Thèse de Doctorat de L'université Sorbonne Paris Cité
Préparée à L'université Paris Diderot
Ecole Doctorale 386 — Sciences Mathématiques de Paris Centre
LSV / Deducteam

Integrating Rewriting, Tableau and Superposition into SMT

par

Guillaume Bury

Thèse de Doctorat d'Informatique

Dirigée par Gilles Dowek & David Delahaye

Présentée et soutenue publiquement à Cachan le 8 janvier 2019

Delia Kesner	Professeur, Université Paris Diderot	Présidente du Jury
Pascal Fontaine	Maître de Conférences (HDR), Université de Lorraine	Rapporteur
Philip Rümmer	Associate Professor (Docent), Uppsala University	Rapporteur
Régine Laleau	Professeur, Université Paris-Est Créteil	Examinatrice
Damien Doligez	Chargé de Recherche, Inria	Examinateur
Mohamed Iguernlala	Docteur, Ingénieur R&D OCamlPro	Examinateur
Gilles Dowek	Directeur de Recherche, Inria	Directeur de thèse
David Delahaye	Professeur, Université de Montpellier	Co-Directeur de thèse

Abstract

Cette thèse doctorale présente ArchSAT, un théorème prouveur capable de générer des preuves formelles, qui est utilisé pour étudier l'intégration à l'algorithme SMT de techniques de raisonnements dits "du premier ordre". ArchSAT intègre la réecriture grâce à une théorie SMT standard, qui permet d'accélérer la vitesse du raisonnement sur les problèmes dont certains axiomes peuvent être vus comme des règles de réécriture. De plus, une extension de cette théorie adaptée à l'algorithme McSAT (plutôt que SMT), permet aussi de gérer les règles de réécriture conditionnelles. ArchSAT intègres aussi la méthode des tableaux au travers d'une théorie SMT traditionnelle, afin de raisonner de manière générique sur tout le premier ordre, ce qui permet de remplacer la transformation en forme normal conjonctive et le mécanisme des triggers habituellement utilisés dans les prouveurs SMT. Cette théorie SMT pour la méthode des tableaux utilise par ailleurs une variante de la superposition afin d'unifier des termes modulo égalités et règles de réécriture. Finalement, ArchSAT est capable de génerer des preuves formelles à la fois pour l'assistant de preuve Coq, et le framework logique dedukti, ce qui permet d'assurer la correction des résultats.

Mots clés— déduction automatique, SMT, réécriture, superposition, tableau, preuve formelle, logique du 1er ordre, unification

Abstract

This PhD thesis presents ArchSAT, an automated theorem prover with formal proof outputs, which is used to study the integration of some first-order reasoning methods into SMT solvers. ArchSAT integrates the notion of rewriting as a regular SMT theory, which allows us to speed up reasoning on problems whose axioms can be turned into rewrite rules. Additionally, an extension of the rewriting theory for the underlying McSat architecture enables ArchSAT to consider conditional rewrite systems as well. ArchSAT also integrates a tableau method presented as a SMT theory able to handle generic first-order reasoning, replacing the conjunctive normal form transformation and trigger mechanism traditionally used in SMT solvers. This tableau theory uses a variant of superposition in order to perform unification modulo equalities and modulo rewrite rules. Finally, ArchSAT is able to generate formal proofs for the Coq proof assistant and the Dedukti logical framework, ensuring the correctness of its results.

Keywords— automated deduction, SMT, rewriting, superposition, tableau, formal proof, first-order logic, unification

Integrating Rewriting, Tableau and Superposition into SMT

Guillaume Bury

8 january 2019

Contents

Acknowledgments

I'd like to thank David & Gilles, for all the work they did motivating me and pushing me forward. I know I haven't been the easiest PhD student, and I'm glad you both kept up with me, even when I lacked the motivation. I'd also like to thank all the people of Deducteam, who only very rarely mentionned how often they didn't see me at the laboratory, particularly Simon, Pierre, Francois, Gaspard, Guillaume, and Rodolphe.

I'd like to thank my friends, most notably[1] the following unordered set of people[2]. Eliot, Rémy and Nathanaël for all those times we talked and played. Simon, with whom I talked about my thesis probably more than with David & Gilles. Mathilde who became friends with me faster than I would have ever imagined. Tito, who was a great flatmate for the two years and some he was in the appartment with me. Guillaume & Léa, who have always been a great help, whether it be about dance as well as other things[3]. All the other people in Team Rocket who brought me fun and distraction: Caroline, Céline, Héloise, Martin, Mathilde, Nathanaël, Pierre-Alexandre, Rares, Thibault, Antonia, Claire, Louise, Ludovic, Ninon, Paul, Sandra, Théo, and Vincent. And all the people who danced with me, and who I occasionally annoyed by rambling about my thesis when they made the mistake of asking what I was working on: Natacha, Auxanne, Juliette, Juliette[4], Marie-Lou, Elsa, Justine, Hélène, Antonia, Romain, Béa, Mélody, and all the others I'm forgetting, or simply whose name I've forgotten[5]. You all provided me the support I very much needed. I don't think I would have ever had the strength to complete this thesis without you.

I'd also like to thank all the people at OCamlPro who all helped me survive through the last stretch, particularly through all the administrative files I've had to complete.

Finally, my family: my Dad, my Mom, my brother and sisters, and my Grand-mother, some of whom were sometimes sceptic about my work, but who always helped me whenever I asked, I thank you.

[1] Though not only those people, of course, I'm rpetty much certain to forget at least one person.

[2] You have each helped me in different ways that can't, and shouldn't be ranked, hence the unordered set.

[3] Special note to Guillaume, who's also trying to finish his thesis, and to who I wish a happy end of thesis.

[4] Yup, no last names, duplicates are voluntary, :D

[5] It happens way too often, unfortunately.

Introduction

Automated theorem proving is the study of programs capable of automatically proving mathematical formulas, a.k.a. automated theorem provers. Its most prominent field of application is the verification of software and hardware: such a verification process typically translates a specification and a piece of software or hardware into an equivalent mathematical formula. Proving the formula proves that the software or hardware verifies its specification. As expected, this process generates a formula whose size depends on the size of the software or hardware considered. Modern software or hardware may generate very large formulas, regularly containing more than hundreds of formulas. Proving such formulas therefore needs to be automated. This is one of the tasks that is assigned to automated theorem provers. In order to prove the correctness of the software or hardware, any formula that is not proved by an automated theorem prover needs to be proved by hand, which is a complex and time-consuming process that need to be repeated for each update to the software or hardware. Therefore, the more theorems a prover is capable of proving, and the faster it does so, the less humans have to do manually, which speeds up the overall process of verifying software and hardware[1].

This thesis is about improving automated theorem proving, through two distinct goals. The first goal is to increase the number of theorem proved, so as to provide even further automatization of processes such as verification of software. This is achieved by combining and adapting multiple existing techniques into one automated theorem prover. The second goal is to increase the confidence in those proved theorems, in order to provide increased reliability on the results of automated theorem provers: as the usage of automated theorem provers grows, so does the need to ascertain their results are correct.

In the following, I introduce the various existing theorem proving techniques used or mentioned throughout in this thesis. Then, I present how these theories are combined. Finally, I talk about the independent verification of automated theorem provers results, and how the results generated by my theorem prover in particular can be independently verified.

Reasoning modulo theories

In this thesis, I will focus on automated theorem provers that target formulas expressed in first-order logic, because it is the most widely used logic to specify problems in practice. Additionally, first-order problems will often use specific theories, such as arithmetic on integers or floating point values, particularly for problems coming from verification of software.

Reasoning modulo theories has thus become standard among the automated deduction community, as can be seen in the various competitions for automated theorem provers [12,84]. Therefore, most of the current automated theorem provers include features to reason modulo some theories. These features can be divided into two main categories : specific and generic. The generic approaches aim at specifying theories through axioms, which are included in a problem, and handling these axioms using a generic algorithm[2], whereas the specific approach relies on having hand-crafted algorithms (typically decision procedures) for each theory to be considered.

[1] Note that it is not the only use of automated theorem proving. Actually, automated theorem provers can be of help to mathematicians when elaborating formal proofs of theorems, by automatizing small steps of reasoning.

[2] The algorithm is generic in the sense that it accepts a wide variety of axioms that can change from one problem to another.

The specific approach is most typically examplified by SMT[1] solvers, such as z3 [41], yices2 [51], VeriT [25], CVC4 [10], and Alt-Ergo [21]. SMT solvers are made up of a core SAT solver, which is extended to cover many different theories by adding decision procedures together with a mechanism for combining these decision procedures. Though the theories can be combined, they are fixed once a solver is released: each theory supported by the solver needs a manually written decision procedure in the source code of the SMT solver. In these cases, the problem statements do not need to include any theory axiomatization, because the solver has an internal representation of the theory. This internal representation of the theory allows for very efficient theory-specific reasoning, enabling SMT solvers to be very fast on problems that only have ground formulas[2]. In the cases where some axioms are not part of their builtin theories, SMT solvers typically use instantiation patterns, or triggers, which are, in general, an incomplete strategy, except in some specific cases [48–50].

On the other hand, first-order theorem provers tend to have more generic and complete approaches, where axioms are explicitly part of the problem, and where the prover performs some reasoning that is generic with respect to these axioms. This is the approach mostly used in first-order theorem provers such as Vampire [62], Zipperposition [38], and E-prover [80], which use superposition, Zenon [24] and Princess [77] which use the tableau method, and iProver [60], which uses the Inst-Gen instantiation framework[3]. While each first-order technique is different, they are generally complete, that is, if a proof exists, the algorithm will find it in a finite amount of time, but may loop indefinitely on problems with no proof[4]. Even though theoretically, any theory can be handled using the generic mechanisms of these provers, most of them still have a specific handling of some theories such as equality, uninterpreted functions and arithmetic. In fact, a lot of work has been done in integrating reasoning about specific theories in first-order automated theorem provers, ranging from integration of equality in tableau provers [16, 46], arithmetic in tableau provers [26, 33] and superposition provers [38]. These extensions are quite intricate, in that they involve modifications of the base algorithm considered, and thus typically result in new algorithms whose soundness and completeness must be proved again.

Thus extending existing first-order provers to specific theories is a complex and difficult task, which explains why these provers tend to have much less built-in theories than SMT solvers. This limitation is one of the motivations of deduction modulo theory [47]. Deduction modulo theory is a generic approach to better handle axioms, replacing them by computation rules (i.e. rewrite rules), when possible. The idea behind deduction modulo theory is then to turn axioms into rewrite rules whenever possible, and make use of the fact that rewrite rules are oriented in order to guide the proof search. In practice, this is done by normalizing terms with regards to the rewrite system induced by the rewrite rules. This effectively allows provers to replace deduction steps with computation steps, which is much more efficient since it eliminates choice points in the proof search. Deduction modulo theory has been integrated into the first-order automated theorem provers iProver [28], Zenon [46], and Zipperposition [32], where increased performances of these automated theorem provers have been observed.

Equational Reasoning

The theory of equality and uninterpreted functions is so pervasive that it has specific reasoning in almost all automated theorem provers. SMT provers usually rely on congruence closure algorithms [41, 69] to perform reasoning on both ground equalities and uninterpreted function symbols involved in problems, and use triggers (among other techniques) with matching modulo equalities [40] in order to reason about quantified formulas modulo equalities. Additionally, theory combination frameworks for SMT solvers handle the task of ensuring equalities are shared among

[1] SMT: Satisfiability Modulo Theories.

[2] The theory axiomatization is implicit in these problems. Hence the quantified formulas that axiomatize the theory are also implicit.

[3] The Inst-Gen framework is similar to resolution, but when two clauses (or rather some of their literals) are matched, instead of resolving the two clauses, it generates new instantiated forms of the input clauses using the matching substitution.

[4] Note that it is theoretically impossible to have a correct, complete and terminating decision procedure for first-order logic.

theories. First-order theorem provers, on the other hand, have different techniques to handle equalities and uninterpreted functions and predicates. Resolution has been extended in order to handle equalities and uninterpreted functions better by integrating them into their calculi, which resulted in techniques such as paramodulation [70], and superposition [38, 78]. Extensive work on better equality handling in tableau provers [14, 15, 74] has also been realized.

One of the main challenges of automated theorem proving is to properly use equalities when deciding how to instantiate quantified formulas: equalities discovered during proof search, and particularly equalities induced by congruence over uninterpreted functions, extend the set of terms that must be considered when instantiating quantified formulas. SMT solvers usually handle this by performing e-matching through their trigger mechanism [40,66], that is matching of terms modulo some equalities. As mentioned above, this strategy is in general incomplete. A complete strategy for solving first-order problems actually requires to perform simultaneous rigid e-unification[1], that is, the simultaneous unification of a list of pair of terms modulo some equalities where each term contains rigid variables (i.e. variables that must be instantiated at most once). This problem is in general undecidable [43, 44]. Therefore, restrictions of this problem are often considered and used in theorem provers. For instance, the tableau theorem prover Princess performs a variant of simultaneous rigid e-unification where each variable's bound term belongs to a finite set [6, 7], and other theorem provers use non-simulaneous rigid e-unification, which motivates the work on finding efficient algorithms to solve this problem [53–56].

Rewriting is an adequate theory to reason with equalities: rewrite rules on terms can be seen as quantified equalities. Procedures such as Knuth-Bendix completion [57,59], or narrowing [5,75], together with term normalization algorithms actually provide a way to unify terms modulo a set of rewrite rules[2]. Going further, paramodulation and superposition can be seen as extensions of these procedures to handle clauses instead of single equalities. ArchSAT notably uses rigid unit superposition, a variant of superposition, in order to perform unification modulo equalities and rewrite rules.

ArchSAT

As exemplified by superposition and paramodulation, combinations of theorem proving techniques happen frequently, and have obtained great results. As mentioned earlier, decution modulo theory has been integrated into the tableau theorem prover Zenon [46], and together with native handling of polymorphism, it has been shown to greatly enhance the capabilities of Zenon [31,46]. Deduction modulo theory has also been integrated in Zipperposition [32], and into the resolution-based theorem prover iProver, resulting in the prover called iProver Modulo, which has been proved to significantly increase performances [28]

The work presented in this thesis is an attempt at integrating some of the techniques used by first-order theorem provers, namely rewriting, tableau, and superposition, into SMT solvers. Starting from an SMT solver, which has good performances on ground problems, integrating first-order techniques aims at bridging the gap and asymmetry of performances between first-order theorem provers. While the tableau method and superposition are standalone algorithms, which have been used as the core of automated theorem provers, rewriting is a more transversal technique, which has been integrated into other theorem provers with much success. This work on integrating first-order techniques into an SMT prover has been implemented in a new theorem prover called ArchSAT, which I developed from scratch specifically for the purposes of this thesis. In the following, we will give an overwiew of how that integration has been done, and how it relates to previous work.

Integration of tableau theory into the SMT architecture of ArchSAT replaces two processing steps of regular SMT solving : the transformation into clausal normal form, and the trigger mechanism. While regular SMT solvers need to pre-process input problems in order to transform them into sets of clauses, the tableau theory in ArchSAT integrates this transformation into the

[1] This unification can be performed in many various ways. For instance, superposition does it incrementally while resolving clauses. The e-unification is not explicit and distinct but rather interleaved with the proof search.

[2] Unification modulo rewrite rules is often characterized as solving equations or systems of equations in the literature.

proof search algorithm, lazily unfolding logical connectives as needed. This is similar to what has been done within the Alt-Ergo solver. ArchSAT also replaces the generally incomplete handling of quantified formulas using triggers in SMT solvers by a built-in strategy using meta-variables, following the tableau method. Meta-variables are introduced for each quantified formula in the same space as other ground terms[1], and once a model is found by the solver, terms are unified in order to find substitutions for the meta-variables. These substitutions are then used to instantiate the quantified formulas. This is actually quite similar to the Inst-Gen framework mentioned above, the main difference being that Inst-Gen considers a saturated set of clauses and simply unifies (without using equalities nor rewrite rules), whereas ArchSAT considers a first-order model of the input clauses and unifies modulo equalities and rewrite rules, using a variant of superposition as described later. Though not proved, this strategy is believed to be complete, which is significant since currently, most of the approaches to handle quantified formulas in SMT solvers are not meant to be complete.

ArchSAT also distinguishes itself from more traditional SMT solvers when dealing with equalities, by having two distinct theories: one for equality, and one for uninterpreted functions and predicates. This is possible because ArchSAT uses an extensions of the SMT algorithm called McSat, which builds a first-order model rather than a propositional model consistent with a theory as the SMT algorithm does. The McSat algorithm uses a notion of assignment of terms to values, which allows theories to exchange information, replacing the theory combination mechanism and allowing ArchSAT to separate the reasoning about equalities from the reasoning about uninterpreted functions and predicates.

Previous work on combining rewriting and SMT solving include the use of rewriting, or rather normalization, inside theories, as a way to extend the internal notion congruence used in a theory's implementation [27], as well as on the use of SMT solvers inside a rewriting engine [76]. Rewriting in ArchSAT is a general mechanism used in order to speed up reasoning when the input problem contains quantified formulas that can be turned into rewrite rules. ArchSAT interprets these quantified formulas as rewrite rules, and uses them to build a rewrite system. This rewrite system then allows ArchSAT to normalize terms without losing any information, thus helping guide proof search. The difference with the previous work is that the rewrite rules come from the problem, rather than being built in the prover, thus resulting in a more generic approach.

Integration of SAT or SMT algorithms within superposition-based provers has first been introduced within the theorem prover Vampire in [87]. Called AVATAR[2], this techniques vastly supercedes the various[3] splitting algorithms that were used previously. Splitting algorithms in superposition tried to emulate the disjunction rule of semantic tableaux, by partitioning a clause C into two (or more) sub-clauses which share no variables, and then deciding one of the sub-clauses. The sub-clauses which has been decided on then replaces the original clause, and a backtracking system is then implemented in order to undo that choice later in the proof search. AVATAR also partitions clauses, but instead of making a decision locally on one of the sub-clauses, it uses a SAT or SMT solver to compute a propositional model of the sub-clauses, and then performs resolution on the resulting clauses. Both splitting and AVATAR aim at improving performances of the ground reasoning performed by superposition provers. ArchSAT uses a dual approach and integrates superposition into SMT solving. A variant of superposition is used in order to unify terms while taking into account equalities as well as rewrite rules. The results of this unification is then used by the tableau mechanism for meta-variables in order to perform instantiations.

Certificates

In addition to the new approaches for handling quantified formulas in SMT solving, one of the key features of ArchSAT is the ability to generate formal proofs.

As automated theorem provers become ever more complex in order to prove more problems, it is increasingly important to be able to check or certify a prover's results. It is a certitude that all automated theorem provers had bugs, and most probably all automated theorem provers still

[1] Contrary to triggers, which are kept separate from the ground terms in the traditional SMT architecture.

[2] AVATAR: Advanced Vampire Architecture for Theories and Resolution.

[3] [87] mentions 481 variants of splitting implemented in the Vampire prover alone.

have bugs. In most cases, these bugs affect only very specific corner cases, however, it is almost impossible to specify a posteriori which results of a prover may be invalidated by a given bug. The easiest way to verify a prover's results is thus for the prover to output some traces or certificates that can then be checked by an independent program.

Certification of automated theorem prover's result, although not ignored, has been less shared than research about increasing performances of provers. Particularly, though the calculi and algorithms that provers rely on usually provide a way to generate certificates or formal proofs, concrete implementations rarely do offer such capabilities. This is due in part to the technical difficulties encountered in proof generation, which can range from purely technical considerations to differences between the theoretical calculi and the implementation. Some of these difficulties for SMT solvers are discussed in [11].

Even when provers can produce proofs, there are significant differences between what each prover might do. These differences range from the language of the produced proof, to the process needed to effectively check a proof: some provers output traces that then need to be processed in order to reconstruct a fully checkable proof.

Among the provers cited above, only a few of them offer a way to certify their results. CVC4 is able to output proofs [65] in the LFSC[1] format [82], VeriT can generate proofs [8] in its own format (described in [17]), which can then be reconstructed and checked in Coq using SMTCoq [52]. z3 does not produce proofs, but traces [22], which can then be reconstructed into formal proofs checked by the Isabelle proof assistant [71]. Lastly, Zenon can output formal proofs in both Coq [24] and Dedukti [36]. ArchSAT follows Zenon, and is capable of producing formal proofs for both Coq and Dedukti.

Outline

This thesis is divided into 6 chapters, each presenting a different aspect of ArchSAT:

- First, Chapter 1 presents the kind of problems solved by automated theorem provers, describes the SAT, SMT and McSat algorithms which are the core of the ArchSAT theorem prover. It also gives an overview of the ArchSAT theorem prover.

- Chapter 2 explains how ArchSAT is built on top of the McSat algorithm in order to perform generic first-order reasoning. This chapter shows how the tableau method is integrated in ArchSAT in order to reason about logical connectives and quantified formulas. It also explains how the McSat algorithm allows us to separate the reasoning about equalities from the one about uninterpreted functions and predicates.

- Chapter 3 focuses on the various ways ArchSAT integrates rewriting. Rewriting integration in ArchSAT is divided in 3 parts: static, dynamic, and unification. Static rewriting in ArchSAT is an incomplete but efficient algorithm that makes use of axioms as rewrite rules. Dynamic rewriting is a complete mechanism to handle rewrite rules that may become true or false during proof search in a way similar to that of triggers. Lastly, in presence of rewrite rules, unification must be adapted to consider them, and we thus define an algorithm for e-unification modulo rewrite rules using a variant of superposition.

- Chapter 4 explains how ArchSAT generates formal proofs. It explains how the SAT, SMT, and McSat algorithms are able to produce proofs, and details some of the design choices and internal representation of proofs in ArchSAT.

- Chapter 5 presents results of the benchmarks performed to compare ArchSAT and various other automated theorem provers, in order to evaluate the usefulness of the techniques implemented.

- Finally, Chapter 6 discusses some more technical details of the implementation of ArchSAT.

[1] LFSC: Logical Framework with Side Conditions.

Chapter 1

ArchSAT: an McSat prover

This chapter presents ArchSAT, a theorem prover I developped during my thesis in order to study the integration of first-order theorem proving techniques, such as Tableaux and Rewriting, into SMT solvers. ArchSAT's goal is to prove arbitrary first-order formulas, with a focus on the handling of quantified formulas, compared to traditional SMT solvers.

This chapter defines the Satisfiability problem, aka SAT, that theorem provers aim at solving, and presents the SAT and SMT[2] algorithm used respectively for solving the propositional and ground first-order satisfiability problems. It then presents McSat, a relatively recent[3] extension of the SMT algorithm, whose aim is to further merge the first-order reasoning into the core SAT algorithm compared to SMT.

Finally, I introduce ArchSAT, my theorem prover implementation, which is based on McSat.

1.1 SAT Solving

The SAT algorithm is nowadays fairly well established and understood. This section nonetheless aims at presenting once again the SAT problem and standard algorithm, first as an introduction for the non-initiated reader, and also as a preliminary for the more recent McSat algorithm which lacks the standardisation that SAT benefits from.

SAT solvers implementation nowadays include a staggering quantity of optimizations, ranging from algorithmic optimizations for the computation of unit propagation, to low-level handling of CPU cache. However, the core algorithm hasn't changed much and is still either one of Davis–Putnam–Logemann–Loveland (i.e. DPLL) or Conflict Driven Clause Learning (i.e. CDCL) algorithm. This section will present CDCL as it is the one that I used in the implementation, which has traditionally better performances on structured problems while DPLL is usually better suited to random problems.

1.1.1 Problem

Given an countable set $\mathcal{V}_\mathcal{P}$, whose elements we will write $P, Q, R, \ldots$ and call atomic propositions, we can build propositional formulas using the usual logical connective:

$$\mathcal{P} := P, Q, R, \ldots |\top|\bot|\neg\mathcal{P}|\mathcal{P} \land \mathcal{P}|\mathcal{P} \lor \mathcal{P}|\mathcal{P} \Rightarrow \mathcal{P}|\mathcal{P} \Leftrightarrow \mathcal{P}$$

We can then define as usual the notion of domain, structure, interpretation, model, evaluation, and formula satisfiability in order to give a semantic to propositional formulas, and its relation to the provability of formulas in classical sequent calculus.

We then consider the propositional satisfiability (SAT) problem:

Definition 1.1.1. Propositional satisfiability, a.k.a SAT: Given a formula F, does there exists a model of F, i.e. an interpretation $\mathcal{M}$ such that $\mathcal{M} \models F$?

[2] SMT: Satisfiability Modulo Theory

[3] McSat was introduced in 2013

Since the valuation of atomic propositions in $\mathcal{M}$ only matters for atoms that actually occur in F, and that each atom can only have one of two different valuation ($\top$ and $\bot$), there are only a finite number of models to consider when checking the satisfiability of a formula, and thus satisfiability is decidable. However, the number of such models is exponential in the number of variables and so also exponential in the size of the formula.

Note that in classical logic, which will be used in the remainder unless specified, we can reformulate satisfiability in terms of provability, and give an alternate definition :

Definition 1.1.2. SAT: Given a formula F, is there a proof of $F \vdash \bot$?

The SAT algorithm actually doesn't work on propositional formulas as defined above, but instead works with clauses because their specific structure allows for specified reasoning steps that greatly speed up the execution of the algorithm in practice. More specifically, the SAT algorithm only works on input problems in CNF[1]. We therefore define the following:

Definition 1.1.3. A literal is either an atomic proposition (also called a positive literal), or the negation of an atomic proposition (also called a negative literal): $l = P | \neg P$.

Definition 1.1.4. A clause is a disjunction of literals: $\mathcal{C} = l_1 \vee l_2 \vee \ldots \vee l_n$.

Definition 1.1.5. A formula in clausal normal form is a conjunction of clauses:

$$F = \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \ldots \wedge \mathcal{C}_m$$

The restriction of the SAT algorithm to formulas in CNF is not a problem however, since it is possible to convert an arbitrary formula into an equivalent formula in CNF, though the resulting formula may increase exponentially in size. There exist other conversion schemes preserving satisfiability (but not producing an equivalent formula) that are used in practice to avoid this problem. These conversions are usually done as a linear pre-processing step in almost all existing implementations of SAT solvers.

Contrary to most SAT solvers however, ArchSAT does not use conversion to CNF, and instead relies on a lazy on-the-fly unfolding of logical connective to encode propositional logic into the clausal calculus implemented by the SAT algorithm, see Section 2.1.2.

1.1.2 Algorithm

As mentioned in the previous section, the SAT algorithm solves the satisfiability problem for formulas in CNF. The SAT algorithm works by maintaining a set $\mathbb{S}$ of clauses to satisfy, and incrementally building a partial model and backtracking when the partial model is found to not satisfy the clauses in $\mathbb{S}$.

We will represent our partial model using a trail, which is an ordered list of assignments. Assignments are either decisions or propagations. A decision is a triple containing a literal l, an integer n called the level of the decision, and a boolean value v, and is written $l \mapsto_n v$. Propagations are triples containing a literal l, a clause C called the reason of the propagation, and a boolean value v, and are written $l \leadsto_C v$. In the following, we will consider that any assignment from a literal l' to $\bot$ is the same as an assignment from $\neg l'$ to $\top$. For that, we must assume that negation is involutive, i.e. $\neg\neg l = l$. Each element in the trail has a level, which is the number of decision appearing in the trail up to (and including) it. For instance, propagations made before any decisions have level 0, and the first decision has level 1. In this manuscript, trails will always be written left to right chronologically, so that the left-most element of the list is the first element of the trail.

A trail t actually represents a partial model $\mathcal{M}$, where an assignment of the form $P \mapsto_n \top$ or $P \leadsto_C \top$ corresponds to $\mathcal{M}(P) = \top$, and an assignment of the form $\neg P \mapsto_n \top$ or $\neg P \leadsto_C \top$ corresponds to $\mathcal{M}(P) = \bot$. We can thus consider that a trail, through its corresponding partial model, defines a partial evaluation function on formulas (with the same semantics as usual), and thus re-use the provability notation, so that given a trail t and an arbitrary formula F, we say that F is true in t and write $t \vdash F$ iff F evaluates to true in the partial model corresponding to t.

[1] clausal normal form

$$\textsc{Propagate}\ \frac{\text{Solve}(\mathbb{S}, t)}{\text{Solve}(\mathbb{S}, t :: a \rightsquigarrow_C \top)} \qquad \begin{array}{l} C \in \mathbb{S}, \neg(a \text{ occurs in } t) \\ C = a \vee C' \qquad t \vdash \neg C' \end{array}$$

$$\textsc{Decide}\ \frac{\text{Solve}(\mathbb{S}, t)}{\text{Solve}(\mathbb{S}, t :: a \mapsto_n \top)} \qquad \begin{array}{l} \neg(a \text{ occurs in } t) \\ n = \text{max\_level}(t) + 1 \end{array}$$

$$\textsc{Conflict-sat}\ \frac{\text{Solve}(\mathbb{S}, t)}{\text{Analyze}(\mathbb{S}, t, C)} \qquad C \in \mathbb{S}, t \vdash \neg C$$

$$\textsc{Analyze-propagation}\ \frac{\text{Analyze}(\mathbb{S}, t :: a \rightsquigarrow_C \top, D)}{\text{Analyze}(\mathbb{S}, t, D)} \qquad \neg(a \text{ occurs in } D)$$

$$\textsc{Analyze-decision}\ \frac{\text{Analyze}(\mathbb{S}, t :: a \mapsto_n \top, D)}{\text{Analyze}(\mathbb{S}, t, D)} \qquad \neg(a \text{ occurs in } D)$$

$$\textsc{Analyze-Resolution}\ \frac{\text{Analyze}(\mathbb{S}, t :: a \rightsquigarrow_C \top, D)}{\text{Analyze}(\mathbb{S}, t, C' \vee D')} \qquad \begin{array}{l} C = a \vee C' \\ D = \neg a \vee D' \end{array}$$

$$\textsc{Learn-sat}\ \frac{\text{Analyze}(\mathbb{S}, t :: \_ \mapsto_{\_} \_ :: \_ :: a \mapsto_{d'} \top :: \_, D)}{\text{Solve}(\mathbb{S} \cup \{D\}, t :: \neg a \rightsquigarrow_D \top)} \qquad \begin{array}{l} D = \neg a \vee D' \\ (t \vdash \neg D') \end{array}$$

Figure 1.1: Inference rules for the SAT algorithm

The maximum level of a trail t, written max_level(t) is the highest level of decisions in t. We will say that a literal l occurs in a trail t, and write "l occurs in t" iff l or $\neg l$ is assigned in t. Similarly, we will also use that notation for clauses: "l occurs in C" iff l, or $\neg l$ is one of the literals in C.

The SAT algorithm internal state carries both the set $\mathbb{S}$ of clauses to satisfy, and a trail t. Furthermore this internal state can be in one of the two following states:

- Solve$(\mathbb{S}, t)$, where propagations and decisions are made, until a conflict D is detected, at which point the algorithm enters
- Analyze$(\mathbb{S}, t, D)$, which carries an additional clause D called the conflict, where the conflict is analyzed in order to determine a suitable backtrack point, after which the solver can revert to its Solver state.

We can now formalize the CDCL algorithm. In order to solve the satisfiability of a set of clauses $\mathbb{S}$, we start in the Solve$(\mathbb{S}, [])$[1] state, and we use the inference rules in Figure 1.1

- In rule Propagate, there is a clause $\mathcal{C} \in \mathbb{S}$ such that, in the current partial model, all literals but one (that we write a) are false. In this case, the only way to extend our partial model in order to satisfy the clause $\mathcal{C}$ is to assign a to $\top$, so we add the propagation $a \rightsquigarrow_C \top$ to the trail. This is usually called boolean constraint propagation.
- Rule Decide introduces choice points in the trail, by deciding on a value for a not yet assigned literal a. The idea is to try and find a model where a is true, and if that fails, then the solver will know that a must be false (see the analyze and Backjump rules for more explanations). In practice, in order to ensure termination, the Decide rule is restricted to decide only on literals that occur in the set of clauses $\mathbb{S}$.
- Rule Conflict-sat detects contradiction between the trail and the set of clauses to satisfy. More precisely, assuming an input state Solve$(\mathbb{S}, t)$, t, the Conflict-sat rule stops propagations and decisions if a clause $D \in \mathbb{S}$ is false according to the trail t. Once such a clause is

[1] [] denotes the empty trail

detected, it switches to the Analyze$(\mathbb{S}, t, D)$ state in order to analyze the conflict clause D and find the decision that created the conflict.

The Analyze-Propagation, Analyze-Decision and Analyze-Resolution rules are there to go back up the trail in order to find where to backtrack, and which decision to reverse. The idea is to undo the propagations in reverse chronological order so that we get to the latest choice point (i.e. decision) involved in the conflict. Importantly, the rules will rely and maintain the following invariants on the Analyze$(\mathbb{S}, t, D)$ state[1]

1. The conflict clause D is entailed by the clauses in $\mathbb{S}$.
2. The conflict clause D is false in the current trail: $t \vdash \neg D$.

Invariant 1 and 2 together ensure the fact that the current trail cannot satisfy the set of clauses $\mathbb{S}$ and thus that the solver needs to backtrack. Invariant 1 will always hold and will be used to add the conflict clause to the set $\mathbb{S}$ at the end of the analyze phase, in order to learn from the conflict and avoid repeating the same bad decision, hence the name of the algorithm: Conflict Driven Clause Learning. We thus need to satisfy the conflict clause, and for that we will look for a point in the trail where invariant 2 does not hold anymore, by looking at the last element of the trail.

When looking at the last element of the trail in order to refine our conflict clause, we can distinguish four cases:

- We last propagated a literal which does not occur in the conflict, in which case there is nothing to do (rule Analyze-Propagation)
- We last decided a literal which does not occur in the conflict, again in which case there is nothing to do (rule Analyze-Decision)
- We last propagated $a \rightsquigarrow_C \top$, and a occurs in D. We know that a being true causes D to not be satisfied, but a being true was not a choice but rather the consequence of choices that were made before propagating it, thus we want to look for these choices, and "eliminate" a from the conflict clause. To do so, we will perform a resolution between the conflict clause D and the reason C of the propagation of a. This is always possible: indeed, using invariant 2, since a occurs in D, we know that $D = \neg a \vee D'$. Also, given how rule Propagate works, we know that $C = a \vee C'$. Thus the resolution always works and yields the clause $C' \vee D'$, which is used to continue the conflict analysis (replacing D). This maintains invariant 1 because the resolution inference rule $\dfrac{C \vee a \qquad D \vee \neg a}{C \vee D}$ is admissible in classical logic. Invariant 2 is also maintained because the Propagate rule ensures that $t \vdash \neg C'$. Also, the SAT algorithm considers clauses as sets of literals, thus redundant occurences of a literal in a clause are systematically eliminated in the resolution result clause.
- Finally, we last decided on a literal a, which occurs in the conflict $D = \neg a \vee D'$. Because of invariant 2, D is not satisfied by the trail, and thus we know that the decision was wrong. The aim is thus to undo that decision, and add the clause D and propagate $\neg a$. Since D is not satisfied by the trail, D' is also not satisfied by the trail, and it may happen that the last decision responsible for D' being unsatisfied is earlier than the decision right before a, so rule Learn-sat allows to backtrack to any point where D' is not satisfied (and thus a can be propagated). Once that point is chosen, We can then undo the trail up to that point, and add the analyzed conflict clause D to the set of clauses to satisfy, which is sound because of invariant 1. Undoing the trail ensures that we can progress: a is not assigned in t, and $t \vdash \neg D'$[2], because $t :: a \mapsto_n \top \vdash \neg D$, thus it is possible to propagate $\neg a \rightsquigarrow_D \top$. Note that it is possible in rule Learn-sat that $a = b$, that is, the rule only backtracks one decision.

The algorithm can then end in one of two ways:

[1] See Chapter 4 for more details

[2] And thus, the side condition $t \vdash \neg D'$ is actually redundant, but will be useful when using McSat, see Section 1.3.

- The algorithm reaches a state Solve$(\mathbb{S}, t)$ in which no further inference rule is applicable. In this case, the input problem is satisfiable. Indeed, if neither rule Propagate or Decide is applicable, then all literals occurring in the input problems are in the trail, which means that every clause in $\mathbb{S}$ can be evaluated, and since the Conflict-sat rule cannot be applied, all clauses must be satisfied. We have thus found a model of the input clauses[1]

- The empty clause $\bigvee_{l\in\emptyset} l = \bot$ is deduced during the analyze phase, in which case the analyze phase will end in a state of the form Analyze$(\mathbb{S}, [], \bot)$[2]. In this case the input problem is unsatisfiable. Indeed the algorithm has proved that the input clauses allow to deduce $\bot$, which is absurd.

1.1.3 Some Examples

Let's look at how the algorithm works on some concrete problems.

First let's try and prove that the following clauses are satisfiable: $\mathcal{C}_1 = \neg P \vee R$, $\mathcal{C}_2 = \neg P \vee \neg R$. We get the following derivation:

$$
\begin{array}{cl}
\text{Solve}(\{\boldsymbol{\mathcal{C}_1 = \neg P \vee R}, \boldsymbol{\mathcal{C}_2 = \neg P \vee \neg R}\}, []) & \\ \hline
\text{Solve}(\{\mathcal{C}_1, \mathcal{C}_2\}, [\boldsymbol{P \mapsto_1 \top}]) & \textsc{Decide} \\ \hline
\text{Solve}(\{\mathcal{C}_1, \mathcal{C}_2\}, [P \mapsto_1 \top, \boldsymbol{R \rightsquigarrow_{\mathcal{C}_1} \top}]) & \textsc{Propagate} \\ \hline
\text{Analyze}(\{\mathcal{C}_1, \mathcal{C}_2\}, [P \mapsto_1 \top, R \rightsquigarrow_{\mathcal{C}_1} \top], \boldsymbol{\mathcal{C}_2 = \neg P \vee \neg R}) & \textsc{Conflict-sat} \\ \hline
\text{Analyze}(\{\mathcal{C}_1, \mathcal{C}_2\}, [P \mapsto_1 \top], \boldsymbol{\neg P}) & \textsc{Analyze-resolution} \\ \hline
\text{Solve}(\{\mathcal{C}_1, \mathcal{C}_2, \boldsymbol{\mathcal{C}_3 = \neg P}\}, [\boldsymbol{\neg P \rightsquigarrow_{\mathcal{C}_3} \top}]) & \textsc{Learn-sat} \\ \hline
\text{Solve}(\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}, [\neg P \rightsquigarrow_{\mathcal{C}_3} \top, \boldsymbol{R \mapsto_1 \top}]) & \textsc{Decide}
\end{array}
$$

At this point, there is no rule left to apply, and we have indeed found a model of the input clauses: P is false and R is true.

Let us look at an unsatisfiable example. Consider the following clauses: $\mathcal{C}_1 = P \vee Q$, $\mathcal{C}_2 = \neg P \vee R$, $\mathcal{C}_3 = \neg Q \vee R$, $\mathcal{C}_4 = S \vee T$, $\mathcal{C}_5 = \neg S \vee \neg R$, $\mathcal{C}_6 = \neg T \vee \neg R$,

We can build the following derivation:

[1] Or more exactly we have found an infinite family of models: any extension of the partial model described by the trail into a complete model, is a model of the input clauses.

[2] Note that this is also a final state, in the sense that no inference rule can be applied.

$$\begin{array}{cl}
\text{Solve}(\{\begin{array}{l}\boldsymbol{\mathcal{C}_1 = P \lor Q, \mathcal{C}_2 = \neg P \lor R, \mathcal{C}_3 = \neg Q \lor R,}\\ \boldsymbol{\mathcal{C}_4 = S \lor T, \mathcal{C}_5 = \neg S \lor \neg R, \mathcal{C}_6 = \neg T \lor \neg R}\end{array}\}, []) & \\
\hline
\text{Solve}(\{\mathcal{C}_1, \dots, \mathcal{C}_6\}, [\boldsymbol{P \mapsto_1} \top]) & \textsc{Decide} \\
\hline
\text{Solve}(\{\mathcal{C}_1, \dots, \mathcal{C}_6\}, [P \mapsto_1 \top, \boldsymbol{R \rightsquigarrow_{\mathcal{C}_2}} \top]) & \textsc{Propagate} \\
\hline
\text{Solve}(\{\mathcal{C}_1, \dots, \mathcal{C}_6\}, [P \mapsto_1 \top, R \rightsquigarrow_{\mathcal{C}_2} \top, \boldsymbol{\neg S \rightsquigarrow_{\mathcal{C}_5}} \top]) & \textsc{Propagate} \\
\hline
\text{Solve}(\{\mathcal{C}_1, \dots, \mathcal{C}_6\}, [P \mapsto_1 \top, R \rightsquigarrow_{\mathcal{C}_2} \top, \neg S \rightsquigarrow_{\mathcal{C}_5} \top, \boldsymbol{\neg T \rightsquigarrow_{\mathcal{C}_6}} \top]) & \textsc{Propagate} \\
\hline
\text{Analyze}(\{\mathcal{C}_1, \dots, \mathcal{C}_6\}, [P \mapsto_1 \top, R \rightsquigarrow_{\mathcal{C}_2} \top, \neg S \rightsquigarrow_{\mathcal{C}_5} \top, \neg T \rightsquigarrow_{\mathcal{C}_6} \top], \boldsymbol{\mathcal{C}_4 = S \lor T}) & \textsc{Conflict-sat} \\
\hline
\text{Analyze}(\{\mathcal{C}_1, \dots, \mathcal{C}_6\}, [P \mapsto_1 \top, R \rightsquigarrow_{\mathcal{C}_2} \top, \neg S \rightsquigarrow_{\mathcal{C}_5} \top], \boldsymbol{S \lor \neg R}) & \textsc{Analyze-resolution} \\
\hline
\text{Analyze}(\{\mathcal{C}_1, \dots, \mathcal{C}_6\}, [P \mapsto_1 \top, R \rightsquigarrow_{\mathcal{C}_2} \top], \boldsymbol{\neg R}) & \textsc{Analyze-resolution} \\
\hline
\text{Analyze}(\{\mathcal{C}_1, \dots, \mathcal{C}_6\}, [P \mapsto_1 \top], \boldsymbol{\neg P}) & \textsc{Analyze-resolution} \\
\hline
\text{Solve}(\{\mathcal{C}_1, \dots, \mathcal{C}_6, \boldsymbol{\mathcal{C}_7 = \neg P}\}, [\boldsymbol{\neg P \rightsquigarrow_{\mathcal{C}_7}} \top]) & \textsc{Learn-sat} \\
\hline
\text{Solve}(\{\mathcal{C}_1, \dots, \mathcal{C}_7\}, [\neg P \rightsquigarrow_{\mathcal{C}_7} \top, \boldsymbol{Q \rightsquigarrow_{\mathcal{C}_1}} \top]) & \textsc{Propagate} \\
\hline
\text{Solve}(\{\mathcal{C}_1, \dots, \mathcal{C}_7\}, [\neg P \rightsquigarrow_{\mathcal{C}_7} \top, Q \rightsquigarrow_{\mathcal{C}_1} \top, \boldsymbol{R \rightsquigarrow_{\mathcal{C}_3}} \top]) & \textsc{Propagate} \\
\hline
\text{Solve}(\{\mathcal{C}_1, \dots, \mathcal{C}_7\}, [\neg P \rightsquigarrow_{\mathcal{C}_7} \top, Q \rightsquigarrow_{\mathcal{C}_1} \top, R \rightsquigarrow_{\mathcal{C}_3} \top, \boldsymbol{\neg S \rightsquigarrow_{\mathcal{C}_5}} \top]) & \textsc{Propagate} \\
\hline
\text{Solve}(\{\mathcal{C}_1, \dots, \mathcal{C}_7\}, [\neg P \rightsquigarrow_{\mathcal{C}_7} \top, Q \rightsquigarrow_{\mathcal{C}_1} \top, R \rightsquigarrow_{\mathcal{C}_3} \top, \neg S \rightsquigarrow_{\mathcal{C}_5} \top, \boldsymbol{\neg T \rightsquigarrow_{\mathcal{C}_6}} \top]) & \textsc{Propagate} \\
\hline
\text{Analyze}(\{\mathcal{C}_1, \dots, \mathcal{C}_7\}, [\neg P \rightsquigarrow_{\mathcal{C}_7} \top, Q \rightsquigarrow_{\mathcal{C}_1} \top, R \rightsquigarrow_{\mathcal{C}_3} \top, \neg S \rightsquigarrow_{\mathcal{C}_5} \top, \neg T \rightsquigarrow_{\mathcal{C}_6} \top], \boldsymbol{\mathcal{C}_4 = S \lor T}) & \textsc{Conflict-sat} \\
\hline
\text{Analyze}(\{\mathcal{C}_1, \dots, \mathcal{C}_7\}, [\neg P \rightsquigarrow_{\mathcal{C}_7} \top, Q \rightsquigarrow_{\mathcal{C}_1} \top, R \rightsquigarrow_{\mathcal{C}_3} \top, \neg S \rightsquigarrow_{\mathcal{C}_5} \top], \boldsymbol{S \lor \neg R}) & \textsc{Analyze-resolution} \\
\hline
\text{Analyze}(\{\mathcal{C}_1, \dots, \mathcal{C}_7\}, [\neg P \rightsquigarrow_{\mathcal{C}_7} \top, Q \rightsquigarrow_{\mathcal{C}_1} \top, R \rightsquigarrow_{\mathcal{C}_3} \top], \boldsymbol{\neg R}) & \textsc{Analyze-resolution} \\
\hline
\text{Analyze}(\{\mathcal{C}_1, \dots, \mathcal{C}_7\}, [\neg P \rightsquigarrow_{\mathcal{C}_7} \top, Q \rightsquigarrow_{\mathcal{C}_1} \top], \boldsymbol{\neg Q}) & \textsc{Analyze-resolution} \\
\hline
\text{Analyze}(\{\mathcal{C}_1, \dots, \mathcal{C}_7\}, [\neg P \rightsquigarrow_{\mathcal{C}_7} \top,], \boldsymbol{P}) & \textsc{Analyze-resolution} \\
\hline
\text{Analyze}(\{\mathcal{C}_1, \dots, \mathcal{C}_7\}, [], \bot) & \textsc{Analyze-resolution}
\end{array}$$

In this case, we indeed reach the empty clause when analyzing the second conflict, proving the input clauses are unsatisfiable. It is also quite easy to extract a full formal proof of unsatisfiability, see Chapter 4.1.

1.1.4 Learning strategy

It can be noted that, while the Propagate, Decide and Conflict-sat rules are standard and present in one form or another in all presentation of the SAT algorithm, the rules related to the analyze of the conflict, and consequently the clauses learned during the algorithm execution can differ in other presentations. Indeed, while given a conflict, there is a unique latest decision that generated the conflict, there are more than one way to analyze the conflict, deduce a new clause to add to the solver state, and choose a point in the trail to backtrack to.

One such instance can be seen in the example above: when analyzing the first conflict (starting with $\mathcal{C}_4$), we deduce the clause $\mathcal{C}_7 = \neg P$, but we could have stopped resolutions one step earlier and instead try and learn the clause $\mathcal{C}_7' = \neg R$. The decision undone would not have changed, but the clause learnt, and thus the propagation done afterwards would be different. In this particular case, learning $\mathcal{C}_7'$ would have lead to solving the problem in fewer steps. However, that would require a more generalized rule Learn-sat.

These different strategies have been studied in [89]. The rules in Figure 1.1 correspond to the last Unique Implication Point, or last UIP strategy for learning, while the first UIP strategy would have learnt clause $\mathcal{C}_7'$.

1.1.5 Soundness, completeness

The SAT algorithm is sound and complete, see [18, 39]. No proof will be presented in this thesis, but the detailed description in the previous section (and particularly the invariants of the analyze phase) should be enough to understand the soundness of the algorithm. Completeness can be proven using the fact that there is a finite number of trails, and the same trail appears at most twice: at most once in a solve phase, and at most once in an analyze phase. This is ensured, among other things, because once a decision $a \mapsto_n \top$ has been undone by the Learn-sat rule, its inverse $\neg a \rightsquigarrow \top$ is propagated, thus preventing to reach an already seen trail.

1.1.6 Implementation

When applying the algorithm in practice, it is useful to use a specific strategy when applying the inference rules of the algorithm. Specifically, the rules are given the following order of priority:

1. The Conflict-sat rule is applied as eagerly as possible, in order to detect absurd trails as soon as possible.

2. The Propagate rule is then applied as much as possible, in order to minimize the number of decisions, so as to reduce the possibilities to make a bad decision.

