2, r), element(co_2, i)]$$
$$concat(co_1, co_2, r) \approx \mathtt{t} \rightarrow index\_type\_\_zero < i \leq length(co_2) + index\_type\_\_zero \rightarrow$$
$$element(r, length(co_1) + i) \approx element(co_2, i)$$
$$\forall co_1, co_2, r : vector, i : int.[concat(co_1, co_2, r), element(r, i)]$$
$$concat(co_1, co_2, r) \approx \mathtt{t} \rightarrow$$
$$length(co_1) + index\_type\_\_zero < i \leq length(r) + index\_type\_\_zero \rightarrow$$
$$element(r, i) \approx element(co_2, i - length(co_1))$$

CONCAT__1__DEF:

$$\forall co_1, co_2 : vector.[concat\_\_1(co_1, co_2)]$$
$$length(co_1) + length(co_2) \leq index\_type\_\_last - index\_type\_\_zero \rightarrow$$
$$concat(co_1, co_2, concat\_\_1(co_1, co_2))$$

CONCAT__2__DEF:

$$\forall co_1 : vector, e : element\_type.[concat\_\_2(co_1, e)]$$
$$length(co_1) < index\_type\_\_last - index\_type\_\_zero \rightarrow$$
$$\exists co_2 : vector.length(co_2) \approx 1 \wedge element(co_2, index\_type\_\_first) \approx e \wedge$$
$$concat(co_1, co_2, concat\_\_2(co_1, e))$$

CONCAT__3__DEF:

$$\forall co_2 : vector, e : element\_type.[concat\_\_3(e, co_2)]$$
$$length(co_2) < index\_type\_\_last - index\_type\_\_zero \rightarrow$$
$$\exists co_1 : vector.length(co_1) \approx 1 \wedge element(co_1, index\_type\_\_first) \approx e \wedge$$
$$concat(co_1, co_2, concat\_\_3(e, co_2))$$

CONCAT__4__DEF:

$$\forall e_1, e_2 : element\_type.[concat\_\_4(e_1, e_2)]$$
$$2 \leq index\_type\_\_last - index\_type\_\_zero \rightarrow length(concat\_\_4(e_1, e_2)) \approx 2 \wedge$$
$$element(concat\_\_4(e_1, e_2), index\_type\_\_first) \approx e_1 \wedge$$
$$element(concat\_\_4(e_1, e_2), index\_type\_\_first + 1) \approx e_2$$

FIND_RANGE:

$$\forall v : vector, e : element\_type, n : int.[find(v, e, n)]$$
$$n \leq find(v, e, n) \leq length(v) + index\_type\_\_zero \vee$$
$$find(v, e, n) \approx index\_type\_\_zero$$

FIND_NO :

$$\forall v : vector, e : element\_type, n : int.[find(v, e, n)]$$
$$find(v, e, n) \approx index\_type\_\_zero \rightarrow$$
$$\forall i : int.[element(v, i)]$$
$$n \leq i \leq length(v) + index\_type\_\_zero \wedge i > index\_type\_\_zero \rightarrow$$
$$equal\_elements(element(v, i), e) \not\approx \mathtt{t}$$

FIND_EXACT:

$$\forall v : vector, e : element\_type, n : int.[find(v, e, n)]$$
$$find(v, e, n) > index\_type\_\_zero \rightarrow$$
$$equal\_elements(element(v, find(v, e, n)), e) \approx \mathtt{t}$$

FIND_OTHERS:

$$\forall v : vector, e : element\_type, n : int.[find(v, e, n)]$$
$$find(v, e, n) > index\_type\_\_zero \rightarrow$$
$$\forall i : int.[element(v, i)]\ n \leq i < find(v, e, n) \wedge i > index\_type\_\_zero \rightarrow$$
$$equal\_elements(element(v, i), e) \not\approx \mathtt{t}$$

REVERSE_FIND_RANGE:

$$\forall v : vector, e : element\_type, n : int.[reverse\_find(v, e, n)]$$
$$index\_type\_\_zero \leq reverse\_find(v, e, n) \leq n \wedge$$
$$reverse\_find(v, e, n) \leq length(v) + index\_type\_\_zero$$