3. Finally the Decide rule is called when no conflict is detected and no propagation can be done.

4. The Analyze-decision, Analyze-propagation, and Analyze-resolution rules have mutually exclusive premises and thus need no priorities.

5. Finally, rule Learn-sat offers a choice point as to how far back to backtrack. Choosing to undo only the decision that generated the conflict (i.e. choosing $a = b$), is typical called backtracking, whereas choosing to backtrack as far as possible is called backjumping. The advantage of backjump is that it allows us to revert to the earliest point where the Learn-sat rule can propagate, whereas with the normal backtrack, the propagation done by the rule may not need some of the most recent decisions.

It is interesting to note that these priorities are not needed for the soundness and completeness of the algorithm. However, they are highly useful to lower the practical complexity of the algorithm, especially together with concrete implementation details of efficient way to apply these rules:

- Decision choices usually rely on some heuristics, prioritizing literals that appear the most often in conflicts in order to reach conflicts faster.

- Propagation can be very efficiently detected and applied using a 2-watch scheme: in order to quickly detect which clause may trigger application of rule Propagate, the solver “watches” 2 unassigned literals in each clause. Once one of the literals becomes assigned, the solver looks for another unassigned literal. If one is found, the solver updates it watched literals for that clause, else the only unassigned literal in the clause is propagated. This allows for very efficient detection of propagation opportunities, and watched literals do not need to be changed upon backtracking.

- As noted, rules Analyze-decision, Analyze-propagation, Analyze-resolution are mutually exclusive and can be merged into a single function analyzing a conflict in a very deterministic and optimized manner, computing directly the final result of the conflict (particularly when using the last UIP strategy), avoiding the need to compute all the intermediary resolutions which can be quite costly.

1.2 SMT Solving

The SAT algorithm described previously only handles purely propositional formulas[1]. However it is often the case that we want to solve problems involving formulas using quantified variables, and first-order theories such as arithmetic. The SMT algorithm is an extension of the SAT algorithm to handle precisely these cases.

[1] After an eventual transformation to clausal normal form.

1.2.1 First-Order Terms and Theory Theory

Given a signature $\mathcal{S} = (\mathcal{V}, \mathcal{S}_\mathcal{F}, \mathcal{S}_\mathcal{P})$, where $\mathcal{V}$ is an infinite set of variables, $\mathcal{S}_\mathcal{F}$ a set of function symbols and $\mathcal{S}_\mathcal{P}$ a set of predicate symbols, we can introduce $\mathcal{T}$ the set of first-order terms and $\mathcal{F}$ the set of first-order formulas over $\mathcal{S}$. An atomic formula is a formula that does not start with a logical connective, so it is either a predicate symbol application, or an equality. We will often use x, y, to denote variables in $\mathcal{V}$, $f, g, \ldots$ to denote function symbols in $\mathcal{S}_\mathcal{F}$, and $p, q, r, \ldots$ to denote predicate symbols in $\mathcal{S}_\mathcal{P}$. To each function and predicate symbol is associated an arity which is the number of arguments that they take. Function symbols of arity 0 will be written $a, b, \ldots$ and their applications written without parenthesis, e.g. a instead of $a()$. For instance :

- $a, f(b), g(x, f(c))$ are terms
- $p(x)$, $q(f(c, y)) \to (p(x) \land q(y))$ are formulas.
- $\forall z. \neg r(z)$ is a quantified formula, universally quantifying the variable z in $r(z)$.

As in Section 1.1.1, the notion of structure, interpretation, model, evaluation, and formula satisfiability is defined as usual. Interpretation of a first-order term (or formula) t in a model $\mathcal{M}$ will be written $[\![t]\!]_\mathcal{M}$.

A first-order theory T is a (potentially infinite) set of first-order formulas. A model of a theory T is a structure $\mathcal{M}$ that satisfies all formulas of T, also written $\mathcal{M} \models T$. A formula F is said to be satisfiable in a theory T, iff there exists a model $\mathcal{M}$ of T which also satisfies F : $\mathcal{M} \models F$.

We can then define the satisfiability modulo theory, i.e SMT problem :

Definition 1.2.1. SMT: Given a theory T and a first-order formula F, does there exist a model of F in T ?

First-order logic, with no theory (or rather an empty theory) is in general only semi-decidable because of universally quantified formulas. Some more restricted problems may however be decidable: for instance satisfiability of ground formulas in the theory of linear rational arithmetic is decidable. SMT solvers commonly aim to solve efficiently these restricted problems: that is problems that only involve ground formulas (i.e no quantified formula) over theories for which satisfiability of ground formulas is decidable, in which case the SMT algorithm provides a complete decision procedure.

Note that this section describes single-sorted first-order terms and theories. In practice, however, most provers use many-sorted first-order terms and theories, for instance for arithmetic. Going even further, I used typed first-order terms with polymorphic types, as described in Section 2.1.1. However, this distinction does not matter much when describing the SMT algorithm.

1.2.2 Algorithm

In order to solve satisfiability modulo theory, the SMT algorithm extends SAT in a very non-invasive manner. As for the SAT algorithm, the SMT algorithm only accept as input a set of clauses, but whose literals are atomic first-order formulas (i.e. either equalities or predicates) instead of atomic propositions. The main idea is to run a SAT solver on the input clauses, in order to find a truth value assignment for the input literals, and then verify that this assignment is satisfiable in the theory. When combined with a transformation into clausal normal form[1], the reasoning is the split into a reasoning on clauses which handles the propositional aspect of the input formulas, and a theory-specific reasoning, which does not need to handle the propositional structure of the formulas.

Formally, the SMT algorithm is the same as the SAT algorithm, with the added Conflict-smt rule from Figure 1.2. The rule allows to trigger an analyze phase and a backtrack whenever the partial truth assignment maintained by the algorithm in the trail is inconsistent with the theory.

The end conditions for the algorithm are the same as for the SAT algorithm:

[1] CNF transformation on quantified formulas is a bit more intricate than in the propositional case, for more information see Section 1.2.5.

$$\textsc{Conflict-smt}\ \frac{\text{Solve}(\mathbb{S}, t)}{\text{Analyze}(\mathbb{S}, t, D)} \qquad \begin{array}{l} t \vdash \neg D \\ \mathcal{T} \vdash D \end{array}$$

$$\textsc{Learn}\ \frac{\text{Solve}(\mathbb{S}, t)}{\text{Solve}(\mathbb{S} \cup \{C\}, t)} \qquad T \vdash C$$

Figure 1.2: Inference rules for the SMT algorithm

- Either the empty clause has been reached, in which case $\perp$ has been proven by the algorithm. Note that Conflict-smt ensures some slightly different analyze phase invariants than described in Section 1.1.2:
  1. The conflict clause D is entailed by the clauses in $\mathbb{S} \cup T$.
  2. The conflict clause D is false in the current trail: $t \vdash \neg D$.

- Or there is no rule left to apply, in which case, the truth assignment formed by the trail is satisfiable in the theory, otherwise a conflict could be found.

Finally, note that, while the current presentation, as well as most presentations, introduce the SMT algorithm in the context of first-order logic, the algorithm in itself is not really tied to first-order logic specifically. More generally, given any logic, the SMT algorithm is suitable for solving the satisfiability of sets of clauses, as long as it is provided with an adequate implementation of a theory.

1.2.3 Examples

Let us show how the SMT algorithm works on some examples. First we consider a problem using only equalities and some uninterpreted functions. Consider the following problem :

$$\begin{array}{lcc} \mathcal{C}_1 & := & a = b \\ \mathcal{C}_2 & := & \neg(f(a) = f(b)) \lor f(a) = c \end{array}$$

Clearly, this set of clauses is satisfiable. Now let us see how the SMT algorithm reaches that conclusion. For readability purposes, let us not use the Solve([...]) syntax but rather present the trace of the algorithm in a table :

Clause Set	Trail	Conflict (if analyzing)	Last Rule used
$\{ \mathcal{C}_1, \mathcal{C}_2 \}$	$[]$		
$\{ \mathcal{C}_1, \mathcal{C}_2 \}$	$[\boldsymbol{a = b \leadsto_{\mathcal{C}_1}} \top]$		Propagate
$\{ \mathcal{C}_1, \mathcal{C}_2 \}$	$[a = b \leadsto_{\mathcal{C}_1} \top, \boldsymbol{f(a) = f(b) \mapsto_1} \perp]$		Decide
$\{ \mathcal{C}_1, \mathcal{C}_2 \}$	$[a = b \leadsto_{\mathcal{C}_1} \top, f(a) = f(b) \mapsto_1 \perp]$	$\boldsymbol{\mathcal{C}_3 = \neg(a = b) \lor f(a) = f(b)}$	Conflict-smt
$\{ \mathcal{C}_1, \mathcal{C}_2, \boldsymbol{\mathcal{C}_3} \}$	$[a = b \leadsto_{\mathcal{C}_1} \top, \boldsymbol{f(a) = f(b) \leadsto_{\mathcal{C}_3}} \top]$		Learn-sat
$\{ \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3 \}$	$[a = b \leadsto_{\mathcal{C}_1} \top, f(a) = f(b) \leadsto_{\mathcal{C}_3} \top, \boldsymbol{f(a) = c \mapsto_1} \top]$		Decide
	SAT		

Here, once both $a = b$ and $f(a) \neq f(b)$ are in the trail, the theory can trigger the rule Conflict-smt, since $a = b \rightarrow f(a) = (b)$ is a tautology in the theory of equality and uninterpreted functions. This tautology can be found, for instance, by using a congruence closure algorithm. This tautology is also a clause: $a \neq b \lor f(a) = f(b)$, and so can directly be used as a conflict clause. We'll discuss in the next section how theories can be implemented in practice.

Now, let's prove that the following set of clauses is unsatisfiable:

$$
\begin{aligned}
\mathcal{C}_1 &\equiv a = b \\
\mathcal{C}_2 &\equiv b = c \lor b = d \\
\mathcal{C}_3 &\equiv \neg(a = d) \\
\mathcal{C}_4 &\equiv \neg(a = c)
\end{aligned}
$$

Clause Set	Trail	Conflict (if analyzing)	Last Rule used
$\{ \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4 \}$	$[]$		
$\{ \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4 \}$	$[\ \boldsymbol{a = b \rightsquigarrow_{\mathcal{C}_1} \top}\ ]$		Propagate
$\{ \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4 \}$	$[\ a = b \rightsquigarrow_{\mathcal{C}_1} \top, \boldsymbol{a = d \rightsquigarrow_{\mathcal{C}_3} \bot}\ ]$		Propagate
$\{ \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4 \}$	$[\ a = b \rightsquigarrow_{\mathcal{C}_1} \top, a = d \rightsquigarrow_{\mathcal{C}_3} \bot,$ $\boldsymbol{a = c \rightsquigarrow_{\mathcal{C}_4} \bot}\ ]$		Propagate
$\{ \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4 \}$	$[\ a = b \rightsquigarrow_{\mathcal{C}_1} \top, a = d \rightsquigarrow_{\mathcal{C}_3} \bot,$ $a = c \rightsquigarrow_{\mathcal{C}_4} \bot, \boldsymbol{b = d \mapsto_1 \top}\ ]$		Decide
$\{ \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4 \}$	$[\ a = b \rightsquigarrow_{\mathcal{C}_1} \top, a = d \rightsquigarrow_{\mathcal{C}_3} \bot,$ $a = c \rightsquigarrow_{\mathcal{C}_4} \bot, b = d \mapsto_1 \top\ ]$	$\mathcal{C}_5 = \neg a = b \lor \neg b = d \lor a = d$	Conflict-SMT
$\{ \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \boldsymbol{\mathcal{C}_5} \}$	$[\ a = b \rightsquigarrow_{\mathcal{C}_1} \top,\ a = d \rightsquigarrow_{\mathcal{C}_3} \bot,$ $a = c \rightsquigarrow_{\mathcal{C}_4} \bot, \boldsymbol{b = d \rightsquigarrow_{\mathcal{C}_5} \bot}\ ]$		Learn-Sat
$\{ \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_5 \}$	$[\ a = b \rightsquigarrow_{\mathcal{C}_1} \top, a = d \rightsquigarrow_{\mathcal{C}_3} \bot,$ $a = c \rightsquigarrow_{\mathcal{C}_4} \bot,\quad b = d \rightsquigarrow_{\mathcal{C}_5} \bot,$ $\boldsymbol{b = c \rightsquigarrow_{\mathcal{C}_2} \top}\ ]$		Propagate
$\{ \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_5 \}$	$[\ a = b \rightsquigarrow_{\mathcal{C}_1} \top, a = d \rightsquigarrow_{\mathcal{C}_3} \bot,$ $a = c \rightsquigarrow_{\mathcal{C}_4} \bot,\quad b = d \rightsquigarrow_{\mathcal{C}_5} \bot,$ $b = c \rightsquigarrow_{\mathcal{C}_2} \top\ ]$	$\neg a = b \lor \neg b = c \lor a = c$	Conflict-Smt
$\{ \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_5 \}$	$[\ a = b \rightsquigarrow_{\mathcal{C}_1} \top, a = d \rightsquigarrow_{\mathcal{C}_3} \bot,$ $a = c \rightsquigarrow_{\mathcal{C}_4} \bot, b = d \rightsquigarrow_{\mathcal{C}_5} \bot,\ ]$	$\neg a = b \lor b = d \lor a = c$	Analyze-Resolution
$\{ \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_5 \}$	$[\ a = b \rightsquigarrow_{\mathcal{C}_1} \top, a = d \rightsquigarrow_{\mathcal{C}_3} \bot,$ $a = c \rightsquigarrow_{\mathcal{C}_4} \bot,\ ]$	$\neg a = b \lor a = d \lor a = c$	Analyze-Resolution
$\{ \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_5 \}$	$[\ a = b \rightsquigarrow_{\mathcal{C}_1} \top, a = d \rightsquigarrow_{\mathcal{C}_3} \bot,\ ]$	$\neg a = b \lor a = d$	Analyze-Resolution
$\{ \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_5 \}$	$[\ a = b \rightsquigarrow_{\mathcal{C}_1} \top,\ ]$	$\neg a = b$	Analyze-Resolution
$\{ \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_5 \}$	$[]$	$\emptyset$	Analyze-Resolution
		UNSAT	

In this example, we have two conflicts involving equality, where transitivity of equality is broken. These conflicts can be found using a simple union-find algorithm [68]. Finally, at the end, the empty clause is reached, proving the input clauses are unsatisfiable.

1.2.4 Implementation, Theory requirements and Combinations

The presentation of the SMT algorithm in Section 1.2.2 is really theoretical and while the description of the SAT algorithm (with the rule priorities) is clear enough to be reasonably implemented, SMT theories as introduced earlier may appear less straightforward to implement in practice. This is intended, as theories are here considered as some kind of blackboxes, with a specified external behavior (i.e. triggering Conflict-smt as soon as possible). This is in order to make the SMT algorithm modular with regards to the theory, and allows us to consider a wide variety of first-order theories that can be implemented as an SMT theory. Examples of first-order theories often implemented in SMT solvers include equality and uninterpreted functions, linear arithmetic, arrays, bitvectors, ...

In order to trigger Conflict-smt as early as possible, the theory only has to ensure the satisfiability of the trail, which is exactly the satisfiability decision problem on atomic formulas of a given theory. Consequently, SMT theories are usually implemented using decision procedures such as union-find [68]. the simplex algorithm, Fourier-Motzkin ... Formally, the theory has to ensure that there exists a model (i.e. an assignment of all variables) that satisfies the trail. As long as that is verified, the SMT algorithm is correct. And assuming the theory is complete (i.e its algorithm terminates), so is the SMT algorithm.

There is, however, a very useful property of SMT theories that is needed in order to achieve good performances: incrementality. Assume an algorithm that accepts a list of inputs, and with an internal state. Incrementality refers to the fact that, starting from an internal state s corresponding to a list of inputs $l = [x_1, \ldots, x_n]$, computing the state s' corresponding to the list of inputs $l' = [x_1, \ldots, x_n, x_{n+1}]$, can be done faster[1] by starting from s and adding the new input x_{n+1}, than by computing from scratch s' from l'. For instance, the simplex algorithm is generally considered incremental, in that adding new constraints and finding a new solution is fast because the work that has been done in order to satisfy the previous constraints is re-used.

Once an incremental theory is implemented, it can be integrated with a SAT solver in order to build an SMT solver. The theory formally has access to the current state of the SAT solver, particularly the trail. Incremental theories can thus update their internal state for each new decision or propagation added on the trail. This internal state is then backtracked and restored when the SAT solvers analyzes a conflict and backtracks the trail. Non-incremental theories cannot run on every new decision of propagation in practice since it would take too long, and thus usually can not raise rule Conflict-Smt as eagerly as incremental theories.

Finally, although it is not required, the rule Learn from Figure 1.2 is quite convenient to consider. Such a rule could replace the Conflict-smt rule, but would break the termination property of the SMT algorithm unless restrictions are placed on the clauses that the theory can add. Indeed, this rule could allow a theory to introduce an infinite number of tautologies (for instance $1+1=2$, $1+2=3$, $1+3=4$, $\ldots$), and thus prevent other rules from being applied. However, the main point of the Learn rule is to allow the theory to encode some of its reasoning into clauses, and rely on the capacity of the clausal calculus implemented by the rules of the SAT algorithm. One example of such a use is to defer some theory-specific decisions back to the inference rules, e.g. considering a theory for arithmetic that can only deal with equalities and inequations, but not disequalities. Note that it is a realistic occurrence, as this is actually the case of arithmetic theories that use the simplex algorithm. Now consider a propagation $x = a + b \rightsquigarrow_n \bot$ (which is equivalent to $\neg(x = a + b) \rightsquigarrow_n \top$), it might be interesting for the theory to "ask" the solver to decide whether $x < a + b$ or $x > a + b$ by adding the tautology $(x = a + b) \lor (x < a + b) \lor (x > a + b)$ as a clause using rule Learn. With such a clause, it is guaranteed that the inference rules will at one point decide or propagate either $x < a + b$ or $x > a + b$. That way, the arithmetic theory does not have to internally implement a case analysis on each disequality.

1.2.5 Quantified Formulas and Triggers

As mentioned earlier, the SMT algorithm, just like the SAT algorithm, only works on clauses. While that was not a problem for propositional formulas, it is more problematic for first-order formulas. Indeed, while Skolemization allows us to eliminate existential quantification, universal quantification is more problematic.

What is usually done in modern SMT solvers it to leave free variables while converting the input formulas into clausal normal form. The resulting clauses are then split into those with only ground terms, and those in which free variables occur. In these clauses with free variables, a heuristic is used to select some subterms, which will be called triggers. The SMT solver is then run using only the ground clauses as input. If it reaches the empty clause, then the problem is unsatisfiable, else if a model is found, this model is used to try and match the selected triggers with the ground terms occurring in the model. The substitutions found are then used to instantiate the clauses with free variables, whose instantiations[2] are added to the set of ground clauses, and the algorithm repeats.

For instance, let us consider the following problem:

$$F = (\forall x.\ p(x) \rightarrow q(x)) \rightarrow p(a) \rightarrow q(a)$$

In order to prove F, we will work by contradiction and use the SMT algorithm to prove that $\neg F$ is unsatisfiable. Converting $\neg F$ into clausal normal form yield the following clauses :

$$\begin{array}{rcc} \mathcal{C}_0 & := & \neg p(x) \lor q(x) \\ \mathcal{C}_1 & := & p(a) \\ \mathcal{C}_2 & := & \neg q(a) \end{array}$$

[1] It has a lower theoretical time complexity, but more importantly a faster implementation.

[2] i.e. ground clauses.

Note the implicitly universally quantified variable x in $\mathcal{C}_0$. Now we can, for instance, choose $q(x)$ as trigger[1]. We can run the SMT algorithm on the input set of clauses $\mathbb{S} = \{\mathcal{C}_1, \mathcal{C}_2\}$ using a theory for uninterpreted functions and predicates, and it trivially returns the following model :

$$\begin{array}{rcl} p(a) & \mapsto & \top \\ q(a) & \mapsto & \bot \end{array}$$

With this model we can try and match any ground term in it with our trigger $q(x)$, and we find the substitution $x \mapsto a$ by matching $q(x)$ with $q(a)$. We can then apply this substitution to the clause where the trigger originated, and we get a new clause $\mathcal{C}_3 = \neg p(a) \vee q(a)$. We then add this clause to the set of ground clauses. We thus run again the SMT algorithm on the input set of clauses $\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$. Naturally, this lead to reaching the empty clause, thus proving the unsatisfiability of the clauses. We have then proved the input formula F.

Triggers thus allow SMT solvers, which are usually mainly used for ground problems, to also handle problems using quantified formulas. However, in the general case, triggers do not yield a complete decision procedure, more precisely, the completeness of the SMT algorithm with triggers depend on which triggers are selected, and there are no generic and complete way of selecting trigger, only heuristics [48–50].

1.3 McSat

McSat, introduced in [42,58], is an extension of the SMT algorithm. It solves the same problem, that is, satisfiability of formulas modulo a theory, but it integrates the theory more than the SMT algorithm does.

Explanations for McSat in detail requires more use of the notions of domain and interpretation for first-order terms and formulas, than for the SMT algorithm. In order to allow for lighter notations and explanations, in the remainder of this thesis, we consider that we have a fixed theory T and domain of interpretation $\mathcal{D}$.

1.3.1 Semantic Decision, Propagations and Models

The main goal of the McSat algorithm is to allow not only decisions on the boolean values of literals in clauses, but also decisions on the values of variables occurring in these literals. These decisions may then allow us to evaluate some literal. For instance, consider that a term x has been given a value of 1 (by a mechanism that we describe later) and another term y the value of 0, then an arithmetic theory could evaluate the formula $x + y > 0$ and add it to the trail. More generally, assignments of variables to concrete values, and not equivalence classes as is typically done, gives more information and allow for efficient propagation. As in the SMT algorithm, McSat uses a theory to reason about first-order terms and formulas, and whose role we will detail later.

Formally, we extend the notion of trail introduced in Section 1.1.2 by adding semantic decision, which are assignments of the form $t \mapsto_n v$ where t is a term (usually occurring in the input clauses), n the level of the decision (as before), and v a value to which t is assigned. Values belong to the chosen domain $\mathcal{D}$ (for instance the relative numbers $\mathbb{Z}$ for integer arithmetic).

We also add semantic propagations of the form $l \rightsquigarrow_n \top$ where l is a literal and n an integer representing a level. Intuitively, a semantic propagation $l \rightsquigarrow_n \top$ means that l can be evaluated to $\top$ by using semantic decisions with level lower (or equal) than n. Using these levels, one can define a notion of level for clauses in a trail : the maximum level of a clause C in a trail t, written $\text{max\_level}_t(C)$, is the highest level of any semantic propagation $l \rightsquigarrow_{\_} \top$ for which l or $\neg l$ occurs in C, and else 0.

Finally, we also extend the notion of occurrence in a trail to also include semantic propagations and decisions: a term u is said to occur in a trail t iff u is assigned in t, and a formula p is said to occur in a trail t iff p or $\neg p$ is assigned in t.

[1] Here we only select one trigger, but it is possible to choose more than one trigger by clause, and in some cases, even select a set of set of triggers for each clause, only instantiating if all the triggers of a set are matched.

Partial Interpretation As in Section 1.1.2, we will use the trail (and particularly the semantic decisions) to represent a partial model of the input formulas. It is, however, not so simple: during proof search, the McSat algorithm maintains in the trail a (partial) mapping from first-order terms and formulas to model values (and not from variables and constant symbols to model values, as is needed for a model). The intention of this mapping is to represent a set of constraints on the model that the algorithm is building. For instance, given a function symbol f of arity 2, and a constant a, one such constraint that we would like to be able to express in our mapping is the following: $f(a) \mapsto 0$, regardless of the values that f takes on other argument, and also regardless of the value mapped to a. To that end we introduce the notion of abstract partial interpretation.

An abstract partial interpretation σ is a mapping from ground expressions to model values. To each abstract partial interpretation we can associate a set of complete models that realizes it:

$$\text{Complete}(\sigma) = \{\mathcal{M} \mid \forall t \mapsto v \in \sigma, [\![t]\!]_{\mathcal{M}} = v\}$$

Coherence An abstract partial interpretation σ is said to be coherent iff there exists at least one model that completes it, i.e. $\text{Complete}(\sigma) \neq \emptyset$. One example of incoherent abstract partial interpretation is the following mapping:

$$\sigma = \begin{cases} a \mapsto 0 \\ b \mapsto 0 \\ f(a) \mapsto 0 \\ f(b) \mapsto 1 \end{cases}$$

Indeed, in the theory of uninterpreted functions, there is no model in which a and b have the same values, but $f(a)$ and $f(b)$ have different values.

Compatibility In order to do semantic propagations, we want a notion of evaluation for abstract partial interpretations. We thus define the partial interpretation function of an abstract partial interpretation σ as the intersection of the interpretation functions of all the completions of σ, i.e. it is the interpretation where all completions agree. This can be axiomatised using the following proposition :

$$\forall t \in \mathcal{T} \cup \mathcal{F}. \forall v \in \mathcal{D}. \, (\forall \mathcal{M} \in \text{Complete}(\sigma). [\![t]\!]_{\mathcal{M}} = v) \rightarrow [\![t]\!]_{\sigma} = v$$

Note that as the name suggests, $[\![.]\!]_{\sigma}$ is partial as it is not defined on terms whose value can change from one completion of σ to another.

Naturally, the semantic decisions in a trail t form an abstract partial model, and thus define a partial interpretation function that we will write σ_t. We can now say that a trail t is coherent iff σ_t is coherent, and for all literals a occurring in t, $\neg([\![a]\!]_{\sigma_t} = \bot)$, or in other words, for every literal a true in the trail, there exists at least one model that completes σ and where a is satisfied.

1.3.2 Algorithm

Just as for the SMT algorithm, McSat is an extension of SAT, thus we extend the inference rules of Figure 1.1 with those of Figure 1.3. These rules suppose that a theory T has been defined and fixed.

- The Decide-semantic allows the theory to decide a semantic assignment for a not-yet decided term. The only constraint is that the resulting trail must be coherent, to avoid triggering a conflict right after a decision.
- The Propagate-semantic rule allows the theory to deduce propagations that are entailed by semantic assignments, i.e. literals that can be evaluated using the semantic decisions. Each such propagation is annotated with the level at which it occurred[1], representing the semantic decisions that are needed for the evaluation.

[1] One could instead annotate the propagation with the level of decision that are actually needed to deduce the propagation, instead of asusming that semantic propagations were done as early as possible. However, this would needlessly complicate the presentation of the McSat algorithm

- The Conflict-mcsat rule mirrors the Conflict-smt rule from Figure 1.2, but also stores the level of the clause (which is useful for treating semantic propagations in clauses).

Correctly analyzing an McSat conflict is not easy. Indeed, compared to an SMT solver, conflict clauses may contain semantic propagations. The reasons behind a semantic propagation are semantic decisions, so just as in the SAT algorithm where we used propagations reasons to find the (boolean) decision at the root of the conflict, and forbid the decision, in this case we would like to forbid the semantic decision that caused the conflict. Identifying the semantic decision is easy, since semantic propagation store the maximum level needed for their evaluation. However, forbidding a semantic decision is trickier: while in the boolean case, the decision could simply be flipped: i.e if the solver had decided $l \mapsto_n \top$, then we knew we could propagate $\neg l \rightsquigarrow_{\_} \top$, effectively preventing the bad decision. In the semantic case, assignements are more generic, for instance the decision could be $x \mapsto_n 42$, in which case, forbidding a single value for x would be very inefficient and probably not lead to an algorithm that terminates. Instead, we will choose one of the semantic propagations in the conflict, e.g. $l \rightsquigarrow_n \top$, and force the next decision of the solver to be its negation: $\neg l \mapsto_n \top$, as this prevents not only the specific value that was responsible for the conflict, but in general also prevents a lot of other values. For instance, a decision $x \mapsto_n 42$, may lead to a propagation $x > 13 \rightsquigarrow_n \top$, in which case after analyzing, instead of adding $x \neq 42$, we would decide $\neg(x > 13) \mapsto_n \top$, thus eliminating for x all values above 13.

In order to achieve that, semantic decisions which did not play a role in the conflict are simply ignored by rule Analyze-semantic-decision, and semantic propagations are kept in the conflict clause by rule Analyze-semantic-propagation (that is, semantic propagations are not removed from the analyzed clause). Notice that this is why we need to store the conflict level in rule Conflict-Mcsat; indeed the Analyze-semantic-decision removes semantic decisions (including those present in the conflict clause), which means that we cannot access the level of these semantic propagations later, particularly when applying future instances of Analyze-semantic-decision. Finally, when we reach the semantic decision that created the conflict, rule Conflict-Mcsat gathers all the semantic propagations that derive from that decision (there may be more than one), choose one of them, and decide its negation, thus progressing since we cannot decide on the same value that created the conflict (among many others).

1.3.3 Examples

Examples are given in Section 2.1.3.1 and 2.1.3.2, which also explains how McSat theories work in practice.

1.3.4 Theory Requirements Combinations

Soundness Taking into account assignments means that theories for McSat have to ensure different invariants compared to theories for SMT solvers. The conditions for raising conflicts are different: in an SMT, the theory raises a conflict (using Conflict-smt) as soon as the set of formulas decided and/or propagated is inconsistent. In an McSat solver, a theory keeps track of potential values for each assignable term, refining this set for each new formulas which is propagated or decided. A conflict is then raised (using Conflict-mcsat), as soon as one term has no potential value left (i.e. its set of potential values is empty).

Additionally, a theory has to produce values in order for terms to be assigned using Decide-semantic. These semantic decisions must ensure the trail stays coherent. Concretely, this means that the new assignment should not allow formulas that are true in the trail to be evaluated to $\bot$.

Completeness In order for the mSAT algorithm to be complete, it needs to assign "enough" terms so that, if the algorithm ends in a SAT state (i.e. no inference rule is applicable, and the empty clause has not been deduced), the trail t defines an abstract partial interpretation that effectively evaluates all terms occuring in the problem (or else, a theory could simply not assign any term, and the algorithm would end quickly without doing much work).

$$\textsc{Decide-Semantic}\ \frac{\text{Solve}(\mathbb{S}, t)}{\text{Solve}(\mathbb{S}, t :: a \mapsto_n v)} \quad \begin{array}{l} \neg(a \text{ occurs in } t) \\ n = \text{max\_level}(t) + 1 \\ t :: a \mapsto_n v \text{ is coherent} \end{array}$$

$$\textsc{Propagate-Semantic}\ \frac{\text{Solve}(\mathbb{S}, t)}{\text{Solve}(\mathbb{S}, t :: a \rightsquigarrow_n \top)} \quad \begin{array}{l} [\![a]\!]_{\sigma_t} = \top \\ n = \text{max\_level}(t) \end{array}$$

$$\textsc{Conflict-Mcsat}\ \frac{\text{Solve}(\mathbb{S}, t)}{\text{Analyze}(\mathbb{S}, t, C^m)} \quad \begin{array}{l} \mathcal{T} \vdash C \\ t \vdash \neg C \\ m = \text{max\_level}_t(C) \end{array}$$

$$\textsc{Analyze-Semantic-decision}\ \frac{\text{Analyze}(\mathbb{S}, t :: a \mapsto_n v, C^m)}{\text{Analyze}(\mathbb{S}, t, C^m)} \quad n > m$$

$$\textsc{Analyze-Semantic-propagation}\ \frac{\text{Analyze}(\mathbb{S}, t :: a \rightsquigarrow_n \top, C^m)}{\text{Analyze}(\mathbb{S}, t, C^m)} \quad n > m$$

$$\textsc{Learn-McSat}\ \frac{\text{Analyze}(\mathbb{S}, t :: u \mapsto_n v, C^n)}{\text{Solve}(\mathbb{S} \cup \{C\}, t :: l \mapsto_n \top)} \quad \begin{array}{l} C = l \vee C' \\ \neg(l \text{ occurs in } t) \end{array}$$

Figure 1.3: McSat specific inference rules

While there are certainly many ways to ensure just that (particularly if the theory can decide which terms to assign depending on the previous assignments[1]), I chose to use a much simpler static and syntactic criterion. The idea is to assign only variables and terms whose head symbol is a non-interpreted function symbol. For instance, in that criterion, assuming the input problem is $x+f(y+z) < 0$, the terms to assign would always be x, y, z, and $f(y+z)$, because $y+z$ can simply be evaluated, once y and z are assigned, and in turn $x+f(y+z)$ (as well as $x+f(y+z) < 0$) can also be evaluated once x and $f(y+z)$ are assigned. Note that it would be perfectly acceptable to also assign $y+z$ (since, as can be seen in that particular example, what might be of interest is the value of the sum, rather than the values of the variables), but that would require performing constraints propagation in the theory. In order to keep theories as simple as possible, I instead chose to only assign variables and uninterpreted terms (i.e. terms whose head symbol is uninterpreted), and then do a bottom-up evaluation of terms.

Usual Structure of Theories In that setting, theories tend to have the following structure: each term to be assigned[2] is mapped to a set of possible values (initially set to the whole domain). Then, each formula that is decided or propagated is "watched", until all its assignable terms except one are assigned. At that point, the formula can be seen as a unit formula (by reference to unit clauses), which can be used to refine the set of possible values for the last unassigned term. For instance, take the formula $x + f(y + z) < 0$; its watched terms are x and $f(y + z)$, so if $f(y + z)$ is at one point assigned to 2, then we could refine the set of potential values for x to remove all values superior or equal to -2.

Values for assignments are then taken from the set of potential values of the corresponding term. Conflicts are triggered as soon as one set of potential values becomes empty. This happens when one, or more (usually 2), formulas have generated constraints on a term t that are incompatible. A formula implied by these incompatible constraints can then be created by "eliminating" the term t from these contraints, and the resulting implication can be used as a conflict clause. For

[1] This would be very interesting in the case of terms containing "if ... then ... else ..." constructions, in order to not assign terms that only appear in an irrelevant branch.

[2] These terms are easy to identify statically at the beginning of the algorithm.

instance, let us consider the following trail $[x+y \geq 43 \mapsto_1 \top :: x-y \leq 13 \mapsto_2 \top :: y \mapsto_3 1]$. There is no remaining potential value for x: indeed since y is assigned to 1, $x+y \geq 43$ constrains x to be greater or equal than 42, and $x-y \leq 13$ constrains x to be lesser or equal than 14. We can then use Fourier-Motzkin elimination to deduce: $x+y \geq 43 \wedge x-y \leq 13 \rightarrow 2*y \geq 30$[1]. It is a suitable conflict clause: $\neg(x+y \geq 43) \vee \neg(x-y \leq 13) \vee (2*y \geq 30)$, because $\neg(x+y \geq 43)$ and $\neg(x-y \leq 13)$ are already false in the trail, and $2*y \geq 30$ can be evaluated to false given that y is assigned to 1.

Combination of Theories As can be seen in the presentation both for SMT and McSat, the algorithms usually consider a single theory. However, we often want to combine more than one theory, for instance in order to reason both about arithmetic and uninterpreted functions and predicates, or also use arrays and/or bitvectors, etc. For SMT solvers, there exist frameworks which allow us to combine some theories, but in the context of McSat, using such frameworks would require to implement SMT theories, which have more constraints than McSat theories. For instance the Nelson-Oppen [67] framework requires from the theories to only share the equality symbol but no other function symbol, and to propagate back to the combinator all equalities that they can deduce. There are very recent work on such framework for McSat [23], but this is not a problem, as the requirements for McSat theories already allow for easy combinations, indeed the assignment mechanisms already looks a lot like the Nelson-Oppen framework: assignments allow us to deduce equalities by comparing values, but also to deduce disequalities. While this is not the topic of this thesis, it has been convenient to not have to implement one of the complex SMT theory combining framework.

1.4 ArchSAT

This section provides a brief overview of ArchSAT from a technical point of view. The rest of this thesis describes original work I did during my thesis and integrated into ArchSAT.

First I present mSAT, a library for creating SAT, SMT and McSat solvers, that I developped in order to keep an abstraction layer between the implementation of the McSat solver in ArchSAT, and the various theories and experiments I performed during my thesis. This helped identify whether the new algorithms I was testing needed modifying the core McSat algorithm, or if they could fit into the already existing framework.

1.4.1 mSAT: A SAT Library

mSAT [30] is an OCaml library available at: `https://github.com/Gbury/mSAT`. See Appendix A for a poster about mSAT that was presented at the OCaml workshop.

As was apparent in the earlier descriptions, the SAT, SMT, and McSat algorithm all share a very consequent part of their internal mechanisms. More precisely, SAT is actually a subset of SMT, which itself is a subset of McSat. This is easy to see: SMT and McSat both extend the inference rules of the SAT algorithm; furthermore the Conflict-mcsat rule is the same as the Conflict-smt, except that it records the maximum level of literals in the conflict clause. This inclusion between the several algorithms was the motivation for also providing SAT and SMT solver facilities when implementing McSat as a library.

Additionally, the SMT and McSat algorithms are actually modular with regards to the theory: any theory that respects some constraints (or, in terms of programming, that implements an interface) can be used to produce an SMT or McSat solver. Furthermore, the actual representation of litterals that the theory uses does not matter much, since only the theory has to inspect these terms: the inference rules presented only need to compare literals, and build negations of literals. This is reflected in the interface provided by the library, which exposes OCaml functors that take as argument a representation of terms, and a theory implementation, and returns an SMT or McSat solver.

[1] This tautology can be obtained by subtracting $x-y \leq 13$ from $x+y \geq 43$, which yields $x+y-(x-y) \geq 43-13 \equiv 2*y \geq 30$.

Compared to other implementations, let us first remark that mSAT is the first implementation to offer a functorized SMT (and McSat) solver. Indeed, other implementations either only provide the SAT algorithm, or provide a full SMT solver, without giving the possibility of completely implementing one's theory. While there are quite a few implementations of SAT or SMT solvers[1] in C or C++, there are far less available in OCaml:

- A few libraries offer bindings to pure SAT solvers written in C or C++: minisat[2], sattools[3], ocaml-sat-solvers[4].
- Some libraries offers full SMT solving facilites: Alt-Ergo Zero[5], yices2, and z3; of which only Alt-Ergo Zero is pure OCaml.
- Alt-Ergo is a pure ocaml SMT solvers, but only provides an executable.

Note that pure SAT solver implementations are not suited to build SMT solvers, since they usually do not provide the necessary hooks (or callbacks) needed to use incremental and efficient theories. As such, one often needs to write a complete SAT implementation when writing an SMT solver. mSAT aims at lessening this burden by allowing to simply write a theory, and plug it in the SMT functor provided.

In matters of performance, mSAT compares reasonably well with existing solutions available in OCaml. It is about 10 times slower than the OCaml bindings to the extremely optimised C solvers, and significantly faster than Alt-Ergo Zero (see the poster in Appendix A, or the github repository `https://github.com/Gbury/sat-bench` for more precise measurements). Some experiments and optimizations have managed to reduce the gap with minisat, reducing the 10x factor to 2x or 3x, though they have not yet been merged in the upstream version of mSAT. In any case, the current performances of mSAT are satisfactory enough: indeed, the main focus of my work has been first-order reasoning, in which it is the theory reasoning that tends to dominate the solving time. In that context, the performance of mSAT on pure SAT problems do not need to be exceptional.

Lastly, one feature of mSAT that most (if not all) other implementations lack is the generation of formal proofs when the solver reaches UNSAT. When an empty clause is reached[6], a proof of that clause can be generated, as a resolution tree whose leaves are lemmas provided by the theory. For more information on the production of formal proofs, see Chapter 4.

1.4.2 Overview of ArchSAT

ArchSAT is a full implementation of an McSat solver, available at `https://github.com/Gbury/archsat`. I developed it as the core method of evaluating the practical feasibility of algorithms I realised during my thesis.

It is written in OCaml, using mSAT, and aims at solving first-order problems. ArchSAT uses the dolmen (see Chapter 6.1 for more information) library to accept a wide range of input syntaxes. More generally, a number of tips and tricks were used during the development in order to get a concise yet expressive code; these are detailed in Chapter 6. ArchSAT is around 40k lines of code[7], not counting mSAT (about 8k lines of code), dolmen (6.5k lines of code), and other external libraries.

Contrary to most provers[8], ArchSAT uses terms that are strongly typed in a first-order polymorphic type system à la ML, see [19] for a full description of the type system. This allows for a very natural and concise description of theories as, for instance, set theory axioms can be polymorphic over the type of values contained in sets. The type-checking algorithm used is very

[1] mSAT being much more recent, it does not yet have many dedicated implementations, only extensions built on top of already existing SMT solvers.

[2] `https://github.com/c-cube/ocaml-minisat`

[3] `https://github.com/ujamjar/sattools`

[4] `https://github.com/tcsprojects/ocaml-sat-solvers`

[5] Which is the core of Alt-Ergo factored out to be standalone.

[6] Actually, any clause reached by the solver can be proved, but it is less common to need a proof of a clause other than the empty clause.

[7] Blank lines, comments and documentation included.

[8] Which tends to either use untyped terms, of strictly first-order typed terms.

close to what is described in [31]. In the rest of the manuscript, we will thus consider that we are using first-order polymorphic terms, though that does not change much from using simply typed first-order terms. Note, however, using simply typed terms rather than untyped ones does increase performance of proof search, as illustrated in [31].

ArchSAT also differs from other traditional SMT provers in that it does not perform CNF transformation before solving. Instead, it uses a lazy CNF conversion, inspired from the tableau proof search method, that is implemented as an McSat theory, and which is detailed in Chapter 2. Following the inspiration from the tableau method, quantified formulas are handled by generating meta-variables and performing unification. This is respectively explained in details in Section 2.3, and Section 2.3.2.

Another feature of ArchSAT is the handling of rewrite rules. Indeed transforming axioms into rewrite rules allows for better performances of automated theorem provers, as show in [28,32,34,46]. ArchSAT thus uses deduction modulo to mainly rewrite formulas (along with their subterms). In order to do so, ArchSAT distinguishes three different uses of rewrite rules:

- Regular Rewrite rules that appear at the top-level of a problem are used to normalize ground atomic formulas. These rewrite rules often axiomatize a theory, such as the set theory of the B method for instance. This is presented in Section 3.2.

- Conditional rewrite rules are handled using a trigger-like mechanism, but using McSat's notion of evaluation to only instantiate rules whose guards are satisfied. This trigger-like mechanism is also used to handle rewrite rules that are not at the root of a problem, that is, rules whose truth value may change during proof search. This is detailed in Section 3.3.

- Rewrite rules are also integrated into the unification algorithm used for finding instantiations. This is explained in Section 3.4.

Finally, whenever ArchSAT finds a proof for an input problem, it is able to generate a complete formal proof of that problem. This formal proof is then exportable to a format that can be checked externally, for instance by the Coq assistant prover [9]. The details of the formal proof structure and the subtleties of its generation are documented in Chapter 4.

Chapter 2

First-Order Reasoning in Archsat

This chapter presents how generic first-order reasoning is handled in ArchSAT, and is divided into 4 sections. First it explains how logical connectives are handled without using a CNF transformation prior to solving (as is usually done in most SMT provers), but rather using a lazy unfolding of logical connectives by adding new clauses. Then the treatment of equalities and uninterpreted functions is explained: instead of relying on a congruence closure algorithm like most SMT solvers, ArchSAT splits the reasoning into two distinct theories: one for purely equational reasoning using a standard union-find algorithm, and another for uninterpreted functions and predicates that makes use of the assignments in McSat to ensure that each uninterpreted function (or predicate) can be given an adequate value. Finally, ArchSAT treats quantified formula much like in tableau theory, by generating fresh constants (or skolem symbols) for existentially quantified formulas, and meta-variables (or free variables) for universally quantified formulas. Meta-variables are then instantiated following the substitutions returned by unification of predicates.

2.1 Pure Ground Reasoning in Archsat

2.1.1 Terms, Formulas and Blackboxes

This section describes the structure of first-order polymorphic terms that are used in ArchSAT. Given a set of variables $\mathcal{V}$, and a set of function symbols $\mathcal{F}$, we can build the set of types $\mathcal{T}_y$, the set of terms $\mathcal{T}$, and the set of formulas $\mathcal{F} = F, G, \ldots$ as described in Figure 2.1. In order to correctly express the type of polymorphic functions, we also define the set of type signatures[2] , which represent the type of constant symbols.

Most syntax constructions that appear are relatively standard but for a few cases. First, type variables can be quantified in formulas. Together with that, term applications take some type arguments before the term arguments. This is what allows polymorphic function symbols to work (and particularly allows to use the quantified types). Second, the syntax includes meta-variables that are used during proof search, and written A_F for type meta-variables, and X_F for term meta-variables. Meta-variables are tied to a formula that introduced them (denoted by F), which can either be a universally quantified formula $F = \forall x : t_y.G$, or the negation of an existentially quantified formula: $F = \neg\exists x : t_y.G$, where G is an arbitrary formula. Lastly, type signatures represent the type of constants. Those can either have a regular type, or a function type, taking a n-uplet of typed arguments[3], or they can be polymorphic in some type variable α.

Some standard notations that help readability will be used throughout this manuscript :

- Variables in $\mathcal{V}$ will usually be written using $\alpha, \beta, \ldots$ for type variables, and $x, y, z, \ldots$ for term variables.

[2] Note that type signatures are not used in terms or formulas. In practice they appear in symbol declarations, since only constant symbols are assigned type signatures (variables are typed using regular types).

[3] In first-order logic, all applications are total, hence the representation of multiple arguments with n-uplets rather than curried arrows.

$$
\begin{array}{rlr}
\mathcal{T}_y = & \alpha, \beta, \ldots & \text{(type variables)} \\
\mid & A_F & \text{(type meta-variable)} \\
\mid & f(t_{y_1}, t_{y_2}, \ldots) & \text{(type constructor application)} \\
\mathcal{S}_y = & t_y & \text{(regular type)} \\
\mid & t_{y_1} * t_{y_2} * \ldots * t_{y_n} \rightarrow t_{y_{n+1}} & \text{(function type)} \\
\mid & \Pi\alpha.\ s_y & \text{(polymorphic signature)} \\
\mathcal{T} = & x, y, z, \ldots & \text{(term variables)} \\
\mid & X_F & \text{(term meta-variable)} \\
\mid & f(t_{y_1}, t_{y_2}, \ldots; t, u, \ldots) & \text{(term application)} \\
\mathcal{F} = & \top \quad \mid \quad \bot & \text{(True and False constants)} \\
\mid & t & \text{(Term predicate as atomic formula)} \\
\mid & t = u & \text{(Equality)} \\
\mid & \neg F & \text{(Negation)} \\
\mid & F \wedge G & \text{(Conjunction)} \\
\mid & F \vee G & \text{(Disjunction)} \\
\mid & F \Rightarrow G & \text{(implication)} \\
\mid & F \Leftrightarrow G & \text{(Equivalence)} \\
\mid & \forall x : \mathsf{Type}.F & \text{(Type universal quantification)} \\
\mid & \exists x : \mathsf{Type}.F & \text{(Type existential quantification)} \\
\mid & \forall x : t_y.F & \text{(Term universal quantification)} \\
\mid & \exists x : t_y.F & \text{(Term existential quantification)}
\end{array}
$$

Figure 2.1: Syntax for the Types, Type Signaturs, Terms, and Formulas in ArchSAT

- Function symbols will usually be written $a, b, c, \ldots, f, g, h, \ldots$. Function symbols that takes no argument will be written without parenthesis: a instead of $a()$.
- Terms will often be written using $t, u, v, \ldots$
- Successive use of the same logical connective may be concatenated according to associativity[1], so as to write $p \wedge q \wedge r \wedge s$ instead of $(((p \wedge q) \wedge r) \wedge s)$. More generally, proof search will not distinguish $(p \wedge q) \wedge r$ from $p \wedge (q \wedge r)$, as both will be considered as the formula $p \wedge q \wedge r$. See Chapter 4 for more details about associativity of logical connectives in formal proofs.
- Successive quantifications may be concatenated so as to write $\forall x_1 : t_{y_1}, x_2 : t_{y_2}, \ldots, x_n : t_{y_n}.F$ instead of $\forall x_1 : t_{y_1}.\forall x_2 : t_{y_2}.\ldots\forall x_n : t_{y_n}.F$.
- In examples, meta-variables such as X_F will often be written X when there is no confusion as to which formula generated the meta-variable.

The type system used is mostly the same as the one presented in [31], itself mostly following the one introduced in [19], and adapted to meta-variables and epsilon terms.