REVERSE_FIND_NO :

$$\forall v : vector, e : element\_type, n : int.[reverse\_find(v, e, n)]$$
$$reverse\_find(v, e, n) \approx index\_type\_\_zero \rightarrow$$
$$\forall i : int.[element(v, i)]$$
$$index\_type\_\_zero < i \leq n \wedge i \leq length(v) + index\_type\_\_zero \rightarrow$$
$$equal\_elements(element(v, i), e) \not\approx \mathtt{t}$$

REVERSE_FIND_EXACT:

$$\forall v : vector, e : element\_type, n : int.[reverse\_find(v,e,n)]$$
$$reverse\_find(v,e,n) > index\_type\_\_zero \rightarrow$$
$$equal\_elements(element(v, reverse\_find(v,e,n)),e) \approx \mathtt{t}$$

REVERSE_FIND_OTHERS:

$$\forall v : vector, e : element\_type, n : int.[reverse\_find(v,e,n)]$$
$$reverse\_find(v,e,n) > index\_type\_\_zero \rightarrow$$
$$\forall i : int.[element(v,i)]\ find(v,e,n) < i \leq n \wedge i \leq length(v) + index\_type\_\_zero \rightarrow$$
$$equal\_elements(element(v,i),e) \not\approx \mathtt{t}$$

C.2 Tests in SPARK 2014

C.2.1 Two_Way_Sort

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Formal_Vectors;

package Sort is
   pragma SPARK_Mode (On);

   subtype Index is Integer range 0 .. 1_000_000;

   function Boolean_Equal (D1, D2 : Boolean) return Boolean is
     (D1 = D2);

   subtype My_Boolean is Boolean;

   package Index_Vectors is new Ada.Containers.Formal_Vectors
     (Index_Type   => Index,
      Element_Type => My_Boolean,
      "="          => Boolean_Equal);

   use Index_Vectors;

   subtype Arr is Vector;

   procedure Two_Way_Sort (A : in out Arr) with
     Post => (if Length (A) > 0 then
                 (for some K in First_Index (A) .. Last_Index (A) =>
                        Element (A, K) = Element (A, K) and then
                    (for all J in First_Index (A) .. K - 1 =>
                         not Element (A, J))
                  and then (for all J in K+1 .. Last_Index (A) =>
```

```
                        Element (A, J))));
end Sort;

package body Sort is
   pragma SPARK_Mode (On);

   procedure Two_Way_Sort (A : in out Arr) is
      I : Integer;
      J : Integer;
   begin
      if Length (A) = 0 then
         return;
      end if;