A substitution is a total function from variables to terms such that there is a finite set of variables that are not bound to themselves. This set of variables that are not bound to themselves is called the domain. The co-domain of a substitution is the set of variables occurring in terms in the

[1] Note that for proof search, the distinction between left and right associative logical connectives does not matter since we instead manipulate lists of formulas.

image of the domain of the substitution. In this thesis, we will consider idempotent substitutions, i.e. substitutions for which the domain and co-domain have an empty intersection.

A boxed formula is of the form $[\![F]\!]$, where F is a formula that does not start with a negation. In the rest of this thesis, we will use boxed formula as atoms for the McSat solver. This means that a literal is now either a boxed formula, or the negation of a boxed formula, and a clause a disjunction of these literals. We thus have two distinct notions of negation: the negation of formulas as described in Figure 2.1, and the negation of atoms as understood in the inference rules for the McSat algorithm. This is the reason why negations are not allowed at the top of a formula in a box: all reasoning about negation will be done by the inference rules of the McSat algorithm. In order to do so, any negation at the top of a formula that is meant to be boxed will "escape" the box: $[\![\neg F]\!]$ will automatically be normalized to $\neg[\![F]\!]$, and $[\![\neg\neg F]\!]$ into $[\![F]\!]$. Clauses are thus disjunctions of (possibly negated) blackboxes, for instance: $\neg[\![p \vee q]\!] \vee [\![p]\!] \vee [\![q]\!]$. As was apparent in the presentation of the SAT and McSat algorithm, the inference rules do not need to look "inside" the blackboxes, only the theories need to look inside.

2.1.2 Unfolding Propositional Connectives using Tableaux Rules

We can now implement the Tableau proof search method as a regular theory for ArchSAT[1]. Tableau proof search in ArchSAT is heavily inspired by the tableau prover Zenon [24].

The main idea is to encode the propositional calculus of logical connectives into the clausal calculus implemented by the inference rules of the McSat algorithm. The way it is done is that, each time the inference rules decide or propagate a boxed formula $[\![F]\!]$, we use the rule Learn to add some clauses that "unfold" the top logical connective of F. For that, we define a function $\lfloor . \rfloor$ in Figure 2.2, that maps boxed formulas to sets of clauses. Each time the inference rules decide or propagate a boxed formula $[\![F]\!]$, the rule Learn is used for each clause in the set $\lfloor [\![F]\!] \rfloor$. In order to avoid having the same clause learned multiple times, this operation is done once per formula F when the positive boxed formula $[\![F]\!]$ is assigned to be true, and once when the negative boxed formula $\neg[\![F]\!]$ is assigned to be true.

More generally, arbitrary tableau inference rules can be translated into clauses. First, let us suppose we have a translation into clauses of tableau rules handling conjunctions.Now consider the following generic tableau rule :

$$\frac{P_1 \qquad \dots \qquad P_n}{f_{1,1}, f_{1,2} \dots \quad \mid \quad f_{2,1}, \dots \quad \mid \quad \dots \quad \mid \quad f_{m,1} \dots}$$

This rule can be translated into a clause :

$$\neg[\![P_1]\!] \vee \dots \vee \neg[\![P_n]\!] \vee [\![f_{1,1} \wedge f_{1,2} \wedge \dots]\!] \vee [\![f_{2,1} \wedge \dots]\!] \vee \dots \vee [\![f_{m,1} \wedge \dots]\!]$$

This translated clause is then to be added to the SAT solver once all the formulas $P_1, \dots, P_n$ are true, and naturally handles branching rules by leaving the SAT solver decide on one of the blackboxes $[\![f_{1,1} \wedge f_{1,2} \wedge \dots]\!], [\![f_{2,1} \wedge \dots]\!], \dots, [\![f_{m,1} \wedge \dots]\!]$, and then backtrack whenever is needed.

A special case for rules with exactly one branch (but possibly multiple formulas in that branch) is needed to correctly handle conjunction. Given a tableau rule:

$$\frac{P_1 \qquad \dots \qquad P_n}{f_{1,1}, f_{1,2} \cdots f_{1,k}}$$

The rule can be translated into k clauses :

$$\begin{aligned} \neg[\![P_1]\!] \vee \dots \vee \neg[\![P_n]\!] \vee [\![f_{1,1}]\!] \\ \neg[\![P_1]\!] \vee \dots \vee \neg[\![P_n]\!] \vee [\![f_{1,2}]\!] \\ \dots \\ \neg[\![P_1]\!] \vee \dots \vee \neg[\![P_n]\!] \vee [\![f_{1,k}]\!] \end{aligned}$$

[1] Meaning that it does not need to be different from any other theories, contrary to CNF conversion which is fundamentally different from arithmetic theories for instance.

Analytic Rules

$$(\alpha_\wedge)\quad \lfloor \llbracket P \wedge Q \rrbracket \rfloor = \left\{ \begin{array}{l} \neg\llbracket P \wedge Q\rrbracket \vee \llbracket P\rrbracket \\ \neg\llbracket P \wedge Q\rrbracket \vee \llbracket Q\rrbracket \end{array}\right. \qquad (\beta_{\neg\wedge})\quad \lfloor \neg\llbracket P \wedge Q \rrbracket \rfloor = \llbracket P \wedge Q\rrbracket \vee \neg\llbracket P\rrbracket \vee \neg\llbracket Q\rrbracket$$

$$(\beta_\vee)\quad \lfloor \llbracket P \vee Q \rrbracket \rfloor = \neg\llbracket P \vee Q\rrbracket \vee \llbracket P\rrbracket \vee \llbracket Q\rrbracket \qquad (\alpha_{\neg\vee})\quad \lfloor \neg\llbracket P \vee Q \rrbracket \rfloor = \left\{ \begin{array}{l} \llbracket P \vee Q\rrbracket \vee \neg\llbracket P\rrbracket \\ \llbracket P \vee Q\rrbracket \vee \neg\llbracket Q\rrbracket \end{array}\right.$$

$$(\beta_\Rightarrow)\quad \lfloor \llbracket P \Rightarrow Q \rrbracket \rfloor = \neg\llbracket P \Rightarrow Q\rrbracket \vee \neg\llbracket P\rrbracket \vee \llbracket Q\rrbracket \qquad (\alpha_{\neg\Rightarrow})\quad \lfloor \neg\llbracket P \Rightarrow Q \rrbracket \rfloor = \left\{ \begin{array}{l} \llbracket P \Rightarrow Q\rrbracket \vee \llbracket P\rrbracket \\ \llbracket P \Rightarrow Q\rrbracket \vee \neg\llbracket Q\rrbracket \end{array}\right.$$

$$(\beta_\Rightarrow)\quad \lfloor \llbracket P \Leftrightarrow Q \rrbracket \rfloor = \left\{ \begin{array}{l} \neg\llbracket P \Leftrightarrow Q\rrbracket \vee \llbracket P \Rightarrow Q\rrbracket \\ \neg\llbracket P \Leftrightarrow Q\rrbracket \vee \llbracket Q \Rightarrow P\rrbracket \end{array}\right. \qquad (\beta_{\neg\Rightarrow})\quad \lfloor \neg\llbracket P \Leftrightarrow Q \rrbracket \rfloor = \begin{array}{l} \llbracket P \Leftrightarrow Q\rrbracket \\ \vee\neg\llbracket P \Rightarrow Q\rrbracket \\ \vee\neg\llbracket Q \Rightarrow P\rrbracket \end{array}$$

δ-Rules

$$\begin{array}{lcll} \lfloor \llbracket \exists x.P(x) \rrbracket \rfloor & = & \neg\llbracket \exists x.P(x)\rrbracket \vee \llbracket P(c)\rrbracket & (\delta_\exists)(\text{c is a fresh constant}) \\ \lfloor \neg\llbracket \forall x.P(x) \rrbracket \rfloor & = & \llbracket \forall x.P(x)\rrbracket \vee \neg\llbracket P(c)\rrbracket & (\delta_{\neg\forall})(\text{c is a fresh constant}) \end{array}$$

γ-Rules

$$\begin{array}{lcll} \lfloor \llbracket \forall x.P(x) \rrbracket \rfloor & = & \neg\llbracket \forall x.P(x)\rrbracket \vee \llbracket P(X_{\forall x.P(x)})\rrbracket & (\gamma_{\forall M}) \\ \lfloor \neg\llbracket \exists x.P(x) \rrbracket \rfloor & = & \llbracket \exists x.P(x)\rrbracket \vee \neg\llbracket P(X_{\neg\exists x.P(x)})\rrbracket & (\gamma_{\neg\exists M}) \\ \lfloor \llbracket \forall x.P(x) \rrbracket \rfloor & = & \neg\llbracket \forall x.P(x)\rrbracket \vee \llbracket P(t)\rrbracket & (\gamma_{\forall\text{inst}}) \\ \lfloor \neg\llbracket \exists x.P(x) \rrbracket \rfloor & = & \llbracket \exists x.P(x)\rrbracket \vee \neg\llbracket P(t)\rrbracket & (\gamma_{\neg\exists\text{inst}}) \end{array}$$

Figure 2.2: Rules of Tableau Theory

Each of these clauses will propagate one element of the conjunction, thus behaving in exactly the same way as a regular tableau prover would.

Logical connectives. This section focuses on the unfolding of logical connectives (rules α and β). A detailed explanation of how quantified formulas are handled can be found in Section 2.2 for existentially quantified formulas, and in Sections 2.3 and 2.3.2 for universally quantified formulas.

A crucial point to remember is that rule Learn is restricted to clauses that can be proven in the considered theory. This is why when, for instance, the inference rules decide or propagate $\llbracket P \wedge Q\rrbracket$ to be true, we cannot simply learn that $\llbracket P\rrbracket$ and $\llbracket Q\rrbracket$ are true. Indeed, the assignment of $\llbracket P \wedge Q\rrbracket$ to true is susceptible to be backtracked, for instance if it was a decision. In order to link the assignment of $\llbracket P\rrbracket$ and $\llbracket Q\rrbracket$ to that of $\llbracket P \wedge Q\rrbracket$, the simplest is to encode it in clauses. By seeing clauses as implications, it is easy to then understand the two clauses to learn: $\llbracket P \wedge Q\rrbracket \Rightarrow \llbracket P\rrbracket \equiv \neg\llbracket P \wedge Q\rrbracket \vee \llbracket P\rrbracket$, and $\llbracket P \wedge Q\rrbracket \Rightarrow \llbracket Q\rrbracket \equiv \neg\llbracket P \wedge Q\rrbracket \vee \llbracket Q\rrbracket$.

In essence, this process simulates a lazy CNF conversion with named intermediate formulas[1], step-by-step. The main reason why this process is done lazily, that is one step at a time, instead of recursively applying the $\lfloor . \rfloor$ function on every input formula is to better handle quantified formulas. Indeed, the tableau theory waits for a formula to be assigned, and then unfolds it according to $\lfloor . \rfloor$. This allows us to only translate either the positive or the negative version of a formula, since we know its truth value. If we did not know its truth value (for instance if we wanted to perform this conversion recursively), then we would need to translate both the positive and negative version of the formula in order to be complete. This becomes a problem for quantified formulas. Indeed, quantified formulas are handled by substituting the bound variable with either

[1] Intermediary formulas are named because traditional SAT and SMT solvers only handle atomic formulas. Thus to refer to any non-atomic formula F, a new atomic proposition equivalent to F must be created, essentially giving a name to the formula F.

an arbitrary term, or a meta-variable, resulting in a new formula that is not a sub-formula of the input formula, contrary to what happens for ordinary logical connectives. Thus translating formulas using $\lfloor . \rfloor$ non-lazily would generate a lot of possibly useless formulas that would parasite the proof search. This is why we wait for a formula to be assigned before unfolding its top-level connective. Performing translation lazily also allows us to possibly find a model or a proof without translating all formulas in a problem, and thus achieve better performances.

This theory works by only using the inference rule Learn described in Section 1.2.4. As a consequence it is a valid theory for both SMT and McSat solvers. Furthermore, since it only interacts with formulas, it should be trivial to integrate it into any theory combination framework used in an SMT solver. This means that it can be effectively used as a drop-in replacement for CNF conversion.

2.1.2.1 Example

Let us try and prove the formula $F \equiv (A \vee C) \Rightarrow (A \vee B \vee C)$. The proof search starts with a single clause containing a single boxed formula, which is the negation of the goal that must be proven:

$$\mathcal{C}_0 \equiv \neg [\![(A \vee C) \Rightarrow (A \vee B \vee C)]\!]$$

The inference rules directly propagate: $[\![(A \vee C) \Rightarrow (A \vee B \vee C)]\!] \rightsquigarrow_{\mathcal{C}_0} \bot$. Upon propagation, the tableau theory generates and adds the two clauses in $\lfloor \neg [\![(A \vee C) \Rightarrow (A \vee B \vee C)]\!] \rfloor$ using rule $\alpha_{\neg\Rightarrow}$:

$$\begin{aligned} \mathcal{C}_1 &\equiv [\![(A \vee C) \Rightarrow (A \vee B \vee C)]\!] \vee [\![A \vee C]\!] \\ \mathcal{C}_2 &\equiv [\![(A \vee C) \Rightarrow (A \vee B \vee C)]\!] \vee \neg [\![A \vee B \vee C]\!] \end{aligned}$$

With these two clauses, the inference rules of the McSat algorithm can now propagate $[\![A \vee C]\!] \rightsquigarrow_{\mathcal{C}_1} \top$ and $[\![A \vee B \vee C]\!] \rightsquigarrow_{\mathcal{C}_2} \bot$. When the formula $[\![A \vee C]\!]$ becomes true, the tableau theory generates the clause in $\lfloor [\![A \vee C]\!] \rfloor$ using rule $\beta_\vee$:

$$\mathcal{C}_3 \equiv \neg [\![A \vee C]\!] \vee [\![A]\!] \vee [\![C]\!]$$

At the same time, the formula $[\![A \vee B \vee C]\!]$ became false, so the tableau theory also generates the clause in $\lfloor \neg [\![A \vee B \vee C]\!] \rfloor$:

$$\begin{aligned} \mathcal{C}_4 &\equiv [\![A \vee B \vee C]\!] \vee \neg [\![A]\!] \\ \mathcal{C}_5 &\equiv [\![A \vee B \vee C]\!] \vee \neg [\![B]\!] \\ \mathcal{C}_6 &\equiv [\![A \vee B \vee C]\!] \vee \neg [\![C]\!] \end{aligned}$$

With these new clauses, the inference rules now allow us to propagate $[\![A]\!] \rightsquigarrow_{\mathcal{C}_4} \bot$, $[\![B]\!] \rightsquigarrow_{\mathcal{C}_5} \bot$, and $[\![C]\!] \rightsquigarrow_{\mathcal{C}_6} \bot$. At that point, a conflict is detected in clause $\mathcal{C}_3$. The analyze phase then inspects the trail, starting from the most recent assignment :

1. Analyze propagation $[\![C]\!] \rightsquigarrow_{\mathcal{C}_6} \bot$. Since $[\![C]\!]$ occurs in the conflcit clause $\mathcal{C}_3$, a resolution is performed between $\mathcal{C}_3$ and $\mathcal{C}_6$, to get $D \equiv [\![A \vee B \vee C]\!] \vee [\![A]\!] \vee \neg [\![A \vee C]\!]$

2. Analyze propagation $[\![B]\!] \rightsquigarrow_{\mathcal{C}_5} \bot$. Since $[\![B]\!]$ does not occur in the analyzed clause D, nothing needs to be done.

3. Analyze $[\![A]\!] \rightsquigarrow_{\mathcal{C}_4} \bot$. Since $[\![A]\!]$ occurs in the analyzed clause D, a resolution is performed between D and $\mathcal{C}_4$ to get $D \equiv [\![A \vee B \vee C]\!] \vee \neg [\![A \vee C]\!]$

4. Analyze $[\![A \lor B \lor C]\!] \rightsquigarrow_{\mathcal{C}_2} \bot$. Again, a resolution is performed between D and $\mathcal{C}_2$, and results in $D \equiv [\![F]\!] \lor \neg[\![A \lor C]\!]$
5. Analyze $[\![A \lor C]\!] \rightsquigarrow_{\mathcal{C}_1} \top$. A resolution is performed between D and $\mathcal{C}_1$, ad results in $D \equiv [\![F]\!]$.
6. Finally, the first propagation $[\![F]\!] \rightsquigarrow_{\mathcal{C}_0} \bot$ is analyzed, and by performing a resolution between D and $\mathcal{C}_0$, we get the empty clause.

In practice, the implementation does not need to perform this analyze phase: any conflict that occurs before any decision is made will always result in the empty clause being reached during the analyze phase, so the solver actually stop right after the conflict is found, and returns UNSAT.

We can thus conclude that the set of clauses $\{\mathcal{C}_0, \ldots, \mathcal{C}_6\}$ is unsatisfiable. Since clauses $\mathcal{C}_1, \ldots, \mathcal{C}_6$ are tautologies that are provable in first-order theory, this means that $\mathcal{C}_0$ is actually unsatisfiable on its own. We can then conclude that the formula F is provable.

An example with splitting. Now, let us consider the following problem :

$$\begin{aligned}
\mathcal{C}_0 &\equiv [\![A \lor B]\!] \\
\mathcal{C}_1 &\equiv [\![A \Rightarrow C]\!] \\
\mathcal{C}_2 &\equiv [\![B \Rightarrow C]\!] \\
\mathcal{C}_3 &\equiv [\![D \lor E]\!] \\
\mathcal{C}_4 &\equiv [\![D \Rightarrow \neg C]\!] \\
\mathcal{C}_5 &\equiv [\![E \Rightarrow \neg C]\!]
\end{aligned}$$

All 6 boxed formulas are directly propagated :

- $[\![A \lor B]\!] \rightsquigarrow_{\mathcal{C}_0} \top$
- $[\![A \Rightarrow C]\!] \rightsquigarrow_{\mathcal{C}_1} \top$
- $[\![B \Rightarrow C]\!] \rightsquigarrow_{\mathcal{C}_2} \top$
- $[\![D \lor E]\!] \rightsquigarrow_{\mathcal{C}_3} \top$
- $[\![D \Rightarrow \neg C]\!] \rightsquigarrow_{\mathcal{C}_4} \top$
- $[\![E \Rightarrow \neg C]\!] \rightsquigarrow_{\mathcal{C}_5} \top$

These propagations prompt the tableau theory to generate and add the following clauses :

$$\begin{aligned}
\mathcal{C}_6 &\equiv \neg[\![A \lor B]\!] \lor [\![A]\!] \lor [\![B]\!] && \beta_\lor \\
\mathcal{C}_7 &\equiv \neg[\![A \Rightarrow C]\!] \lor \neg[\![A]\!] \lor [\![C]\!] && \beta_\Rightarrow \\
\mathcal{C}_8 &\equiv \neg[\![B \Rightarrow C]\!] \lor \neg[\![B]\!] \lor [\![C]\!] && \beta_\Rightarrow \\
\mathcal{C}_9 &\equiv \neg[\![D \lor E]\!] \lor [\![D]\!] \lor [\![E]\!] && \beta_\lor \\
\mathcal{C}_{10} &\equiv \neg[\![D \Rightarrow \neg C]\!] \lor \neg[\![D]\!] \lor \neg[\![C]\!] && \beta_\Rightarrow \\
\mathcal{C}_{11} &\equiv \neg[\![E \Rightarrow \neg C]\!] \lor \neg[\![E]\!] \lor \neg[\![C]\!] && \beta_\Rightarrow
\end{aligned}$$

Interestingly, none of these clause can propagate. In that case, the solver will decide on one of the five undecided literals, which are : $[\![A]\!], [\![B]\!], [\![C]\!], [\![D]\!]$ and $[\![E]\!]$. Let us suppose it decides: $[\![C]\!] \mapsto_1 \top$. That decision allows ArchSAT to perform two propagations:

- $[\![D]\!] \rightsquigarrow_{\mathcal{C}_{10}} \bot$

- $[\![E]\!] \leadsto_{\mathcal{C}_{11}} \bot$

At that point, a conflict is detected on clause $\mathcal{C}_9$, where all 3 literals are false. Conflict analysis then does the following :

- Start with the conflict clause $\mathcal{D} \equiv \mathcal{C}_9 \equiv \neg[\![D \vee E]\!] \vee [\![D]\!] \vee [\![E]\!]$
- Analyze the last element on the trail: $[\![E]\!] \leadsto_{\mathcal{C}_{11}} \bot$. Since $[\![E]\!]$ occurs in $\mathcal{D}$, a resolution is performed between $\mathcal{D}$ and $\mathcal{C}_{11}$, which result in a new conflcit clause $\mathcal{D} \equiv \neg[\![D \vee E]\!] \vee [\![D]\!] \vee \neg[\![E \Rightarrow \neg C]\!] \vee \neg[\![C]\!]$
- Analyze the next element on the trail: $[\![D]\!] \leadsto_{\mathcal{C}_{10}} \bot$. Since $[\![D]\!]$ occurs in $\mathcal{D}$, a resolution is performed between $\mathcal{D}$ and $\mathcal{C}_{10}$, which result in a new conflcit clause $\mathcal{D} \equiv \neg[\![D \vee E]\!] \vee \neg[\![E \Rightarrow \neg C]\!] \vee [\![D \Rightarrow \neg C]\!] \vee \neg[\![C]\!]$
- Analyze the next element on the trail: $[\![C]\!] \mapsto_1 \top$. Rule Learn-Sat can apply, which undoes the decision and uses $\mathcal{C}_{12} \equiv \mathcal{D} \equiv \neg[\![D \vee E]\!] \vee \neg[\![E \Rightarrow \neg C]\!] \vee [\![D \Rightarrow \neg C]\!] \vee \neg[\![C]\!]$ to instead propagate: $[\![C]\!] \leadsto_{\mathcal{C}_{12}} \bot$.

At that point, a very similar processus takes place with clauses $\mathcal{C}_6$, $\mathcal{C}_7$ and $\mathcal{C}_8$. ArchSAT can perform 2 propagations :

- $[\![A]\!] \leadsto_{\mathcal{C}_7} \bot$
- $[\![B]\!] \leadsto_{\mathcal{C}_8} \bot$

These propagation create a conflict in clause $\mathcal{C}_6$, where all 3 literals are false. Since we have reached a conflict before any decision is made (the conflict analysis above backtracked before the decision on $[\![C]\!]$, thus the trail now does not contain any decision), this means that the problem is unsatisfiable.

2.1.3 Congruence Closure without Congruence Closure

This section describes how equality and uninterpreted functions (and predicates) are handled in ArchSAT, relying on McSat to implement them in distinct theories. Indeed, McSat allows us to easily implement equality in its theory, and uninterpreted functions (and predicates) in another, whereas traditional SMT solvers rely on a congruence closure algorithm [69] to do both tasks at the same time. This way of separating equality reasoning from uninterpreted functions and predicates is already present in [42, 58], but is an integral part of ArchSAT, and useful to better understand how assignments work, which will help in Section 3.3.

2.1.3.1 Equality

Simple Equality theory for McSat. Let us first consider the theory of equality. As stated in Section 1.3, the invariants that the theory will try to enforce is to keep for each term the constraints on its potential values. Equalities are quite simple in that regard as the only constraint that derives from an equality $t = t'$ is that when t (resp. t') is assigned to some value v, then t' (resp. t) must then also be assigned to v. Similarly, a disequality $t \neq t'$ implies a constraint on the potential values of t (resp. t') as soon as t' (resp. t) is assigned to a value v: t (resp. t') cannot be assigned to v. There are two cases where these constraints may be incompatible:

- if an equality $t = t'$ and a disequality $t \neq t''$ have been propagated or decided, t' assigned to a value v and t'' also assigned to v, then t has no remaining potential value. We can then create a conflict that will be sent to the McSat algorithm using the tautology: $t = t' \wedge t \neq t'' \Rightarrow t' \neq t''$, which gives the clause $\neg[\![t = t']\!] \vee [\![t = t'']\!] \vee \neg[\![t' = t'']\!]$[1]. This is indeed a conflict clause for McSat: $[\![t = t']\!]$ is true in the trail (since it was propagated or decided), $[\![t = t'']\!]$ is false in the trail for the same reason, and lastly, $[\![t' = t'']\!]$ can be evaluated to true, using the assignments for t' and t''.

[1] This is clearly a tautology as it states the transitivity of equality: it is equivalent to $t = t' \wedge t' = t'' \Rightarrow t = t''$

- if two equalities $t = t'$ and $t = t''$ have been propagated or decided, and t' assigned to v' and t'' to a v'' different from v'. In that case just as for the other case, a conflict can be raised using transitivity of equality: $\neg[\![t = t']\!] \vee \neg[\![t = t'']\!] \vee [\![t' = t'']\!]$. The two equalities $[\![t = t']\!]$ and $[\![t = t'']\!]$ are true in the trail, and $[\![t' = t'']\!]$ evaluates to false because of the assignments.

Now that we have seen when and how conflicts are raised, the other task the theory has to perform is to return adequate values for assignments. Whenever the McSat algorithm asks the equality theory for a value to assign to a term t, either an equality $t = t'$ has already constrained t to be assigned the same value v as t' (and no constraint has forbidden that value, or else a conflict would have been raised as explained in the above paragraph), in which case we return v. Or, in the absence of an equality constraint (i.e. if there are no constraints, or only disequality constraints), the assignment value can be chosen freely: indeed, in first-order logic, we assume that each type is non-empty and we have access to an infinity of distinct constants of that type[1]. In order to not generate too many of these constants, in ArchSAT, we choose to assign to t a value that we will write $\hat{t}$. The only thing that we can do with values such as $\hat{t}$ is to compare them, essentially giving them the same semantics as fresh constants without having to generate such constants. Note that it is done only in cases where the type of t is uninterpreted and thus there is not much information on the values of that type. In cases where the type is interpreted (such as arithmetic), more traditional values such as $1, 42, \frac{3}{8}, \ldots$ would be chosen and assigned by the arithmetic theory rather than the generic equality theory.

This description is enough to specify and implement a theory of equality for McSat. However, it still has the inconvenience of not directly handling long chains of equality: indeed consider a scenario where equalities $t_0 = t_1, t_1 = t_2, \ldots t_n = t_{n+1}$, and disequality $t_0 \neq t_{n+1}$ have been propagated at level 0. If we then try and assign the t_i (in an unspecified order), we will need to raise quite a lot of conflict to get to an unsatisfiable result. Indeed, suppose we assign t_0 to $\hat{t_0}$ (since there is no constraint on it), and then t_2 to $\hat{t_2}$ (also since there is no constraint on it). We then have a conflict because t_1 has to be assigned to both $\hat{t_0}$ and $\hat{t_1}$ which are different constants. We then learn through the conflict that t_0 and t_2 are equal. This process can then repeat with t_0 and t_i, and learn about any equation $t_i = t_j$, before finally learning $t_0 = t_{n+1}$ and concluding that the problem is unsatisfiable. This is quite inefficient both in terms of steps needed to find the result, and in terms of how many clauses have been added which, in situations where all formulas are not at level 0 (and thus proof search continues after finding that the disequality conflicts with the equalities), could slow down the unit propagation mechanism considerably. In order to avoid this problem, ArchSAT uses an union-find [85,86] structure to discover conflicts earlier when possible.

Equality theory in ArchSAT. Formally, consider an extended union-find structure with proof producing capabilities [68,69], i.e. whenever two terms t and u are in the same equivalence class, the union-find can produce a list[2] of terms $x_1, \ldots, x_n$, such that $t = x_1$, $x_1 = x_2$, $x_2 = x_3$, $\ldots$, $x_n = u$ are equalities that have been added to the union find (up to equality symmetry), effectively explaining why two terms are in the same equivalence class. The extension of the union-find structure is that each equivalence class c in the union-find will carry:

- an optional pair of terms (t, v) called its tag, and representing the fact that a term t, which belongs to the equivalence class, has been assigned to v. v will be called the tagged value and t the tagged term of the equivalence class.
- a list of disallowed equivalence class, which are forbidden to be merged with c.

The theory for equality in ArchSAT then does the following:

- As soon as a term t is assigned to a value v, the tag of the equivalence class c of t is set to (t, v). If c has no tags, or the tag is of the form $(\_, v)$, there is no problem setting the new tag, else a conflict is raised (see below).

[1] An axiom may restrict the use of these constants in a valid model, but that does not prevent from using as many of these constants in proof search.

[2] In practice, the algorithm also tries and returns the smallest explanation, in order to generate smalle, and thus better, conflicts

- As soon as an equality $t = u$ is decided or propagated, then this equality is added to the union-find algorithm, thus merging the equivalence classes of t and u. When merging the two equivalent classes, the tags are compared, potentially raising a conflict (see below), and the list of disallowed classes are also compared, potentially raising a conflict (see below).

- As soon as a disequality $t \neq u$ is decided or propagated, the equivalence class of t is added to the list of disallowed classes of the equivalence class of u (and vice-versa). This can raise a conflict if t and u have the same equivalence class (see below).

The above operations have two ways to raise a conflict : when comparing two tags, and when merging classes that are not allowed to be merged.

- A conflict involving tags occurs either when the theory attempts to set the tag of an already tagged equivalence class, or when two classes with incompatible tags are attempted to be merged. In any case, the conflict arises in the presence of two tags (t, v) and (t', v') such that $v \neq v'$. The union-find is queried for the explanation $x_1, \ldots, x_n$ of why t and t' are in the same equivalence class. The following conflict can then be raised:

  $$\neg[\![t = x_1]\!] \vee \neg[\![x_1 = x_2]\!] \vee \ldots \vee \neg[\![x_n = t']\!] \vee [\![t = t']\!]$$

  This is obviously a tautology, expressing transitivity of equality on a list of terms. It is a correct conflict clause because the equalities $t = x_1$,..., $x_n = t'$ have been added to the union-find which only occurs when the equality has been decided or propagated (see next point), and equality $t = t'$ can be evaluated to false because of the different assignments for t and t'.

- Merging conflicts occur when two classes c and c' are either:

  - merged and the theory is trying to disallow their merging (in the case of a new disequality).
  - to be merged, although they are already disallowed to be merged (in the case of a new equality). In this case, the two classes are still merged temporarily before raising the conflict (in order to have a more homogeneous way of treating conflicts).

  In both cases, the theory only has to get the disequality $t = t'$ that is the reason why both classes should not be allowed to be merged, and then query the union-find for the explanation $x_1, \ldots, x_n$ of why t and t' should be in the same equivalence class. The theory can then raise the following conflict:

  $$\neg[\![t = x_1]\!] \vee \neg[\![x_1 = x_2]\!] \vee \ldots \vee \neg[\![x_n = t']\!] \vee [\![t = t']\!]$$

  Which is again a trivial tautology, and is a correct conflict clause because all the equalities in it have been decided and/or propagated so that the clause is false in the current trail.

Whenever asked for a value to assign to an unassigned term t, the theory finds the root t' of the equivalence class of t, and returns the value $\hat{t'}$. Additionally, ArchSAT performs a non-necessary optimisation, which is to merge any two equivalence classes that have been tagged with the same value[1].

While the description of the naive equality theory was explained in [58], it did not explore the use of a union-find to make the theory faster, which is a contribution of this thesis. More generally, most decision procedures usually used in SMT solvers can be adapted to McSat in order to retain their excellent speed and conflict detection, while also interfacing properly with the assignment mechanism which serves as a way for theories to exchange information.

[1] Note that the only change it makes is in the theoretical justification of the correction of conflicts clause, where some equalities may be have to be evaluated instead simply being decided or propagated.

Example. Let us consider the following simple example. We consider an input problem that consists of 4 clauses:

$$\begin{aligned}
\mathcal{C}_0 &\equiv \llbracket a = b \rrbracket \\
\mathcal{C}_1 &\equiv \llbracket a = c \rrbracket \\
\mathcal{C}_2 &\equiv \neg\llbracket a = d \rrbracket \\
\mathcal{C}_3 &\equiv \llbracket b = d \rrbracket \vee \llbracket c = d \rrbracket
\end{aligned}$$

When starting to solve this problem, the inferences rules immediately propagate $\llbracket a = b \rrbracket \leadsto_{\mathcal{C}_0} \top$, $\llbracket a = c \rrbracket \leadsto_{\mathcal{C}_1} \top$, and $\llbracket a = d \rrbracket \leadsto_{\mathcal{C}_2} \bot$. At this point, a decision has to be made. Let's suppose that the algorithm chooses to decide on a value for a. Since we do not have any constraint on the value of a, we assign $\hat{a}$: $a \mapsto_1 \hat{a}$. Then b is chosen to be assigned. Since we have a value for a, we have exactly one constraint on the value of b, which must be the same as the value of a, because $\llbracket a = b \rrbracket$ is true. We thus assign the same value to b: $b \mapsto_2 \hat{a}$. Suppose the algorithm now chooses to decide on a value for d. The only constraints on this value is that it must not be $\hat{a}$ since $\llbracket a = d \rrbracket$ is false. We thus assign d to its own distinct value: $d \mapsto_3 \hat{d}$. At that point, the theory of equality can evaluate $\llbracket b = d \rrbracket$ since both b and d are assigned. It can thus propagate: $\llbracket b = d \rrbracket \leadsto_3 \bot$. And finally, $\mathcal{C}_3$ can then be used to propagate: $\llbracket c = d \rrbracket \leadsto_{\mathcal{C}_3} \top$. The state of the McSat algorithm is then:

Clauses	Trail
$\{\ \mathcal{C}_0, \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\ \}$	$[\llbracket a = b \rrbracket \leadsto_{\mathcal{C}_0} \top,\quad \llbracket a = c \rrbracket \leadsto_{\mathcal{C}_1} \top,\quad \llbracket a = d \rrbracket \leadsto_{\mathcal{C}_2} \bot,$ $a \mapsto_1 \hat{a},\quad b \mapsto_2 \hat{a},\quad d \mapsto_3 \hat{d},\quad \llbracket b = d \rrbracket \leadsto_3 \bot,\quad \llbracket c = d \rrbracket \leadsto_{\mathcal{C}_3} \top]$

At that point the equality theory can detect that there is no more adequate value left to assign to c. Indeed there are two constraints on the value of c, on the one hand it must be the same as the value of a, which is $\hat{a}$, because $\llbracket a = c \rrbracket$ is true in the trail (at level 0), and on the other hand it must be the same as the value of d, which is $\hat{d}$, because $\llbracket c = d \rrbracket$ is also true (at level 3). These two constraints cannot be satisfied since by definition $\hat{a}$ and $\hat{d}$ are different values[1].

Thus a conflict can be generated using the transitivity of equality: $\neg\llbracket a = c \rrbracket \vee \neg\llbracket c = d \rrbracket \vee \llbracket a = d \rrbracket$. This is indeed a conflict clause, since $\llbracket a = c \rrbracket$ and $\llbracket c = d \rrbracket$ are true in the trail (and thus $\neg\llbracket a = c \rrbracket$ and $\neg\llbracket c = d \rrbracket$ false), and $\llbracket a = d \rrbracket$ can be evaluated to $\bot$ thanks to the assignments on a and d. The analyze of this conflict then goes up the trail (in reverse chronological order); it performs a resolution with $\mathcal{C}_3$ in order to reach the following clause: $\mathcal{C}_4 \equiv \neg\llbracket a = c \rrbracket \vee \llbracket b = d \rrbracket \vee \llbracket a = d \rrbracket$. Since this clause can propagate new information at level 0, the algorithm thus backtracks to just before the first decision (i.e. the assignment of a) was made, and adds $\mathcal{C}_4$ to the set of clauses that the solver tries to satisfy, and the solver is now in the following state:

Clauses	Trail
$\{\ \mathcal{C}_0, \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4\ \}$	$[\llbracket a = b \rrbracket \leadsto_{\mathcal{C}_0} \top,\quad \llbracket a = c \rrbracket \leadsto_{\mathcal{C}_1} \top,\quad \llbracket a = d \rrbracket \leadsto_{\mathcal{C}_2} \bot,\quad \llbracket b = d \rrbracket \leadsto_{\mathcal{C}_4} \top]$

Continuing solving, the inference rules may for instance choose to decide on a, and get the semantic decision $a \mapsto_1 \hat{a}$ (since there is no constraint on the value of a), and then decide on b (which must have the same value as a, because $\llbracket a = b \rrbracket$ is true in the trail), resulting in the decision $b \mapsto_2 \hat{a}$. At that point, there is already no more adequate value for d left, since it should be distinct from the value of a ($\llbracket a = d \rrbracket$ is false in the trail), and equal to the value of b ($\llbracket b = d \rrbracket$ is true in the trail). However, a and b have the same value, thus a conflict is raised: $\neg\llbracket a = d \rrbracket \vee \neg\llbracket b = d \rrbracket \vee \llbracket a = d \rrbracket$. Resolution steps are then performed between this conflict and $\mathcal{C}_4$, $\mathcal{C}_2$, $\mathcal{C}_1$ and finally $\mathcal{C}_0$, reaching the empty clause and proving that the input clauses are unsatisfiable.

In this example, there is no difference between the naive theory for equality and using a union-find structure, since we do not need to consider chains with more than two equalities.

[1] Values can be syntactically compared to determine their semantic equality.

2.1.3.2 Uninterpreted Functions and Predicates

Dealing with uninterpreted functions and predicates using assignments is actually quite easier than what a congruence closure algorithm usually entails. All that the theory has to do is check that assignments are coherent with the semantics of functions.

Formally, the theory of uninterpreted functions has to ensure the coherence of the partial abstract interpretation σ built by the McSat solver, with regards to uninterpreted functions and predicates. That means ensuring that for each uninterpreted function symbol f, there exists a function F that can be its value in a model that completes σ. The easiest way to ensure that is for the theory to actually build such a function F during proof search, or rather its projection on the values considered in the problem, so that each tuple of values is mapped to a single value.

In order to ensure the existence of such a mapping, the theory must ensure that for any pair of terms $f(t_1, \dots, t_n)$ and $f(u_1, \dots, u_n)$, if for all $1 \leq i \leq n$, t_i and u_i are assigned to the same value v_i, then $f(t_1, \dots, t_n)$ and $f(u_1, \dots, u_n)$ are assigned to the same value v. Whenever that is not the case, a conflict can be raised: $t_1 = u_1 \wedge \dots \wedge t_n = u_n \rightarrow f(t_1, \dots, t_n) = f(u_1, \dots, u_n)$, which gives the clause: $\neg[\![t_1 = u_1]\!] \vee \dots \vee \neg[\![t_n = u_n]\!] \vee [\![f(t_1, \dots, t_n) = f(u_1, \dots, u_n)]\!]$ where every literal can be evaluated using assignments. For an uninterpreted predicate p (instead of an uninterpreted function f), if $p(t_1, \dots, t_n)$ and $p(u_1, \dots, u_n)$ have different truth assignment, then one must be true while the other is false. Without loss of generality, let us suppose that $p(t_1, \dots, t_n)$ is true, then the conflict will be: $t_1 = u_1 \wedge \dots \wedge t_n = u_n \wedge p(t_1, \dots, t_n) \rightarrow p(u_1, \dots, u_n)$ which gives the clause: $\neg[\![t_1 = u_1]\!] \vee \dots \vee \neg[\![t_n = u_n]\!] \vee \neg[\![p(t_1, \dots, t_n)]\!] \vee [\![p(u_1, \dots, u_n)]\!]$.

This is enough to fully implement a theory of uninterpreted functions and predicates: it specifies exactly what trails are incoherent, and what conflicts to raise in such cases. Note that this theory does not decide assignments: it leaves these decisions to other theories, and only checks the coherence of these assignments.

A little caveat is present for uninterpreted predicates, in that terms that have the proposition type ($\$o$ in TPTP), i.e. terms that result from the application of an uninterpreted predicate, should not be targets of assignments. Indeed, these terms of the form $p(\dots)$ already appear directly in blackboxes as $[\![p(\dots)]\!]$ and the McSat algorithm can already decide on these blackboxes. For all intents and purposes, this decision is equivalent to an assignment $p(\dots) \mapsto \top$ (i.e. if a conflict relies on the fact that $p(\dots)$ is true because of its assignment, it will work just as well because $[\![p(\dots)]\!]$ is true in the trail). Consequently, atomic formula will not be assigned (as terms are), but in ArchSAT, a dummy assignment is added following decision on blackboxed atomic formulas, in order to avoid special-casing them in theories.

Example Let us consider the following formula $P \equiv a = b \Rightarrow f(a) = f(b)$. In order to prove such a formula, we start with a single clause :

$$\mathcal{C}_0 \equiv \neg[\![P]\!]$$

The solve directly propagates, $P \rightsquigarrow_{\mathcal{C}_0} \bot$. The solver then uses the tableau rule $\alpha_{\neg\Rightarrow}$ to add the two clauses :

$$\begin{aligned} \mathcal{C}_1 &\equiv [\![P]\!] \vee [\![a = b]\!] \\ \mathcal{C}_2 &\equiv [\![P]\!] \vee \neg[\![f(a) = f(b)]\!] \end{aligned}$$

Using these two clauses, the following two propagations can be made: $[\![a = b]\!] \rightsquigarrow_{\mathcal{C}_1} \top$, and $[\![f(a) = f(b)]\!] \rightsquigarrow_{\mathcal{C}_2} \bot$. At that point, all propagations have been done, so the McSat algorithm chooses to decide on a value for a. Since there is not yet any constraint on it, the value chosen is $\hat{a}$: $a \mapsto_1 \hat{a}$. After that, let us suppose the algorithm decides to choose on b; as explained above in the theory of equality, the only value allowed for b is $\hat{a}$, which results in the decision: $b \mapsto_2 \hat{a}$. Then suppose a value for $f(a)$ is to be chosen; as there is no constraint, $\widehat{f(a)}$ is chosen: $f(a) \mapsto_3 \widehat{f(a)}$. Then, a value is chosen for $f(b)$, which must be distinct from the value of $f(a)$, thus: $f(b) \mapsto_4 \widehat{f(b)}$. At that point, the theory of uninterpreted functions will detect a conflict, because although a and b are assigned to the same value, $f(a)$ and $f(b)$ are not. It will therefore add the following clause:

$$\mathcal{C}_3 \equiv \neg[\![a = b]\!] \vee [\![f(a) = f(b)]\!]$$

The analyze phase then inspects each element of the trail, starting from the most recent. The fours semantic decisions are ignored as no literal in the clause is false because of a semantic propagation. The algorithm then inspects the propagation $[\![f(a) = f(b)]\!] \rightsquigarrow_{\mathcal{C}_2} \bot$. Since $[\![f(a) = f(b)]\!]$ occurs in the conflict clause, a resolution is performed between the conflict clause $\mathcal{C}_3$, and $\mathcal{C}_2$, that yields the clause $D \equiv [\![P]\!] \vee \neg[\![a = b]\!]$. Then the propagation, $[\![a = b]\!] \rightsquigarrow_{\mathcal{C}_1} \top$ is inspected, and since $[\![a = b]\!]$ occurs in D, another resolution is performed, yielding the clause $D \equiv [\![P]\!]$. Finally, the first propagation is analyzed, a last resolution performed, and the empty clause is reached, demonstrating the unsatisfiability of the problem, and therefore the provability of P.

2.2 Existentially-Quantified Formulas

This section describes how existential formulas are handled in ArchSAT. More specifically some technical aspect of how generation of Skolem terms is handled during proof search.

2.2.1 Existential Constants

As shown in Figure 2.2, the reasoning rules of the tableau theory for existentially quantified formulas are relatively simple. Each time an existentially quantified formula $\exists x : t_y.P(x)$ (resp. a universally quantified formula $\forall x : t_y.P(x)$) becomes true (resp. false), a fresh constant c is generated for that formula[1], and a clause $\neg[\![\exists x : t_y.P(x)]\!] \vee [\![P(c)]\!]$ (resp. $[\![\forall x : ty.P(x)]\!] \vee \neg[\![P(c)]\!]$) is added to the solver, following rule $\delta_\exists$ (resp. $\delta_{\neg\forall}$). Fresh type constants are also technically generated for formulas that quantify over types, although there are not many problems that existentially quantify over types. Comparatively, meta-variables for types will be much more frequent, as universal quantification is quite useful to generalize axioms.

More interesting is actually the interaction between these fresh constants and meta-variables. As mentioned earlier, and detailed in Section 2.3, terms will at one point be unified in order to find substitutions from meta-variables to terms. This brings an interesting problem, which is that the generated constants are actually free of constraints during unification: a constant generated from a formula containing a meta-variable can be unified with that same meta-variable without any problem. As we will see this is not a problem with the way instantiation is done as explained in Section 2.3. However, for other instantiation scheme[2], it would be interesting to use Skolem terms instead of epsilon-terms. Indeed Skolem terms are more restrictive with respect to unification, which would allow for more subtle instantiation schemes, or even would help reduce the number of substitutions found during unification. To that end, the next section explains the technical aspect of generating Skolem terms dynamically during proof search.

2.2.2 Generating Skolems on the Fly

Unlike fresh constants, generating Skolem terms pose a more technical challenge. Indeed, Skolem terms are quite easy to generate for initial formulas (which still retain free variables bound by outer quantifiers), but it becomes harder once these free variables have been substituted by ground terms (possibly including meta-variables). For instance, let us consider the formula:

$$F \equiv \forall x : t_y.\exists y : t_y.P(x, y)$$

In this form, it would be quite easy to introduce a Skolem term for the existentially quantified variable, in order to get the formula:

$$F_{\text{sk}} \equiv \forall x : t_y.P(x, \text{sk}(x))$$

[1] This handling of existential formulas is different from that of Zenon which uses epsilon-terms instead of fresh generated constants.

[2] for instance, arithmetic.

However, during proof search, we do not usually have access to F, but rather to the formula $F' \equiv \exists y : t_y.P(t, y)$, where x has been substituted with an arbitrary ground term t. The problem is that looking at F' alone, it is impossible to determine what the free variables of F were, and with what terms they have been substituted.

In order to overcome that problem, each quantified formula maintains a list of free terms. At creation, this list is initialized with the list of free variables of the formula. Subsequently, each substitution that is applied to the quantified formula is also applied to its list of free terms. When a quantified formula then needs to generate its associated Skolem term, the list of free terms can be used as arguments for the Skolem symbol.

Following the example before, when we create F, the existential quantifier records that it has exactly one free variable, and initializes its free term list with $[x]$. When x is substituted with t, this also applies to the list of free terms of the existential quantifier, so that F' has the free term list $[t]$. When we try and generate a Skolem symbol for F', we use its list of free variables to generate the Skolem term $\mathrm{sk}(t)$, which is the expected result.

While this is partly a technical issue, it is also a non-trivial theoretical point, as the instantiation rule for Skolem terms, that is the analog of the δ rules for Skolem terms, are not directly expressible with the structure of terms given in this thesis. Indeed, to express such rules would need to integrate the notion of free arguments as described above into the structure of formulas presented in Figure 2.1.

Finally, note that since ArchSAT actually uses polymorphic terms, there is actually a list of free types along the list of free terms, following the same idea, and Skolem symbols can thus be polymorphic.

2.3 Meta-Variables

Handling universally quantified formulas in ArchSAT is done in rounds, each with three steps. A round starts by solving the clauses currently in the solver state using the tableau, equality and uninterpreted function theories defined earlier. During this search, some meta-variables are generated (very similarly to how epsilon-terms are generated), as described in Section 2.3.1. Once proof search on the clauses in the solver state has finished, it results in a ground model of the input problem (meta-variables are considered ground terms). At that point, some substitution from meta-variables to terms are determined as explained in Section 2.3.2. Section 2.3.3 explains how these substitutions are then used to instantiate the meta-variables, or rather their defining formulas, so that new clauses are added to the solver state. At that point, proof search can then re-start using these new clauses, and this starts another round. This process continues to be applied until either no substitutions can be found, or an unsat result is reached.

2.3.1 Generating Meta-Variables

Meta-variables are generated for each universally quantified formula (resp. existentially quantified formula) that becomes true (resp. false), and then introduced using rule $\gamma_{\forall M}$ (resp. $\gamma_{\neg\exists M}$).

For reasons explained later, we may need to generate more than one meta-variable per quantified formula. Each meta-variable is thus uniquely identified by a pair of a formula and an increasing index used to distinguish distinct meta-variables generated from the same formula.

Semantically, meta-variables are ground terms, whose existence is guaranteed because first-order logic requires all types to be inhabited. In that sense, they behave the same way as epsilon-terms during proof search, and will be assigned to a value by the McSat inference rules. The only difference is during unification, when meta-variables actually behave as unification variables (i.e substitutions bind meta-variables to terms or types).

2.3.2 Finding Instantiations

Once meta-variables are generated, proof search proceeds as usual, until a model is found by the inference rules of the McSat algorithm[1]. Once such a model is found, we will try and find a substitution (from meta-variables to types or terms) such that its application invalidates the propositional model[2]. For instance, suppose we reach a propositional model where $p(a)$ is true and $p(X)$ false, for some meta-variable X. Then the substitution $\{X \mapsto a\}$ would actually invalidate the propositional model, because under the substitution, $p(a)$ is both true and false. In the next subsection we will see how such a substitution is then used in practice.

The way this search for substitutions is done in ArchSAT is the following. When we get a propositional model $\mathcal{M}$, we first keep only the atomic formulas, since the semantics of formulas starting with a logical connective are already fully encoded in the clausal calculus of the McSat inference rules. We can then split the atomic formulas into four mutually exclusive lists:

- The equalities that are true
- The equalities that are false
- The predicates[3] that are true
- The predicates that are false

Then, for each pair (u, v) where either u is a true predicate and v a false predicate, or $u = v$ is one of the false equalities, we try and unify u and v. As usual when unifying terms, we look for a substitution σ with no cycles, so that fixpoint application of σ to a term t, written $t\sigma$, is well-defined. Unification of u and v thus either fails, or returns a substitution σ, such that $u\sigma = v\sigma$ syntactically.

Note that if the list of true equalities is non-empty, this unification should be done modulo these equalities. Section 3.4 will treat this in more details. For now, we will thus consider examples without equalities, where simple unification is enough[4].

For instance, suppose we want to prove the drinker's paradox: $D \equiv \exists x : t_y.\ p(x) \Rightarrow (\forall y : t_y.\ p(y))$. Using the tableaux rules described in Figure 2.2, we start with the clause:

$$\mathcal{C}_0 \equiv \neg[\![\exists x : t_y.\ p(x) \Rightarrow (\forall y : t_y.\ p(y))]\!]$$

This clause propagates $[\![D]\!] \leadsto_{\mathcal{C}_0} \bot$, which allows the tableau theory to generate a meta-variable $X \equiv X_{\exists x : t_y.\ p(x) \Rightarrow (\forall y : t_y.\ p(y))}$ using rule $\gamma_{\neg\exists}$, and add the following clause:

$$\mathcal{C}_1 \equiv [\![\exists x : t_y.\ p(x) \Rightarrow (\forall y : t_y.\ p(y))]\!] \vee \neg[\![p(X) \Rightarrow \forall y : t_y.\ p(y)]\!]$$

This in turn, propagates $[\![p(X) \Rightarrow \forall y : t_y.\ p(y)]\!] \leadsto_{\mathcal{C}_1} \bot$, which causes the tableau theory to create the following two clauses using rule $\alpha_{\neg\Rightarrow}$:

$$\begin{aligned}\mathcal{C}_2 &\equiv [\![p(X) \Rightarrow \forall y : t_y.\ p(y)]\!] \vee [\![p(X)]\!] \\ \mathcal{C}_3 &\equiv [\![p(X) \Rightarrow \forall y : t_y.\ p(y)]\!] \vee \neg[\![\forall y : t_y.\ p(y)]\!]\end{aligned}$$

These two clauses allow to propagate $[\![p(X)]\!] \leadsto_{\mathcal{C}_2} \top$, and $[\![\forall y : t_y.\ p(y)]\!] \leadsto_{\mathcal{C}_3} \bot$. Finally, the tableau theory uses rule $\delta_{\neg\forall}$ to generate a fresh constant τ and add the clause :

$$\mathcal{C}_4 \equiv [\![\forall y : t_y.\ p(y)]\!] \vee \neg[\![p(\tau)]\!]$$

[1] Or, as noted earlier, the SMT inference rules, since the tableau theory, including the handling of meta-variables, is actually compatible with the SMT algorithm.

[2] ArchSAT is primarily aimed at proving formulas, which is done by proving there is no model of the axioms and the negation of the goal, hence why it tries to invalidate models.

[3] In this context, a predicate is a formula whose head symbol is a predicate symbol.

[4] Robinson's unification algorithm is currently used.

At that point, SMT solvers have nothing more to do, whereas McSat solvers would still decide on values for the terms occurring in the problem, however, these additional assignments would not raise a conflict and thus not change the propositional model. The solver has thus reached the following propositional model[1]:

$$\begin{aligned}
[\![\exists x : t_y.\ p(x) \Rightarrow (\forall y : t_y.\ p(y))]\!] &\mapsto \bot \\
[\![p(X) \Rightarrow \forall y : t_y.\ p(y)]\!] &\mapsto \bot \\
[\![p(X)]\!] &\mapsto \top \\
[\![\forall y : t_y.\ p(y)]\!] &\mapsto \bot \\
[\![p(\tau)]\!] &\mapsto \bot
\end{aligned}$$

We thus get exactly one true predicate: $p(X)$ and one false predicate $p(\tau)$. We can now try and unify $p(X)$ and $p(\tau)$, which succeeds and yields the substitution $\sigma = \{X \mapsto \tau\}$. We will see in the next section how this substitution is subsequently used to finish the proof of the drinker's paradox.

Meta-Variables and Skolem Terms Generated constant for existential formulas are quite adequate with the schema of instantiation that is explained in the next section, where new clauses representing the instantiations are added, without affecting the already existing meta-variables. However, there are other ways to use the substitutions found (for instance, substituting the meta-variables in place, or adding new equalities corresponding to the bindings of the substitution) which could be unsound when unifying while using these generated constants.

Indeed, let us assume we are trying to prove the following satisfiable formula: $\exists x : t_y.\forall y : t_y.p(y,x) \Rightarrow p(x,y)$. We get the following clauses and propositional model:

$$\begin{aligned}
&\mathcal{C}_0 \equiv \neg[\![\exists x : t_y.\forall y : t_y.p(y,x) \Rightarrow p(x,y)]\!] \\
&\mathcal{C}_1 \equiv [\![\exists x : t_y.\forall y : t_y.p(y,x) \Rightarrow p(x,y)]\!] \vee \neg[\![\forall y : t_y.p(y,X) \Rightarrow p(X,y)]\!] \\
&\mathcal{C}_2 \equiv [\![\forall y : t_y.p(y,X) \Rightarrow p(X,y)]\!] \vee \neg[\![p(\tau,X) \Rightarrow p(X,\tau)]\!] \\
&\mathcal{C}_3 \equiv [\![p(\tau,X) \Rightarrow p(X,\tau)]\!] \vee [\![p(\tau,X)]\!] \\
&\mathcal{C}_4 \equiv [\![p(\tau,X) \Rightarrow p(X,\tau)]\!] \vee \neg[\![p(X,\tau)]\!] \\
&X \equiv X_{\exists x:t_y.\forall y:t_y.p(y,x)\Rightarrow p(x,y)}
\end{aligned}$$

$$\begin{aligned}
[\![\exists x : t_y.\forall y : t_y.p(y,x) \Rightarrow p(x,y)]\!] &\mapsto \bot \\
[\![\forall y : t_y.p(y,X) \Rightarrow p(X,y)]\!] &\mapsto \bot \\
[\![p(\tau,X) \Rightarrow p(X,\tau)]\!] &\mapsto \bot \\
[\![p(\tau,X)]\!] &\mapsto \top \\
[\![p(X,\tau)]\!] &\mapsto \bot
\end{aligned}$$

We can unify $p(\tau,X)$ and $p(X,\tau)$ with the substitution $\sigma = \{X \mapsto \tau\}$. However, if we try and add the equality $X = \tau$, or try and substitute X by τ everywhere, we would allow the McSat algorithm to reach UNSAT and wrongly conclude that the input formula is provable. Indeed, replacing X with τ, or otherwise constraining the value of X to be the same as τ is unsound, since τ is introduced after X, and more precisely τ depends on X. This dependency relation between X and τ is exactly what the Skolem symbols naturally explicit: generating Skolem terms instead

[1] In ArchSAT, we actually get a full first-order model which also includes assignments for X and τ, but this information is not used for finding instantiations.

of fresh constants would replace τ by $\mathrm{sk}(X)$, which would prevent unification of $p(X, \mathrm{sk}(X))$ and $p(\mathrm{sk}(X), X)$, as the only possible to do that is through the cyclic substitution $\{X \mapsto \mathrm{sk}(X)\}$.

Generating Skolem symbol instead of epsilon terms would also prevent unification from finding a solution for the following variation of the drinker's paradox: $\exists x : t_y.\forall y : t_y.p(x) \Rightarrow p(y)$ which would make ArchSAT incomplete. In order to overcome this problem, we can generate more than one meta-variable for each quantified variable. In the example of the above variation of the drinker's paradox, using Skolem terms, this would produce:

$$\begin{aligned}
\mathcal{C}_0 &\equiv \neg[\![\exists x : t_y.\forall y : t_y.p(x) \Rightarrow p(y)]\!]\\
\mathcal{C}_1 &\equiv [\![\exists x : t_y.\forall y : t_y.p(x) \Rightarrow p(y)]\!] \vee \neg[\![\forall y : t_y.p(X) \Rightarrow p(y)]\!]\\
\mathcal{C}_2 &\equiv [\![\forall y : t_y.p(X) \Rightarrow p(y)]\!] \vee \neg[\![p(X) \Rightarrow p(\mathrm{sk}(X))]\!]\\
\mathcal{C}_3 &\equiv [\![p(X) \Rightarrow p(\mathrm{sk}(X))]\!] \vee [\![p(X)]\!]\\
\mathcal{C}_4 &\equiv [\![p(X) \Rightarrow p(\mathrm{sk}(X))]\!] \vee \neg[\![p(\mathrm{sk}(X))]\!]\\
\mathcal{C}_5 &\equiv [\![\exists x : t_y.\forall y : t_y.p(x) \Rightarrow p(y)]\!] \vee \neg[\![\forall y : t_y.p(X') \Rightarrow p(y)]\!]\\
\mathcal{C}_6 &\equiv [\![\forall y : t_y.p(X') \Rightarrow p(y)]\!] \vee \neg[\![p(X') \Rightarrow p(\mathrm{sk}(X'))]\!]\\
\mathcal{C}_7 &\equiv [\![p(X') \Rightarrow p(\mathrm{sk}(X'))]\!] \vee [\![p(X')]\!]\\
\mathcal{C}_8 &\equiv [\![p(X') \Rightarrow p(\mathrm{sk}(X'))]\!] \vee \neg[\![p(\mathrm{sk}(X'))]\!]
\end{aligned}$$

So that we have the following atomic predicates:

- $p(X) \mapsto \top$
- $p(X') \mapsto \top$
- $p(\mathrm{sk}(X)) \mapsto \bot$
- $p(\mathrm{sk}(X')) \mapsto \bot$

As expected, unification fails for pairs $(p(X), p(\mathrm{sk}(X)))$ and $(p(X'), p(\mathrm{sk}(X')))$, but succeeds for pairs $(p(X'), p(\mathrm{sk}(X)))$ and $(p(X), p(\mathrm{sk}(X')))$. Yielding two substitutions $\{X \mapsto \mathrm{sk}(X')\}$ and $\{X' \mapsto \mathrm{sk}(X)\}$. This shows that generating more than one meta-variable per quantified formula is a way to solve the problem of incompleteness[1] that Skolem-terms introduce. However, the number of meta-variables to generate may be arbitrarily high, which means that we need to periodically generate new meta-variables.

In conclusion, Skolem terms are a way to explicit the dependency relations between meta-variables and existential constants, which may help filter the substitutions found, but introduces other challenges when it come to completeness and fairness of the proof search.

2.3.3 Instantiating Meta-Variables

In this section, we suppose that we have a substitution σ from meta-variables to terms (we will see later what adding bindings from meta-variables to types changes), and we would like to use rules $\gamma_{\forall\mathrm{inst}}$ and $\gamma_{\neg\exists\mathrm{inst}}$ to instantiate the corresponding formulas. The main idea is that for each binding $X_F \mapsto t \in \sigma$, formula F should be instantiated with t using either rule $\gamma_{\forall\mathrm{inst}}$ or $\gamma_{\neg\exists\mathrm{inst}}$.

If we continue with the example of the drinker's paradox, we had the substitution $\sigma = \{X \mapsto \tau\}$, with $X \equiv X_{\exists x:t_y.\ p(x) \Rightarrow (\forall y:t_y.\ p(y))}$. In order to perform the instantiation, we thus use rule $\gamma_{\neg\exists\mathrm{inst}}$ to generate the clause $\mathcal{C}_5 \equiv [\![\exists x : t_y.\ p(x) \Rightarrow (\forall y : t_y.\ p(y))]\!] \vee \neg[\![p(\tau) \Rightarrow \forall y : t_y.\ p(y)]\!]$ which is added to the problem, and propagates $\neg[\![p(\tau) \Rightarrow \forall y : t_y.\ p(y)]\!] \rightsquigarrow_{\mathcal{C}_5} \top$, which in turn triggers the tableau theory to unfold it, so that we reach the following state:

[1] This is not a formal statement of completeness.

$$
\begin{aligned}
&\mathcal{C}_0 \equiv \neg[\![\exists x : t_y.\ p(x) \Rightarrow (\forall y : t_y.\ p(y))]\!]\\
&\mathcal{C}_1 \equiv [\![\exists x : t_y.\ p(x) \Rightarrow (\forall y : t_y.\ p(y))]\!] \lor \neg[\![p(X) \Rightarrow \forall y : t_y.\ p(y)]\!]\\
&\mathcal{C}_2 \equiv [\![p(X) \Rightarrow \forall y : t_y.\ p(y)]\!] \lor [\![p(X)]\!]\\
&\mathcal{C}_3 \equiv [\![p(X) \Rightarrow \forall y : t_y.\ p(y)]\!] \lor \neg[\![\forall y : t_y.\ p(y)]\!]\\
&\mathcal{C}_4 \equiv [\![\forall y : t_y.\ p(y)]\!] \lor \neg[\![p(\tau)]\!]\\
&\mathcal{C}_5 \equiv [\![\exists x : t_y.\ p(x) \Rightarrow (\forall y : t_y.\ p(y))]\!] \lor \neg[\![p(\tau) \Rightarrow \forall y : t_y.\ p(y)]\!]\\
&\mathcal{C}_6 \equiv [\![p(\tau) \Rightarrow \forall y : t_y.\ p(y)]\!] \lor [\![p(\tau)]\!]\\
&\mathcal{C}_7 \equiv [\![p(\tau) \Rightarrow \forall y : t_y.\ p(y)]\!] \lor \neg[\![\forall y : t_y.\ p(y)]\!]\\
&X \equiv X_{\exists x : t_y.\ p(x) \Rightarrow (\forall y : t_y.\ p(y))}
\end{aligned}
$$

$$
\begin{aligned}
[\![\exists x : t_y.\ p(x) \Rightarrow (\forall y : t_y.\ p(y))]\!] &\mapsto \bot\\
[\![p(X) \Rightarrow \forall y : t_y.\ p(y)]\!] &\mapsto \bot\\
[\![p(X)]\!] &\mapsto \top\\
[\![\forall y : t_y.\ p(y)]\!] &\mapsto \bot\\
[\![p(\tau)]\!] &\mapsto \bot\\
[\![p(\tau) \Rightarrow \forall y : t_y.\ p(y)]\!] &\mapsto \bot
\end{aligned}
$$

In this state, the inference rules detect a conflict in clause $\mathcal{C}_6$, which after analysis, allows us to conclude that the problem is unsatisfiable, thus proving the drinker's paradox. Note that we did not need to do an explicit contraction (as is usually needed for the drinker's paradox), since every application of the tableau theory actually does a contraction implicitly, and indeed the quantified formula $\neg\exists x : t_y.\ p(x) \Rightarrow (\forall y : t_y.\ p(y))$, is "used" twice by the tableau theory, in clauses $\mathcal{C}_1$ with rule $\gamma_{\neg\exists M}$, and clause $\mathcal{C}_5$ with rule $\gamma_{\neg\exists\text{inst}}$.

This shows what happens in the simple examples. The general case, when we want to instantiate multiple variables at the same time is a bit more complicated. This is where we start to diverge a bit from the tableau proof search implemented in Zenon, which only instanciate a single meta-variable at a time.

Nested Quantifiers Nested quantification needs special care. Consider for instance the formula: $\forall x : t_y, y : t_y.P(x, y)$. Generating meta-variables from this formula will yield the following clauses and meta-variables.

$$
\begin{aligned}
&\mathcal{C}_0 \equiv \neg[\![\forall x : t_y, y : t_y.P(x, y)]\!] \lor [\![\forall y : t_y.P(X, y)]\!]\\
&\mathcal{C}_1 \equiv \neg[\![\forall y : t_y.P(X, y)]\!] \lor [\![P(X, Y)]\!]\\
&X := X_{\forall x : t_y, y : t_y.P(x,y)}\\
&Y := Y_{\forall y : t_y.P(X,y)}
\end{aligned}
$$

Suppose we now have the substitution $\{X \mapsto t_X; Y \mapsto t_Y\}$. Using rule $\gamma_{\forall\text{inst}}$ for the binding $X \mapsto t_X$ yields clause $\mathcal{C}_2$, which will generate a new meta-variable in clause $\mathcal{C}_3$.

$$
\begin{aligned}
&\mathcal{C}_2 \equiv \neg[\![\forall x : t_y, y : t_y.P(x, y)]\!] \lor [\![\forall y : t_y.P(t_X, y)]\!]\\
&\mathcal{C}_3 \equiv \neg[\![\forall y : t_y.P(t_X, y)]\!] \lor [\![P(t_X, Y')]\!]\\
&Y' := Y_{\forall y : t_y.P(t_X,y)}
\end{aligned}
$$

However, directly applying rule $\gamma_{\forall\text{inst}}$ to the binding $Y \mapsto t_Y$ yield the clause:

$$\mathcal{C}_4 \equiv \neg[\![\forall y : t_y.P(X,y)]\!] \vee [\![P(X,t_Y)]\!]$$

The problem is that we treat each instantiation separately, and deduce two separate new formulas $P(t_X,Y')$ and $P(X,t_Y)$, whereas we'd prefer to deduce $P(t_X,t_Y)$. This problem with separate instantiations can be solved by grouping successive quantified variables in clusters, and instantiating clusters together. This actually mirrors the notation for successive quantifications: instead of considering that a quantifiers binds a single variable, each quantifiers binds a non-ordered list of variables. In the implementation, the list is ordered, but the instantiation mechanism looks at all the bound variables. This allows us to correctly treat formulas such as $\forall x : t_y, y : t_y, z : t_y.P(x,y,z)$ when the substitution maps x and z, or rather the meta-variables generated from x and z, but not the one generated from y.

However grouping variables into clusters is not always possible, for instance consider the formula: $\forall x : t_y.\exists y : t_y.\forall z : t_y.P(x,y,z)$. Applying tableau rules for quantified formulas generates these clauses:

$$\begin{aligned}
&\mathcal{C}_0 \equiv \neg[\![\forall x : t_y.\exists y : t_y.\forall z : t_y.P(x,y,z)]\!] \vee [\![\exists y : t_y.\forall z : t_y.P(X,y,z)]\!]\\
&\mathcal{C}_1 \equiv \neg[\![\exists y : t_y.\forall z : t_y.P(X,y,z)]\!] \vee [\![\forall z : t_y.P(X,\tau,z)]\!]\\
&\mathcal{C}_2 \equiv \neg[\![\forall z : t_y.P(X,\tau,z)]\!] \vee [\![P(X,\tau,Z)]\!]\\
&\ X := X_{\forall x:t_y.\exists y:t_y.\forall z:t_y.P(x,y,z)}\\
&\ \ \tau := \epsilon\, y : t_y.\forall z : t_y.P(X,y,z)\\
&\ Z := Z_{\forall z:t_y.P(X,\tau,z)}
\end{aligned}$$

Now consider a substitution $\sigma = \{X \mapsto t_X; Z \mapsto t_Z\}$. In this case, applying the bindings separately is even worse than in the first case, since an epsilon variable and a meta-variable are generated, resulting in these clauses:

$$\begin{aligned}
&\mathcal{C}_3 \equiv \neg[\![\forall x : t_y.\exists y : t_y.\forall z : t_y.P(x,y,z)]\!] \vee [\![\exists y : t_y.\forall z : t_y.P(t_X,y,z)]\!]\\
&\mathcal{C}_4 \equiv= \neg[\![\exists y : t_y.\forall z : t_y.P(t_X,y,z)]\!] \vee [\![\forall z : t_y.P(t_X,\tau',z)]\!]\\
&\mathcal{C}_5 \equiv \neg[\![\forall z : t_y.P(t_X,\tau',z)]\!] \vee [\![P(t_X,\tau',Z')]\!]\\
&\mathcal{C}_6 \equiv \neg[\![\forall z : t_y.P(X,\tau,z)]\!] \vee [\![P(X,\tau,t_Z)]\!]\\
&\tau' := \epsilon\, y : t_y.\forall z : t_y.P(t_X,y,z)\\
&\ Z := Z_{\forall z:t_y.P(t_X,\tau,z)}
\end{aligned}$$

We end up with the following atomic predicates: $P(t_X,\tau',Z'), P(X,\tau,t_Z)$, whereas we would have ideally wanted to have $P(t_X,\tau',t_Z)$.

In order to avoid these situations, any substitution that is considered for instantiation is simplified in order to only keep the meta-variables corresponding to variables that are quantified in the outer-most clusters. Note that it is possible to have more than one outer-most cluster, for instance if we consider two distinct quantified formulas. Some bindings are lost during this simplification however, it should be simpler to find them again after instantiating the outer-most variables, rather than try and instantiate deep variables. For instance, considering two formulas $\forall x : t_y.P(x)$ and $\forall y : t_y.\exists e : t_y.\forall z : t_y.Q(y,z)$, and a substitution $\sigma = \{X \mapsto t_X; Y \mapsto t_Y; Z \mapsto t_Z\}$, we split σ into two substitutions $\sigma_1 = \{X \mapsto t_X\}$ and $\sigma_2 = \{Z \mapsto t_Z\}$ that we'll each apply "naively", resulting in two instantiations :

$$\begin{aligned}
&\mathcal{C}_x \equiv \neg[\![\forall x : t_y.P(x)]\!] \vee [\![P(t_X)]\!]\\
&\mathcal{C}_y \equiv \neg[\![\forall y : t_y.\exists e : t_y.\forall z : t_y.Q(y,z)]\!] \vee [\![\exists e : t_y.\forall z : t_y.Q(t_Y,z)]\!]
\end{aligned}$$

Polymorphism and Quantified Types Handling polymorphism brings some technical difficulties with instantiation. When instantiating a type variable, and replacing it in the body of the quantified formulas, it may (and probably will) change the type of some quantified term variables. In that case, the substitution will generate fresh variables. This means that when instantiating a type meta-variable and a term meta-variable which depends on it, both substitution have to be done at the same time in order for the term variable to be adequately substituted. Not doing so would not affect soundness, nor completeness, but means that the term instantiation is not done, and would force to find another substitution involving the fresh term meta-variable, even though we have already found the substitution. For instance, consider the following formula $F = \forall \alpha : \mathsf{Type}.\forall x : \alpha.P(x)$, and the substitution $\sigma\{\alpha \mapsto t_y; x \mapsto t\}$. Distinguishing the substitution into one type substitution $\sigma_{t_y} = \{\alpha \mapsto t_y\}$, and a term substitution $\sigma_t = \{x \mapsto t\}$, and applying the type substitution first would generate the formula $F' = \forall x' : t_y.P(x')$, where x has been renamed x' because its type has changed due to the substitution. Afterwards, applying σ_t would not change anything since the variable x doesn't appear anymore in F'.

This means that it is better for the representation of quantified formulas to quantify at the same time on both a list of type variables and a list of term variables, in order to avoid some back-and forth between the instantiation mechanism and the search for substitutions.

Another problem with polymorphism is that polymorphic substitution may inadvertently capture more variables than intended. For instance, consider a substitution $\sigma = \{\alpha \mapsto t_y; x \mapsto t\}$, where a type variable α is assigned to an arbitrary type t_y, and the term variable x, of type α, is assigned to a term t of type t_y. Applying σ directly to a term $x = y$, would fail because the variable y, of type α (same as x), isn't substituted, and thus applying the substitution would try and build the equality $t = y$, which is ill-typed (assuming $t_y \neq \alpha$). While that does not happen when dealing only with meta-variable, it is a common occurrence whenever substitution on variables are handled. In order to avoid this problem, substitutions that are to be applied to a term u, must first be extended with respect to the free variables in u in order to ensure that any variable whose type changes under the substitution is mapped to a fresh variable of the adequate type. This actually can bring quite some problems when trying to compare substitutions and applying a substitution to many terms. Indeed, in order for comparison to be as precise as possible, extending substitutions should be delayed as much as possible, because it makes substitutions bigger, and because the same substitution can be extended into different substitutions depending on which term the substitution is meant to be applied to. However, extending a substitution has side-effects: most notably, it creates fresh variables, which means that extending the same substitution twice to the same term may produce different substitutions since each extension may generate fresh variables. This problem is currently handled by very carefully controlling when and where substitutions are extended in ArchSAT, though a more generic solution would probably be better.

Heuristics Heuristics play an important role in theorem proving: the same algorithm with different heuristics can have very different results. In the case of instantiation in ArchSAT, the main choice point is, once unification has been run to find substitutions, which substitutions should actually be used to instantiate the corresponding formula, and add clauses to the solver. This is an important choice because most realistic problems can have so many terms that unification easily finds at least a thousand substitutions. The problem is that, if all of these substitutions are used to instantiate formulas, this adds a huge number of terms to the problem which means that ground solving afterwards will take proportionally longer, and more importantly the next round will have considerably more terms to unify, leading to even more terms, which in turns will yield more terms to unify at the next round, . . .

In order to avoid this explosion in the number of terms, ArchSAT restricts the number of formula instantiations done at each round[1]. This creates a choice point as to which instantiations to choose among all that were found. Currently a very naive heuristic is used to chose instantiations that were found using terms that were part of the goal, in order to introduce some sort of goal-oriented proof search.

[1] At the time of writing, the default was to perform the best 10 instantiations, according to some heuristic that can be selected (the number of instantiations done is also configurable).

Chapter 3

Rewriting in ArchSAT

This chapter presents how rewriting has been integrated into ArchSAT. First, a short introduction on rewrite systems and the associated challenges is given in Section 3.1. One of these challenges is the fact that the rewrite system considered may change during proof search, leading to two different strategies: one where we consider the rewrite system constant during proof search, yields the strategy which is called static rewriting in Section 3.2, and the other where the rewrite system may freely change during proof search, which is called dynamic rewriting, and presented in Section 3.3. Finally, rewrite rules are integrated into the unification procedure used for finding instantiations, as described in Section 3.4.

3.1 Rewrite Systems

3.1.1 Theoretical Presentation

In the following, we borrow some of the notations and definitions of [47]. We call FV the function that returns the set of free variables of a term or a formula. A term rewrite rule is a pair of terms of the same type[2] denoted by $l \longrightarrow r$, where $\mathrm{FV}(r) \subseteq \mathrm{FV}(l)$. A formula rewrite rule is a pair of formulas denoted by $l \longrightarrow r$, where l is an atomic formula and r is an arbitrary formula, and where $\mathrm{FV}(r) \subseteq \mathrm{FV}(l)$. A class rewrite system is a pair of rewrite systems, denoted by $\mathcal{RE}$, consisting of $\mathcal{R}$, a set of formula rewrite rules, and $\mathcal{E}$, a set of term rewrite rules.

Given a class rewrite system $\mathcal{RE}$, the relations $=_{\mathcal{E}}$ and $=_{\mathcal{RE}}$ are the congruences generated respectively by the sets $\mathcal{E}$ and $\mathcal{R} \cup \mathcal{E}$. In the following, we use the standard concepts of subterm and term replacement: given an occurrence ω in a proposition P, we write $P_{|\omega}$ for the term or proposition at ω, and $P[t]_{\omega}$ for the proposition obtained by replacing $P_{|\omega}$ by t in P at ω. Given a class rewrite system $\mathcal{RE}$, the proposition P $\mathcal{RE}$-rewrites to P', denoted by $P \longrightarrow_{\mathcal{RE}} P'$, if $P =_{\mathcal{E}} Q$, $Q_{|\omega} = \sigma(l)$, and $P' =_{\mathcal{E}} Q[\sigma(r)]_{\omega}$, for some rule $l \longrightarrow r \in \mathcal{R}$, some proposition Q, some occurrence ω in Q, and some substitution σ.

The relation $=_{\mathcal{RE}}$ is not decidable in general, but there are some cases where this relation is decidable depending on the class rewrite system $\mathcal{RE}$ and the rewrite relation $\longrightarrow_{\mathcal{RE}}$. In particular, if the rewrite relation $\longrightarrow_{\mathcal{RE}}$ is confluent and (weakly) terminating, then the relation $=_{\mathcal{RE}}$ is decidable.

In the rest of the thesis, we will assume that the rewrite system (i.e. the $\mathcal{RE}$ rewrite relation) that we use is confluent and strongly terminating. Considering weakly terminating systems would not be a problem theoretically, but would present challenges from the technical point of view, since the rewriting strategy that allows to normalize terms would have to somehow be given to the prover. Hence we will, in practice consider confluent and strongly terminating systems, but the theory and algorithms described also work with weakly terminating rewrite systems. Confluence is another point that could be relaxed: obviously, any rewrite system that can be completed into a confluent (and terminating) system can be used with the techniques described in this section. Completion of rewrite systems was not implemented in ArchSAT for two main reasons: first, it is

[2] Note that this type can contain type variables that are quantified in the rewrite rules as any other variable

technically quite complex to properly implement and relate to the original rewrite system, second and most importantly, the unification algorithm presented in Section 3.4 actually does perform completion of the rewrite system.

3.1.2 Examples of Rewrite Systems

Rewrite systems can be as simple as a single rewrite rule $f(gx) \longrightarrow x$ stating that a function f is the inverse of g. Associativity of an operator $+$ can be expressed as a rewrite rule $(a+b)+c \longrightarrow a+(b+c)$[1].

Complex first-order theories, can often be expressed as rewrite systems. One example if the set theory of the B method [3], which was turned into a rewrite system in [35].

Finally, conditional rewriting can be used to axiomatize the theory of arrays (or maps). We first introduce the necessary function symbols and associated types:

$$\begin{aligned} \text{array} &: \mathsf{Type} \to \mathsf{Type} \\ \text{create} &: \Pi\alpha.\alpha \to \text{array}(\alpha) \\ \text{get} &: \Pi\alpha.\text{int} \to \text{array}(\alpha) \to \alpha \\ \text{set} &: \Pi\alpha.\text{int} \to \alpha \to \text{array}(\alpha) \to \text{array}(\alpha) \end{aligned}$$

We can then declare 3 rewrite rules defining the results of the get function depending on the shape of the array argument:

$$\begin{aligned} \text{get}(\_;\_,\text{create}(\_,x)) &\longrightarrow x \\ i = j \to \text{get}(\_;i,\text{set}(\_;j,x,\_)) &\longrightarrow x \\ i \neq j \to \text{get}(\_;i,\text{set}(\_;j,\_,t)) &\longrightarrow \text{get}(\_;i,t) \end{aligned}$$

Where the _ are arbitrary variables that are not used in the right-hand side of the rewrite rule and thus can be safely ignored (assuming the terms that are matched are well-typed). Note that the second rewrite rule could be replaced by $\text{get}(\_;i,\text{set}(\_;i,x,\_)) \longrightarrow x$. This replaces the condition $i = j$ by using a non-linear rule (i.e. a rule where some variable has multiple occurrences in its left-hand side). Besides the problems that non-linearity brings[2], the condition of the third rewrite rule cannot be erased the same way, so conditional rewriting is needed to handle such an axiomatization of arrays.

3.1.3 Encoding Rewrite Rules using Boxes

In order to make use of rewrite rules during proof search, the rewrite system must first be detected by ArchSAT. To do so, we use the boxes introduced in Section 2.1.1. Indeed, a term rewrite rule (resp. a formula rewrite rule) $l \longrightarrow r$ can be easily encoded as a formula $\forall \overline{x} \in \text{FV}(l).l = r$ (resp. $\forall \overline{x} \in \text{FV}(l).l \leftrightarrow r$). Reciprocally, any formula of the form $\forall \overline{x}.l = r$ (resp. $\forall \overline{x}.l \leftrightarrow r$), can be understood as a rewrite rule provided that that $\text{FV}(r) \subset \text{FV}(l)$, and that the equality (resp. equivalence) can be oriented using a reduction ordering[3].

ArchSAT offers two ways to detect (and orient) input formulas that can be treated as rewrite rules:

- Manual mode. Input problems can specify that some axioms or formulas are actually rewrite rules. For instance the Zipperposition's format format parsed by ArchSAT has a special

[1] Commutativity can also be expressed as a rewrite rule $a+b \longrightarrow b+a$, but it is non-terminating.

[2] Confluence on non-linear systems is harder to check, pattern-matching is also less efficient since some optimizations suppose linearity.

[3] A reduction ordering is a well-founded ordering closed under substitution and congruence.

syntax to declare rewrite rules. In other languages such as TPTP, one can annotate formulas to specifically tag rewrite rules. In both cases, it is important in the internal representation of formulas to keep the order in which equalities (and equivalences) were formulated in the input problem, since it tells how to orient the rule. This is done using the tag mechanism described in Section 6.3.

- Automatic mode. In the absence of manual rules, ArchSAT can use some heuristics to try and automatically detect what formulas could be used as rewrite rules. This heuristic uses a LPO [88][1] to try and orient quantified equalities[2]. Quantified equivalences between atomic formulas are also oriented using this strategy. Equivalences where one side is non-atomic are not currently detected as rewrite rules for the following reason:
  - Having a non-atomic left-hand side for a rewrite rule is not allowed in the definitions of rewrite rules because it raises some problems concerning matching, for instance should conjunction and disjunction be considered up to symmetry when pattern matching, should more complex formulas be matched modulo De Morgan's law, . . . Thus, equivalences with non-atomic left-hand side are left to the general purpose instantiation.
  - Non-atomic right hand sides are currently not supported in automatic mode (but are in manual mode) because ArchSAT currently makes a difference between terms and formulas, and the LPO ordering is only implemented on terms, thus ArchSAT cannot compare the term that forms the atomic left-hand formula with the formula in the right-hand side.

Thus, in ArchSAT, rewrite rules from the input problems are encoded as regular quantified formulas. This means that any theory can simply analyze the formulas that become true (i.e. the boxes that are decided and/or propagated by the inference rules) in order to detect rewrite rules. This is what the rewriting theory in ArchSAT does.

3.1.4 Static and Dynamic Rewriting

Since rewrite rules are encoded as quantified formulas, it means that they are considered as literals by the inference rules of the McSat algorithm, and as such, their truth value can vary during the proof search. This means that the rewrite system $\mathcal{RE}$ that we want to use, may vary during proof search, which significantly complicates its use. Indeed, it means that, for instance, normal forms may evolve during proof search. While this may seem to be a specificity tied to the architecture of ArchSAT, it is actually more general, in the sens that the actual rewrite system that should be considered can actually change even when the rewrite rules are statically known at the beginning.

Consider the following simple unsatisfiable problem:

$$\begin{aligned} f(g(x)) &\longrightarrow x \\ a &= g(b) \\ b &\neq f(a) \end{aligned}$$

There is only one rewrite rule, and all ground terms are in normal form, and neither the theory of equality nor the theory of uninterpreted function can find a conflict using the two ground formulas $a = g(b)$ and $b \neq f(a)$. This show that simply normalizing every term according to a set of rewrite rules is not a complete strategy[3]. This is because theorem provers have to reason about the congruence induced by both the rewrite system and the equalities present in the problem. Both the rewrite system and the equalities must thus be considered together. One interesting and useful point is that those equalities are ground: indeed, in ArchSAT all decision

[1] LPO: Lexicographic Path Ordering

[2] Note that if $l \prec_{\text{LPO}} r$ according to the LPO, then it implies that $\text{FV}(r) \subseteq \text{FV}(l)$.

[3] The implementation of deduction modulo theories in Zenon actually fails to find a proof for this example.

and propagations are done on boxes, and all formulas inside boxes are ground (there may be variables quantified in the formula, but all variables in the formula are bound; however, there can be meta-variables, which act like ground terms). Similarly in most SMT provers, quantified axioms are handled separately (as explained in Section 1.2.5), and thus only ground equalities may occur.

There are two ways to reason with both a rewrite system and some equations:

- Rewrite rules can use the equivalence classes induced by the equalities in the problem in order to perform matching modulo these classes. Using this strategy on the example above, all the terms are no longer in normal form because $f(a)$ can match $f(g(b))$ using the equality $a = g(b)$, and thus can rewrite to b, which allows us to prove the unsatisfiability of the problem.

- Alternatively, one can see the equalities as rewrite rules: reduction ordering are almost always total on ground terms, thus equalities can always be oriented (for instance using a LPO, which is total on ground terms), thus all ground equalities can be seen as rewrite rules. In this case, we can see that not all terms are in normal form. Indeed, suppose that the equality $a = g(b)$ is oriented as $a \longrightarrow g(b)$. We can then rewrite $f(a) \longrightarrow b$. If the equality is oriented in the other direction: $g(b) \longrightarrow a$, then the rewrite system is not confluent anymore because of the critical pair: $f(g(b)) \longrightarrow b$ and $f(g(b)) \longrightarrow f(a)$. Completing the rewrite system can be done by adding the rewrite rule $f(a) \longrightarrow b$ (resp. $b \longrightarrow f(a)$), which allows us to normalize $f(g(b))$ into b (resp. b into $f(a)$). In both cases, we can conclude that the problem is unsatisfiable.

As the example above showed, even with what seems like a static set of rewrite rules, the rewrite relation that has to be considered changes during proof search. This means that normal forms are dependent on the proof search context: normalization of a term may have used an equality that is only true in one propositional branch, and so in other branches the normal form will be different. This means that normalizing terms according to a static rewrite system defined at the beginning of a problem is an incomplete strategy. However, it is quite useful in practice.

ArchSAT thus distinguishes three ways to use rewrite rules. The first way is static rewriting, which corresponds to the normalization of ground terms according to a fixed, static, rewrite system, and which is explained in Section 3.2. Dynamic rewriting, which considers the ground equalities in the problem, is presented in Section 3.3. Finally, rewrite rules must be integrated to the instantiation mechanism, more particularly to the unification algorithm. This is done using rigid unit superposistion, a variant of regular superposision which is described in Section 3.4.

3.2 Static Rewriting in ArchSAT

3.2.1 Normalizing Ground Terms in ArchSAT

In this section, we suppose that we have a set of rewrite rules. These rules are, in practice, detected by ArchSAT as explained in Section 3.1.3. We assume that this rewrite system is confluent and strongly terminating[1]. It is thus very easy to compute the normal form of a given term (or a formula) t according to this rewrite system. To do so, a very naive algorithm is to try and match each subterm of t with each left side of a rewrite rule, replace with the right-hand side if the match is successful, and repeat until no match can be found. Confluence guarantees unicity of the normal form if it exists, and strong termination of the rewrite system ensures that this process terminates (independently of the order in which the rewrites rules are used), and thus that the normal form exists.

Rewriting can then be implemented as a regular theory in ArchSAT with the simple strategy of trying to normalize every new atomic formula. More precisely, for each atomic formula (or rather formula in a box) that is decided or propagated, the theory will try and compute its normal form (if it has not already tried to do so). If a normal form (different form the initial atomic formula) is found, then the theory will push a new clause to the solver. Whenever the theory computes the

[1] Though, in practice it does not matter much, see Section 3.4.

normal form P' of an atomic formula P, using a list of rewrite rules $l = r_1, \ldots, r_n$, it will push the new following clause:

$$\mathcal{C} \equiv \vee_{1 \leq i \leq n} \neg [\![r_i]\!] \vee [\![P \leftrightarrow P']\!]$$

This clauses explicitly mentions the rewrite rules that were used in order to normalize P. This is because, similarly to the tableau theory described in Section 2.1.2, the clauses pushed by the rewriting theory must be tautologies (i.e. they should not need any context to be provable). Additionally, having the explicit list of rewrite rules used is very useful for generating formal proofs (which are detailed in Chapter 4), or even simply to compute the unsat core.

Note that the normalization is only done on atomic formulas, and not terms, nor arbitrary formulas. Normalization on terms would be somewhat redundant with the normalization on atomic formulas: since all terms considered in the problem occur inside formulas, normalization of the formula also normalizes any term inside it. Normalizing only atomic formulas, and not all formulas, is done in order to normalize only formulas that are actually used: some sub-formulas are sometimes not needed to prove a theorem. In these cases, normalizing only atomic formulas helps avoid having to compute the normal form of sub-formulas that are not reached during proof search.

This strategy is actually quite close to the trigger mechanism usually implemented in SMT solvers: watch the ground terms that occur during proof search and when some match is detected, instantiate the corresponding quantified formula. The main differences are:

- Instead of simply instantiating the quantified formula, rewriting actually substitutes inside terms using the instantiated formula (which is either an equality or an equivalence).

- Rewriting may perform multiple instantiations in a single clause: if multiple rewrite steps are needed to compute a normal form, all the steps are expressed using a single clause, rather than each step in its own clause[1].

Regarding completeness, this rewriting theory is, in general, incomplete as shown in Section 3.1.4. That is not a problem, as this theory is meant to speed up ground reasoning by performing automatic instantiations that do not introduce new propositional branches. Complete proof search is achieved using the unification algorithm presented in Section 3.4 that can unify terms while taking into account equalities as well as rewrite rules.

Finally, this strategy for handling rewriting is fully compatible with regular SMT solvers. Indeed, since the rewrite rules are supposed to be true for the whole problem, they could be omitted from the clause pushed, which then becomes clause with a single literal $[\![P \leftrightarrow P']\!]$, which can very easily be transformed into CNF before being added. The only potential problem is that it potentially introduces new terms, which can be expensive for regular SMT solvers[2], and would thus require to run the theory at the same time as the instantiation mechanism, compared to ArchSAT where it runs during the proof search and push new clauses immediately.

3.2.2 Example

Let us show what happens on a simple example using set theory. First, we introduce the type constructor and function symbols for set theory:

$$\begin{aligned} \text{set} &: \mathsf{Type} \to \mathsf{Type} \\ \in &: \Pi\alpha.\alpha \to \text{set}(\alpha) \to \mathsf{Prop} \\ \subseteq &: \Pi\alpha.\text{set}(\alpha) \to \text{set}(\alpha) \to \mathsf{Prop} \end{aligned}$$

[1] This makes proof generation slightly more complex, see Chapter 4.

[2] Congruence closure algorithms typically operate on a known and fixed set of terms, and adding new terms during proof search thus conflict with that. Thus, it can happen that adding new terms requires re-starting the ground proof search from scratch.

We consider the following rewrite rule defining set inclusion in terms of set membership::

$$R \equiv \forall \alpha : \mathsf{Type}. \forall s, t : \mathrm{set}(\alpha).\ s \subseteq t \Leftrightarrow (\forall x : \alpha.\ x \in s \Rightarrow x \in t)$$

Now let's try and prove the simple goal:

$$G \equiv \forall \alpha : \mathsf{Type}. \forall a : \mathrm{set}(\alpha). a \subseteq a$$

This problem translates to the following sequent $R \vdash G$. We will thus try and prove the sequent $R, \neg G \vdash \bot$ which is equivalent in classical logic. Hence, we start with two clauses: $\mathcal{C}_0 \equiv [\![R]\!]$, and $\mathcal{C}_1 \equiv \neg[\![G]\!]$. The solver directly propagates $R \leadsto_{\mathcal{C}_0} \top$ and $G \leadsto_{\mathcal{C}_1} \bot$. Once propagated, R is seen as a rewrite rule $s \subseteq t \longrightarrow (\forall x : \alpha.\ x \in s \Rightarrow x \in t)$ by the rewriting theory, hence we do not generate meta-variables for it since rewrite rule are handled better by the rewriting theory than by the generic theory for quantified formulas. On the other hand, since G is false, the solver applies the rules of the tableau theory and generate a fresh type constant and a fresh term constant for the negated universal quantifications, that we'll write τ_α and τ respectively. We then get the following clauses:

$$\begin{aligned}\mathcal{C}_2 &\equiv [\![G]\!] \vee \neg[\![\forall a : \mathrm{set}(\tau_\alpha). a \subseteq a]\!] \\ \mathcal{C}_3 &\equiv [\![\forall a : \mathrm{set}(\tau_\alpha). a \subseteq a]\!] \vee \neg[\![\tau \subseteq \tau]\!]\end{aligned}$$

The solver then propagates $[\![\tau \subseteq \tau]\!] \leadsto_{\mathcal{C}_3} \bot$. At that point, the rewriting theory can normalize $\tau \subseteq \tau$ into $(\forall x.\ x \in \tau \Rightarrow x \in \tau)$, and generate the clause:

$$\mathcal{C}_4 \equiv \neg[\![R]\!] \vee [\![\tau \subseteq \tau \Leftrightarrow (\forall x.\ x \in \tau \Rightarrow x \in \tau)]\!]$$

It is now enough for the tableau theory to apply its rules in order to prove that the problem is unsatisfiable. The tableau theory will, among other things, generate another fresh term constant, that we will write τ_x, and create the following clauses:

$$\begin{aligned}\mathcal{C}_5 &\equiv \neg[\![\tau \subseteq \tau \Leftrightarrow (\forall x.\ x \in \tau \Rightarrow x \in \tau)]\!] \vee [\![\tau \subseteq \tau \Rightarrow (\forall x.\ x \in \tau \Rightarrow x \in \tau)]\!] \\ \mathcal{C}_6 &\equiv \neg[\![\tau \subseteq \tau \Leftrightarrow (\forall x.\ x \in \tau \Rightarrow x \in \tau)]\!] \vee [\![(\forall x.\ x \in \tau \Rightarrow x \in \tau) \Rightarrow \tau \subseteq \tau]\!] \\ \mathcal{C}_7 &\equiv \neg[\![(\forall x.\ x \in \tau \Rightarrow x \in \tau) \Rightarrow (\tau \subseteq \tau)]\!] \vee [\![\tau \subseteq \tau]\!] \vee \neg[\![(\forall x.\ x \in \tau \Rightarrow x \in \tau)]\!] \\ \mathcal{C}_8 &\equiv [\![(\forall x.\ x \in \tau \Rightarrow x \in \tau)]\!] \vee \neg[\![\tau_x \in \tau \Rightarrow \tau_x \in \tau]\!] \\ \mathcal{C}_9 &\equiv [\![\tau_x \in \tau \Rightarrow \tau_x \in \tau]\!] \vee \neg[\![\tau_x \in \tau]\!] \\ \mathcal{C}_{10} &\equiv [\![\tau_x \in \tau \Rightarrow \tau_x \in \tau]\!] \vee [\![\tau_x \in \tau]\!]\end{aligned}$$

Upon propagation of $[\![\tau_x \in \tau]\!]$, a conflict will be detected in $\mathcal{C}_9$[1]. Since that conflict occurs before any decision is made, we can conclude that the problem is unsatisfiable.

3.2.3 Narrowing, Re-Quantification and Virtual Meta-variables

3.2.3.1 Motivation

Narrowing [72] is the idea of trying to unify rather than match the left-hand side of rewrite rules. Basically, it allows us to reason about the shape of potential meta-variables instantiations: for instance, if some meta-variable X was to be instantiated by a term that starts with g, then the term $f(X)$ could be rewritten using the rewrite rule $f(g(x)) \longrightarrow x$. Thus it might be interesting to instantiate X with a term $g(\ldots)$, which contains a hole yet to be determined. Fortunately, meta-variables exist especially to represent holes in terms that have not yet be chosen, thus we can try and instantiate X with $g(Y)$ where Y is a new meta-variable. Narrowing thus helps reduce (or narrow) the search space of potential instantiations for meta-variables, hence its name.