      I := First_Index (A);
      J := Last_Index (A);
      while I <= J loop
         pragma Loop_Variant (Decreases => J - I);
         pragma Loop_Invariant
           (I in First_Index (A) .. Last_Index (A)
              and then J in First_Index (A) .. Last_Index (A)
              and then (for all K in First_Index (A) .. I-1 => not Element (A, K))
              and then (for all K in J+1 .. Last_Index (A) => Element (A, K)));
         if not Element (A, I) then
            I := I+1;
         elsif Element (A, J) then
            J := J-1;
         else
            Swap (A, I, J);
            I := I+1;
            J := J-1;
         end if;
      end loop;
   end Two_Way_Sort;
end Sort;
```

C.2.2 N_Queens

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Formal_Vectors;

package Queen is
   pragma SPARK_Mode (On);

   N : constant Positive := 8;
   subtype Index is Positive range 1 .. N;

   function Index_Equal (I1, I2 : Index) return Boolean is
```

```
     (I1 = I2);

  package Index_Vectors is new Ada.Containers.Formal_Vectors
    (Index_Type   => Index,
     Element_Type => Index,
     "="          => Index_Equal);

  use Index_Vectors;

  subtype Board is Vector (Capacity => Count_Type (N));

  function Consistent_Index (B : Board; I1 : Index; I2 : Index)
                        return Boolean is
    (Integer (Length (B)) = N and then
         (Element(B, I1) /= Element(B, I2) and then
          I1 - I2 /= Element(B, I1) - Element(B, I2) and then
          I1 - I2 /= Element(B, I2) - Element(B, I1)));

  function Consistent (B : Board; K : Index) return Boolean is
        (Integer (Length (B)) = N and then
           (for all I in Index'First .. K =>
              (for all J in Index'First .. I - 1 =>
                   (Element(B, I) /= Element(B, J) and then
                    I - J /= Element(B, I) - Element(B, J) and then
                    I - J /= Element(B, J) - Element(B, I)))));

  procedure Add_next (B : in out Board; I : Index; Done : in out Boolean;
                          C : in Board)
  with Pre => Integer (Length (B)) = N and then Integer (Length (C)) = N and then
    ((not Done) and
    (for all J in Index'First .. I => Element(C, J) = Element(B, J)) and
    (if I > 1 then Consistent (B, I - 1))),
       Post => Integer (Length (B)) = N and
    ((if Done then Consistent (B, N) else not Consistent (C, N)) and
       (for all J in Index'First .. I => Element(B, J) = Element(B'Old, J)));

 function Copy_Until (B : in Board; I : Index; C : in Board) return Board
 with Pre => Integer (Length (B)) = N and then Integer (Length (C)) = N,
      Post => Integer (Length (Copy_Until'Result)) = N and
    (for all J in Index'First .. I =>
       Element(Copy_Until'Result, J) = Element(B, J));

  procedure Try_Row (B : in out Board; I : Index; Done : in out Boolean;
                     C : in Board)
    with Pre => Integer (Length (B)) = N and then Integer (Length (C)) = N and then
    ((not Done) and
         (for all J in Index'First .. I-1 => Element(C, J) = Element(B, J)) and
       (if I > 1 then Consistent (B, I - 1))),
         Post => Integer (Length (B)) = N and
```

```
     ((if Done then Consistent (B, N) else not Consistent (C, N)) and
       (for all J in Index'First .. I-1 => Element(B, J) = Element(B'Old, J)));

end Queen;

package body Queen is
  pragma SPARK_Mode (On);

   procedure Add_next (B : in out Board; I : Index;
                       Done : in out Boolean; C : in Board)
   is
   begin
      if Consistent (B, I) then
         if N = I then
            Done := True;
         else
            Try_Row (B, I + 1, Done, C);
         end if;
         return;
      else
         pragma Assert (not Consistent (C, I));
         pragma Assert (not (for all J in I .. N => Consistent (C, J)));
      end if;
   end Add_next;

   function Copy_Until (B : in Board; I : Index; C : in Board) return Board is
      R : Board;
   begin
      Clear (R);
      pragma Assert (Integer (Capacity (R)) = N);
      for J in Index'First .. I loop
         pragma Loop_Invariant
           (Capacity (R) = Capacity (R'Loop_Entry) and then
            Integer (Length (R)) = J - 1 and then
                (for all K in Index'First .. J - 1 =>
                     Element(R, K) = Element(B, K)));
         Append (R, Element(B, J));
      end loop;
      for J in I + 1 .. Index'Last loop
         pragma Loop_Invariant
           (Capacity (R) = Capacity (R'Loop_Entry) and then
            Integer (Length (R)) = J - 1 and then
                (for all K in Index'First .. I =>
                     Element(R, K) = Element(B, K)));
         Append (R, Element(C, J));
      end loop;
      return R;
   end Copy_Until;
```

```
   procedure Try_Row (B : in out Board; I : Index; Done : in out Boolean;
                      C : in Board)
   is
   begin
      for R in Index'Range loop
         pragma Loop_Invariant
           (Length (B) = Length (B'Loop_Entry) and then
                (not Done and
                     (for all J in 1 .. I - 1 =>
                            Element(B, J) = Element(B'Loop_Entry, J)) and
                   (if Element(C, I) < R then
                        not Consistent (C, N))));
         Replace_Element (B, I, R);
         if Element(C, I) = R then
            Add_next (B, I, Done, C);
         else
            Add_next (B, I, Done, Copy_Until (B, I, C));
         end if;
         if Done then
            exit;
         end if;
      end loop;
   end Try_Row;

end Queen;
```

C.2.3 Ring_Buffer

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Formal_Vectors;

package Ring_Buf is
   pragma SPARK_Mode (On);

   Buf_Size : constant := 10000;

   subtype Ar_Index is Integer range 0 .. Buf_Size - 1;
   subtype Length_Type is Integer range 0 .. Buf_Size;

   function Eq (I1 : Integer; I2 : Integer) return Boolean is
     (I1 = I2);

   package My_Vectors is new Ada.Containers.Formal_Vectors
     (Ar_Index, Integer, Eq);
   use My_Vectors;

   subtype Buf_Array is Vector (Buf_Size);
```

```
   type Ring_Buffer is record
      Data   : Buf_Array;
      First  : Ar_Index;
      Length : Length_Type;
   end record;