[1] Or, alternatively, if $\neg[\![\tau_x \in \tau]\!]$ is propagated, the conflict will be in $\mathcal{C}_{10}$

Narrowing has been implemented in ArchSAT, though its usefulness in practice is much more dependant on the choices that are made by the solver during proof search. First let us show an example of problem where narrowing may be needed (depending on what choice the solver makes). We consider once again, set theory, this time with some additional constructions to represent tuples:

$$\begin{aligned}
\mathrm{tup} &: \mathsf{Type} \to \mathsf{Type} \to \mathsf{Type} \\
\mathrm{fst} &: \Pi\alpha, \beta. \mathrm{tup}(\alpha, \beta) \to \alpha \\
\mathrm{snd} &: \Pi\alpha, \beta. \mathrm{tup}(\alpha, \beta) \to \beta \\
\mathrm{pair} &: \Pi\alpha, \beta. \alpha \to \beta \to \mathrm{tup}(\alpha, \beta) \\
\\
\mathrm{set} &: \mathsf{Type} \to \mathsf{Type} \\
\in &: \Pi\alpha. \alpha \to \mathrm{set}(\alpha) \to \mathsf{Prop} \\
\subseteq &: \Pi\alpha. \mathrm{set}(\alpha) \to \mathrm{set}(\alpha) \to \mathsf{Prop}
\end{aligned}$$

With the following rewrite rules:

$$\begin{aligned}
\mathrm{fst}(\_, \_; \mathrm{pair}(\_, \_; x, y)) &\longrightarrow x && R_1 \\
\mathrm{snd}(\_, \_; \mathrm{pair}(\_, \_; x, y)) &\longrightarrow y && R_2
\end{aligned}$$

And consider a situation with the following hypotheses, that actually occurred in practice[1], with some meta-variable X:

$$\begin{array}{ll}
a : \mathsf{Type} & \\
u, v : a & X : \mathrm{tup}(a, a) \\
t : \mathrm{set}(a) & r : \mathrm{set}(\mathrm{tup}(a, a))
\end{array}$$

$$\begin{aligned}
\mathcal{C}_1 &\equiv \neg[\![(v \in t)]\!] \\
\mathcal{C}_2 &\equiv [\![\mathrm{pair}(a, a : u, v) \in r]\!] \\
\mathcal{C}_3 &\equiv \neg[\![X \in r]\!] \vee [\![\mathrm{snd}(a, a; X) \in t]\!]
\end{aligned}$$

It then all depends on which literal in $\mathcal{C}_3$ is chosen by the solver. If it chooses that $\neg[\![X \in r]\!]$ is true, then the instantiation mechanism in ArchSAT will easily unify this with $[\![\mathrm{pair}(a, a : u, v) \in r]\!]$, and find the instantiation $X \mapsto \mathrm{pair}(a, a; u, v)$, which allows us to conclude that the problem is unsatisfiable. On the other hand, if it chooses that $[\![\mathrm{snd}(a, a; X) \in t]\!]$ is true, then simple unification is not enough, as the only atomic formulas in the branch are:

- $\neg[\![(v \in t)]\!]$
- $[\![X \in r]\!]$
- $[\![\mathrm{pair}(a, a : u, v) \in r]\!]$
- $[\![\mathrm{snd}(a, a; X) \in t]\!]$

Simple unification (without taking into account rewrite rules) is not enough to find any substitution in this case. The problem is still unsatisfiable, but in this case, proving it requires more than simple unification and normalization (all terms shown above are in normal form). The solution would be to somehow unify $\mathrm{snd}(a, a; X) \in t$ with $v \in t$. There are then two ways to do that:

[1] Problem `mem_ran_1.zf` of the bset problem benchmark, see Section 5.1.

- Either perform unification taking into account the rewrite rules. This is what the algorithm in Section 3.4 does.
- Or, perform narrowing, which in this case, means unifying $\text{snd}(a, a; X)$ with the left-hand side of the rewrite rule R_2: $\text{snd}(\_, \_; \text{pair}(\_, \_; x, y))$, which yields the substitution $\sigma = \{X \mapsto \text{pair}(a, a; \ldots, \ldots)\}$ which contains some holes. This substitution can then be used to instantiate the corresponding quantified formula, substituting the holes with new meta-variables. After that process, simple unification will be able to unify the generated term in order to correctly instantiate the meta-variables generated.

3.2.3.2 Narrowing and Re-Quantifying

Narrowing in ArchSAT works in the following way. Once a ground model has been found, at the same time as regular unification tries to find substitution to instantiate, rewriting also tries to look for substitutions by performing narrowing. In order to find substitutions, the rewriting theory tries and unify each term (or sub-term) that occurs in the problem with the left-hand side of a rewrite rule. When such a substitution succeeds, it will typically bind meta-variables to terms that have free variables, namely the variables used in the rewrite rule. These variables must therefore be re-bound when instantiating the rewrite rule.

More precisely, suppose that a term t has been unified with a rewrite rule $l \longrightarrow r$, resulting in a substitution σ, in which some variables $w_1, \ldots, w_n$ occur. The substitution σ is then split using clusters of quantified formulas as explained in Section 2.3.3. We thus get a finite set of substitutions $\sigma_0, \sigma_1, \ldots$. Each substitution σ_j binds some meta-variables that were generated by successive quantifications in some formula F (by definition of clusters). By definition, F_j starts with some quantifications, so let us write $F_j = \forall v_1, \ldots, v_m.\ P(v_1, \ldots, v_m)$. The substitution σ_j maps a subset $v_{i_1}, \ldots, v_{i_k}$ of the variables quantified in F_j, to terms that may contain variables in $\{w_1, \ldots, w_n\}$. Let us write $v_{i_{k+1}}, \ldots, v_m$ the variables quantified in F but not bound in the substitution. We can then generate the following clause in order to perform the instantiation:

$$\neg[\![\forall v_1, \ldots, v_m.\ P(v_1, \ldots, v_m)]\!] \vee [\![\forall w_1, \ldots, w_n, v_{i_{k+1}}, \ldots, v_m. P(\sigma(v_1), \ldots, \sigma(v_m))]\!]$$

With the convention that if x is not bound in σ, then $\sigma(x) = x$[1].

3.3 Dynamic and Conditional Rewriting

Dynamics rewriting is a trigger-like mechanism used by ArchSAT to correctly handle the general case of rewriting, taking into account equalities as well as rewrite rules that can become true or false during proof search. Additionally, the specificities of the McSat architecture allows us to extend this mechanism to also handle conditional rewrite rules with almost no performance cost.

3.3.1 Rewrite Rules as Triggers

As mentioned previously, there are two main ways to consider that the rewrite system can change during proof search:

- Consider that the pattern matching the left-hand side of rewrite rules can change, since it should match modulo a set of ground equalities (which can change during proof search)
- Consider equalities as new rewrite rules (which may break confluence of the rewrite system and require a completion mechanism to recover confluence)

ArchSAT mainly consider the first solution, and performs pattern matching modulo equality in order to find instances of rewrite rules. Pattern matching modulo a set of ground equalities is decidable: the ground equalities induce finite congruence classes, thus pattern matching can simply iterate over all elements of the congruence classes. In practice, some additional filtering

[1] Actually, in the definition, substitutions are always assumed to be total functions from variables to types and terms, and instead "not being bound" actually means being bound to itself.

is done in order to be more efficient, but the idea still remains the same. While this solves the problem of finding adequate instances of rewrite rules, there is still the problem of how to use these instances. A scheme similar to the one used for static rewriting could be used, building clauses using the rewrite rules used, the equalities used, and finally the equivalence between a term and its normal form. However, this could be problematic as each term could have a different form in each propositional branch of the solver, leading to an explosion of the number of clauses, and such lemmas would be extremely complex to prove formally (see Section 4). Instead, for each instance of rewrite rule found by pattern matching modulo equalities, ArchSAT simply instantiate the corresponding rewrite rule as a regular quantified formula.

Let us consider the example introduced in Section 3.1.4:

$$\begin{aligned}
\mathcal{C}_0 &\equiv [\![\forall x : t_y . f(g(x)) = x]\!] \\
\mathcal{C}_1 &\equiv [\![a = g(b)]\!] \\
\mathcal{C}_2 &\equiv [\![b \neq f(a)]\!]
\end{aligned}$$

The quantified formula in $\mathcal{C}_0$ is understood as a rewrite rule $f(g(x)) \longrightarrow x$ (either because it has been specified as one in the input problem, or because it has been detected automatically be ArchSAT). Static rewriting does not do anything on this example as mentioned previously, however, matching modulo equalities finds an instance $\sigma = \{x \mapsto b\}$ for the rewrite rule, by unifying $f(a)$ and $f(g(x))$ using the equality $a = g(b)$. ArchSAT then instantiates the rewrite rule using the substitution, as it would do for a substitution found by unifying terms for meta-variable instantiation, and generate the clause:

$$\mathcal{C}_3 \equiv \neg[\![\forall x : t_y . f(g(x)) = x]\!] \vee [\![f(g(b)) = b]\!]$$

Then, other theories, such as the ones for equality and uninterpreted functions, are able to complete the proof.

This strategy is complete[1] in cases where the only quantified formulas in the problem are rewrite rules. Indeed, this strategy computes all potential instances of rewrite rules, thus after these instantiations are added to the problem, ground reasoning is all that needs to be done. Thus assuming the rewrite system is confluent and strongly normalizing, and the ground reasoning is also complete, this strategy for instantiating rewrite rules is complete.

ArchSAT also handles adding and removing rewrite rules during proof search: since rewrite rules are actually quantified formulas, they can become true or false during proof search. The same mechanism that allows it to correctly handle equalities becoming true or false, also allows it to handle rewrite rules becoming true or false. This makes it able to handle, for instance, local rewriting.

Note that this is extremely similar to the trigger mechanism often used by SMT solvers. The main difference is that rewrite rules (like other quantified formulas) are handled the same as ground formulas, whereas in SMT solvers, quantified formulas are traditionally separated from the ground reasoning. Another important difference is that this strategy only applies to quantified formulas that can be understood as rewrite rules, compared to triggers which are used on all quantified formulas. This should thus provide a fragment of problems on which trigger offers a complete proof search strategy: problems with only rewrite rules and ground formulas, assuming the triggers chosen are the left-hand side of the rewrite rules.

Finally, the tableau theory presented in Section 2.1.2 could be viewed as some kind of higher-order rewriting or meta-triggers: each time a formula that matches some pattern is decided or propagated, the tableaux theory generate a new clause to be added to the solver. These new clauses can be seen as instances of higher-order lemmas which encode rewrite rules that can pattern match on non-atomic formulas[2].

[1] Although no proof is provided in this manuscript.

[2] With some special casing in order to generate clauses rather than equivalences.

3.3.2 Conditional Rules and McSAT

Interestingly, thanks to the assignments in McSat, this trigger-like strategy can be extended to also handle conditional rewrite rules. Conditional rewrite rules are rewrite rules that only apply when some condition is true, for instance, as show in Section 3.1.2, arrays can be axiomatized using conditional rewrite rules:

$$
\begin{aligned}
\mathrm{get}(\_;\_,\mathrm{create}(\_,x)) &\longrightarrow x\\
i = j \rightarrow \mathrm{get}(\_;i,\mathrm{set}(\_;j,x,\_)) &\longrightarrow x\\
i \neq j \rightarrow \mathrm{get}(\_;i,\mathrm{set}(\_;j,\_,t)) &\longrightarrow \mathrm{get}(\_;i,t)
\end{aligned}
$$

More generally, ArchSAT can consider conditional rules of the form $C_1 \Rightarrow C_2 \Rightarrow \ldots \Rightarrow C_n \Rightarrow l \longrightarrow r$ which are encoded as formulas of the form $\forall x_1,\ldots,x_m.C_1 \Rightarrow C_2 \Rightarrow \ldots \Rightarrow C_n \Rightarrow l \Leftrightarrow r$ or $\forall x_1,\ldots,x_m.C_1 \Rightarrow C_2 \Rightarrow \ldots \Rightarrow C_n \Rightarrow l = r$ modulo some conditions explained at the end of the section[1].

Whenever the left-hand side of a conditional rewrite rule is matched by the dynamic strategy explained above, instead of directly instantiating it, ArchSAT can wait until the conditions truth value can be computed, and then only instantiate the rule if all conditions are true. Computing whether some arbitrary formula (such as a condition of a rewrite rule) is true or false in a given proof search context is, generally, costly as it basically is the entailment problem that SMT solvers are trying to solve: is a given formula entailed (or rather provable) given a set of hypotheses ? Fortunately, assignments in McSat provide an easy and efficient way to compute the truth value of a formula: wait for it to be assigned, and look at whether the assigned value is $\top$ or $\bot$.

This requires conditions of rewrite rules to be able to be evaluated once a match for the left-hand side has been found. In cases where this does not hold, ArchSAT's current behaviour will be to consider that the conditions are not true, and thus it will not do anything. Particularly this requirement forbids conditional rewrite rules where a variable occur in a condition but not in the left-hand side of the rewrite rule. Such conditions in rewrite rules are used for instance in some axiomatization of the B method in order to encode typing. However, this requirement is not sufficient to ensure that a condition can be evaluated. A sufficient criterion would be to define the notion of an evaluable term as follows.

A term t is said to be evaluable with respect to a set of terms S iff :

- $t \in S$, or
- t is the application of an interpreted function symbol[2] to terms that are all evaluable with respect to S.

Intuitively, this defines the set of terms t whose value can be determined once the values of terms in S is known.

A conditional rewrite rule $C_1 \Rightarrow C_2 \Rightarrow \ldots \Rightarrow C_n \Rightarrow l \longrightarrow r$ is said to be evaluable iff all conditions $C_1, C_2, \ldots, C_n$ are evaluable with respect to the set S of all variables in l. This ensures that whenever a match σ is found for l, if all terms bounds to variables of l in σ are assigned, then all the guards of the rewrite rule will have a computable value. In practice, matching (even modulo equalities) a pattern p with a term t only binds variables to terms that already exist in t (or in the equalities), thus these terms will be assigned. Hence, an evaluable rewrite rule is guaranteed to effectively evaluate its conditions for every potential match. ArchSAT currently expects conditional rewrite rules to be evaluable[3], in order for this trigger-like mechanism to work correctly.

[1] ArchSAT also allows rewrite rules conditions can also be grouped using conjunctions instead or nested implications

[2] Interpreted symbols include the equality, addition, multiplication, ...

[3] However, no check and/or enforcement mechanism is currently implemented to ensure that property on conditional rewrite rules.

3.4 Unifying Modulo Equalities and Modulo Rewriting

3.4.1 Motivation

In the general case, the instantiation mechanism for the tableau theory presented in Section 2.3 requires to unify terms while taking into account the equalities that are present. This is illustrated on the following simple example, which starts with clauses $\mathcal{C}_0$, $\mathcal{C}_1$, $\mathcal{C}_2$, and where the tableau theory generates the clause $\mathcal{C}_3$, introducing the meta-variable X.

$$\begin{aligned}
&\mathcal{C}_0 \equiv [\![a = f(b)]\!]\\
&\mathcal{C}_1 \equiv [\![\neg p(a)]\!]\\
&\mathcal{C}_2 \equiv [\![\forall x : t_y.p(f(x))]\!]\\
&\mathcal{C}_3 \equiv \neg[\![\forall x : t_y.p(f(x))]\!] \vee [\![p(f(X))]\!]
\end{aligned}$$

A ground model is then easily found, and the instantiation mechanism can start looking for potential instantiations. In order to do so, as explained in Section 2.3.2, the true predicates are unified with the false predicates[1]. In this example, the only such pair of terms is $(p(f(X)), p(a))$, which cannot be unified directly. However, taking into account the equality $a = f(b)$, they could be unified using $\sigma = \{X \mapsto b\}$.

The actual problem to solve in the general case is rigid E-unification (also called rigid unification modulo equalities), and is defined as follows.

Definition 3.4.1. Rigid E-unification: Consider a set of equalities $\mathcal{S}$ and two terms (or atomic formulas) a and b, in all of which can occur some variables (in a, b, as well as in the equalities in $\mathcal{S}$). The rigid E-unification problem is then to find a substitution σ from variables to terms, such that $\mathcal{S}\sigma \vdash a\sigma = b\sigma$, i.e. once the substitution is applied a and b can be proven equal using the equalities in $\mathcal{S}$.

The rigid qualification is tied to the fact that in the substitution σ, each variable may be bound at most once. This problem can then be extended into the rigid E-unification modulo rewriting problem, which takes into account not only equalities, but also rewrite rules:

Definition 3.4.2. Rigid E-unification modulo rewriting: Consider a set of equalities $\mathcal{E}$, and two terms (or atomic formulas) a and b, in all of which can occur some variables (in a, b, as well as in the equalities in $\mathcal{E}$), and a formula rewrite system $\mathcal{R}$ and a term rewrite system $\mathcal{E}$. The rigid E-unification modulo rewriting problem is then to find a substitution σ from variables to terms, such that $\mathcal{S}\sigma \vdash a\sigma =_{\mathcal{RE}} b\sigma$, i.e. once the substitution is applied a and b can be proven equal using the equalities in $\mathcal{E}$ and the rewrite rules in $\mathcal{RE}$.

ArchSAT has to solve the rigid E-unification problem modulo rewriting in order to be complete, with the distinction that the rigid variables from the definition of the rigid E-unification modulo rewriting problem, are the meta-variables in ArchSAT.

We propose here an approach based on superposition with rigid variables, as in previous work by Degtyarev and Voronkov [45] and earlier work on rigid paramodulation [73], but with significant differences. First, in order to avoid constraint solving, we do not use basic superposition nor constraints. Second, we introduce a merging rule, which factors together intermediate (dis)equations that are alpha-equivalent: with multiple instances of some of the quantified formulas (amplification), it becomes important not to duplicate work. In this aspect, our calculus is quite close to labeled unit superposition [61] when using sets as labels. Third, unlike rigid paramodulation, we use a term ordering[2] to orient the equations.

[1] Members of disequalities are also unified, but there is none in the given example.

[2] The implementation uses a LPO.

3.4.2 Preliminary Definitions

We write $s \approx t \mid \Sigma$ (resp. $s \not\approx t \mid \Sigma$) for the unit clause that contains exactly one equation (resp. disequation) under hypothesis Σ (which is a set of substitutions). We write $\emptyset \mid \Sigma$ for the empty clause under hypothesis Σ. The meaning of $s \approx t \mid \Sigma$ is that for every $\sigma \in \Sigma$, $s \approx t$ is provable using the substitution σ for the meta-variables. We also define $\mathsf{rename}(e)$, where e is an (dis)equation, as follows: let σ map every meta-variable of e to a fresh variable, then $\mathsf{rename}(e) = e\sigma \mid \{\sigma\}$. For example, $\mathsf{rename}(p(X) \approx a)$ is $p(v_1) \approx a \mid \{X \mapsto v_1\}$.

As can be noticed, we keep a set of substitutions, rather than unit clauses paired with individual substitutions, in order to avoid duplicating the work for alpha-equivalent clauses. Indeed, because of rigid variables, we may have to generate multiple distinct meta-variables for the same formulas, leading to some alpha-equivalent clauses. It would be inefficient to repeat the same inference steps with each variant of these clauses. These sets of substitutions are initialized for every input formula by using the $\mathsf{rename}(.)$ function. Thus, clauses do not share any variable, though they may share meta-variables in their attached sets of substitutions.

The composition of two substitutions σ and σ', denoted by $\sigma \circ \sigma'$, is well-defined if and only if the domains of σ and σ' have no intersection. In this case, $\sigma \circ \sigma' \triangleq \{x \mapsto (x\sigma)\sigma' | x \in \text{domain}(\sigma)\}$. This definition extends to sets of substitutions: $\Sigma \circ \sigma' \triangleq \{\sigma \circ \sigma' | \sigma \in \Sigma\}$. We say that a substitution σ is compatible with another substitution σ' if and only if $\sigma \circ \sigma'$ is well-defined, and that it is compatible with a set of substitutions Σ if and only if $\sigma \circ \Sigma$ is not empty. We then have $\sigma \leq \sigma'$ if and only if $\exists \sigma''.\ \sigma \circ \sigma'' = \sigma'$. This notion also extends to sets of substitutions: $\Sigma \leq \Sigma'$ if and only if $\forall \sigma' \in \Sigma'.\ \exists \sigma \in \Sigma. \sigma \leq \sigma'$. The merging of two substitutions $\sigma \uparrow \sigma'$ is the supremum of $\{\sigma, \sigma'\}$ for the order $\leq$, if it exists, or $\bot$ otherwise. The merging of sets of substitutions is $\Sigma \uparrow \Sigma' \triangleq \{\sigma \uparrow \sigma' \mid \sigma \in \Sigma, \sigma' \in \Sigma'\ , \sigma \uparrow \sigma' \neq \bot\}$.

Inference step will almost always involve the merging of some sets of substitutions. It is important to note that the merging of two non-empty set of substitutions can yield an empty set of substitution, as this is what enforces the rigidity of meta-variables. For example, the resolution step between $p(x, x) | \{X \mapsto a\}$ and $\neg p(y, b) | \{X \mapsto y\}$ is not possible, because the result would need to map X to a and b, which is impossible because X is rigid.

3.4.3 Inference System

In Fig. 3.1, we present the rules for unit superposition with rigid variables. We adopt notations and names from Schulz's paper on E [79]. A single bar denotes an inference, i.e. we add the result to the saturation set, whereas a double bar is a simplification in which the premises are replaced by the conclusion(s). The relation $\prec$ is a reduction ordering, used to orient equations and restrict inferences, thus pruning the search space. Typically, $\prec$ is either a RPO or a KBO. The rules of Fig. 3.1 work as described below:

ER is equality resolution, where a disequation $s \not\approx t \mid \Sigma$ is solved by syntactically unifying s and t with σ, if σ is compatible with Σ.

SN/SP is superposition into positive or negative literals. A subterm of u is rewritten using $s \approx t$ after unifying it with s by σ. The rewriting is done only if $s\sigma \not\preceq t\sigma$, a sufficient (but not necessary) condition for a ground instance of $s\sigma \approx t\sigma$ to be oriented left-to-right.

TD1 deletes trivial equations that will never contribute to a proof.

TD2 deletes clauses with an empty set of substitutions. In practice, we only apply a rule if the conclusion is labeled with a non-empty set of substitutions.

ME merges two alpha-equivalent clauses into a single clause, by merging the sets of substitutions. This rule is very important in practice, to prevent the search space from exploding due to the duplication of most formulas. Superposition deals with this explosion by removing duplicates using subsumption, but in our context subsumption is not complete because rigid variables are only proxy for ground terms: even if $C\sigma \subseteq D$, the ground instance of C might not be compatible with the ground instance of D.

$$\text{ER}\ \frac{s \not\approx t \mid \Sigma}{\emptyset \mid \Sigma \circ \sigma}\ \text{if } \sigma = \text{mgu}(s,t)$$

$$\text{TD1}\ \frac{s \approx s \mid \Sigma}{\top} \qquad \text{TD2}\ \frac{s\ R\ t \mid \emptyset}{\top}\ R \in \{\approx, \not\approx\}$$

$$\text{ME}\ \frac{\rho(u) \approx \rho(v) \mid \Sigma \qquad u \approx v \mid \Sigma'}{\rho(u) \approx \rho(v) \mid \Sigma \cup (\Sigma' \circ \rho)}\ \rho \text{ is a variable renaming}$$

$$\text{SN/SP}\ \frac{s \approx t \mid \Sigma \qquad u\ R\ v \mid \Sigma'}{\sigma''(u[p \leftarrow t]\ R\ v) \mid \sigma'''}\ \text{if} \begin{cases} \sigma'' = \text{mgu}(u_{|p}, s) \quad u_{|p} \notin V \\ \sigma''(s) \not\preceq \sigma''(t) \qquad \sigma''(u) \not\preceq \sigma''(v) \\ \sigma''' \in (\Sigma \circ \sigma'') \uparrow (\Sigma' \circ \sigma'') \\ R \in \{\approx, \not\approx\} \end{cases}$$

$$\text{ES}\ \frac{s \approx t \mid \Sigma \qquad u[p \leftarrow s] \approx u[p \leftarrow t] \mid \Sigma' \cup \Sigma''}{s \approx t \mid \Sigma \qquad u[p \leftarrow s] \approx u[p \leftarrow t] \mid \Sigma'}\ \text{if} \begin{cases} \Sigma'' \neq \emptyset \\ \Sigma \leq \Sigma'' \end{cases}$$

$$\text{RP}\ \frac{s \approx t \mid \Sigma \qquad u \approx v \mid \Sigma'}{s \approx t \mid \Sigma \qquad u[p \leftarrow t] \approx v \mid \Sigma'}\ \text{if} \begin{cases} u_{|p} = s \\ s \succ t \\ \Sigma \leq \Sigma' \\ u \not\succeq v \text{ or } p \neq \lambda \end{cases}$$

$$\text{RN}\ \frac{s \approx t \mid \Sigma \qquad u \not\approx v \mid \Sigma'}{s \approx t \mid \Sigma \qquad u[p \leftarrow t] \not\approx v \mid \Sigma'}\ \text{if} \begin{cases} u_{|p} = s \\ s \succ t \\ \Sigma \leq \Sigma' \end{cases}$$

Figure 3.1: The Set of Rules for Unit Rigid Superposition

ES is a restricted form of equality subsumption. The active equation $s \approx t \mid \Sigma$ can be used to delete another clause, as in E [79]. However, ES only works if s and t are syntactically equal to the corresponding subterms in the subsumed clause C. Otherwise, there is no guarantee that further instantiations will not make $s \approx t$ incompatible with C. Moreover, C needs not be entirely removed. Only its substitutions that are compatible with Σ are subsumed.

RN/RP are rewriting of clauses, limited to using clauses with a single empty substitution in their set of substitutions, in order not to constrain the rewritten clause to use more specific substitutions.

Note that the rule **SN/SP** generates as many equations as there are in the set $(\Sigma\circ\sigma'') \uparrow (\Sigma'\circ\sigma'')$ because all substitutions may not always be merged. For instance, given two unit clauses:

$$f(x) = t | \{\{X_1 \mapsto x\}, \{X_2 \mapsto x\}\}$$
$$f(a) = v | \{\{X_1 \mapsto a\}\}$$

Rule **SP** allows us to derive two distinct equations $(t = v)\{x \mapsto a\}|\{\{X_1 \mapsto a\}\}$ and $(t = v)\{x \mapsto a\}|\{\{X_1 \mapsto a; X_2 \mapsto a\}\}$, which are non-mergeable, because the substitutions in their set of substitutions are not alpha-equivalent (they do not even have the same domain).

Rewrite rules can be integrated into the rigid unit superposition easily. In fact, a rewrite rule $l \longrightarrow r$ can be expressed as an equality with an hypothesis set consisting of a single trivial substitution $s \approx t \mid \{\emptyset\}$[1]. Since the trivial substitution is compatible with every substitution, it will never prevent any inference, thus allowing us to use the unit clause as many times as needed to rewrite terms without accumulating constraints, particularly using the rules RP and RN, whose

[1] The singleton set containing the trivial (or identity) substitution $\{\emptyset\}$, must not be confused with $\emptyset$, the empty set.

side conditions are always verified by rewrite rules. Rigid unit superposition therefore provides an algorithm for rigid E-unification modulo rewrite rules, as detailed in the next paragraph.

In order to add rewrite rules, we use the $\mathsf{rewr}(.)$ function, that takes as argument a rewrite rule $l \longrightarrow r$ (or a formula representing one), and returns the unit clause $l \approx r \mid$.

3.4.4 Main Loop

The goal of this variant of superposition is to solve the rigid E-unification modulo rewriting problem. Given a rewrite system $\mathcal{RE}$, a set of equalities $\mathcal{S}$, and two terms to unify a and b, the process is then the following:

1. Start with an empty state (i.e. a set of superposition clauses).
2. For each rewrite rule $l \longrightarrow r$ in the rewrite system $\mathcal{RE}$, add to the superposition state the clause $l \approx r \mid \{\emptyset\}$.
3. For each equality $t = u$ in $\mathcal{S}$, add to the superposition state the clause $\mathsf{rename}(t \approx u)$.
4. Add the clause $\mathsf{rename}(a \not\approx b)$ to the superposition state.
5. Saturate the superposition state using the inference rules, until the empty clause $\emptyset \mid \Sigma$ is found. The substitutions in Σ are solutions to the rigid E-unification modulo rewriting problem.

Note that the saturation could be run even after the first solutions have been found, in order to provide a decision procedure for the rigid E-unification modulo rewriting problem, which is at the very least semi-complete.

3.4.5 Example

To illustrate the previous procedure, let us consider the following problem, coming from set theory, where pair, fst, and snd are the constructor and destructors of tuples, f a function on tuples, and X a rigid variable:

$$\begin{array}{rcl} a & \preceq b & \\ \mathrm{pair}(\mathrm{fst}(x), \mathrm{snd}(x))) & \longrightarrow & x \\ \mathrm{fst}(a) & \approx & \mathrm{fst}(b) \\ p(a) & \not\approx & p(\mathrm{pair}(\mathrm{fst}(b), X)) \end{array}$$

The saturation process is then the following. We start with 3 clauses corresponding to the rewrite rule, the equality, and the pair of terms to unify respectively:

$$\begin{array}{llr} 1 & \mathsf{rewr}(\mathrm{pair}(\mathrm{fst}(x), \mathrm{snd}(x))) \longrightarrow x) & \mathrm{pair}(\mathrm{fst}(x), \mathrm{snd}(x)) \approx x \mid \{\} \\ 2 & \mathsf{rename}(\mathrm{fst}(a) \approx \mathrm{fst}(b)) & \mathrm{fst}(a) \approx \mathrm{fst}(b) \mid \{\} \\ 3 & \mathsf{rename}(p(a) \not\approx p(\mathrm{pair}(\mathrm{fst}(b), X))) & p(a) \not\approx p(\mathrm{pair}(\mathrm{fst}(b), y)) \mid \{X \mapsto y\} \end{array}$$

The saturation process then applies rule RN using clauses 2 and 3, replacing $\mathrm{fst}(b)$ by $\mathrm{fst}(a)$ in clause 3, because it is smaller (since our precedence is $a \preceq b$). Then, rule RN effectively performs narrowing using the rewrite rule in clause 1, in order to deduce that if $X \mapsto \mathrm{snd}(a)$, then $\mathrm{pair}(\mathrm{fst}(a), X)$ could be rewritten to a. This yields clause 5, which can then produce the empty clause using rule ER:

$$\begin{array}{llc} 4 & \mathrm{RN}(2,3) & p(a) \not\approx p(\mathrm{pair}(\mathrm{fst}(a), y)) \mid \{X \mapsto y\} \\ 5 & \mathrm{SN}(1,4) & p(a) \not\approx p(a) \mid \{X \mapsto \mathrm{snd}(a)\} \\ 6 & \mathrm{ER}(5) & \emptyset \mid \{X \mapsto \mathrm{snd}(a)\} \end{array}$$

This produce the substitution $\sigma = \{X \mapsto \mathrm{snd}(a)\}$ as solution of the unification modulo equalities and modulo rewriting. Note that none of the instantiation mechanisms presented earlier could solve this problem:

- Simple unification (using the robinson algorithm) cannot unify $p(a)$ and $p(\mathrm{pair}(\mathrm{fst}(b), X))$.
- All terms are in normal form (even considering pattern matching modulo equalities)
- Narrowing does not unify modulo equalities, and thus can not discover any potential instantiation in this case.

Chapter 4

Proof Generation in ArchSAT

Confidence in an automated theorem prover's results is complicated. On the one hand, most automated theorem provers have now been heavily tested on known problems so they may be considered as reliable. On the other hand, they still remain quite complex in terms of implementation, which means that bugs are almost unavoidable. Particularly, once a bug has been found in an automated theorem prover, it is hard to evaluate all the problems on which the bug is triggered, and thus in general all previous results of the prover might be invalidated. In order to avoid these situations, the best solution is to certify prover's results. This way, even bugs in the prover do not matter as long as the certificate can be checked independently.

There are two ways to certify the results of a prover : either certify the prover itself (i.e. the code of the prover) in order to get a certified prover, or make the prover output certificates along its results, to get what is called a certifying prover. Certifying a whole prover is actually quite hard, or even unrealistic in most cases, because of all the complex strategies and optimizations that are implemented. Nevertheless, there exist some certified provers [63] but none of the best current automated theorem provers are currently formally certified. Thus, producing a certificate for each result is the best way to certify results for provers. In the case of theorem provers, whose aim is to prove theorems, such certificates take the form of proof certificates, which can be checked by an external proof checker.

We can distinguish two different categories of proof certificates currently implemented in theorem provers: traces and formal proofs. Most automated theorem provers provide traces, which are certificates with no strict definition or requirements, and which are often a series of reasoning steps defined by each prover. These traces are thus relatively easy to produce, but quite difficult to verify, because each prover needs a dedicated checker for its traces, which is then not that independent from the prover. On the other hand, formal proofs are checked by formal proof systems (such as the Coq proof assistant), which define very strict reasoning rules. Formal proofs are therefore harder to produce, since the reasoning steps of the prover have to be translated into reasoning steps accepted by the formal system. In addition, formal proofs carry increased confidence, because formal proof systems are much more independent from a prover than a dedicated checker.

Unfortunately, most of the time, producing formal proofs is more a technical challenge than a theoretical one, and it is often ignored by automated theorem provers in favor of traces. That is not the case for ArchSAT, which directly outputs formal proofs in independently verifiable formats. In order to increase confidence even more, ArchSAT targets two independent formats, in order to decrease the impact of potential bugs in proof checkers. Currently, ArchSAT therefore produces proofs that are checkable either by the Coq proof assistant or the Dedukti logical framework.

This formal proof output relies on the Curry-Howard correspondence, which establishes a link between formulas and types as well as proofs and terms. In that correspondence, for each logic corresponds a type system in which types can represent formulas, and typing derivation represent proofs. For type systems which are syntax directed (as is frequently the case), the term at the root of a type derivation adequately encodes the whole derivation. This allows us to use the concept of proof term, since a term can then be used to represent a proof of the formula corresponding to its type. In the following, we will therefore repeatedly use types to represent formulas, and terms to represent proofs. ArchSAT currently expresses its proofs in the calculus of constructions [37], because it is the type system used in the proof assistant Coq, which also relies on the Curry-Howard correspondence. On the other hand, the Dedukti logical framework is not a proof assistant, but rather a type checker that can be parametrized over a set of rewrite rules. Thus, proofs expressed in Dedukti need to manually rely on the correspondence, encoding the required logic into rewrite rules, and then expressing the provability of a formula by exhibiting a term that has the corresponding type. ArchSAT will therefore use an encoding of the calculus of constructions in Dedukti, presented in Appendix C.

In order to output formula proofs, ArchSAT has its own internal representation of proof trees and proof terms following the Curry-Howard correspondence, which allows it to output different forms of the same proof, namely:

- Two Coq proof scripts (using simple tactics such as apply and intro). One where the whole proof is expressed using the internal proof tree presentation, and one where the resolution part of the proof is kept in an internal format[1].
- Following the Curry-Howard correspondence present in both tools, Coq and Dedukti proof terms can also be generated. These terms are elaborated by ArchSAT from its representation of the proof tree[2].
- Normalized Coq and Dedukti proof terms, in which β-redexes are reduced[3] (once again produced by ArchSAT by working on the proof term).

In the following, Section 4.1 explains how rudimentary proofs are obtained from the SAT, SMT or McSat algorithms, Section 4.2 explains how these proofs are then expressed formally. Finally, Section 4.3 and 4.4 deal with some important details in the implementation of the proof output, respectively the difficulty of relating proof search and formal proofs, and the structured representation of proofs in ArchSAT.

4.1 Resolution Proofs for SAT/SMT/McSat

4.1.1 Pure Resolution Proof Trees for SAT

It is known that the SAT, SMT, and McSat algorithms are capable of providing proofs whenever they reach UNSAT. From a theoretical point of view, the full derivation from any one of these algorithms is already a proof of its conclusion, since the algorithms are sound. However, in practice, these derivations are too big, and more importantly, they are not kept in memory when executing the algorithm[4]. Fortunately, it is possible to extract a small proof from both the derivation and the execution of these algorithms. In fact, we only have to be able to prove the clauses learnt after the analyze phase.

For example, let us consider a SAT derivation (using the inferences rules defined in Figure 1.1 of Chapter 1.1). It can be seen as a succession of Solve() and Analyze() phases, where each Analyze() phase adds a new clause to the set of known clauses through the Learn-Sat inference rule. We can therefore extract from any SAT derivation the ordered sequence of clauses learnt using the Learn-Sat rule. Let us write $\mathbb{H} = H_0, \dots, H_n$ the set of hypothesis clauses, and $\mathcal{C}_0, \mathcal{C}_1, \dots, \mathcal{C}_m$ the learnt clauses. According to the analyze phase invariants, we have that for all $0 \leq i \leq n$, $\mathcal{C}_i$ can be deduced from $\mathbb{H} \cup \{\mathcal{C}_0, \dots, \mathcal{C}_{i-1}\}$. Furthermore, as mentioned in Section 1.1.2, the way the $\mathcal{C}_i$ clause can be deduced is by using resolution between clauses in $\mathbb{H} \cup \{\mathcal{C}_0, \dots, \mathcal{C}_{i-1}\}$. Thus, for each $\mathcal{C}_i$, we can extract clauses from each application of the Analyze-resolution rule during its analyze phase, resulting in a list of clauses $\mathcal{C}_{i,0}, \mathcal{C}_{i,1}, \dots, \mathcal{C}_{i,k_i}$. The invariants of the Analyze() phase ensures that:

[1] Available with the options `--coqscript file.v` and `--coq file.v` respectively.

[2] Available using the `--dkterm file.dk` and `--coqterm file.v` options.

[3] Available using the `--dknorm file.dk` and `--coqnorm file.v` options.

[4] Indeed, most implementations of the SAT, SMT, or McSat algorithms tend to heavily rely on an imperative style of programming mutating a large data structure, and thus keeping the history of the proof search would be costly.

- $\mathcal{C}_{i,j} \in \mathbb{H} \cup \{\mathcal{C}_0, \dots, \mathcal{C}_{i-1}\}$,
- $\mathcal{C}_i$ is obtained by chaining resolutions steps using the list of clauses $\mathcal{C}_{i,0}, \dots, \mathcal{C}_{i,k_i}$, i.e. the following derivation is admissible[1]:

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{C}_{i,0} \qquad \mathcal{C}_{i,1}}{D_1}\text{resolution} \qquad \mathcal{C}_{i,2}}{D_2}\text{resolution}}{\vdots}}{} \qquad \dfrac{D_{k_i-1} \qquad \mathcal{C}_{i,k_i}}{\mathcal{C}_i}\text{resolution}$$

We can thus prove every $\mathcal{C}_i$ using previous clauses. This provides a way to prove every $\mathcal{C}_i$ using only hypotheses: using $\mathcal{C}_{0,0}, \dots, \mathcal{C}_{0,k_0}$, one can prove $\mathcal{C}_0$ using only the hypotheses, let us call this proof Π_0. Then, one can prove $\mathcal{C}_1$ using the hypotheses and $\mathcal{C}_0$, but since we can prove $\mathcal{C}_0$ using only the hypotheses, we can use the substitution lemma of the considered logic to simply replace in the proof of $\mathcal{C}_1$ any use of $\mathcal{C}_0$ as an axiom by its proof Π_0 in order to get a proof of $\mathcal{C}_1$ that only uses hypotheses.

This shows how to extract, for any clause $\mathcal{C}$ reached in a SAT derivation, a resolution proof tree where the conclusion is $\mathcal{C}$, and the leaves are the hypotheses. This includes the empty clause that is reached when the conclusion of the problem is UNSAT. In the UNSAT case, we can thus have a resolution proof tree whose leaves are the hypotheses of the problem, and whose conclusion is $\bot$, the empty clause, thus formally proving that the problem is unsatisfiable.

In the UNSAT example used in Section 1.1.3, we have proved that the following set of clauses is unsatisfiable:

$$\begin{aligned}
\mathcal{C}_1 &= P \vee Q \\
\mathcal{C}_2 &= \neg P \vee R \\
\mathcal{C}_3 &= \neg Q \vee R \\
\mathcal{C}_4 &= S \vee T \\
\mathcal{C}_5 &= \neg S \vee \neg R \\
\mathcal{C}_6 &= \neg T \vee \neg R
\end{aligned}$$

The resolution proof tree extracted from the UNSAT derivation is shown in Figure 4.1[2], where clauses are represented as rectangles, each containing a list of literals separated with a comma (instead of the usual $\vee$ logical disjunction), as well as a unique name (including for the clauses generated during resolution steps). Resolution steps are drawn using arrows pointing from the two clauses used in a resolution toward the conclusion of the resolution (with the resolved literal explicited in an oval shape).

Note that this extraction is actually quite easy and has very little cost. All that is needed is to store, for each learnt clause, the list of clauses used to deduce it during its analyze phase. This is one of the methods that can be used by SAT solvers in order to compute what is called an unsat core and which is actually a set of clauses that is enough to deduce UNSAT[3]. Note that computing the unsat core is similar to computing the set of clauses that occur in the leaves of a resolution proof tree; most provers simply only compute the set of leaves whereas in ArchSAT, the whole resolution proof tree is computed.

[1] The derivation is admissible in any logic where the resolution inference rule is admissible. This includes intuitionistic and classical logic, as well as most usual logics.

[2] This graph was automatically generated by ArchSAT using the `--res-dot` option

[3] In a lot of problems, not all hypotheses are needed to deduce UNSAT.

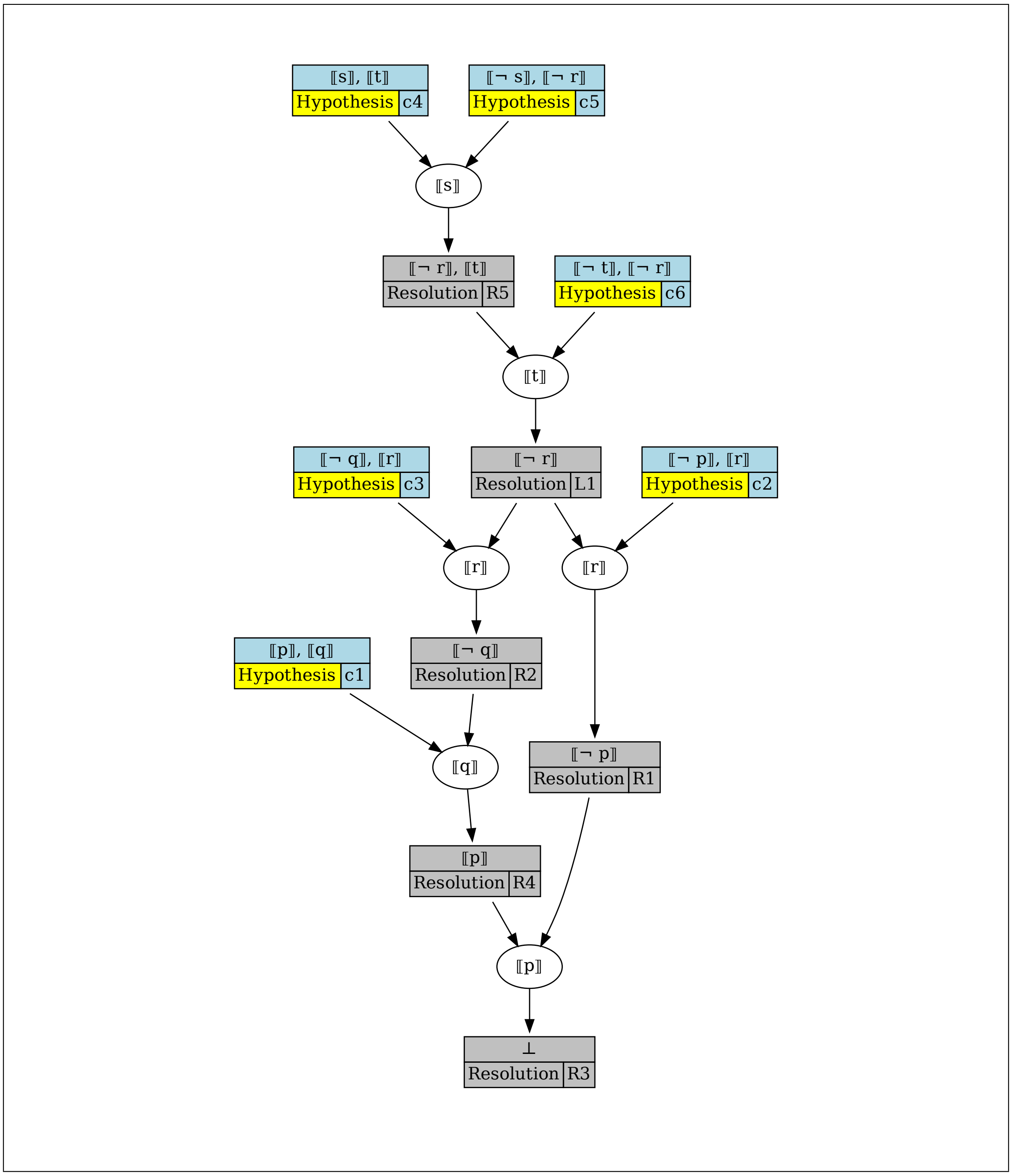


Figure 4.1: Resolution Proof Tree for a Simple SAT Problem

4.1.2 Resolution Trees for SMT and McSat

Producing proof for SMT provers is basically the same as for the SAT algorithm. It uses the same Analyze-resolution and Learn-Sat inference rules as the SAT algorithm. The only difference is that, due to the Conflict-Smt inference rule, the leaves of the resolution proof tree are now either an hypothesis or a tautology produced by the theory. This means that computing a full formal proof requires the theory to provide a way to output formal proofs for the tautologies that it produces and sends to the SMT algorithm (via the Conflict-Smt or Learn inference rules). Producing proofs in the case of McSat is actually exactly the same as for SMT, because the changes between SMT and McSat affect how conflicts are found rather that what conflicts are found[1].

For instance, consider the example shown in Section 1.2.3, where we have proved the following set of clauses to be unsatisfiable:

$$\begin{aligned}
\mathcal{C}_1 &\equiv a = b \\
\mathcal{C}_2 &\equiv b = c \lor b = d \\
\mathcal{C}_3 &\equiv \neg(a = d) \\
\mathcal{C}_4 &\equiv \neg(a = c)
\end{aligned}$$

We get the resolution proof tree of Figure 4.2, where some of the leaves are tautologies from the theory of equality, more precisely application of the transitivity of equality. Proving tautologies formally can range from being quite easy to being quite hard and/or tedious depending on the theory and formal proof system considered. For instance, the theory of equality (without uninterpreted functions) is quite simple: what we need to prove is only some instances of reflexivity, symmetry[2] and transitivity. On the other hand, theories such as rewriting, may have reasonably simple lemmas but are much harder to correctly handle as they require to do a lot of substitutions in terms (typically, in order to normalize a term), which can be quite hard to do properly. Finally some theories actually create tautologies that are themselves quite complicated: an example would be congruence closure theories, which can generate tautologies mixing pure equality reasoning with uninterpreted functions.

4.1.3 Theory Tautologies in ArchSAT

This section describes the tautologies (and associated proof strategies) currently used by the theories implemented in ArchSAT.

ArchSAT handles logical connectives, as well as first-order quantified formulas using the tableau theory (rather than pre-processing input formulas to transform them into CNF), as explained in Section 2.1.2. The tableau theory tautologies are presented in Figure 2.2. In this theory, quantified formulas offer some interesting choice points:

- First, existential formulas (or negated universal ones) can either use fresh constant symbols, of skolem symbols (as mentioned in Section 2.2). To simplify[3], whatever choice has been made during proof search, both are translated using epsilon-terms in the proof tree.

- Secondly, it may happen that meta-variables are used in the final proof. However, meta-variables are specific to proof search and are not directly translatable in most proof assistants. In order to overcome this problem, meta-variables are replaced by an arbitrary ground term of the corresponding ground type when generating the proof tree. In first-order logic this should always be possible as all types are considered inhabited. See Section 4.3.1 for more details on the interactions with the logics of proof assistants.

[1] More precisely, a McSat solver quite surely finds different conflicts that a SMT prover would, but both produce conflicts that are instances of the same theorems (such as transitivity for equality).

[2] Actually, symmetry of equality can be a little tricky depending on the internals of a theorem prover, see Section 4.3.2 for more information.

[3] And because Skolemization would not be easy to do considering the proof structure of ArchSAT.

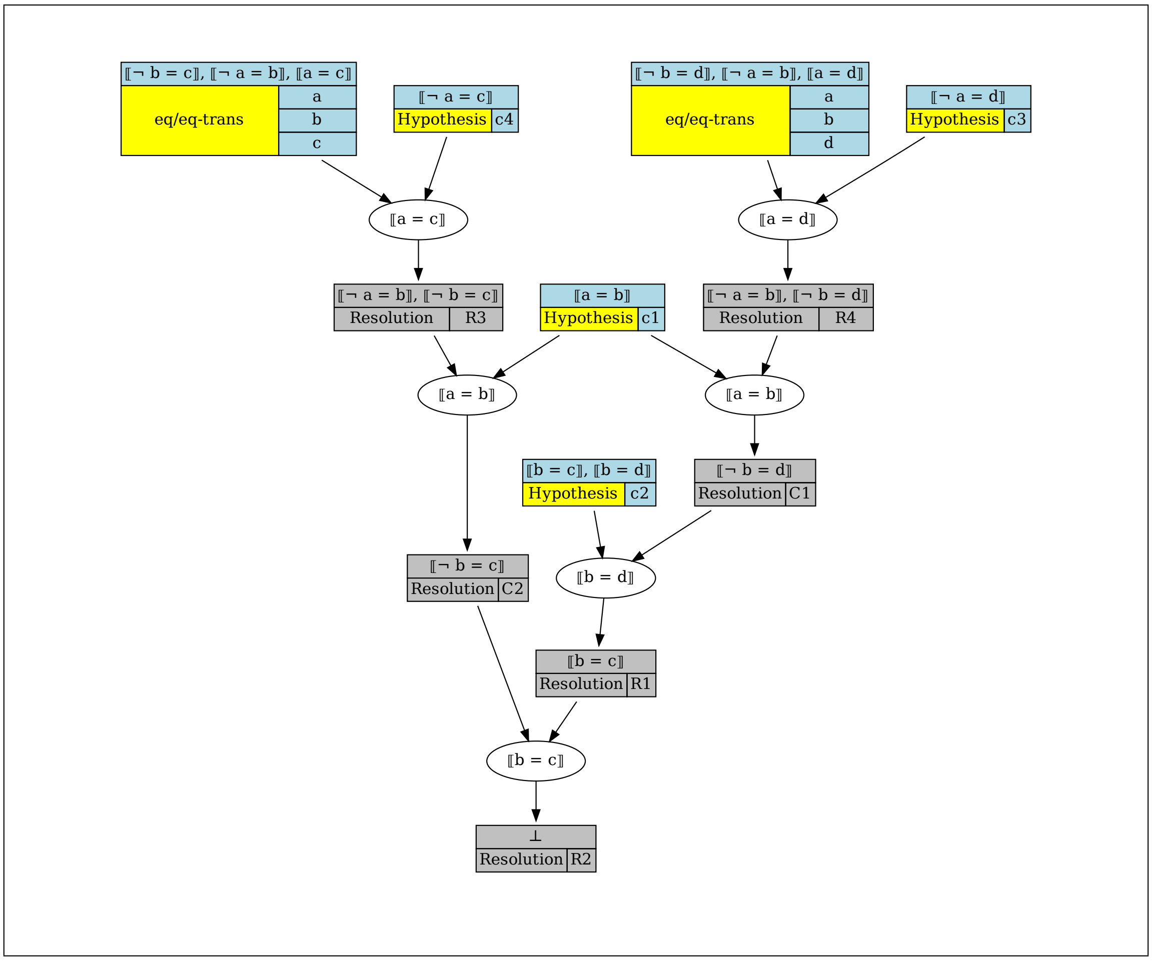


Figure 4.2: Resolution proof tree for a simple SMT problem

As mentioned earlier, the theory of equality has two main categories of tautologies:

Reflexivity tautologies are of the form $\mathcal{C} \equiv x = x$, where x is an arbitrary term.

Transitivity tautologies are of the form

$$\mathcal{C} \equiv \neg(x_1 = x_2) \vee \neg(x_2 = x_3) \vee \ldots \vee \neg(x_{n-1} = x_n) \vee (x_1 = x_n)$$

and are proven by using the basic transitivity lemma as many times as necessary.

Uninterpreted functions and uninterpreted predicates have respectively a category of tautologies:

Function tautologies are of the form

$$\mathcal{C} \equiv \neg(a_1 = b_1) \vee \neg(a_2 = b_2) \vee \ldots \vee \neg(a_n = b_n) \vee f(a_1, \ldots, a_n) = f(b1, \ldots, b_n)$$

Predicate tautologies are of the form

$$\mathcal{C} \equiv \neg(a_1 = b_1) \vee \neg(a_2 = b_2) \vee \ldots \vee \neg(a_n = b_n) \vee \neg p(a_1, \ldots, a_n) \vee p(b1, \ldots, b_n)$$

Finally, rewriting produce two kinds of lemmas:

Dynamic rewriting only generates pure instantiations of rewrite rules are produced, which can be discharged exactly like any other quantified formula instantiation.

Static rewriting produces lemmas of the form

$$\neg[\![r_1]\!] \vee \neg[\![r_2]\!] \vee \ldots \vee \neg[\![r_n]\!] \vee [\![P \Leftrightarrow P']\!]$$

where each r_i is a rewrite rule (i.e. a quantified formula interpreted as a rewrite rule), P an atomic formula, and P' is the normal form, computed using the rules $r_1, r_2, \ldots$.

4.2 Resolution Encoding and its Implication on the Classical Nature of Proofs

This section will describe in more details the formal logic in which ArchSAT internally expresses its proofs before printing them in an external format (for Coq, for example). One of the aim of ArchSAT is to be able to output formal proofs in different formats, in order to maximize confidence in the results. As bugs may also happen in formal proof assistants such as Coq, producing proofs for more than one proof assistant would therefore increase confidence.

Thus, ArchSAT targets more than one formal output, namely it currently can output proofs in the Coq proof assistant format, and the Dedukti logical framework. In order to support multiple proof outputs without duplicating work, ArchSAT has a structured internal representation of proofs (see Section 4.4 for more details), that uses higher-order terms with dependent types. This choice was made so that internal proofs would be able to be exported to Coq or Dedukti in a relatively straightforward manner. Particularly, this is one reason why internal proofs in ArchSAT do not use inductive types, since they are not present in Dedukti[1].

4.2.1 Clause encoding

Although clauses and resolution can be expressed and used in any axiomatization of first-order logic, there are some benefits in encoding them in a special way. Mainly, resolution between clauses may be complex to automatize because in formal logic, disjunction is binary. Thus, suppose we want to resolve two clauses $C = c_0 \vee \ldots \vee c_i \vee a \vee c_{i+1} \vee \ldots \vee c_n$ and $D = d_0 \vee \ldots \vee d_i \vee \neg a \vee d_{i+1} \vee \ldots \vee d_m$, using a, in order to deduce $E = c_0 \vee \ldots \vee c_i \vee c_{i+1} \vee \ldots \vee c_n \vee d_0 \vee \ldots \vee d_i \vee d_{i+1} \vee \ldots \vee d_m$. There are basically two options in order to prove E:

[1] Of course, inductive types can be encoded in Dedukti, but it is simpler to directly axiomatize first-order logic in Dedukti rather than encoded inductive types so that they can then be used to axiomatize first-order logic.

- Use a generic resolution lemma, but this requires to use symmetry lemmas for disjunction in order to re-order the clauses, so as to have: $C' = a \vee c_0 \vee \ldots \vee c_n$ and $D' = \neg a \vee d_0 \vee \ldots \vee d_m$, on which the generic resolution lemma can be applied.
- Manually split all disjunctions in C, with two main different cases:
  - For c_i (for any i), prove the corresponding atom in E
  - For a, split all disjunctions in D, where we also have two main cases:
    - For d_j (for any j), prove the corresponding atom in E
    - For $\neg a$, since we also have a, we can deduce E from the exfalso axiom, which states that any theorem is provable if $\bot$ is provable.

Both of these are actually quite cumbersome, and may end up generating quite big proofs. On the contrary, ArchSAT encodes clauses as implications (and thus, proofs of clauses as functions), following what is done in [29] for iProver Modulo. Note however that only clauses are encoded following [29], and even then with a slight difference. In superposisiton provers such as iProver Modulo, clauses are not simply disjunction of literals, but are also implicitly quantified over all free variables occurring in the clause. This is not the case in ArchSAT where clauses are disjunctions of ground arbitrary first-order formulas. Thus a clause $\mathcal{C} = c_0 \vee \ldots \vee c_n$ is encoded as:

$$\overset{\circ}{\mathcal{C}} =_{\text{def}} \neg c_0 \Rightarrow \neg c_1 \Rightarrow \cdots \Rightarrow \neg c_n \Rightarrow \bot$$

Note that the literals c_i are either formulas (without negation at the top) or single negations of formulas.

We will call this encoding of a clause, the weak form of a clause, or simply weak clause. It has the following three properties which makes it suitable for use in resolution proofs:

1. $\overset{\circ}{\bot} = \bot$
2. $\overset{\circ}{\mathcal{C}}$ can be proven from $\mathcal{C}$
3. Resolution is admissible between weak clauses (and results in the weak form of the usual resolution result): given two clauses $C = c_0 \vee \ldots \vee c_i \vee a \vee c_{i+1} \vee \ldots \vee c_n$ and $D = d_0 \vee \ldots \vee d_j \vee \neg a \vee d_{j+1} \vee \ldots \vee d_m$, which can be resolved into a clause $E = c_0 \vee \ldots \vee c_i \vee c_{i+1} \vee \ldots \vee c_n \vee d_0 \vee \ldots \vee d_j \vee d_{j+1} \vee \ldots \vee d_m$, we can perform resolution on $\overset{\circ}{C}$ and $\overset{\circ}{D}$, which results in the following proof of $\overset{\circ}{E}$:

$$\begin{aligned}
&\text{fun}(x_{c_0} : \neg c_0) \ldots (x_{c_n} : \neg c_n)(x_{d_0} : \neg d_0) \ldots (x_{d_n} : \neg d_n) \rightarrow \\
&\quad \overset{\circ}{C} x_{c_0} \ldots x_{c_i} (fun(x_a : a) \rightarrow \\
&\qquad \overset{\circ}{D} x_{d_0} \ldots x_{d_j} (fun(x_{\neg a} : \neg a) \rightarrow \\
&\qquad\quad x_{\neg a} x_a \\
&\qquad) x_{d_{j+1}} \ldots x_{d_m} \\
&\quad) x_{c_{i+1}} \ldots x_n
\end{aligned}$$

With these three properties, we can encode a whole resolution proof, with the following strategy:

- First, for each leaf of the resolution tree, prove the weak form of its clause.
  - For hypotheses, this can be done using property 2 of weak clauses.
  - For theory lemmas, it is better for the theory to directly prove the weak form of the lemma (see next section).
- Then each resolution of the proof tree can be performed on weak clauses, using property 3.

- Finally, we get the weak form of the empty clause, which is the empty clause as stated by property 1.

We can thus encode whole resolution proofs using weak clauses. If we again take the pure propositional example used in Section 1.1.3, whose resolution graph is in Figure 4.1, we can produce the proofs (in Coq as well as Dedukti) using weak clauses which is presented in Appendix D.

4.2.2 Tableaux Lemmas and Constructive Proofs

The encoding of clauses raises an interesting question about the proof of theory lemmas. For each theory lemma used in a resolution proof, one could either prove the lemma as a traditional clause and then prove the weak clause from that proof, or directly prove the weak form of the lemma. While both seem equivalent at first glance, there is one major difference in that proving implications (i.e. weak clauses) is easier than proving disjunctions (i.e. traditional clauses) in intuitionistic logic (without using the excluded middle).

It is easy to see that on the lemmas generated by the tableau theory presented in Section 2.1.2. Let us take for instance the lemma generated by rule $\beta_{\neg\wedge}$: $[\![P \wedge Q]\!] \vee \neg[\![P]\!] \vee \neg[\![Q]\!]$. This disjunction is not provable in intuitionistic logic. However, its weak form $\neg[\![P \wedge Q]\!] \Rightarrow \neg\neg[\![P]\!] \Rightarrow \neg\neg[\![Q]\!] \Rightarrow \bot$ is easily provable in intuitionistic logic.

This allows ArchSAT, a prover dedicated to classical logic, to output intuitionistic proofs whenever possible, i.e. when the proof found can be expressed in intuitionistic logic, it is output as a Coq proof that does not require classical axioms. There are two conditions for a proof found by ArchSAT to be intuitionistic:

- The goal has to be $\bot$, or be the negation of a formula. Indeed, ArchSAT always does proof by contradiction (like most theorem provers), using the negation of the goal to prove $\bot$. However, when the goal is $\bot$ (or alternatively, when there are no goal[1]), its negation cannot be useful for the proof, thus it is left as it is. When the goal is a negated formula, doing a proof by contradiction is actually the same as performing an introduction : since $\neg P$ if traditionally defined as $P \rightarrow \bot$, a proof of $\neg P$ is actually a function that takes some element of type P and returns an element of type $\bot$ according to the Curry-Howard correspondence.
- The tableau rules $\delta_{\neg\forall}$, $\gamma_{\neg\exists\text{M}}$, or $\gamma_{\neg\exists\text{inst}}$ are not used in the proof. These rules are used to transform a negated quantified formula into a (not negated) quantified formula using classical De Morgan's laws for quantifiers, which state that $\neg\exists x, P(x) \Leftrightarrow \forall x, \neg P(x)$, and $\neg\forall x, P(x) \Leftrightarrow \exists x, \neg P(x)$.

If these two criteria are fulfilled, the classical proof found by ArchSAT can be expressed in intuitionistic terms, and ArchSAT therefore produces an intuitionistic proof. Examples of sequents for which ArchSAT produce classical proofs therefore include the two examples in the previous section :

$$P \vee Q, \neg P \vee R, \neg Q \vee R, S \vee T, \neg S \vee \neg R, \neg T \vee \neg R \vdash \bot$$

And :

$$a = b, b = c \vee b = d, \neg(a = d), \neg(a = c) \vdash \bot$$

4.3 Relating Proof Search and Formal Proofs

This section explains some of the challenges associated with proof generation after proof search. More specifically, once proof search has found a proof, or rather has reached the conclusion that a proof exists, extracting a formal proof is not always straightforward because of many reasons, some due to the architecture of the code of a prover, and some of which require to answer some non-trivial theoretical questions. This section deals with the latter kind of challenges which I encountered during implementation of proof output.

[1] as it is often the case for problems expressed for SMT solvers in the SMT-LIB language.

4.3.1 Synthetizing Terms

The first challenge (and perhaps one of the most interesting from a theory point of view), is the creation of a term of a given type, which I called synthetizing. This should always be possible, or at least authorized in the context of first-order theorem provers, as first-order theory traditionally requires all types (or sorts) to be non-empty.

There are two cases in ArchSAT that require to synthetize terms:

- The use of epsilon-terms in proofs: formal epsilon-term specifications require the quantified type to be inhabited, which means generating a term of this type to prove it is inhabited[1].
- There may be meta-variables left in the proof tree at the end of proof search. For example, this case is encountered is when proving $\exists x : t_y, x = x$, where we generate a meta-variable X and discover that the formula $[\![X = X]\!]$ must be false, which is unsatisfiable regardless of the actual instance of X.

Since the specification of first-order logic actually allows us to use as axioms the inhabitability of types, the problem could be solved easily by adding axioms to the initial problem. However, it should not be necessary for a prover to add axioms to a problem to make it formally provable. Furthermore, in formal proofs generated by provers, the context of the proof (that is, the type definitions, the axioms and the goal that constitute the problem), ideally should **not** be generated by the prover, since the prover could (inadvertently) introduce inconsistent axioms, thus negating the use of the formal proof. It is therefore far better for the original formulation of a problem to allow the prover to prove type inhabitance without additional hypotheses.

For regular first-order problems, this could imply adding constants of all the types that occur in a given problem (which is already very often the case). The prover could then maintain an association table from types to terms. However, this way is incompatible with polymorphism : handling polymorphic symbols means that proof search may encounter types that do not exist in the original problem, particularly if the goal to prove is a universally quantified lemma[2], in which case a fresh type constant (or a skolem type constructor) would be generated and terms of this fresh type would be used during proof search. Moreover, the existence of type meta-variables brings another problem, since it means that when generating a term of a certain type, this type may contain type meta-variables which also should be substituted for another type. The choice of how to replace type meta-variables could therefore influence the possibility of generating terms of given types.

This demonstrates the need for a smarter algorithm to synthetize terms in order to properly handle all cases. However, this is not the focus of this thesis, and I choose to have an hybrid implementation that implements the following:

- Type meta-variables are always replaced by the same basic type defined by ArchSAT[3].
- Synthetizing a term of a given type operates a breadth-first search by iterating on the constants symbols defined in the input problem.
- The search is limited to a rather shallow depth[4]. If no appropriate term is found, a new constant of the required type is generated and added to the hypotheses of the problem, in order to not prevent the rest of the proof generation[5].

This strategy succeeds in most cases, since problems using polymorphism usually define basic constructors for polymorphic types (such as the empty list, the empty set, etc), which are found almost instantly.

[1] Adding an axiom specifying that all types are inhabited is wrong in proofs produced by ArchSAT since, in the Coq output for instance, no embedding is used, there is no way to distinguish a first-order type from other types, and specifying all types to be inhabited would obviously be incoherent in Coq.

[2] For instance if the goal is to prove some property on the length of appended polymorphic lists

[3] ArchSAT defines a basic type for elements. Among others, the TPTP $i type, which is the default type implicitly declared and used for terms in TPTP, is translated using this basic type.

[4] Currently the maximum depth is 5, as synthetizing do not require to create big terms on typical problems.

[5] A warning is also emitted to alert the user to this potential problem.

4.3.2 Proof Search Congruences

The other main challenge is the disparity between the congruences that automated theorem provers usually implement in order to speed up proof search, and their expression in a formal setting.

In practice, during proof search, the more terms (and/or formulas) can be identified with each other (i.e. considered equal), the better it is, since it reduces the total number of terms (or formulas) and limits potential instantiations. Provers tend to identify terms which usually have the same semantics, for instance $a = b$ and $b = a$ are often considered equal, logical connectives such as conjunction and disjunction do not have fixed arity (and instead take a list of arguments), etc. This is perfectly sound and helps proof search a lot, however, most formal proof assistants do not handle these congruences between terms (or formulas). Thus, when producing a formal proof, great care must be taken to correctly distinguish terms that were congruent during proof search but not in formal proofs, and then relate them using lemmas.

4.3.2.1 True and False

The most obvious congruence is the one between $\top$ and $\bot$. It is quite reasonable during proof search to consider $\neg\top \equiv \bot$ and $\neg\bot \equiv \top$. However formal proof systems do not consider those terms equal, though proving one side knowing the other is easy. Since this congruence is actually quite simple, it could be handled directly by adding a few corner cases to some part of the formal proof generation, though there are many places where it must be considered, and thus a likely source of easily fixable but annoying bugs.

In order to avoid having to manually handle these cases, ArchSAT actually has a generic mechanism to identify terms which are equivalent modulo some simple lemma applications, by extending environment lookups, see Section 4.4 for more details.

4.3.2.2 Equality Symmetry and Substitution

Equality symmetry is another usual congruence that considers equal the two formulas $a = b$ and $b = a$ for any pair of terms (a, b). However, contrary to the congruence between $\top$ and $\bot$, the special mechanism mentioned above cannot handle completely this case because it is much more complex.

The main problem is that in order to identify $a = b$ and $b = a$, one easy way is to order them using an arbitrary term ordering, thus having a canonical representation of the equality. However, applying substitutions may re-order equalities: consider the formula $\exists x : t_y, \exists y : t_y, x = y$, such that in the arbitrary term ordering used, x is smaller than y. Then, when we generate epsilon-terms ϵ_1 and ϵ_2 for x and y respectively, it may happen that ϵ_1 is bigger than y. Thus, after generating ϵ_1, we get the formula $\exists y : t_y, y = \epsilon_1$, where the order of equality has been reversed. This is a problem because the specification of epsilon-terms in formal proofs helps us prove $\exists y : t_y, \epsilon_1 = y$, which is a different formula in a formal proof assistant. There are two solutions to palliate this problem:

- After every instantiation or generation of epsilon-term, prove that the translated internal formula and the formula expected by the proof assistant are equal, which requires a lot of work and is costly for big formulas. Note that this option requires the prover to be able to compute the formula expected by the proof assistant, which most likely means having a separate structure for proof terms, which correctly keeps equality ordering through substitutions (thus not honoring the congruence) in order to know which formulas are to be proven equal.

- In the proof search terms, while keeping the congruence as is, annotate equalities with their original order and maintain these tags through substitutions, so that when they are translated to proof terms, the correct order can be used.

ArchSAT uses the latter option, making use of of the generic system for tagging terms and formulas as described in Section 6.3.

4.3.2.3 Binary Connectors vs. N-Ary Connectors

Quite identical to equality symmetry, logical connectors such as conjunction and disjunction are naturally treated as associative (and even commutative in some cases) by automated theorem provers, whereas they are not by proof assistants. This brings the same problems as equality symmetry does (formulas translated from proof search do not match those expected by the proof assistant), and can be solved the same way. Once again, ArchSAT tags disjunctions and conjunctions with their original tree structure, in order to translate them correctly when generating proofs.

4.4 A Structured Representation of Proofs

This section describes some technical details concerning the implementation of the proof output of ArchSAT. Historically, ArchSAT's first proof output targeted Coq proof scripts, and was achieved through a combination of carefully crafted printing functions. This required manually adjusting a lot of corner cases in the printing functions until everything worked. Then, Dedukti was targeted for a second proof output, but Dedukti does not have a notion of proof scripts like Coq has, which means that proof terms must be generated directly. Since Coq proof scripts generated from ArchSAT used only simple tactics (`apply`, `intro`, `exact`, `cut`, `pose proof`), the proof constructed by the printed proof script was actually directly expressible in Dedukti. However, since it used only printing functions, there was no real way to reuse the work that was done for Coq proof scripts. That was a real problem concerning the sharing of code between proof outputs, as it meant that the same work had to be done again (including fixing bugs).

In order to avoid duplicating work, I thus choose to have a structured representation of the formal proof generated by ArchSAT: instead of having an informal representation (such as the resolution tree) that is then printed in a formal language, the informal representation is first transformed into a structured formal proof, which then can be printed in various formats. This way, the reasoning part of the proof is shared between different formats, which then differ mainly by the function used to print terms in their syntax. As demonstration of the usefulness of this split, once the proof generation was implemented and tested (in conjunction with the Coq output), implementing the Dedukti output took only about a day of work.

More specifically, in the calculus of constructions in which ArchSAT expresses its proofs, terms and types are not syntactically distinct, thus proof terms will be used to represent proofs as well as types (and by extension, formulas). During proof generation, ArchSAT will first build a proof tree, similar to a sequent calculus proof tree or, equivalently, a typing derivation. This tree will naturally use proof terms to represent the goals and hypotheses that occur in sequents. This proof tree can be printed as a Coq proof script. In order to also support proofs in Dedukti, in which the proof tree can not be expressed directly, ArchSAT also implements the elaboration of this proof tree into the corresponding proof term. This term can then be printed in the Dedukti language, but also in the Coq language.

Since they are used pervasively in proof generation, proof terms will first be presented in Section 4.4.1, and then proof trees in Section 4.4.2.

4.4.1 Proof Terms

ArchSAT uses polymorphic, higher-order terms with dependant types for proof generation. This allows an easy correspondence with the calculus of constructions terms that are used in Coq. The syntax of proof terms in ArchSAT is presented in Figure 4.3. In order to keep the code as simple as possible, the term structure is really basic: application is curryfied and represented as a binary constructor[1] (a term is applied to another term) and a binder binds a single variable[2]. An additional term constructor is added to explicitly represent let-bindings, which basically are β-redexes: let $x = u$ in v is the same as $(\lambda x : t_y.v)u$ (assuming u is of type t_y). This redundant

[1] Whereas in proof search, application is n-ary

[2] Whereas in proof search binders binds a list of type variables and a list of term variables

$\mathcal{P} = \mathsf{Type}$	Type
$\mid\ x, y, z, \ldots$	variable
$\mid\ uv$	application
$\mid\ \text{let } x = u \text{ in } v$	let-binding
$\mid\ \lambda x : t.\ u$	Lambda
$\mid\ \forall x : t.\ u$	Universal quantification
$\mid\ \exists x : t.\ u$	Existential quantification

Figure 4.3: Syntax for the Proof Terms in ArchSAT

term constructor is introduced in order to enable a linear treatment of proofs[1]. This is particularly useful in resolution proofs, which may be extremely large, and where let-bindings offer a way to linearize the resolution graphs so that large proofs can be more easily constructed[2].

As usual, this thesis will use some basic notations for proof terms:

- Some basic constants are declared with their usual type: $\mathsf{Prop}, \top, \bot, =, \vee, \wedge$.
- As for proof search terms, successive quantifications may be concatenated.
- Implication is defined as $\Rightarrow \equiv \lambda P : \mathsf{Prop}, Q : \mathsf{Prop}.\ \forall v : P.\ Q$
- Negation is defined as $\neg \equiv \lambda P : \mathsf{Prop}.\ P \Rightarrow \bot$
- Equivalence is defined as $\Leftrightarrow \equiv \lambda t : \mathsf{Prop}, u : \mathsf{Prop}.\ (t \Rightarrow u) \wedge (u \Rightarrow t)$.

The type system used for proof terms by ArchSAT is shown in Figure 4.4. It uses the standard notions and notations for typing environments and typing statements. In this type system, the Type constructor is defined as impredicative, which although technically unsound does not really pose a problem, mainly because the proof generated is destined to be checked by a consistent proof checker such as Coq or Dedukti. Moreover, the aim of proof terms in ArchSAT is more to check that the conclusion of a reasoning step match the pre-condition of the next one, checking low-level details such as order of equalities. Most reasoning steps in proofs generated by ArchSAT are effectively first-order reasoning, and the higher-order and dependant types are mainly used in order to state the theorems needed in proofs (such as introduction and elimination principles, epsilon-term specification, . . .).

Proof terms in ArchSAT naturally have a notion of reduction, which actually is the β-reduction, extended to also reduce let-bindings, as well as a notion of definitions, where constants can be defined equal to terms, and are then expanded when reducing[3]. The notion of definition is useful to define usual logical connectives such as negation, and keeping them in non-reduced terms in order to have shorter and more readable terms.

This reduction is used in the typing system, where matching of types (in rules such as App, which requires types to have a certain shape) is performed on the reduced form of a type. Let us note that reducing all terms eagerly is not a good idea as some terms, such as proof terms, are rarely compared, contrary to others, such as proof term types, which are frequently compared. This motivates the need to distinguish a term and its reduced form.

However, reduction and polymorphism raised some non-trivial challenge during implementation concerning term comparison : in order to compare terms, one must consider the reduced forms

[1] Let-bindings are meant to be used in cases where the bound term t is small compared to u, allowing to perform some treatment on t before continuing treatment of u in very large terms, whereas regular term application does not have that kind of implicit usage constraints.

[2] It is easier to implement tail-recursive functions on let-bindings than applications.

[3] Rather than δ- reduction, this is more similar to having syntactic sugar for defined constants.

$$\frac{}{\Gamma \vdash \mathsf{Type} : \mathsf{Type}}\ \text{Type} \qquad \frac{}{\Gamma, x : t \vdash x : t}\ \text{Ax}$$

$$\frac{\Gamma \vdash u : \forall x : t_y.t \qquad \Gamma \vdash v : t_y}{\Gamma \vdash uv : t[x/v]}\ \text{App} \qquad \frac{\Gamma, x : t_y \vdash u : t}{\Gamma \vdash \lambda x : t_y.u : \forall x : t_y.t}\ \text{Abs}$$

$$\frac{\Gamma, x : t_y \vdash u : t}{\Gamma \vdash \forall x : t_y.u : t}\ \text{All} \qquad \frac{\Gamma, x : t_y \vdash u : t}{\Gamma \vdash \forall x : t_y.u : t}\ \text{Ex}$$

$$\frac{\Gamma \vdash u : t_y \qquad \Gamma, x : t_y \vdash v : t}{\Gamma \vdash \text{let } x = u \text{ in } v : t}\ \text{Let}$$

Figure 4.4: Type System for Proof Terms in ArchSAT

of the terms, and compare them modulo α-equivalence. Since type variables may be bound, reduction may change the type of a bound variable, thus resulting in a new fresh variable bound in the reduced term. This poses the problem of how to compare two non-equal terms, in order to have a total coherent order. Indeed, consider the following situation[1]

- Negation is defined as $\neg \equiv \lambda P : Prop, v : P.\bot$.
- Implication is defined as $\Rightarrow \equiv \lambda P : Prop, Q : Prop.\forall v : P.Q$.
- A term $t_1 \equiv \neg A$ is created. Its reduced form is $t_1' \equiv \forall v_1 : A.\bot$ (a fresh variable v_1 is created when substituting P in the definition of $\neg$).
- A term t_2 is created[2], whose reduced form is $t_2' \equiv \forall v_2 : B.C$.
- A term $t_3 \equiv A \Rightarrow \bot$ is created, whose reduced form is $t_3' = \forall v_3 : A.\bot$ (again, a fresh variable v_3 is created when reducing the definitions of the implication and negation).
- As expected t_1' and t_3' are alpha equivalent, thus t_1 and t_3 are equal.
- Suppose that for non alpha-equivalent terms, in the case of variables quantified by the same binder, comparison of terms performs the comparison of the unique id of the quantified variables. Then, we would get $t_1' < t_2' < t_3'$, which is inconsistent[3].

This example shows the difficulty of comparing term modulo alpha-equivalence (compared to computing equality modulo alpha-equivalence). While there may be algorithms, that correctly compare terms modulo alpha-equivalence (for instance, using deBruijn indices), I choose to hash-cons proof terms modulo alpha-equivalence. This way, each hashconsed terms can be assigned a unique integer, which can then be used to obtain a total order on terms.

Finally, for reference the OCaml type definition of terms can be found in Appendix B.

4.4.2 Proof Trees

Now that proof terms are defined, we can define sequents as a pair of a set of typed term identifiers (the axioms), and a term (which is the goal). A proof tree is then a tree[4] where each node is annotated with a sequent and a proof step. A proof step is basically a function from sequents to arrays of sequents, which returns the sequents needed to prove the input according to the proof step (if it returns an array of length 0, it means that there is no sequent left and the proof step actually closes the current branch). Proof trees can also have nodes annotated with a

[1] Which actually occurred during the implementation.

[2] for instance $B \Rightarrow C$.

[3] And leads to a subtle bug where a key is not found in a map although it is equal to one of the keys bound in the map.

[4] i.e. a directed acyclic graph.

sequent but no proof steps, in order to represent open (or incomplete) proof trees. This allows us to build a proof tree starting from the root, progressively eliminating open nodes using proof steps.

4.4.2.1 Proof Steps

Here is the description of the (fairly standard) basic proof steps used in ArchSAT:

- The introduction proof step takes as argument a sequent of the form $(\Gamma, \forall x : t_y, t)$, and compute an array of length 1 : $[|(\Gamma \cup \{x : t_y\}, t)|]$.
- The cut proof step takes as argument a sequent of the form (Γ, t) and a formula F, and produce an array containing two sequents to prove: $[|(\Gamma, F); (\Gamma \cup \{h : F\}, t)|]$, where h is a fresh identifier.
- The let-in proof step takes as argument a sequent of the form (Γ, t) and a term u of type F, and produces an array containing a single sequent: $[|(\Gamma \cup \{h : F\}, t)|]$ where h is a fresh identifier.
- The apply proof step takes as argument a sequent of the form (Γ, g) and a term f of type $g_1 \Rightarrow g_2 \Rightarrow \ldots \Rightarrow g_n \Rightarrow g$, and generate an array of n sequents: $[|(\Gamma, g_1); \ldots; (\Gamma, g_n)|]$. Note that the term f may contain function symbols declared globally (such as the transitivity lemma for equality), but also variables bound in the environment Γ. It thus subsumes the axiom rule in a lot of sequent-like calculus. This rule is also the only one that can generate 0 sequents, thus closing the current branch.

Additionally, ArchSAT defines a few extended proof steps to make the generation of proof trees from resolution trees easier:

- The duplicate proof step takes as argument a sequent of the form $(\Gamma \cup \{c_1 : \mathring{\mathcal{C}}_1\}, t)$, where $\mathring{\mathcal{C}}_1$ is the weak form of the clause $\mathcal{C}_1$, which contains some duplicate atoms. The proof step then returns an array with a single sequent: $(\Gamma \cup \{c_1 : \mathring{\mathcal{C}}_1, c_2 : \mathring{\mathcal{C}}_2\}, t)$ where C_2 is the same clause as $\mathcal{C}_1$ but where duplicates have been removed.
- the resolve proof step takes as argument a sequent of the form $(\Gamma \cup \{c_1 : \mathring{\mathcal{C}}_1, c_2 : \mathring{\mathcal{C}}_2\}, t)$, where $\mathring{\mathcal{C}}_1$ and $\mathring{\mathcal{C}}_2$ are two weak clauses, and produce an array consisting of a single sequent: $[|(\Gamma \cup \{c_1 : \mathring{\mathcal{C}}_1, c_2 : \mathring{\mathcal{C}}_2, c_3 : \mathring{\mathcal{C}}_3\}, t)|]$, where $\mathcal{C}_3$ is the result of resolution beetween $\mathcal{C}_1$ and $\mathcal{C}_2$ (and $\mathring{\mathcal{C}}_3$ its weak form).
- The mSAT backend is a big proof step that takes as argument a sequent of the form $(\Gamma, \bot)$, and a resolution tree and returns an array containing as many sequents as leaves of the resolution tree. Each sequent is of the form (Γ, L), where L is the lemma corresponding to the resolution tree leaf[1].

While these steps could be expressed using the basic steps described above, it is nonetheless useful to be able to have them expressed as single reasoning steps.

4.4.2.2 Positions and Tactics

Proof trees are built incrementally, starting from the root, and applying proof steps to open nodes until the tree is closed i.e. all nodes are annotated with a proof step. To do so, ArchSAT has a notion of position in proof trees, which intuitively represent open nodes. Proof steps may then be applied to a position, and generate an array of positions, which correspond to the array of sequents returned by the proof step. Positions also give access to their underlying sequent, which

[1] In practice, this proof step generates an empty array and takes care of proving each of the leafs internally, but it is due to purely technical considerations.

means that proof generation can match on the form of the goal, and query the axioms (also called the environment) before deciding what proof step to apply. This is interesting as it allows the implementation of tactics.

A tactic is a function that takes some positions within a proof, performs in-place modifications of that proof (typically, by applying some proof steps), and returns some value. The returned value can depend on the tactic, and is useful for tactics to either returns new positions within the proof, or indicate whether the tactic closed a branch. Tactics can also inspect the shape of sequents in order to determine what modifications to apply to the proof tree. Some concrete examples of tactics are shown in Appendix B.

4.4.2.3 Congruences

Congruences, as explained in Section 4.3.2 are classes of formulas that even though not equal, can be easily proven from one to another using known global constants (such as the equality symmetry lemma for relating $a = b$ and $b = a$, for example). Handling these cases manually is quite tricky and very quickly tedious because every proof step and tactic would need to take into account all of the congruences. In order to avoid this, a special mechanism is implemented. This is done by modifying the function performing lookups in the environment (i.e. the axioms). Most tactics perform lookup in the environment in order to look for a specific axiom before applying a proof step. Usually, lookups try and find the exact term that is looked up. ArchSAT instead allows us to register congruences that permit to lookup a list of alternate terms, and in case one of these term is found, wrap it appropriately in order to return a term of the required type.

Let us consider the case of equality symmetry. Suppose we have an environment $\Gamma = \{eq : a = b\}$. When a tactic tries and finds a term of type $b = a$ in the environment, the lookup fails. Instead of stopping here, the congruence for equality symmetry actually tries and look for $a = b$, which is found, and returns eq. In order to return a term of type $b = a$ as required, the congruence then wraps it using the equality symmetry lemma, which results in a term with the expected type. The interesting point of this mechanism is that congruences are defined once, and then every lookup in the environment will benefit from them.

ArchSAT currently uses two congruences as described in Section 4.3.2:

True and False There are 4 cases depending on which term is looked up:

- if it is $\top$, then no lookup is performed and the constant proof of true (let us call it I) is returned.
- if it is $\bot$ then $\neg\top$ is looked up. If the lookup is successful and returns e, then eI is returned
- if it is $\neg\top$, then $\bot$ is looked up. If the lookup is successful and returns e, then $\lambda x : \top.\ e$ is returned
- if it is $\neg\bot$, then no lookup is performed and the constant term $\lambda x : \bot.\ x$ is returned.

Equality Symmetry Again, there are 2 cases depending on which term is looked up:

- if it is $a = b$, then $b = a$ is looked up. If it is successful, then the term found is wrapped using the equality symmetry lemma[1].
- if it is $\neg(a = b)$, then $\neg(b = a)$ is looked up. If it is successful, then the term found is wrapped using the disequality symmetry lemma[2].

Note that these congruences deal with terms with at most one negation at the top. The reason why this is actually enough is explained in the next Section.

[1] The equality symmetry lemma states that $\forall t : \mathsf{Type}, a : t, b : t.a = b \Rightarrow b = a$

[2] The disequality symmetry lemma states that $\forall t : \mathsf{Type}, a : t, b : t.a \neq b \Rightarrow b \neq a$

4.4.3 From Resolution Tree to Proof Tree

ArchSAT transforms the resolution tree into a proof tree by folding over the nodes of the resolution tree in a topological order (from the leaves to the root), in order to ensure that when translating a node of the resolution tree, its children have been translated (and more importantly their conclusion added to the environment). Each node is translated using a let-binding that brings its conclusion into the environment (i.e. the axioms). There are then 4 distinct cases depending on the node of the resolution tree:

- The node is a resolution. The resolution proof step is used to deduce the conclusion of the node, which is the result of resolution.
- The node is a duplication node. This happens when an input clause[1] contains more than one occurrence of some literal. Since some algorithms for boolean constraint propagation in the core solver relies on clauses not containing duplicates, these duplicates are eliminated. The duplicate proof step is used to deduce the de-duplicated weak clause.
- The node is a leaf, corresponding to a hypothesis. In this case, we have a hypothesis $\mathcal{C} \equiv c_0 \vee \ldots \vee c_n$, and we want to prove $\overset{\circ}{\mathcal{C}} \equiv \neg c_0 \Rightarrow \ldots \Rightarrow \neg c_n \Rightarrow \bot$. This is done easily by introducing all of the $\neg c_i$, and then splitting the disjunction of the hypothesis clause. Each case c_j is then easily closed because $\neg c_j$ has been introduced.
- The node is a lemma. The proof of the weak clause is left to the theory. This is typically done by introducing all atoms of the weak clause, and then proving $\bot$, which is usually quite easy. These proofs vary much more than the proofs done for other nodes, as each theory typically has multiple distinct lemma patterns to prove (for instance, equality reflexivity, equality transitivity, equality symmetry, ...). The code building these proofs is of significant size and complexity (due to the number of different lemmas that need to be proven). It is also code that would need to be duplicated if proof output was done with nested printing function. Therefore, it is the part of proof generation that benefits the most from the structured proof representation.

Theory lemmas in ArchSAT are mostly quite easy to prove. The various lemmas that theories in ArchSAT currently produce are described in Section 4.1.3. Following is an explanation of how each are proved. Note that these proofs rely on an axiomatization of first-order logic to provide some basic lemmas. The required axiomatization lemmas can be found in the files `cc.dk`, `logic.dk`, `classical.dk` and `epsilon.dk` in Appendix C. This appendix shows the full first-order axiomatization that is used in Dedukti proofs produced by ArchSAT. For Coq proofs, the standard library is used, as it already contains all the necessary axioms and lemmas.

Equality reflexivity lemmas are direct applications of the reflexivity of equality found in the axiomatisation of first-order logic.

Equality transitivity lemmas are also found in the axiomatisation of first-order logic.

Equality symmetry lemmas are essentially used in the congruence system, and are also taken from the first-order axiomatization.

Function application lemmas in general are proved by using as many times as needed the Leibniz characterisation of equalities : $\forall A : \mathsf{Type}, x : A, y : A, P : A \Rightarrow \mathsf{Prop}.\ x = y \Rightarrow Px \Rightarrow Py$. As a special case, when there are only 1 or 2 arguments, specialized lemmas (such as Coq's `f_equal` and `f_equal2`) are used to produce shorter, more readable proofs.

Predicate applications lemmas are substantially the same as the ones for function applications.

Logical connective lemmas corresponding to tableau rules α and β (see Figure 2.2) are simple to prove using the elimination principles[2] on logical connectives[3].

[1] Input for the McSat inference rules, i.e. either a hypothesis or a theory tautology

[2] The elimination principles are present in the standard library of Coq, as well as in the axiomatization of first-order used by Dedukti proofs (see Appendix C).

[3] Again, sometimes with argument re-ordered so as to prove branches in the intuitive order in Coq proof scripts.

Epsilon-terms are introduced using Hilbert's choice operator, eventually after having applied de Morgan's Laws for quantified formulas (in the case of a negated universal formula).

Instantiations are more complex. As explained in Section 3.2.3, it happens that quantified formulas are instantiated with terms that contain free variables, in which case these free variables are quantified in the resulting formula. As for epsilon-terms, de Morgan's law for quantified formulas are used on negated existential formulas in order to get universal formulas

Dynamic rewriting is exactly the same as instantiation.

Static rewriting lemmas are of the form:

$$\neg[\![r_1]\!] \vee \neg[\![r_2]\!] \vee \ldots \vee \neg[\![r_n]\!] \vee [\![P \Leftrightarrow P']\!]$$

where each r_i is a rewrite rule (i.e. a quantified formula interpreted as a rewrite rule), P an atomic formula, and P' its normal form, computed using the rules r_i. In order to prove such lemmas, each rewrite rule is instantiated with appropriate ground terms using the same process as for regular instantiations. Then the goal $P \Leftrightarrow P'$ can be proven starting from the reflexivity of equivalence $P \Leftrightarrow P$, and applying formula rewrites using transitivity of equivalence, and term rewrites using Leibniz's characterization of equality.

At that point, it might be interesting to note that since regular clauses allow at most one negation at the top of each formula, and the weak encoding of clauses adds one negation to all atoms of a clause, when proving a lemma (after introduction of the atoms), all formulas have at most two negations at the top. Moreover, since the goal at that point is to prove $\bot$, double negation in axioms can be eliminated. Suppose a sequent of the form $(\Gamma \cup \{h : \neg\neg H\}, \bot)$, then by applying h and then doing an introduction, we get the following sequent $(\Gamma \cup \{h : \neg\neg H, h' : H\}, \bot)$. This allows us to only ever have to consider term with at most one negation at the top, particularly in congruences. On the other hand, some care must be taken when eliminating double negations, as it might change the produced proofs. Consider a trivial clause $\mathcal{C} \equiv [\![P]\!] \vee \neg[\![Q]\!]$ where $P = Q$ (but we keep distinct names in order to better follow how each blackbox is translated). Its weak form is : $\neg[\![P]\!] \Rightarrow \neg\neg[\![Q]\!] \Rightarrow \bot$. We thus have to prove $\bot$ knowing $H_0 : \neg[\![P]\!]$, $H_1 : \neg\neg[\![Q]\!]$. The simplest way to do so is to apply H_1 to H_0, which is to apply the encoding of $\neg[\![Q]\!] = \neg[\![P]\!]$ to the encoding of $[\![P]\!]$, which is intuitive. However, if we eliminate the double negation in H_1 in order to get $H_2 : [\![Q]\!]$, then the simplest way is to apply H_2 to H_1. While in this example it is not complex, most lemma proofs need to be careful in how they handle double negations. It is actually one point where having a structured proof helps, because the lemmas can then simply look in the environment to see whether a term is present, and decide how to prove the lemma using this information, whereas proving lemmas using printing functions would not usually have this information.

ArchSAT can then output the whole proof tree:

- as a graph using the `--full-dot file.gv` command line option
- as a coq proof script using either the `--coq proof.v` (using the mSAT backend proof step to translate the resolution proof) or the `--coqscript script.v` (translating each resolution using the resolution proof step). The difference between these two output is that the mSAT backend does not build the structured proof of each resolution, and instead uses an arrangement of printing functions to handle the resolutions (the theory lemmas are sill proved using the structured representation)[1].

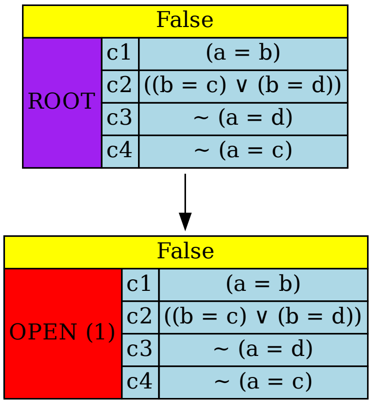
4.4.4 Proof Tree Example

Let us see how the transformation from resolution tree to proof tree operates. We will consider the small example using equalities shown before :

[1] The current implementation of structured proof is slower than simple printing functions, hence on large pure SAT problems, the structured proof can take a long time to be built.

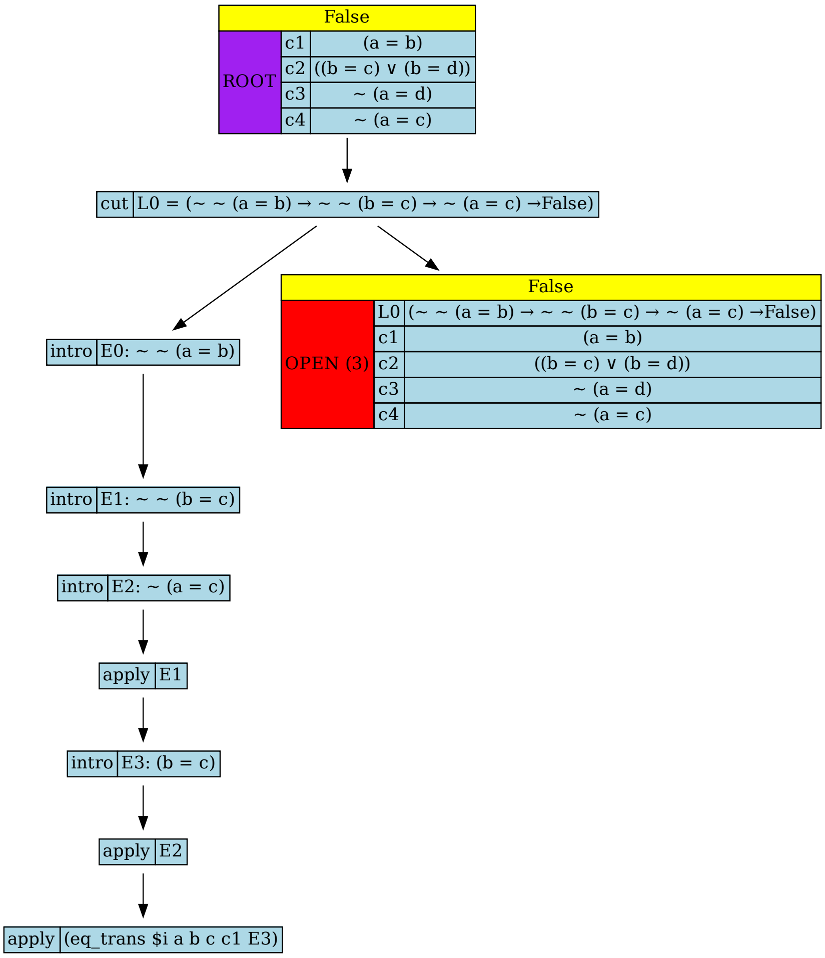
$$\begin{aligned}
\mathcal{C}_1 &\equiv a = b \\
\mathcal{C}_2 &\equiv b = c \lor b = d \\
\mathcal{C}_3 &\equiv \neg(a = d) \\
\mathcal{C}_4 &\equiv \neg(a = c)
\end{aligned}$$

This problem is unsatisfiable, and the resolution tree for the unsatisfiability proof is shown in Figure 4.2. In order to build the structured proof tree from that resolution tree, ArchSAT starts with a single node stating the sequent that needs to be proven. We will represent proof trees using graphs, where each node is drawn using a rectangle. The root of the tree, at the top of the graph, shows the goal with yellow background, while the hypotheses are listed under it. Arrows are then used to link a node to its children. Open nodes use a red background[1] and show the sequent that needs to be proven in the same fashion as the root node. The graph for the empty proof tree corresponding to the SMT problem above is the following:



Then, each node of the resolution tree is translated, starting from the leaves. The current order of these translations is unspecified (apart from being a topological order). In this example, the first node to be translated is the proof of the lemma $L_0 \equiv \neg(a = b) \lor \neg(b = c) \lor (a = c)$. This results in the following proof tree:

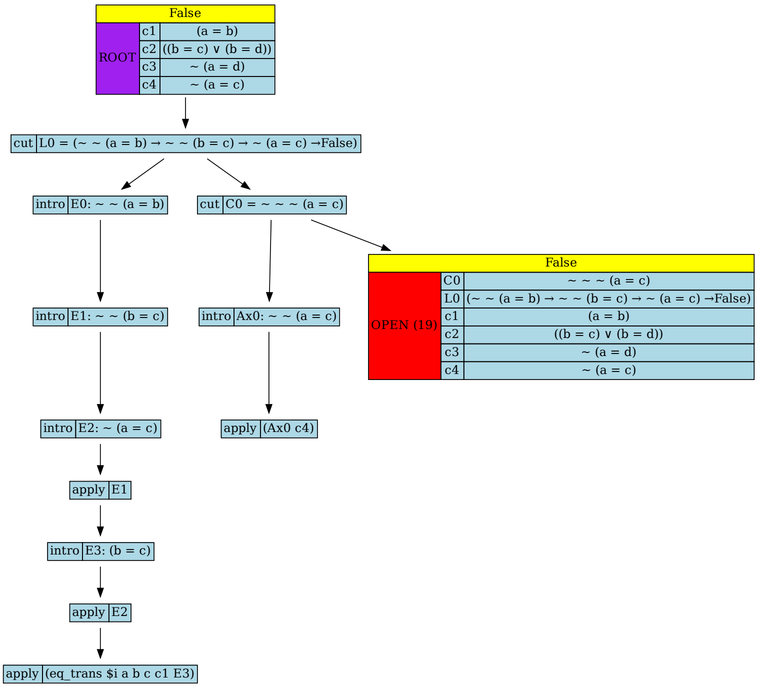
[1] The number in parenthesis in open nodes is a unique node identifier useful for debugging.