   function Inv (R : Ring_Buffer) return Boolean is
     (Integer (Length (R.Data)) = Buf_Size);

   function Is_Full (R : Ring_Buffer) return Boolean is (R.Length = Buf_Size);

   function Is_Empty (R : Ring_Buffer) return Boolean is (R.Length = 0);

   package P_Model is

      function Is_Model (R : in Ring_Buffer; M : in Buf_Array) return Boolean
        with Pre  => Inv (R);

   end P_Model;

   use P_Model;

   function Model (R : Ring_Buffer) return Buf_Array with
     Pre  => Inv (R),
     Post => Is_Model (R, Model'Result);

   procedure Clear (R : in out Ring_Buffer)
      with Post => (Is_Empty (R));

   function Head (R : in Ring_Buffer) return Integer
     with Pre => Inv (R);

   procedure Push (R : in out Ring_Buffer; X : Integer)
      with Pre => (Inv (R) and then not Is_Full (R)),
     Post => Inv (R) and
     (if Is_Model (R, Model (R)) and Is_Model (R'Old, Model (R'Old)) then
        Model (R) = Model (R'Old) & X);

   procedure Pop (R : in out Ring_Buffer; Top : out Integer)
      with Pre => (Inv (R) and then not Is_Empty (R)),
           Post => Inv (R) and
     (if Is_Model (R, Model (R)) and Is_Model (R'Old, Model (R'Old)) then
        Head (R'Old) & Model (R) = Model (R'Old));

end Ring_Buf;

package body Ring_Buf is
   pragma SPARK_Mode (On);
```

```
function Head (R : in Ring_Buffer) return Integer is
  (Element (R.Data, R.First));

package body P_Model is

   function Is_Model (R : in Ring_Buffer; M : in Buf_Array) return Boolean
   is
     (Length (M) = Count_Type (R.Length) and then
          (if R.Length < Buf_Size - R.First then
               ((for all I in R.First .. R.First + R.Length - 1 =>
                       Element (R.Data, I) = Element (M, I - R.First)) and
                (for all I in 0 .. R.Length - 1 =>
                       Element (R.Data, I + R.First) = Element (M, I)))
           else
             ((for all I in R.First .. Buf_Size - 1 =>
                    Element (R.Data, I) = Element (M, I - R.First)) and
               (for all I in 0 .. R.Length - (Buf_Size - R.First) - 1 =>
                     Element (R.Data, I) =
                    Element (M, I + Buf_Size - R.First)) and
                 (for all I in 0 .. Buf_Size - 1 - R.First =>
                       Element (R.Data, I + R.First) = Element (M, I)) and
               (for all I in Buf_Size - R.First .. R.Length - 1 =>
                     Element (R.Data, I - (Buf_Size - R.First)) =
                    Element (M, I)))));

end P_Model;

use P_Model;

function Model (R : Ring_Buffer) return Buf_Array is
   M  : Buf_Array := Copy (R.Data);
   Mt : Buf_Array := Copy (R.Data);
begin
   if R.Length < Buf_Size - R.First then
      Delete_First (M, Count_Type (R.First));
      Delete_Last (M, Count_Type (Buf_Size - (R.First + R.Length)));
   else
      Delete_First (M, Count_Type (R.First));
      Delete_Last (Mt, Count_Type (2 * Buf_Size - R.First - R.Length));
      Append (M, Mt);
   end if;
   return M;
end;

procedure Clear (R : in out Ring_Buffer) is
begin
   R.Length := 0;
end Clear;
```

```
   procedure Push (R : in out Ring_Buffer; X : Integer)
   is
   begin
      if R.Length < Buf_Size - R.First then
         Replace_Element (R.Data, R.First + R.Length, X);
      else
         Replace_Element (R.Data, R.Length - (Buf_Size - R.First), X);
      end if;
      R.Length := R.Length + 1;
   end Push;

   procedure Pop (R : in out Ring_Buffer; Top : out Integer)
   is
   begin
      Top := Element (R.Data, R.First);
      if R.First < Buf_Size - 1 then
         R.First := (R.First + 1);
      else
         R.First := 0;
      end if;
      R.Length := R.Length - 1;
   end Pop;

end Ring_Buf;
```

C.2.4 Amortized_Queue

```
with Ada.Containers; use Ada.Containers;
with Ada.Containers.Formal_Vectors;

package Amortized_Queue is
   pragma SPARK_Mode (On);

   subtype Index is Integer range 1 .. 1_000;

   subtype Val is Integer range -2 ** 31 .. 2 ** 31 - 1;

   function Eq (I1 : Val; I2 : Val) return Boolean is
     (I1 = I2);

   package My_Vectors is new Ada.Containers.Formal_Vectors
     (Index, Val, Eq);
   use My_Vectors;

   Capacity : constant Count_Type := Count_Type (Index'Last);

   type Queue is record
      Front : Vector (Capacity);
```

```
      Rear  : Vector (Capacity);
   end record;

   function Inv (Q : in Queue) return Boolean is
     (Length (Q.Front) >= Length (Q.Rear) and then
      Q.Front.Capacity - Length (Q.Front) >= Length (Q.Rear) and then
      Q.Front.Capacity = Q.Rear.Capacity);

   package P_Model is

      function Is_Model (Q : in Queue; M : in Vector) return Boolean;

   end P_Model;

   use P_Model;

   function Model (Q : in Queue) return Vector with
     Pre  => Inv (Q),
     Post => Is_Model (Q, Model'Result);

   function Front (Q : in Queue) return Val with
     Pre  => Inv (Q) and then Length (Q.Front) > 0,
     Post => (if is_Model (Q, Model (Q)) then
                Last_Element (Model (Q)) = Front'Result);

   function Tail (Q : in Queue) return Queue with
     Pre  => Inv (Q) and then Length (Q.Front) > 0,
     Post => Inv (Tail'Result) and then
     (if is_Model (Q, Model (Q)) and
         is_Model (Tail'Result, Model (Tail'Result)) then
        Model (Q) = Model (Tail'Result) & Last_Element (Model (Q)));

   function Enqueue (Q : in Queue; V : in Val) return Queue with
     Pre  => Inv (Q) and then
     Q.Front.Capacity - Length (Q.Front) > Length (Q.Rear),
     Post => Inv (Enqueue'Result) and then
     (if is_Model (Q, Model (Q)) and
         is_Model (Enqueue'Result, Model (Enqueue'Result)) then
        V & Model (Q) = Model (Enqueue'Result));

end Amortized_Queue;

package body Amortized_Queue is
  pragma SPARK_Mode (On);

   package body P_Model is

      function Is_Model (Q : in Queue; M : in Vector) return Boolean is
        (Length (Q.Front) + Length (Q.Rear) = Length (M) and then
```

```
          (for all I in Index range 1 .. Integer (Length (Q.Rear)) =>
                Element (M, I) =
                 Element (Q.Rear, Integer (Length (Q.Rear)) - I + 1)) and then
        (for all I in Index range 1 .. Integer (Length (Q.Front)) =>
              Element (M, I + Integer (Length (Q.Rear))) =
             Element (Q.Front, I)) and then
          (for all I in Index range 1 .. Integer (Length (Q.Rear)) =>
                Element (M, Integer (Length (Q.Rear)) - I + 1) =
                 Element (Q.Rear, I)) and then
        (for all I in Index range
             Integer (Length (Q.Rear)) + 1 .. Integer (Length (M)) =>
              Element (M, I) =
             Element (Q.Front, I - Integer (Length (Q.Rear)))));

end P_Model;

function Model (Q : in Queue) return Vector is
   RevRear : Vector := Copy (Q.Rear);
begin
   Reverse_Elements (RevRear);
   return RevRear & Q.Front;
end Model;

function Tail (Q : Queue) return Queue is
   Front : Vector := Copy (Q.Front);
   Rear : Vector := Copy (Q.Rear);
begin
   Delete_Last (Front);
   if Length (Front) < Length (Rear) then
      Reverse_Elements (Rear);
      Insert (Front, 1, Rear);
      Clear (Rear);
   end if;
   return Queue'(Front => Front, Rear => Rear);
end Tail;

function Enqueue (Q : in Queue; V : in Val) return Queue is
   Front : Vector := Copy (Q.Front);
   Rear : Vector := Copy (Q.Rear);
begin
   Append (Rear, V);
   if Length (Front) < Length (Rear) then
      Reverse_Elements (Rear);
      Insert (Front, 1, Rear);
      Clear (Rear);
   end if;
   return Queue'(Front => Front, Rear => Rear);
end Enqueue;
```

```
   function Front (Q : Queue) return Val is
   begin
      return Last_Element (Q.Front);
   end Front;

end Amortized_Queue;
```