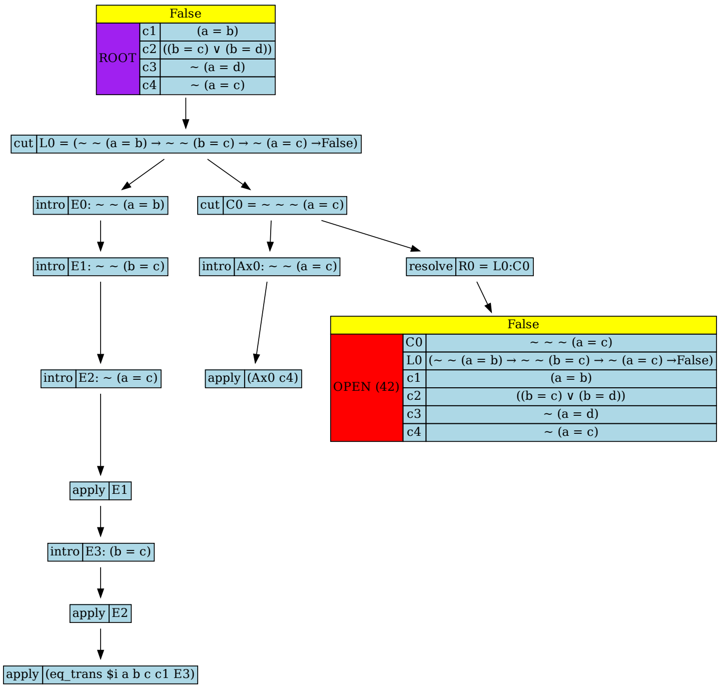
One can see the cut step introducing the weak form the clause L_0. On the left of the cut step is the proof of the clause : it starts by introducing each of the literals of the weak clause, then eliminates the double negation of $E_1 : \neg\neg(b = c)$ in order to get an axiom $E_3 : (b = c)$. Interestingly, the same process (eliminating the double negation), is not done for $E_0 : \neg\neg(a = b)$, because there is already an hypothesis stating that $a = b$. After that, the disequality $a \neq c$ can be proven to be inconsistent because of the transitivity of equality.

On the right of the cut step, there is an open node where the rest of the translation of the resolution tree will be performed. The next operation performed is to translate the input clause $\mathcal{C}_4 \equiv \neg(a = c)$, into its weak form $\mathring{\mathcal{C}}_4 \equiv \neg\neg(a = c) \Rightarrow \bot$, which is the same as $\neg\neg\neg(a = c)$. Proving the weak clause is done by introducing $Ax_0 : \neg\neg(a = c)$, and then proving $\bot$ using the input clause $\mathcal{C}_4$[1]. This results in the following proof tree :

[1] In the general case, when clauses have more than one literal, this part requires to split the disjunctions in the input clause.

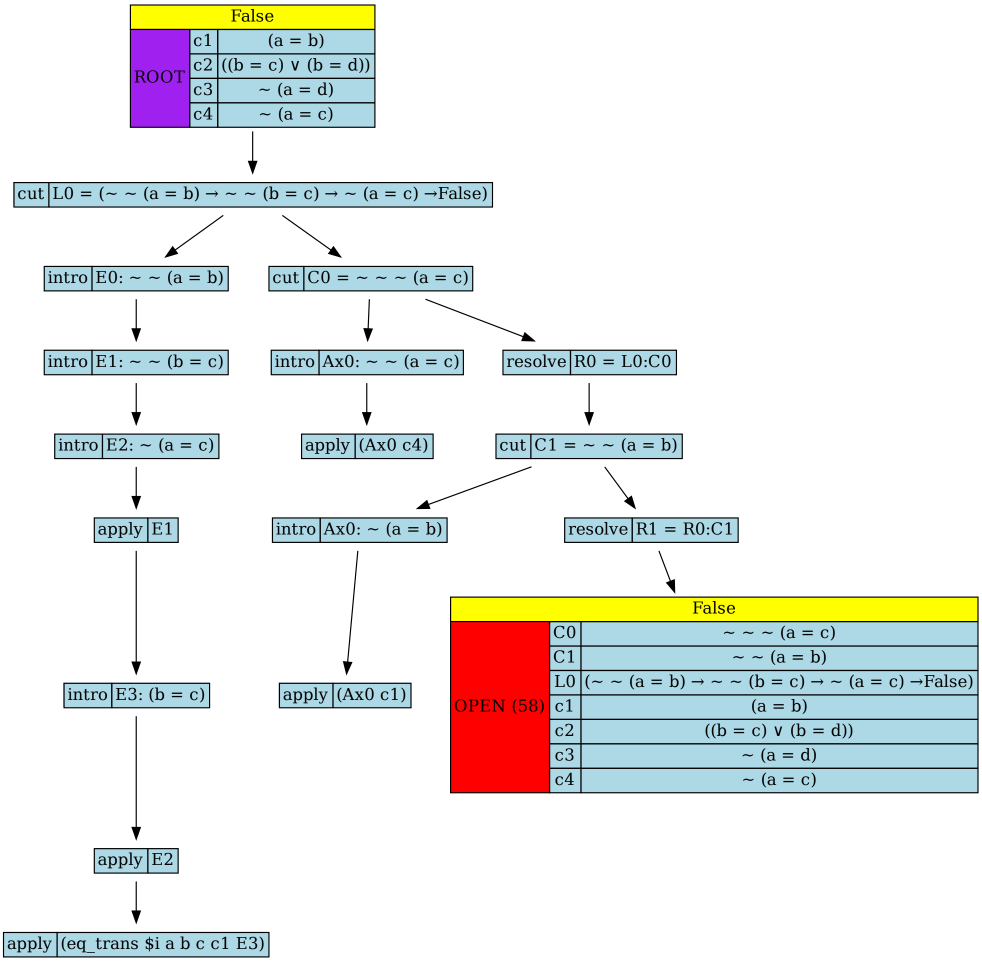


We then translate the resolution step between L_0 and C_0, whose resulting clause is called R_0. As mentioned previously, the resolution is represented as a single step. We thus get the next proof tree.

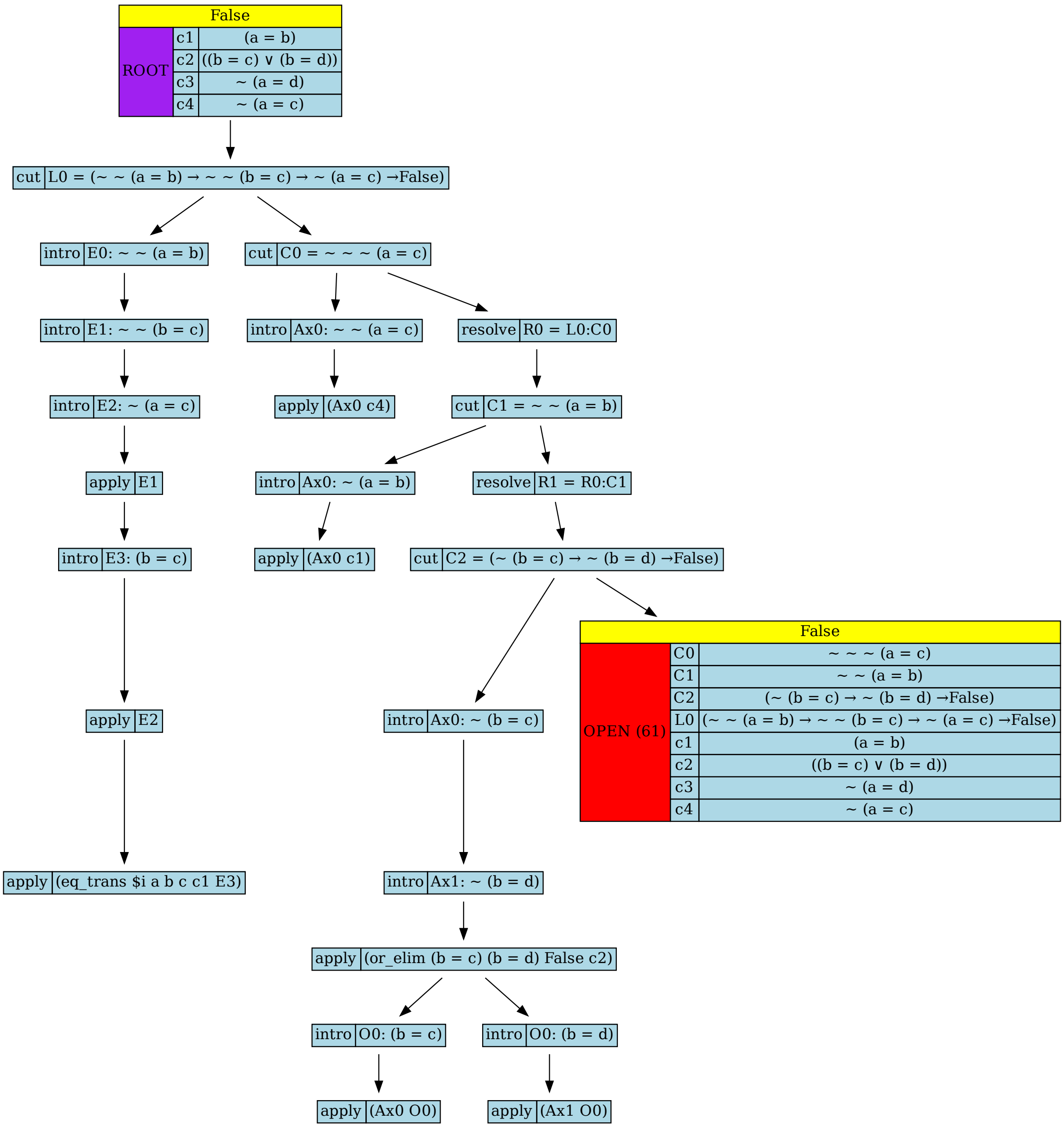


Note that, in order to keep proof trees more readable, not all current hypotheses are shown in open nodes. In particular, clauses resulting from resolutions are not shown because in typical problems they vastly outnumber the other hypotheses and do not bring much information.

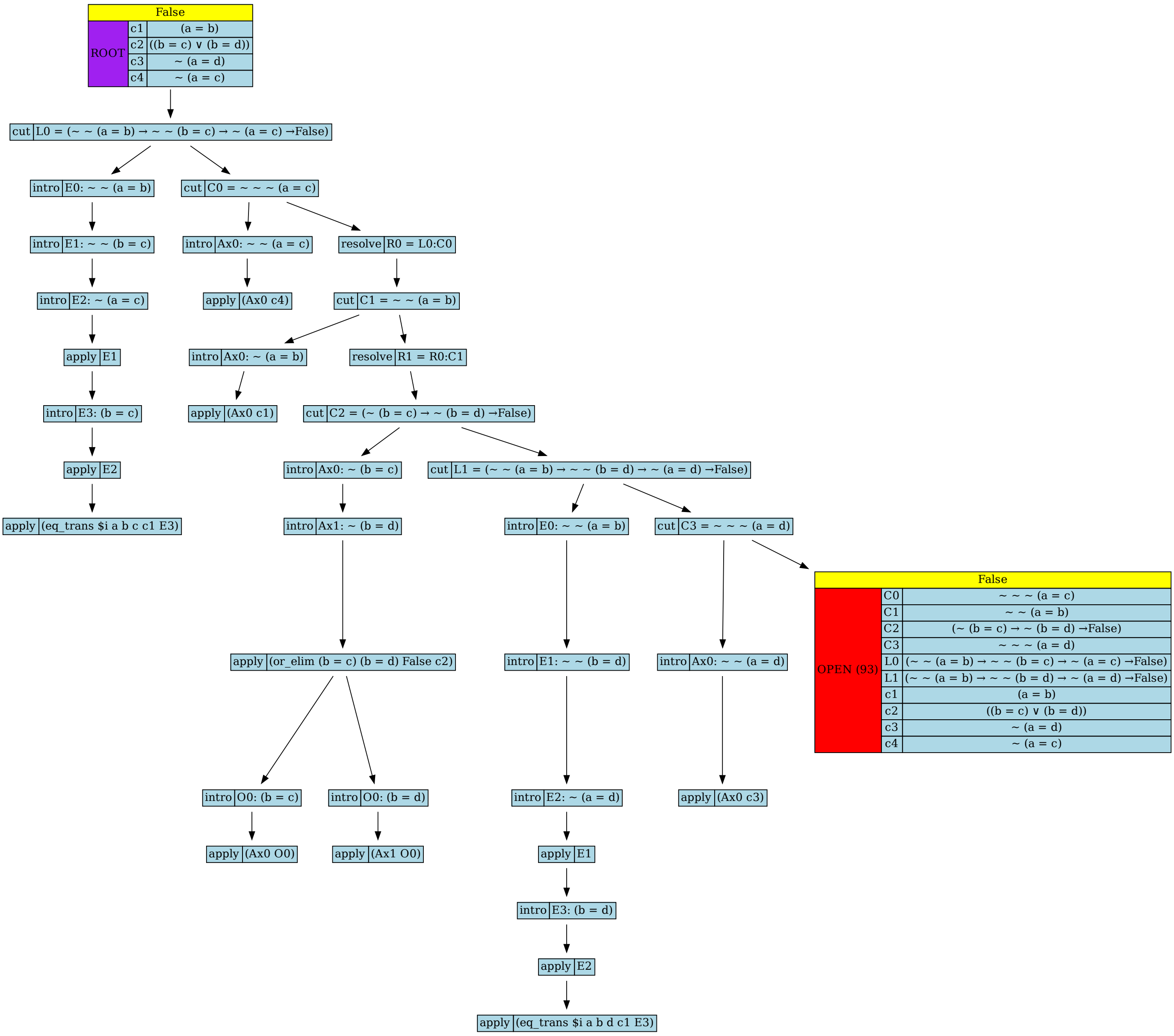
We then prove the weak clause $\overset{\circ}{\mathcal{C}}_1$, and perform a resolution between this weak clause and the previously introduced clause R_0 :



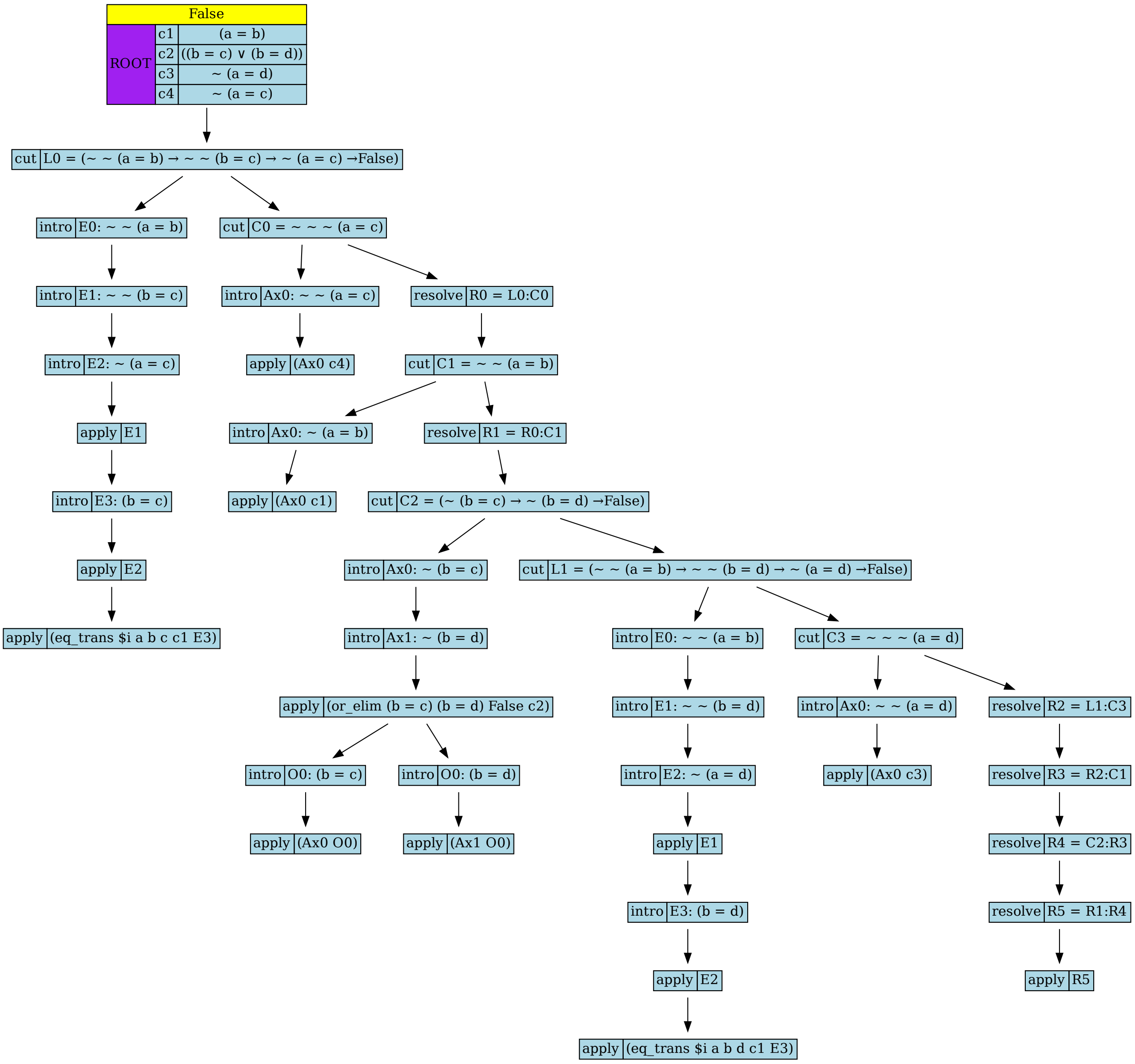
We then prove the weak clause $\overset{\circ}{\mathcal{C}}_2$. In this case, the clause $\mathcal{C}_2$ actually does contain a disjunction, so the proof needs to split this disjunction. This is done using the or_elim lemma from the first-order logic axiomatization. This lemma allows to prove some formula by performing a pattern matching on a disjunction. The exact type of the or_elim lemma is or_elim : $\forall P : \mathsf{Prop}, Q : \mathsf{Prop}, R : \mathsf{Prop}, (P \vee Q) \Rightarrow (P \Rightarrow R) \Rightarrow (Q \Rightarrow R) \Rightarrow R$. The application of this lemma creates two branches, each of which is proved by introducing the given literal, after which an immediate contradiction can be found using this introduced literal and one of the literals introduced earlier when deconstructing the weak clause that is being proven.



We then prove the last theory lemma, stating transitivity of equality on terms a, b, and d, introduced as the weak clause L_1. This lemma is proved exactly the same way as clause L_0. Afterwards, we introduce the weak clause $\mathring{\mathcal{C}}_3$, whose proof is trivial because it does not contain any disjunction. We then have the following proof tree :



All that is left is to perform the last 4 resolutions, which yield the empty clause R_5. Finally, R_5 is applied to close the last branch of the proof. The complete proof tree is thus the following :



All these graphs are automatically generated by ArchSAT :

- The graph of the complete proof tree can be printed using the `--full-dot file.gv` option
- Each intermediary proof tree can be printed as graph using the `--incr-dot basename` option, which instructs ArchSAT to print the temporary proof trees built during proof generation. For each node of the resolution tree translated into the proof tree, a graph will be printed in a different file. These files will be named using the basename provided to the option and an increasing counter : `basename.000.gv`, `basename.001.gv`, . . .

4.4.5 From Proof Tree to Proof Term

The proof tree can also be transformed into a proof term. This is actually quite easy to do, by requiring each proof step to also be able to elaborate a proof into a term, provided the proof terms for the branches it created. All the basic proof steps described in Section 4.4.2.1, can very easily elaborate the proof trees they produce into terms:

- The introduction proof step directly elaborates into a lambda abstraction. Suppose an introduction proves $(\Gamma, \forall x : t_y, t)$ using a proof of $(\Gamma \cup \{x : t_y\}, t)$, which elaborates into t. Then the proof of $(\Gamma, \forall x : t_y, t)$ can be elaborated into $\lambda x : t_y.t$.
- The cut step elaborates into a let-binding. More precisely, suppose a cut proves a sequent of the form (Γ, t) from a proof of (Γ, F) and a proof of $(\Gamma \cup \{x : F\}, t)|]$, which elaborate into proof terms e and u respectively. Then the cut proof step elaborates into let $x = e$ in u, as it often happens that term e is actually quite small compared to u.
- The let-in proof step can similarly elaborated into a let-binding. The only difference with the cut is that the term e is provided during the application of the proof step.
- The apply proof step is elaborated into an application. Suppose an apply proof step proves a sequent (Γ, g) using a function f, and sequents of the form : $(\Gamma, g_1), \ldots, (\Gamma, g_n)$, which each elaborates into p_i. Then, the proof of (Γ, g) can be elaborated into $f(p_1, \ldots, p_n)$.

ArchSAT can then output proof terms:

- in Coq using the `--coqterm term.v` command line option.
- in Dedukti using the `--dkterm term.dk` command line option.

Finally, as mentioned previously, β-reduction of terms can be computed (and sometimes must be computed in order to build proof terms). ArchSAT thus offers a way to print a normalized proof term, where all definitions have been expanded, and all β-redex (and let-bindings) reduced. This is available using the command line option:

- `--coqnorm norm.v` for a term checkable by Coq.
- `--dknorm norm.dk` for a term checkable by Dedukti.

Examples of all outputs are given in appendices D, E, and F. As can be seen, normalized output sometimes greatly simplifies the extra-reasoning introduced by the SMT/McSat architecture of ArchSAT. This happens particularly on first-order problems with very few logical connectives, whereas pure SAT proofs tend to become more complex because reducing let-bindings may duplicate sub-proofs. Finally, normalizing proof terms is currently quite expensive in term of time, because type-checking is still performed during normalization. This could be greatly optimized in further developments.

Chapter 5

Benchmarks

This Chapter presents the various benchmarks that were done to compare ArchSAT to other theorem provers.

These benchmarks include problems using rewriting in Section 5.1, in order to measure the usefulness of rewriting in ArchSAT and compared to the approaches of other provers. Section 5.2 and 5.3 present benchmarks using more generic problems coming from general purpose libraries of problems, in order to compare the general approach of ArchSAT. These two benchmarks also allow to establish a baseline for the generic performance of ArchSAT, so that the comparison on problems involving rewriting can be evaluated and compared to this control group of benchmarks[2].

In these benchmarks, we considered open and state of the art provers using similar techniques to the one used in ArchSAT. These include tableau provers, SMT solvers, superposition solvers, and solvers integrating rewriting, as described in each benchmark section.

5.1 Set Theory of the B Method

In order to test the implementation of rewriting in ArchSAT, we consider the set theory of the B method [3]. This method is supported by some tool sets, such as Atelier B [81] and Rodin [4], which are used in industry to specify and build, by stepwise refinements, software that is correct by design. This B set theory is suitable as it can be easily turned into a theory that is compatible with deduction modulo theory, i.e. where a large part of axioms can be turned into rewrite rules, and for which the rewriting theory proposed previously in Chapter 3 should work. Starting from the theory described in Chap. 2 of the B-Book [3], we therefore transform whenever possible the axioms and definitions into rewrite rules. The resulting theory has been introduced in [35], and a summary is presented in Fig. 5.1, with the three rewriting rules corresponding to the axiomatic core of the B set theory that we consider.

As can be seen, the proposed theory is typed, using first-order logic extended to polymorphic types à la ML, through a type system in the spirit of [20]. This extension to polymorphic types offers more flexibility, and in particular it allows us to deal with theories that rely on elaborate type systems, like the B set theory (see Chap. 2 of the B-Book [3]). The complete type system that is used in this formalization can be found in [35]. The type constructors, i.e. tup for tuples and set for sets, and type schemes of the considered set constructs are provided in Fig. 5.1 as well, where Type is the type of types and o the type of formulas, and where type arguments are subscript annotations of the constructs.

To test ArchSAT in this theory, we consider 319 lemmas coming from Chap. 2 of the B-Book [3]. These lemmas are properties of various difficulty regarding the set constructs introduced by the B method. It should be noted that these constructs and notations are, for a large part of them, specific to the B method, as they are used for the modeling of industrial projects, and are not necessarily standard in set theory.

[2] As it happens, ArchSAT tends to solve less problems than other provers in these generic benchmarks, but more in benchmarks involving rewriting. Hence, we can conclude that the impact of rewriting is predominant and allows ArchSAT to surpass other provers, despite a weaker general-purpose reasoning.

Type Constructors

$\mathsf{tup} : \mathsf{Type} \to \mathsf{Type} \to \mathsf{Type} \qquad \mathsf{set} : \mathsf{Type} \to \mathsf{Type}$

Type Schemes of the Set Constructs

$$\begin{array}{lcl}
\_ \in \_ & : & \Pi\alpha : \mathsf{Type}.\alpha \to \mathsf{set}(\alpha) \to o \\
(\_, \_) & : & \Pi\alpha_1, \alpha_2 : \mathsf{Type}.\alpha_1 \to \alpha_2 \to \mathsf{tup}(\alpha_1, \alpha_2) \\
\_ \times \_ & : & \Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\alpha_1) \to \mathsf{set}(\alpha_2) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \\
\mathbb{P}(\_) & : & \Pi\alpha : \mathsf{Type}.\mathsf{set}(\alpha) \to \mathsf{set}(\mathsf{set}(\alpha)) \\
\_ = \_ & : & \Pi\alpha : \mathsf{Type}.\alpha \to \alpha \to o
\end{array}$$

Axioms of Set Theory

$(x, y)_{\alpha_1,\alpha_2} \in_{\mathsf{tup}(\alpha_1,\alpha_2)} s \times_{\alpha_1,\alpha_2} t \longrightarrow x \in_{\alpha_1} s \wedge y \in_{\alpha_2} t$

$s \in_{\mathsf{set}(\alpha)} \mathbb{P}_\alpha(t) \longrightarrow \forall x : \alpha.x \in_\alpha s \Rightarrow x \in_\alpha t$

$s =_{\mathsf{set}(\alpha)} t \longrightarrow \forall x : \alpha.x \in_\alpha s \Leftrightarrow x \in_\alpha t$

Figure 5.1: Rewriting Rules of the Axiomatic Core of the B Set Theory

	Alt-Ergo	ArchSAT	Zenon Modulo	Zipperposition
Proofs (total: 320)	232	272	138	306
Rate	72.7%	85.3%	43.3%	95.9%
Total time (s)	8.42	268.69	2.86	109.88

Table 5.1: Bset Benchmarks results

As tools, we consider ArchSAT, as well as other automated theorem provers, able to deal with first-order logic with polymorphic types and rewriting natively. In particular, we consider Zipperposition (version 1.5), a prover based on superposition and rewriting, as well as Zenon Modulo (version 0.4.2), a tableau-based prover that is an extension of Zenon to deduction modulo theory. To show the impact of rewriting over the results, we also include the Alt-Ergo SMT solver (version 1.01). It would have been possible to also consider provers dealing with pure first order logic and encode the polymorphic layer. But preliminary tests have been conducted and very low results have been obtained even for the best state-of-the-art provers (we have considered E and CVC4 in particular), which tends to show that polymorphism encoding adds a lot of noise in proof search and is not effective in practice.

The experiment was run on an Intel Xeon E5-1650 v3 3.50 GHz computer, with a timeout of 90 s (beyond this timeout, results do not change) and a memory limit of 1 GiB. The results are summarized in Tab. 5.1. In these results, we observe that ArchSAT obtains better results, in terms of proved problems, than Zenon Modulo and Alt-Ergo, which tends to show the effectiveness of the rewriting theory in practice. However, ArchSAT places second behind Zipperposition, which means that there is still room for improvement regarding the implementation. Looking at the cumulative times, Alt-Ergo is not really faster than ArchSAT and Zipperposition, which take more time to find few more difficult problems (with a timeout of 3 s, they respectively find 260 and 303 proofs in 16.61 s and 17.61 s, while Alt-Ergo obtains the same results). These experimental results have been published at the ABZ 2018 conference [32].

5.2 TPTP Library

The TPTP[1] library [83] is a collection of first-order problems meant to be used as benchmarks for first-order theorem provers. It is separated into categories, which represent different kinds and shapes of problems. It is interesting to gauge the overall performance of a theorem prover on a variety of problems. This benchmark was meant to evaluate the global performances of ArchSAT without rewriting, as no problems in the TPTP library is currently axiomatized using rewrite rules. Additionally, only problems from the TPTP library that are first-order with no arithmetic are considered in this benchmark (as arithmetic is currently not handled by ArchSAT).

A number of other theorem provers have been considered in order to be compared to ArchSAT:

- CVC4 [10], currently one of the best SMT provers, winner of the 2018 edition of SMT-COMP [2].
- E-prover [80], is a superposition theorem prover for first-order logic with equality.
- Princess, a tableau theorem prover for Presburger arithmetic with uninterpreted predicates.
- Zenon [24] is also a tableau theorem prover, recently extended to handle polymorphism [31], linear arithmetic [33], and rewriting [34].
- Zipperposition [38] is a superposition prover developped to handle integer linear arithmetic.

This benchmark evaluates the number of solved problems for each prover. A problem is said to be solved by a prover, if the theorem has been proved for first-order theorem provers such as Zenon, Princess and E-prover, or the problem (including the negation of the goal) is found to be UNSAT for SMT provers (ArchSAT and CVC4). When relevant, the running time of the provers on the problems solved is shown and discussed. This will often be done by using the cumulative time of provers on the problems solved. Cumulative time for each prover is computed by taking the list of the times spent on each problem solved, sorting that list in increasing order, and then plotting the cumulative sum of these sorted running times. While this does not offer a point-wise comparison of the running times of provers, it is nonetheless useful to discern global trends in running time for each prover.

This section will first compare in Section 5.2.1 various configurations of the ArchSAT theorem prover to evaluate the impact of some of the options that can be chosen, then ArchSAT will be compared to the other provers listed above in Section 5.2.2.

All the benchmarks in this section have been run on an Intel Xeon E5-2660 v2 2.20 GHz, with a timeout of 60 seconds, and a maximum memory of 2Go.

5.2.1 Archsat Variants

We first present the results of benchmarking for some configurations of ArchSAT on TPTP. We distinguish 6 different versions of ArchSAT:

- The auto configuration of ArchSAT disables rewriting, and uses the tableau theory described in Chapter 2, using the unification procedure presented in Section 3.4 in order to unify modulo equalities, and the heuristic described in Section 2.3.3 to only perform the best 10 instantiations at each round.
- The auto-all configuration disables the heuristic and instead performs all the instantiations found at each round.
- The R configuration is similar to the auto configuration, except unification is done using only the Robinson [64] unification algorithm, which does not unify modulo equalities. Completeness is thus lost, but unification is significantly faster.
- The R-all configuration disable the heuristic and performs all instantiations at each round using Robinson unification.

[1] Thousands of Problems for Theorem Provers.

		archsat	archsat	archsat	archsat	archsat	archsat
		(auto)	(auto-all)	(R)	(R-all)	(rwrt)	(rwrt-R)
tptp-v7.2.0/Problems	AGT (53)	8 (0)	9 (0)	9 (0)	9 (0)	8 (0)	11 (2)
	ALG (457)	178 (0)	180 (0)	184 (0)	189 (4)	177 (0)	183 (0)
	ANA (91)	8 (0)	11 (3)	5 (0)	7 (2)	8 (0)	5 (0)
	BIO (4)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
	BOO (140)	0 (0)	1 (0)	1 (0)	1 (0)	0 (0)	3 (2)
	CAT (130)	3 (0)	3 (0)	4 (0)	3 (0)	3 (0)	5 (1)
	COL (239)	42 (1)	43 (2)	25 (0)	27 (2)	27 (0)	26 (1)
	COM (177)	7 (0)	9 (0)	6 (0)	7 (0)	7 (0)	7 (2)
	CSR (786)	46 (0)	57 (0)	54 (1)	58 (0)	47 (0)	51 (1)
	FLD (281)	7 (0)	12 (0)	9 (0)	12 (0)	7 (0)	9 (0)
	GEG (1)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
	GEO (776)	113 (0)	105 (6)	111 (0)	89 (2)	118 (0)	123 (2)
	GRA (34)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
	GRP (1090)	6 (0)	26 (13)	10 (1)	12 (0)	18 (0)	22 (0)
	HAL (10)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
	HEN (67)	2 (0)	3 (0)	3 (0)	3 (0)	4 (0)	5 (0)
	HWC (6)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
	HWV (411)	7 (0)	10 (0)	14 (0)	21 (6)	7 (0)	24 (10)
	KLE (241)	0 (0)	0 (0)	0 (0)	0 (0)	5 (0)	9 (4)
	KRS (299)	43 (0)	32 (0)	45 (0)	32 (0)	42 (0)	45 (2)
	LAT (733)	31 (1)	31 (2)	20 (0)	18 (0)	36 (0)	29 (2)
	LCL (1225)	52 (0)	52 (0)	94 (4)	92 (11)	66 (0)	118 (14)
	LDA (50)	2 (0)	2 (0)	0 (0)	0 (0)	0 (0)	0 (0)
	MED (12)	1 (0)	0 (0)	3 (1)	0 (0)	1 (0)	2 (0)
	MGT (157)	17 (0)	14 (1)	21 (0)	15 (2)	17 (0)	21 (0)
	MSC (35)	6 (0)	6 (0)	6 (0)	8 (2)	6 (0)	6 (0)
	NLP (520)	2 (0)	0 (0)	1 (0)	0 (0)	2 (0)	1 (0)
	NUM (1053)	73 (1)	74 (4)	95 (4)	91 (10)	90 (5)	100 (3)
	NUN (25)	2 (0)	2 (0)	2 (0)	2 (0)	2 (0)	2 (0)
	PHI (7)	3 (0)	2 (0)	4 (1)	2 (0)	3 (0)	3 (0)
	PLA (74)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
	PRD (3)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
	PRO (72)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
	PUZ (149)	24 (0)	41 (2)	25 (0)	36 (0)	22 (0)	26 (0)
	REL (220)	0 (0)	0 (0)	0 (0)	0 (0)	2 (0)	2 (0)
	RNG (263)	3 (0)	3 (0)	6 (1)	5 (0)	14 (0)	18 (2)
	ROB (45)	2 (0)	2 (0)	0 (0)	0 (0)	0 (0)	0 (0)
	SCT (291)	23 (0)	23 (0)	39 (3)	31 (0)	24 (0)	36 (2)
	SET (1268)	93 (0)	98 (8)	98 (3)	100 (15)	89 (1)	123 (25)
	SEU (906)	70 (2)	49 (0)	81 (7)	55 (0)	68 (2)	75 (4)
	SEV (7)	0 (0)	0 (0)	1 (0)	0 (0)	0 (0)	1 (0)
	SWB (204)	14 (0)	21 (1)	14 (0)	20 (0)	9 (0)	13 (2)
	SWC (846)	87 (0)	167 (50)	108 (6)	109 (0)	92 (0)	102 (5)
	SWV (1399)	189 (1)	197 (7)	223 (8)	224 (13)	172 (0)	232 (10)
	SWW (492)	15 (0)	15 (0)	35 (4)	20 (1)	23 (0)	41 (8)
	SYN (1223)	326 (0)	424 (0)	417 (0)	425 (9)	322 (0)	416 (0)
	SYO (104)	3 (0)	3 (0)	4 (0)	3 (0)	2 (0)	5 (1)
	TOP (131)	5 (0)	5 (0)	8 (3)	5 (0)	5 (0)	5 (0)
	Total (16807)	1514 (6)	1733 (99)	1786 (47)	1732 (79)	1546 (8)	1906 (105)

Table 5.2: TPTP Benchmarks Results (ArchSAT Variants)

- The rwrt configuration is the same as the auto configuration, but with rewriting activated as described in Chapter 3. In this case, the rewriting theory uses a heuristic to automatically turn some axioms into rewrite rules.
- The rwrt-R configuration activates rewriting like the rwrt configuration, and uses the Robinson unification algorithm instead of rigid superposition.

Table 5.2 presents the number of problems solved (i.e. on which ArchSAT find UNSAT, that is a proof), for each category of the TPTP library. In parenthesis is the number of unique problems solved by a particular configuration, that is the number of problem that a particular configuration solves and that no other configuration (in that same table) solves.

Rewriting. Overall, rewriting in ArchSAT increases performances (comparing the auto and R configurations). This is due to two factors: first, there are not that many problems in TPTP that have an axiomatization that can be turned into a rewrite system[1]; second, activating rewriting has some runtime cost, slowing down ArchSAT so that some problems are no longer proved.

Unification. What is more surprising is that restricting ArchSAT to perform only simple unification improves the results, although the procedure is incomplete. This is mostly due to the rigid superposition algorithm taking too much time in cases where unification is not possible[2]. This suggests that the algorithm is not efficient enough in its current state. There are two main ways this could be solved: better heuristics and more simplifications and redundancy criterions. Heuristics could help rigid superposistion find solution faster, while additional redundancy criterion could help cut research space, ideally enough so that problems with no solution are solved fast enough to try other problems.

Instantiation Heuristic. As can be seen, when comparing the auto, auto-all, R and R-all configurations, the usefulness of the heuristic varies. With the rigid superposition, deactivating the heuristic improves significantly the number of solved problems, whereas when using regular Robinson unification, using the heuristic is slightly better overall. This can be understood as on complex problems that require unification modulo, there are likely few substitutions found by unification[3] hence performing all these instantiations is fine. On the other hand, in cases where simple Robinson unification is enough, there is usually a very important number of potential substitutions found. In this case, filtering and choosing substitutions actually proves to be of some help, though not as much as it could be hoped.

Importantly, each configuration has unique problems that only it manages to solve (shown in parenthesis in Table 5.2). This shows the usefulness of each of the approaches described in this thesis.

5.2.2 Comparison with Other Provers

The results of the comparison between ArchSAT and other provers are shown in Table 5.3. In this table, the results of all configurations of ArchSAT have been aggregated, and then compared to the other provers. Additionally, Figure 5.2 shows the cumulative times of each prover.

While the results show that ArchSAT is a little behind the other theorem provers, ArchSAT still manages to solve a decent number of problems. In addition, it has the lowest total cumulative time of all provers, and importantly also has a number of unique problems solved.

The main reason for the rather low performances of ArchSAT is the differences in heuristics. Heuristics are quite omnipresent in theorem provers currently, and are often one of the main reason for differences in performances. ArchSAT currently has some very naive heuristics for meta-variable instantiation as described in Section 2.3.3. However, tests have shown that varying

[1] Sometimes because of the absence of typing, or simply because axioms are not expressed in an adequate form.

[2] In practice, the superposition algorithm is restricted to search only within a maximum depth, but this depth could probably be tuned better.

[3] Otherwise, given the current less than ideal performance of rigid superposition, the unification process may simply take all the time.

		archsat	cvc4	eprover	princess	zenon	zipperposition
tptp-v7.2.0/Problems	AGT (53)	12 (0)	36 (18)	17 (0)	6 (0)	17 (0)	18 (0)
	ALG (457)	192 (15)	158 (5)	140 (2)	216 (38)	52 (0)	156 (15)
	ANA (91)	13 (0)	0 (0)	0 (0)	11 (1)	17 (0)	32 (11)
	BIO (4)	0 (0)	1 (1)	0 (0)	0 (0)	0 (0)	0 (0)
	BOO (140)	2 (0)	0 (0)	0 (0)	3 (0)	3 (0)	43 (39)
	CAT (130)	5 (0)	10 (6)	6 (2)	28 (8)	13 (0)	31 (10)
	COL (239)	38 (0)	0 (0)	0 (0)	77 (13)	34 (0)	131 (50)
	COM (177)	9 (1)	24 (1)	30 (3)	15 (0)	30 (1)	63 (24)
	CSR (786)	68 (0)	366 (24)	418 (65)	49 (0)	106 (0)	142 (1)
	FLD (281)	13 (0)	0 (0)	0 (0)	8 (0)	29 (4)	56 (28)
	GEG (1)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
	GEO (776)	146 (0)	223 (8)	294 (10)	167 (6)	247 (3)	384 (62)
	GRA (34)	1 (0)	8 (0)	17 (6)	5 (0)	4 (0)	10 (0)
	GRP (1090)	29 (0)	30 (5)	68 (29)	63 (4)	80 (10)	589 (458)
	HAL (10)	0 (0)	4 (0)	5 (1)	0 (0)	0 (0)	0 (0)
	HEN (67)	5 (0)	0 (0)	0 (0)	9 (0)	9 (0)	44 (34)
	HWC (6)	0 (0)	0 (0)	0 (0)	2 (2)	0 (0)	1 (1)
	HWV (411)	27 (1)	2 (0)	6 (0)	60 (12)	54 (2)	67 (10)
	KLE (241)	8 (0)	110 (8)	159 (37)	13 (0)	6 (0)	75 (0)
	KRS (299)	52 (1)	62 (1)	85 (13)	99 (5)	136 (20)	65 (0)
	LAT (733)	39 (0)	79 (23)	76 (23)	28 (0)	42 (0)	68 (18)
	LCL (1225)	128 (1)	60 (10)	142 (77)	112 (14)	117 (6)	340 (169)
	LDA (50)	1 (0)	0 (0)	0 (0)	3 (0)	1 (0)	6 (2)
	MED (12)	2 (0)	4 (0)	9 (0)	0 (0)	6 (0)	3 (0)
	MGT (157)	23 (0)	60 (1)	66 (2)	46 (1)	92 (3)	100 (7)
	MSC (35)	8 (0)	4 (0)	5 (0)	6 (0)	14 (3)	14 (4)
	NLP (520)	2 (0)	22 (0)	22 (0)	20 (0)	22 (0)	28 (6)
	NUM (1053)	137 (1)	404 (19)	433 (34)	254 (8)	139 (6)	360 (20)
	NUN (25)	2 (0)	0 (0)	1 (1)	0 (0)	6 (0)	5 (0)
	PHI (7)	3 (0)	7 (0)	7 (0)	4 (0)	6 (0)	7 (0)
	PLA (74)	0 (0)	0 (0)	0 (0)	0 (0)	5 (0)	29 (24)
	PRD (3)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
	PRO (72)	0 (0)	37 (11)	35 (1)	1 (0)	3 (0)	11 (0)
	PUZ (149)	42 (0)	24 (0)	31 (0)	61 (11)	65 (2)	76 (5)
	REL (220)	2 (0)	4 (0)	75 (55)	4 (0)	0 (0)	39 (17)
	RNG (263)	18 (0)	112 (5)	112 (11)	64 (5)	32 (0)	98 (17)
	ROB (45)	0 (0)	0 (0)	0 (0)	6 (0)	1 (0)	10 (4)
	SCT (291)	34 (0)	27 (9)	23 (5)	27 (7)	39 (7)	53 (13)
	SET (1268)	169 (0)	280 (31)	331 (41)	323 (1)	240 (8)	457 (80)
	SEU (906)	99 (0)	405 (72)	448 (70)	181 (10)	93 (0)	268 (0)
	SEV (7)	1 (0)	0 (0)	1 (0)	0 (0)	1 (0)	1 (0)
	SWB (204)	22 (0)	68 (6)	76 (13)	42 (2)	24 (1)	42 (0)
	SWC (846)	187 (0)	214 (2)	336 (80)	343 (32)	111 (0)	471 (75)
	SWV (1399)	284 (2)	291 (25)	263 (1)	372 (20)	277 (6)	464 (88)
	SWW (492)	38 (0)	156 (33)	191 (76)	40 (12)	39 (4)	83 (18)
	SYN (1223)	461 (1)	203 (1)	278 (0)	402 (18)	557 (9)	743 (155)
	SYO (104)	5 (0)	5 (0)	7 (1)	7 (0)	14 (5)	6 (1)
	TOP (131)	8 (0)	23 (7)	26 (9)	7 (0)	7 (0)	11 (1)
	Total (16807)	2335 (23)	3523 (332)	4239 (668)	3184 (230)	2790 (100)	5700 (1467)

Table 5.3: TPTP Benchmark Results

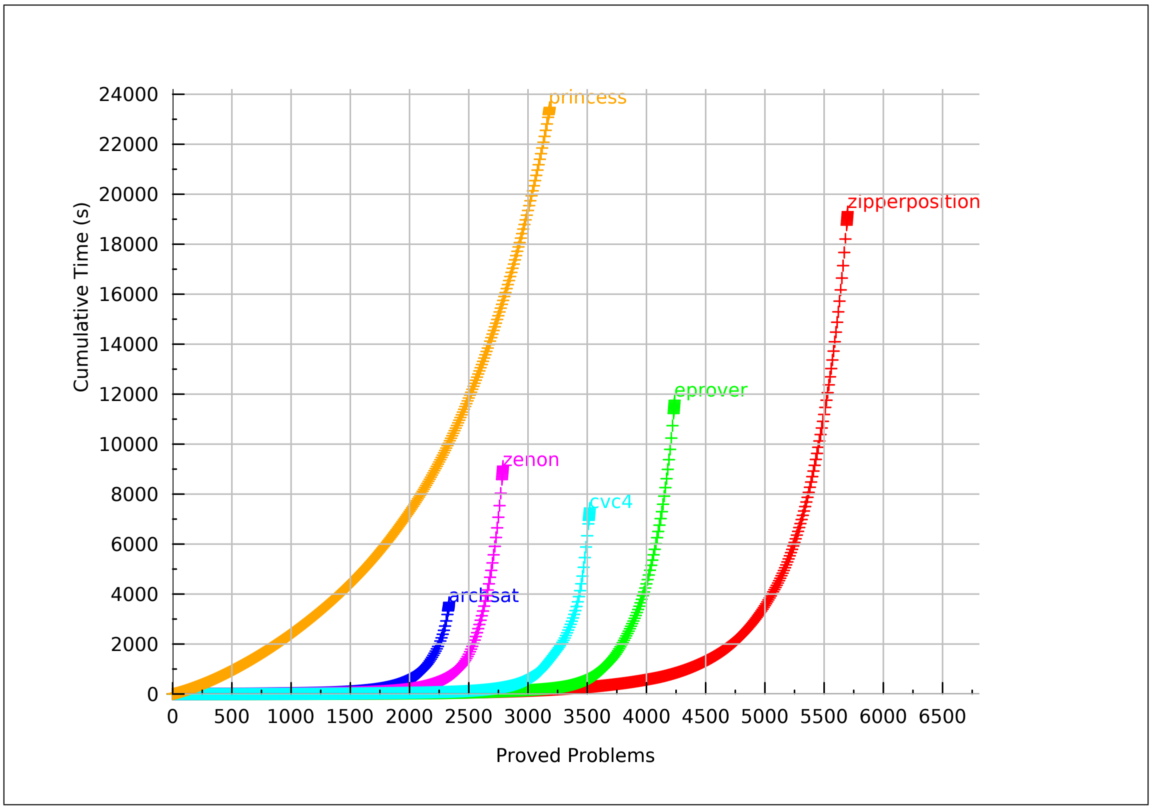


Figure 5.2: Cumulative Times for TPTP Benchmark

even slightly these heuristics, such as changing the number of instantiations done at each round, can significantly change the behaviour of ArchSAT on some problems, from solving a problem in less than a second to reaching a timeout after a few minutes.

The overall difference between ArchSAT and other provers is likely to be caused by a difference of heuristics, particularly since the other provers considered are well-established and benefit from several years of development, and thus have better and more refined heuristics. ArchSAT is still experimental and was written from scratch for the purposes of this thesis, which focus on rewriting and production of formal proofs, rather than performance on generic problems. This also reinforces the importance of the good results obtained on the benchmarks of problems of the B set theory presented in Section 5.1: ArchSAT manages to best Zenon for example[1] although it has low performances in the general case. This means that the treatment of rewriting in ArchSAT is really quite useful and allows it to surpass other provers in the presence of a genuine rewrite system.

It is interesting to take a look at the provers' results for each category of the TPTP library, as the overall results are actually biased due to varying number of problems in each category: for instance the SYN category contains a lot more problems that most other categories, therefore, a prover that is 10% more efficient than others on problems in that particular category would gain an overall number of problems greater than a prover that is 10% more efficient than other provers on another category. Measuring the total number of problems proved introduce a bias that favors provers that do well in categories with a lot of problems. A notable example are the GRP and LCL categories, where Zipperposition gains most of its lead compared to all the other provers.

ArchSAT is actually very competitive in the ALG category, where it is the second best prover after Princess, and has 15 unique problems solved. It also compares favorably in the SYN category, where it places 3rd. Overall, the results for ArchSAT are encouraging considering that no heuristics or algorithms in ArchSAT have been tailored for the TPTP benchmark.

[1] As well as CVC4 and Princess, which solve respectively 87 and 46 problems, and thus were omitted in the results because of their low performances (mainly due to the encoding of polymorphism, which was necessary because they cannot handle polymorphic terms).

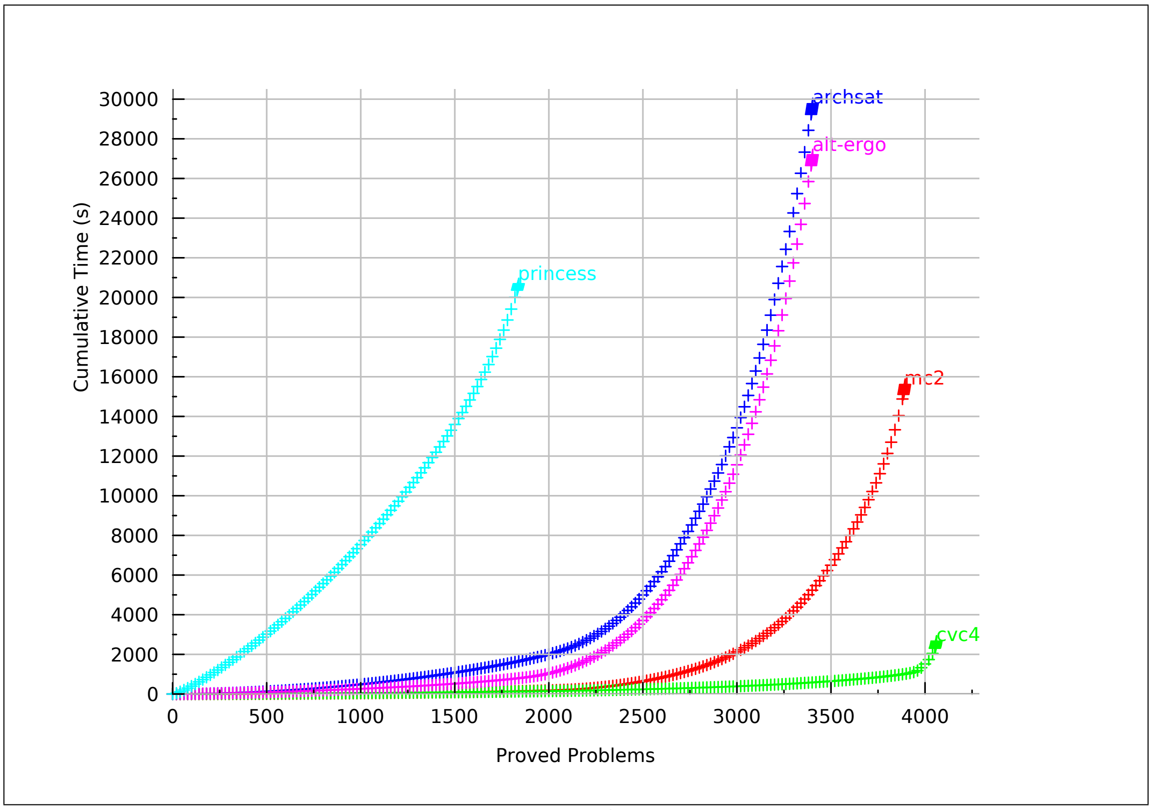


Figure 5.3: Cumulative times for SMT-LIB QF_UF benchmarks

5.3 SMTlib Library

The SMT-LIB library [13] is a collection of problems meant to be benchmarks for SMT solvers. It provides an interesting point of comparison on general problems. Whereas the TPTP library categories inform more about the general shape and/or provenance of the problems, the SMT-LIB libray is split into categories according to the theories required to solve the problem. This benchmark presents results on the QF_UF category, which only contains problems with quantifier-free formulas that use uninterpreted functions and predicates, and equality. ArchSAT is unfortunately not competitive with other provers for the UF category, which contains quantified formulas using uninterpreted functions and predicates, and thus the results for that category are not shown.

Compared to the previous section, the Zenon and E-prover provers are not show in result tables because they cannot parse problems in the SMT-LIB language. However, the results include the following provers that could not read the TPTP format :

- Alt-Ergo is an SMT solver written in OCaml introduced in Section 5.1.
- mc2 is an OCaml implementation of the McSat algorithm forked from mSAT by Simon Cruanès. It can handle ground problems containing equalities and uninterpreted functions and predicates.

All the benchmarks in this section have been run on an Intel Xeon E5-2660 v2 2.20 GHz, with a timeout of 60 seconds, and a maximum memory of 2Go. The results of all provers on the QF_UF catgory of SMT are summarized in Table 5.4, and the cumulative times in Figure 5.3. Different configurations of ArchSAT are not shown here because these configurations affect mostly how quantified formulas are handled, and thus do not have a significative impact on this experiment[1].

[1] Some non-negligible difference do appear, most likely because, although the different configurations perform the exact same proof search on quantifier-free formulas, they each have a slight overhead, which alters the final results.

			alt-ergo	archsat	cvc4	mc2	princess
smtlib/QF.UF	NEQ (48)		0 (0)	15 (0)	40 (12)	28 (1)	0 (0)
	PEQ (47)		0 (0)	9 (0)	24 (11)	12 (0)	1 (0)
	QG-classification	loops6 (448)	286 (0)	285 (0)	288 (0)	288 (0)	37 (0)
		qg5 (5286)	2980 (0)	2868 (0)	3335 (5)	3328 (0)	1754 (0)
		qg6 (244)	34 (0)	31 (0)	121 (87)	34 (0)	1 (0)
		qg7 (418)	78 (0)	73 (0)	114 (36)	77 (0)	22 (0)
		Total (6396)	3378 (0)	3257 (0)	3858 (128)	3727 (0)	1814 (0)
	SEQ (56)		6 (0)	20 (0)	37 (12)	25 (0)	4 (0)
	TypeSafe (3)		3 (0)	3 (0)	3 (0)	3 (0)	3 (0)
	e.d (100)		17 (0)	100 (0)	100 (0)	100 (0)	19 (0)
	Total (6650)		3404 (0)	3404 (0)	4062 (163)	3895 (1)	1841 (0)

Table 5.4: SMTlib QF_UF Benchmarks results

Very interestingly, ArchSAT is quite competitive in this fragment. It is basically ex-aequo with Alt-Ergo, although the algorithms used are different: Alt-Ergo has a more traditional SMT architecture with a congruence closure algorithm compared to ArchSAT, which relies on an McSat solver where equality and uninterpreted function are handled separately. This shows the strength of the McSat handling of equality, particularly on problems in the eq_diamond category (abbreviated as “e.d” in the table), meant to be hard for union-find and congruence closure algorithms, where ArchSAT solves all of the 100 problems, whereas Alt-Ergo only solves 17. The good results of mc2, which places 2nd, also show the worth of the McSat algorithm, that allows it to be very close to CVC4 in the number of problem proved.

ArchSAT is also better than Princess, which was expected since first-order provers such as Princess are typically less performant than SMT solvers on pure ground problems. Finally, cumulative times show Alt-Ergo and ArchSAT tied, Princess in the same order of magnitude, while CVC4 is about 10 times faster.

Chapter 6

Implementation of ArchSAT

The code of ArchSAT is available on github: `https://github.com/Gbury/archsat`.

This chapter explains some technical challenges met during the implementation of ArchSAT, and the solutions that were found. While Chapters 1, 2, 3, and 4 explained the main algorithms used in ArchSAT, this Chapter deals with more technical aspect of the implementation of ArchSAT. In particular, it explains how ArchSAT internals were carefully designed to be modular and easily extensible, as well as some aspects of the user interface of ArchSAT: parsing of input languages, typechecking of input problems and syntaxically correct output.

Most of these solutions take advantage of features of the OCaml language. Some knowledge of the recent developments of the OCaml compiler is therefore recommended to understand some of the sections in this chapter. The following features of OCaml will be used: polymorphic types, generalized algebraic datatypes, extensible types, and functors.

6.1 Dolmen: A Library for Uniform Parsing of Languages

The first challenge was parsing of the input problems. Actually, there are many different syntaxes for expressing first-order problems:

- The Dimacs format[2] describes pure SAT problems.
- The iCNF format[3] contains descriptions of pure SAT problems, as well as instructions for the solvers to solve the problems with specific local assumptions.
- The SMT-LIB syntax[4] defines first-order terms, and SMT-LIB files consist of a list of statements to be executed by the prover.
- The Zipperposition's format[5] describes first-order problems, and has dedicated syntax for rewrite rules and datatype declarations.
- The TPTP syntax[6] describes first-order and higher order problems.

TPTP and SMT-LIB are the most usual syntaxes used by automated theorem provers. However, a lot of provers (Zenon, Alt-Ergo, Zipperposition, ...) have their own specific syntax, This is a problem as some provers are very restrictive in what syntax they are able to parse: until recently[7], the Alt-Ergo theorem prover only accepted problems in its own syntax, which made it rather difficult to use in benchmarks where problems were expressed in SMT or TPTP syntax. Similarly,

[2] `http://www.satcompetition.org/2009/format-benchmarks2009.html`

[3] `http://www.siert.nl/icnf/`

[4] `http://smtlib.cs.uiowa.edu/language.shtml`

[5] `https://github.com/c-cube/zipperposition/blob/master/src/parsers/Parse_zf.mly`

[6] `http://tptp.cs.miami.edu/~tptp/TPTP/SyntaxBNF.html`

[7] Support for SMT-LIB in Alt-Ergo was removed in 2015, and then finally re-introduced in version 2.2.0 of Alt-Ergo, which was released on April, 21, 2018

a lot of provers only parse a very small number of input languages, even though almost all languages describe the same first-order logic.

At the same time, although they typically refer to the same logic of first-order problems, languages have small syntactic differences, which makes the support of many different formats possibly complicated. For instance, some syntaxes (Dimacs, TPTP, Zipperposition's format) describe problems, i.e. represent a list of axioms and goals, while others (iCNF, SMT-LIB) are series of instructions to be executed by the prover. The builtin constants and typing rules also differ slightly between languages, and will be the focus of Section 6.2. However, these differences are mainly technical details, and they are not enough to justify the various languages restrictions and the associated problems when experimenting. As seen in Section 5, some provers are excluded from whole sets of problems simply because they cannot parse a language.

This was the main motivation for writing the dolmen library. dolmen is a library that intends to provide parsers for a large part of the traditional input languages used in automated theorem proving, and partially abstract over those input languages. dolmen is written in OCaml, and more specifically provides functors that allow the user to create parsers for all the languages listed at the beginning of this Section. These parsers take as argument modules that implement terms and statements, and returns a parser for the desired language. This architecture was designed to facilitate the use of dolmen as a drop-in replacement for hand-written parsers in theorem provers, as it allows users to keep the same representation of terms. More specifically, for each supported language, dolmen exposes a functor of the following type:

```
module Make
    (L : ParseLocation.S)
    (I : Id)
    (T : Term with type location := L.t
               and type id := I.t)
    (S : Statement with type location := L.t
                    and type id := I.t
                    and type term := T.t) :
  Language_intf.S with type statement = S.t
```

This functor takes 4 modules as arguments:

- `L : ParseLocation.S` is a module used to define location in source files, as well as the exception that will be raised by the parser in case of lexing or parsing error. These error definitions are required because the errors need to contain the location in the file where the error occurs, hence the definition of the exceptions depends on the type definition for location in files.

- `I : Id` is a module used to represent identifiers in source files. Identifiers are usually more complex that simple strings. In many languages, identifiers are scoped, so that the same string can be used to refer to different entities depending on whether it occurs in a type or a term. Additionally, each TPTP statement has a name, which needs an identifier. This module is thus used to define the scopes used in a language.

- `T : Term` is the module implementing the structure of terms (including types, formulas, etc). This is the most significant module as it defines the abstract syntax tree, and the corresponding functions to build it. It typically has to implement application (either currified or not, depending on the language), term annotations, quantifications, . . .

- `S : Statement` is a module implementing the top-level statements (or directives) of a language. These are mostly used to specify the hypotheses and goal of a problem, but can also be used to set some prover options or ask the prover to output some values in SMT-LIB.

Once provided with these modules, the functor returns a module with the following interface:

```
module type S = sig

  type token
  (** The type of tokens produced by the language lexer. *)

  type statement
  (** The type of top-level directives recognised by the parser. *)

  module Lexer : Lex_intf.S
    with type token := token
  (** The Lexer module for the language. *)

  module Parser : Parse_intf.S
    with type token := token
     and type statement := statement
  (** The Parser module for the language. *)

  val find : ?dir:string -> string -> string option
  (** Helper function to find a file using a language specification.
      Separates directory and file because most include directives in languages
      are relative to the directory of the original file being processed. *)

  val parse_file : string -> statement list
  (** Parse the given file. *)

  val parse_input :
    [ `Stdin | `File of string ] -> (unit -> statement option) * (unit -> unit)
  (** Incremental parsing. Given an input to read (either a file, or stdin),
      returns a generator that will incrementally parse the statements,
      together with a cleanup function to close file descriptors.
      In case of a syntax error, the current line will be completely
      consumed and parsing will restart at the beginning of the next line.
      Useful to process input from [stdin], or even large files where it would
      be impractical to parse the entire file before processing it. *)
end
```

This signature exposes both low-level and high-level parsing, to adapt to the user's needs:

- High-level functions such as `find` and `parse_file` respectively help locate included files[1] and parse files directly into a list of directives.
- Incremental parsing can be accessed using the `parse_input` function, to deal with very big files, as well as interactive mode, as required by the SMT-LIB specification.
- Low level details are exposed in the `Lexer` and `Parser` modules, if more control over parsing is required.

Language class. dolmen also provides an abstraction over classes of languages: a language class in dolmen is a set of languages that all describe the same objects. Thus all languages in a class are actually different syntaxes used to describe the same objects. For instance, dolmen defines the class of languages used in automated theorem proving, currently called `Logic`[2] Language

[1] Most languages have include statements which make provers read another file (for instance, containing axioms common to multiple problems), which may be in another directory than the original input problem file. In order to help in these cases, some languages, for instance TPTP, specify that included files should be searched in the directory specified by an environment variable. The `find` function is used to make these specificities abstract.

[2] This is currently the only class defined in dolmen.

classes in dolmen allows us to transparently parse any language supported by dolmen in that class: similarly to the single language parsers, a language class is a functor that takes as argument an implementation of terms and statements, and returns a function that can automatically detect the language of a file using its extension, select the appropriate parsing function, parse the given file, and return the parsed abstract syntax tree. This is especially interesting because adding a parser for a new language can be done in dolmen alone, and does not require the user to change her/his code. A resume of the interface provided by the automated theorem prover class functor is in Figure 6.1.

More information on dolmen can be found on its github repository[1], which features a tutorial, an online documentation and some examples.

6.2 An Extensible Typechecker

The next challenge due to the number of different languages and their differences is typechecking. While some provers do not have types, having at least simple types is necessary for some theories (such as arithmetic, where we have to distinguish integers from rationals), and may increase performances [31]. Even for provers that internally do not use types, checking that input problems are well-typed is important, as incoherent input problems may make the prover crash or make it output a wrong result.

However, typechecking multiple languages, each with some little differences, might be a litlle more difficult than parsing them, and at the same time, slightly easier. While the syntax of languages are quite different from each other, the typing systems of each language are almost the same, except for a very few features. This means that while parsers for each language could not re-use code, for typechecking, most of the code can actually be shared.

Before presenting how the extensible typechecker of ArchSAT was designed to easily handle multiple languages, here is an excerpt of differences in type-checking of languages:

- Most builtin symbols have different names depending on the language. For instance, the type of propositions is $o in TPTP, and Bool in SMT-LIB.
- Some languages have more builtin notions than others:
  - In TPTP, conjunction, disjunction, and equality have dedicated syntax rules, whereas in SMT-LIB, they are simply special cases of the generic application: conjunction is simply application of the “and” function symbol.
  - Zipperposition’s format specifies integers as different tokens than regular function symbols, whereas in TPTP integers are regular function symbols whose name is the integer string representation.
- The same builtin functions that exist in different languages may have different typing rules, for instance in arithmetic. TPTP has some kind of overloading for arithmetic symbols (and only for them), so that the same symbol for addition may be used for integers, rationals, and reals. SMT-LIB has distinct theories for integers only, reals only, and both of them, so the type of addition vary depending on the theory.
- The typing system may differ: Zipperposition’s format and TPTP’s TFF1 system has first-order prenex polymorphic terms, whereas SMT-LIB has parametric functions.
- Some languages have type inference whereas other do not. TPTP specifies that the type of unknown function symbols has to be inferred, and similarly the type of a quantified variable should be inferred if it is not specified.

Despite all these differences, most of the typechecking is the same for all of these languages: there is a global environment of declared constant symbols, a local environment of variables bound by quantifiers or let-binders, typing of application is the same, . . . Duplicating work in order to

[1] `https://github.com/Gbury/dolmen`

```
exception Extension_not_found of string
(** Raised when trying to find a language given a file extension. *)

(** {2 Supported languages} *)

type language =
  | Dimacs          (** Dimacs CNF format *)
  | ICNF            (** iCNF format *)
  | Smtlib          (** Smtlib format *)
  | Tptp            (** TPTP format (including THF) *)
  | Zf              (** Zipperposition format *)
(** The languages supported by the Logic class. *)

(** {2 High-level parsing} *)

val find :
  ?language:language ->
  ?dir:string -> string -> string option
(** Tries and find the given file, using the language specification. *)

val parse_file :
  ?language:language ->
  string -> language * S.t list
(** Given a filename, parse the file, and return the detected language
    together with the list of statements parsed.
    @param language specify a language; overrides auto-detection. *)

val parse_input :
  ?language:language ->
  [ `File of string | `Stdin of language ] ->
  language * (unit -> S.t option) * (unit -> unit)
(** Incremental parsing of either a file (see {parse_file}), or stdin.
    Returns a triplet [(lan, gen, cl)], containing the language detected
    [lan], a generator function [gen] for parsing the input, and a cleanup
    function [cl] to call in order to cleanup the file descriptors.
    @param language specify a language for parsing, overrides auto-detection
    and stdin specification. *)

(** {2 Mid-level parsing} *)

module type S = Language_intf.S with type statement := S.t
(** The type of language modules. *)

val of_language   : language  -> language * string * (module S)
val of_extension  : string    -> language * string * (module S)
val of_filename   : string    -> language * string * (module S)
(** These function take as argument either a language, a filename,
    or an extension, and return a triple:
    - language
    - language file extension (starting with a dot)
    - appropriate parsing module

    Extensions should start with a dot (for instance : [".smt2"])
    @raise Extension_not_found when the extension is not recognized. *)
```

Figure 6.1: dolmen Automated Theorem Prover Class Interface

have a typechecker for each language would therefore be wasteful and particularly difficult to maintain (any modification of the common part would need to be done on each typechecker).

ArchSAT relies on a unique typechecker, extensible in order to correctly deal with the specificities of each language. ArchSAT's typechecker has a notion of local environment, used for isntance to record the local bound variables. This local typing environment contains a function designed to handle the cases specific to a language, which is called the builtin function. This allows us to have a very compact description of each language specificities. More specifically, this function for builtins of each language has the following type:

```
type symbol =
  | Id of Dolmen.Id.t
    (** Identifiers from Dolmen, basically a string and a scope/namespace *)
  | Builtin of Dolmen.Term.builtin
    (** Builtin symbols for languages. *)
(** Wrapper around potential function symbols from the Dolmen AST. *)

type res =
  | Ttype   : res
  | Ty      : Expr.ty -> res
  | Term    : Expr.term -> res
  | Formula : Expr.formula -> res
  | Tags    : tag list -> res (**)
(** The results of parsing an untyped Dolmen term. *)

type builtin_symbols =
  env -> Dolmen.Term.t ->
  symbol -> Dolmen.Term.t list -> res option
(** Function to handle extensibility of typechecking. Called in order to
    type applications where the head symbol is unknown to the typechecker.
    The function is given the local environment, the whole Dolmen term being
    typechecked, the head symbol and the argument list, and should return
    the typecheked term. *)
```

The builtin function (of type `builtin_symbols`) of the local environment is called each time the head symbol of an application is unknown for the typechecker. The goal of the function is then to look at the symbol, potentially look at the terms to which it is applied, and then decide how to typecheck the arguments terms, and then the application. This allows us to easily implement the overloading of arithmetic function symbols for each language for example, although the common parts of the typechecker do not handle overloading.

For instance all the specificities of the TPTP language (excluding arithmetic) fit into this simple function :

```
let tptp_builtins env ast s args =
  match s with
  | Type.Id ({ Id.name = "$_"; ns = Id.Term } as id) ->
    Some (Type.wildcard env ast id args)
  | Type.Id { Id.name = "$tType"; ns = Id.Term } ->
    Some Type.Ttype
  | Type.Id { Id.name = "$o"; ns = Id.Term } ->
    Some (Type.parse_app_ty env ast Expr.Id.prop args)
  | Type.Id { Id.name = "$i"; ns = Id.Term } ->
    Some (Type.parse_app_ty env ast Expr.Id.base args)
  | Type.Id { Id.name = "$true"; ns = Id.Term } ->
    Some (Type.parse_app_formula env ast Expr.Formula.f_true args)
  | Type.Id { Id.name = "$false"; ns = Id.Term } ->
    Some (Type.parse_app_formula env ast Expr.Formula.f_false args)
  | Type.Id id when Id.equal id Id.tptp_role ->
    Some (Type.Tags [])
  | _ -> None
```

In this case, it can be seen that the TPTP builtin symbols, such as `$i`, `$o` and `$_` are parsed as regular identifiers, since the TPTP syntax has no special case for them. It then follows that they are represented as regular constants but with names that can be recognized. They must then be specifically handled during typechecking, but only when the input language is TPTP. This is easily done by pattern-matching the unknown identifiers that are given to the builtin function. As shown in the above code there are very few special cases for TPTP. This shows the usefulness of having a generic typechecker, extensible for the few special cases of each language.

Thanks to this separation beetween the main code of the typecher and the builtins of each language, the common parts of the typechecker fits in about 1000 lines of code, and has very useful features such as typo suggestions, human-readable error messages with full locations, unused quantified variable detection, inference of type arguments in polymorphic symbol applications, and inference of non-declared function symbols. The code defining the builtin functions for all languages supported by ArchSAT (without arithmetic) is about 100 lines of code. The interested reader can take a look at files `src/base/type.ml` and `src/base/builtin.ml` of the ArchSAT Git repository[1] for the code of the typechecker and builtin functions implementations respectively. Additionally, this extensible typechecker is planned to be integrated into dolmen[2].

6.3 Tags and Builtins for Preserving Semantic Data

The mechanisms presented in this thesis often need to attach some arbitrary information to expressions, for instance the original tree structure of flattened conjunction and disjunction (as explained in Section 4.3.2.2). This kind of information is difficult to store safely. First, because equal expressions may need to carry different information: for instance consider the two formulas $A \land (B \land C)$ and $(A \land B) \land C$. Both are interpreted as the conjunction of the list of formulas $[A; B; C]$. However, in order to store for each formula its original conjunction tree, hashtables (or other storage that uses equality on expressions) cannot be used since they require that equal expressions carry the same information. Instead, in ArchSAT uses polymorphic tags that can be attached to expressions.

Polymorphic tags rely on the ability to have type-safe heterogeneous maps in OCaml, that is, structures with the following interface:

[1] `https://github.com/Gbury/archsat`

[2] See `https://github.com/Gbury/dolmen/pull/4`

```
type map
(** The type of immutable maps from tags to values. *)

type 'a t
(** A tag to which is mapped values of type ['a] in maps. *)

val create : unit -> 'a t
(** Create a new tag. *)

val empty : map
(** The empty map. *)

val get : map -> 'a t -> 'a option
(** Get the value of a tag (if it exists). *)

val add : map -> 'a t -> 'a -> map
(** Add a value to a tag in a map. *)
```

In order to have such maps, ArchSAT uses the `CCMixMap` module from the OCaml-containers [1] library, which interested reader can read in more details on its Github repository: `https://github.com/c-cube/ocaml-containers`.

In ArchSAT, each expression carries such an heterogeneous map, which is thus tied to the physical representation of the expression, allowing equal expressions to have different maps. This is how each expression is able to carry information about its original shape, but is also used for other uses. Current uses of tags in ArchSAT includes :

- Identification of formulas that are marked as rewrite rules.
- Pretty-printing information on identifiers.
- Memoization of various functions that read and/or modify other tags on expressions (and thus cannot be memoized using a hashtable).

6.4 Backtracking Using a Global Stack

All SMT and McSat solvers need a way to backtrack the state of the theories in order to keep the internal state of theories synchronized with the state of the core SAT solver. To do so, ArchSAT uses a global backtrack stack: each action that modifies the internal state of a theory registers the corresponding reverse modification on the stack, and when the solver backtracks, every action on the backtrack stack is executed until the backtrack point is reached. Wrappers for data structures can be built using this backtrack stack in order to provide, for instance, hashtables whose basic operations automatically registers the needed actions on the backtrack stack.

While not particularly new nor original, this greatly simplifies the implementation of theories, which often store their internal state in a hashtable. A theory then only has to use a backtrackable hashtable (which offers the same functionalities as a regular hashtable), and its internal state is automatically backtracked, without the need of any code for backtracking in the theory itself.

The backtrack stack, as well as backtrackable hashtables that are used in ArchSAT, are implemented in the file `src/util/backtrack.ml` of the repository.

6.5 Extensible and Type-safe Message Passing Between Extensions

SMT and McSat solvers are traditionally very modular since theories are quite independant from each other, and can be added or removed easily. Modularity is possible because theories usually only have a static interface that they need to implement in order to work with the rest of the prover[1]. In the case of SMT solvers, this interface is basically a single function, which is given the newly assumed literals, and must either return SAT, or UNSAT with a conflict clause. However, in practice a lot of additional features may be used (though not required) by theories. For instance, a very common feature of SMT solvers is to allow theories to do a final check once a propositional model is found; this allows theories to run costly non-incremental procedures at that point. Another example is the mechanisms of rounds in ArchSAT: once a first-order model has been found, instantiations are done, adding clauses to the solver, and then the solver restarts. To do so, some theory must be able to detect when a first-order model has been found, in order to look for instantiations. All these examples show that adding new features often need to extend some theories, but not all: not all theories look for instantiations, or need a final check of the propositional model found by the SAT solver.

For these applications, ArchSAT has a mechanism of extensible message passing and handling for extensions. This means that any extension can send messages and respond to messages, and that the type of messages is extensible so that new theories (or other modules within ArchSAT) can define their own messages. The type of messages is defined as an extensible generalized algebraic datatype using a type parameter to specify the expected type of answers to that message: a message of type `'a msg` is a message that expect an answer of the `'a`:

```ocaml
type 'ret msg = ..
(** Messages are arbitrary data that can be sent to extensions,
    and expect an answer of type ['ret option].
    Note that since it is an extensible type, extensions will
    most likely ignore most messages *)

type _ msg += If_sat : ((Expr.formula -> unit) -> unit) -> unit msg
(** This message contains a function to iter over current assumptions.
    It is sent whenever the sat solver has found a propositional model.
    Extensions interested in the final check have the possibility of
    performing side-effects to add new clauses, or raise a conflict
    if necessary. *)
```

Each extension then has to implement a handler defined as follows:

```ocaml
type handle = { handle : 'ret. 'ret msg -> 'ret option; }
(** Type for message handlers. Enclosed in a record to ensure full polymorphism *)
```

The `handle` function inside the an arbitrary message ocamlm, of type `'ret msg`, and returns a value of the type `'ret option`. This way, the typechecker guarantees that no problem can arise at runtime, while allowing communications of any type between the extensions. Note that wrapping the handler function in a record is necessary to ensure that the handler is polymorphic enough to handle all possible messages[2]. For messages unknown to the extension, or messages that the extension does not need to answer, the handler function simply returns `None`. Thus, the simplest handler simply ignores all messages and always returns `None`.

Finally, messages can be sent and the answers analyzed using the following functions:

[1] This is particularly explicit when looking at the functors provided by the mSAT library: each theory only has to implement the interface recquired by the functor.

[2] This wrapping in a record is necessary for the current OCaml compiler, as the function requires an explicitly polymorphic type and the compiler cannot infer such types currently, hence wrapping in a record is a way to provide the compiler with enough type annotations.

```
val handle : ('acc -> 'ret option -> 'acc) -> 'acc -> 'ret msg -> 'acc
(** Send a message to all extensions, and fold over the answers. *)

val send : unit msg -> unit
(** Send the given message to all extensions, ignoring any exception raised
    during the handling of the message. *)

val ask : string -> 'a msg -> 'a option
(** Send the given message to the extension with the given name,
    and returns the return value (if the extension replied). *)
```

Whenever a message is sent to the extensions, each handler is queried for an answer, which is then provided to a folding function. Additionally, some wrappers are provided to simplify the case of messages that do not expect an answer (that is, messages of type `unit msg`), or messages that are meant to be sent to a particular extension (in which case, folding is unnecessary, since there will be only one answer).

This mechanism provides a very simple way of extending the behaviour of theories without modifying all the theories each time a new event or message is created. It is notably used during proof generation, for theories to prove their lemmas, but also for theories to provide information when printing resolution proof in the graphviz dot format.

6.6 A Pipeline for the Main Execution Loop

An automated theorem prover has to do a lot of tasks besides proof search: parse options, set some configuration variables accordingly, parse the input problem, typecheck the input problem, solve the problem, export the solution to various formats if some options have been set, . . . The code orchestrating all of these tasks can quickly become difficult to maintain and read. This is particularly true in the case of SMT-LIB, which requires a prover to have two distinct modes: the regular mode where an input file is parsed, and an interactive mode where the user can give commands one by one. This makes computational limits such as timeouts and maximum memory harder to enforce since it must consider two distinct situations: in interactive mode, each statement should have its own limits, whereas when solving an input file, the limits should apply to the processing of all statements in the file.

In order to alleviate these problems, ArchSAT introduces a notion of pipeline. A pipeline is a sequence of computations phases, that correspond to the successive tasks done by a prover. The pipeline can then be run on an input generator of statements, which can be either the statements written by the user in interactive mode, or a single statement that includes the input file to consider. The main complexity of the pipeline mechanism comes from the expansion of includes. An include statement is actually processed into a generator of statements, which requires the pipeline to handle computation phases that evaluate to a generator of statements (statements which themselves can evaluate to a generator of statements recursively). Additionally, pipelines are useful to abstract over computation limits: each run of the pipeline on a statement is subject to the computational limits set by the options. This makes limits apply to each individual statement in interactive mode, and apply to a whole file when it is included.

A pipeline is basically a way to structure a computation: when run, it takes some inputs and returns some outputs. It is therefore defined as a polymorphic type, which can be evaluated on some inputs :

```
type ('a, 'b) pipeline
(** The type of pipelines from values of type ['a] to values of type ['b]. *)

val eval : ('a, 'b) pipeline -> 'a -> 'b
(** Evaluate a pipeline to a function. *)
```

The simplest pipeline is the one that does nothing and directly returns its input :

```ocaml
val _end : ('a, 'a) pipeline
```

Non-trivial pipelines are then built by chaining operators. An operator represents an atomic computation, whereas a pipeline is a composite computation, built from a possibly complex arrangement of operators. The type of operators is also a polymorphic type :

```ocaml
type ('a, 'b) op
(** An operator from values of type ['a] to values of type ['b]. *)
```

Operators are thin wrappers around functions. Additionally, each operator is given a name, to better identify it:

```ocaml
val apply : ?name:string -> ('a -> 'b) -> ('a, 'b) op
(** Create an operator from a function *)
```

The main way to build pipelines is to compose an operator with a pipeline. This order (operator first, then the rest of the pipeline) is chosen in order to ensure that pipelines can be implemented using tail-recursive calls. This is done via the `(@>>>)` operator[1]. Another combinator allows to concatenate two pipelines, but this breaks the tail-recursive evaluation of pipelines, thus the preferred way of building pipelines is to use operators with the `(@>>>)` infix symbol :

```ocaml
val (@>>>) : ('a, 'b) op -> ('b, 'c) t -> ('a, 'c) t
(** Add an operator at the beginning of a pipeline. *)

val (@|||) : ('a, 'b) t -> ('b, 'c) t -> ('a, 'c) t
(** Concatenate two pipelines. Whenever possible it is best to use [(@>>>)],
    which creates tail-rec pipelines. *)
```

All of the above is fairly standard and easy to implement. The main advantage of pipelines is to correctly manage includes statements, which make the prover insert the contents of another file in the current file. This is tricky to do because include statements can be nested, i.e. an included file can itself include another file, and so on... This requires to perform a fixpoint computation on include statements. Pipelines in ArchSAT offer a way to insert the fixpoint computation of an operator in a pipeline using the following functions :

```ocaml
type 'a fix = [ `Ok | `Gen of bool * 'a Gen.t ]
(** Type used to fixpoint expanding statements such as includes. *)

val fix : ('a * 'b, 'a * 'b fix) op -> ('a * 'b, 'a) t -> ('a * 'b, 'a) t
(** Perform a fixpoint expansion. *)
```

The `fix` function above is meant to compute the fixpoint of a function that generates values of type `'b`. To do so, it maintains a stack of generators of values of type `'b`. The given operator can then add a new generator `g` on the stack by returning a `` `Gen (_, g) ``. This would typically be a statement generator that corresponds to the results of parsing the included file. On values that do not need to be expanded (that is, any statement that is not an include), the operator can simply return `` `Ok ``. In this case, the statement is given to the pipeline `p`, which is the second argument of the `fix` function. Additionally, a value of type `'a` is “threaded” through the pipeline,

[1] Note that the composition operators for pipelines are expressed using infix symbols in order to be more readable.

so that it can be modified by the pipeline p, and then used when generating the next statement. This is useful for languages such as SMT-LIB where statements executed later in the pipeline can modify the prover options.

Once a pipeline is built, running it can then be done using the eval function shown above, which simply calls operators in the correct order. In practice however, a more complex function is used so that when evaluating the pipeline, the time and memory limits given to the prover are correctly enforced. This is relatively easy to do as it allows us to separate the implementation of the concrete computations that need to be done, from the handling of computation limits. Pipelines make the source code of the main binary much more readable. For instance, using some infix combinators, the code of the main loop of ArchSAT looks like :

```
(fix (apply ~name:"expand" Pipe.expand) (
  (apply ~name:"execute" Pipe.execute)
    @>|> (f_map ~name:"typecheck" ~test:Pipe.run_typecheck Pipe.typecheck)
    @>|> (f_map ~name:"solve" Pipe.solve)
    @>|> (iter_ ~name:"print_res" Pipe.print_res)
    @>>> (f_map ~name:"translate" ~test:Pipe.run_translate Pipe.translate)
    @>|> (iter_ ~name:"export" Pipe.export)
    @>>> (iter_ ~name:"print_proof" Pipe.print_proof)
    @>>> (iter_ ~name:"print_model" Pipe.print_model)
    @>>> (apply fst) @>>> _end)
)
```

Which is much more readable than the hundreds of lines that would have been required in order to write a loop, manually order the computation phases, and correctly reset limits in the presence of exceptions. The full code for defining pipelines is present in file src/middle/pipeline.ml of the ArchSAT repository, while the implementation of computations phases, such as typechecking, is in file src/middle/pipe.ml.

6.7 Escaping Identifiers for Correct Outputs

Finally, one of the most technical problems is the escaping of identifiers in exported files. Whenever a file is written by the prover, it is usually in a defined format, such as the graphviz language for proof graphs, the Coq or Dedukti language for formal proofs, etc. All of these formats have different definitions of what characters are allowed to be used:

- Proof graphs in graphviz heavily use html tables, consequently characters such as ’<’, ’>’, ’&’ can be used but have to be escaped into strings such as "&gt;".
- Coq requires identifiers not to start with a digit, but they can contain (and start with) underscores and unicode letters (a non-exhaustive list of allowed character languages is given in the Coq documentation, but there is no precise definition of what is not allowed).
- Dedukti does not currently have a public documentation of its syntax.
- TPTP has different rules for variables and constant function symbols:
  - Variables must start with an uppercase ASCII letter, and can then contain any ASCII letter, digit, or underscore.
  - Constants must start with a lowercase ASCII letter, or a dollar sign, and can then contain any ASCII letter, digit, or underscore.

For instance, a lot of TPTP builtin symbols such as the type of propositions $o, and the default type of term $i are not valid identifiers in Coq. Thus, this can result in syntax errors in produced files if identifiers are not properly escaped according to each language’s specification before being printed. However, most languages (except graphviz) do not offer a simple way to have a bijection between identifier names and allowed names. This means that naive escaping (or actually renaming) of a symbol name can create artificial conflicts in the target language that do not exist in the source.

For instance, consider two identifers named $i and #i used in an ArchSAT proof. $i comes from the input problem in the TPTP language, and #i was generated by ArchSAT[1]. None of these identifiers can be printed as such in the Coq language. A simple way to escape such names for Coq is to replace illegal characters by an underscore. That way, most of the names are preserved and can be linked between the original file and the proof, and escaped names are close to the original. However, in this example, this makes $i and #i be escaped to the same name, which is not intended, and may cause errors in the generated proof.

Thus, there is a need to check for collisions and possibly do some renaming after escaping identifiers names. This is implemented in a generic way in ArchSAT, in the source file `src/util/escape.ml`. This module handles both escaping and renaming, using user-provided functions, and correctly handling unicode names. This allows us to have relatively nice variable names such as ϵ_2 in languages that support unicode such as Coq, while not breaking the output on languages that have no unicode support. The usefulness of this generic implementation is that it can be used differently for each language in order to conform to its syntax by simply providing the escaping and renaming functions.

ArchSAT therefore ensures that it produces syntaxically correct proofs and graphs regardless of the input format, the input names of constants and variables, and regardless of the names of ArchSAT-generated values.

[1] Names can be generated for various reasons, ranging from disambiguation of nested quantification during typechecking to names generated for epsilon-terms, or for new variables.

Conclusion

In this thesis, we have presented ArchSAT, an McSat solver in which several theories have been integrated, such as the tableau method, rewriting, and superposition, and which can generate formal proofs. ArchSAT's use of rewriting successfully allows it to prove more theorems than most other provers on problems that use rewriting, and its generated proofs make it suitable for applications requiring high confidence in its results. Additionally, its modular architecture makes it a very powerful tool to experiment with new theories. These strong points make ArchSAT a unique theorem prover, which fills a currently vacant place in the automated theorem proving community.

The tableau method is integrated into ArchSAT as a regular SMT/McSat theory, replacing both the transformation in clausal normal form and the trigger mechanism used in most SMT solvers. To do so, it unfolds logical connectives by generating clauses, and uses rigid meta-variables to deal with quantified formulas. Formulas containing the meta-variables are then unified in order to find substitutions, which are used to instantiate the corresponding formulas. As a replacement for the clausal normal form transformation, the tableau theory seems very adequate, as the performances of ArchSAT on pure ground problem are satisfactory. Concerning quantified formulas, the tableau theory appears to work quite well on the TPTP library where, with a few adjustments, it should become competitive with other automated theorem provers. This shows that the approach is promising in practice, in addition to being one of the few methods for quantified formulas that aim at being complete.

Rewriting is currently studied as one serious way to increase automated theorem provers performances. Rewrite rules are expressive enough to axiomatize most theories, and have enough structure so that automated theorem provers can use them much better than regular axioms. This is shown by the integration of rewriting in ArchSAT, which is split into two parts. The first consists in speeding up reasoning on ground terms by normalizing them, either using the static rewrite engine, or the dynamic trigger-like mechanism. This integration allows ArchSAT to have good results on problems axiomatized using rewrite rules: on those problems, ArchSAT surpasses most of the other provers. This shows the usefulness of turning axioms into rewrite rules, and validates the practical approach of its implementation in ArchSAT.

Rewriting is also integrated into the unit rigid superposition algorithm, which unifies terms modulo equalities and rewrite rules. The algorithm relies on unit superposition, where considered each clause only has one atom, and tags each clause with substitutions in order to enforce rigidity of some variables. Rewrite rules are managed by considering their variables as not rigid, allowing the algorithm to instantiate the variables in the rules as many times as required. The implementation of this algorithm allows ArchSAT to solve problems involving both ground equalities and rewrite rules. However, it seems currently limited to handle small or medium-sized problems. In order to better scale, some more techniques and heuristics from existing state-of-the-art superposition provers could be used.

Finally, the two formal proof outputs of ArchSAT offer a very strong confidence in its results. ArchSAT has a structured representation of proofs using proof trees, allowing us to eliminate most of the duplication that would usually come with supporting proof output in multiple different formats. This structured representation is therefore crucial to maintain multiple proof outputs. Additionally, this means that proofs found by ArchSAT are checked twice: a first time internally, when building the structured proof tree, and then checked a second time by two independent proof assistants: Coq and Dedukti. There are very few automated theorem provers with an independently checkable proof output, and less with two such outputs. This places ArchSAT in a unique position concerning formal verification of automated theorem provers results.

Perspectives As show by the experiments, there is still room for improvement, for ArchSAT, mostly regarding the handling of quantified formulas using meta-variables. The use of meta-variables in ArchSAT allows us to have intrinsically different handling of quantified formulas than most traditional SMT solvers, by being closer to techniques used by first-order theorem provers. Following this idea, the instantiation mechanism used by ArchSAT currently could be replaced by a notion of first-order conflict inspired by the technique used by the Princess theorem in [7]. This would solve the current problem where ArchSAT generates too many terms: currently, either ArchSAT performs all instantiations that it finds by unification, or it uses a heuristic to choose some of these instantiations. This tends to either lead to too many new terms in the former case, or is heavily dependant on the heuristic in the latter case.

Rigid unit superposistion could be improved significantly, by adding more redundancy criteria, inference rules, and some priorization heuristics. Particularly, the version presented in this thesis has a scalability problem when duplicating meta-variables. This could be solved by using more targeted redundancy criteria, inspired by the work done in other superposition provers. Additionally, fine-tuning the implementation to use better prioritization of enqueued clauses in the loop of the superposition algorithm seems to be a critical point to achieve high practical performances. Other algorithms for solving rigid E-unification, such as in [6], could also be implemented in order to compare them to rigid unit superposition. The drawback of using such algorithms is that they would not deal with rewrite rules.

Completing the implementation of an arithmetic theory for ArchSAT[1] would also allow us to realize interesting experiments. More specifically, just like for the theory of equality in ArchSAT, where a union-find algorithm was added to the standard McSat theory for equality, the standard theory of linear rational (or real) arithmetic could be extended to use the simplex algorithm in order to find conflicts faster. The standard theory of arithmetic, and the one extended with the simplex could then be compared to assert the usefulness of such extensions. This experiment would also be interesting for the theory of equality.

Finally, the formal proof output of ArchSAT is currently very heavily checked: each reasoning step in the structured proof tree is meticulously checked, and all the proof term types are checked, which currently slows down proof generation[2]. This could be solved by using simpler proof terms with less typechecking internally, as a compromise between the pure printing solution, which has no structured representation of proofs, and the current very structured proof tree, which has probably too much structure. This should allow ArchSAT to generate formal proofs faster than it currently does.

[1] which was started during the internship of Thomas Bernardi.

[2] Proof generation by ArchSAT currently takes about as much time as the checking of generated proofs by either Coq of Dedukti.

Bibliography

[1] The ocaml-containers library. https://github.com/c-cube/ocaml-containers.

[2] The smt competition 2018. http://smtcomp.sourceforge.net/2018/results-competition-main.shtml?v=1531410683, June 2018.

[3] Jean-Raymond Abrial. *The B-Book, Assigning Programs to Meanings.* Cambridge University Press, Cambridge (UK), 1996. ISBN 0521496195.

[4] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in event-b. *International journal on software tools for technology transfer*, 12(6):447–466, 2010.

[5] Sergio Antoy, Rachid Echahed, and Michael Hanus. A needed narrowing strategy. *Journal of the ACM (JACM)*, 47(4):776–822, 2000.

[6] Peter Backeman and Philipp Rümmer. Efficient algorithms for bounded rigid e-unification. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 70–85. Springer, 2015.

[7] Peter Backeman and Philipp Rümmer. Theorem proving with bounded rigid e-unification. In *International Conference on Automated Deduction*, pages 572–587. Springer, 2015.

[8] Haniel Barbosa. *New techniques for instantiation and proof production in SMT solving.* PhD thesis, Université de Lorraine, Universidade Federal do Rio Grande do Norte, 2017.

[9] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual: Version 6.1.* PhD thesis, Inria, 1997.

[10] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. Snowbird, Utah.

[11] Clark Barrett, Leonardo De Moura, and Pascal Fontaine. Proofs in satisfiability modulo theories. *All about proofs, Proofs for all*, 55(1):23–44, 2015.

[12] Clark Barrett, Leonardo De Moura, and Aaron Stump. Smt-comp: Satisfiability modulo theories competition. In *International Conference on Computer Aided Verification*, pages 20–23. Springer, 2005.

[13] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[14] B. Beckert. Semantic Tableaux with Equality. *Journal of Logic and Computation*, 1994.

[15] B. Beckert and R. Hähnle. An Improved Method for Adding Equality to Free Variable semantic Tableaux. In D. Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *LNCS*, pages 507–521. Springer, 1992.

[16] Bernhard Beckert and Christian Pape. Incremental theory reasoning methods for semantic tableaux. In *International Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, pages 93–109. Springer, 1996.

[17] Frédéric Besson, Pascal Fontaine, and Laurent Théry. A flexible proof format for smt: A proposal. In *First International Workshop on Proof eXchange for Theorem Proving-PxTP 2011*, 2011.

[18] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Conflict-driven clause learning sat solvers. *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, pages 131–153, 2009.

[19] Jasmin Blanchette and Andrei Paskevich. TFF1: The TPTP Typed First-Order Form with Rank-1 Polymorphism. In M.P. Bonacina, editor, *CADE - 24th International Conference on Automated Deduction - 2013*, volume 7898 of *LNCS*, pages 414–420, Lake Placid, NY, United States, June 2013. Springer.

[20] Jasmin Christian Blanchette and Andrei Paskevich. TFF1: The TPTP Typed First-Order Form with Rank-1 Polymorphism. In *Conference on Automated Deduction (CADE)*, volume 7898 of *LNCS*, pages 414–420, Lake Placid (NY, USA), June 2013. Springer.

[21] FranCcois Bobot, Sylvain Conchon, E Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The alt-ergo automated theorem prover, 2008, 2013.

[22] Sascha Böhme and Tjark Weber. Fast lcf-style proof reconstruction for z3. In *International Conference on Interactive Theorem Proving*, pages 179–194. Springer, 2010.

[23] Maria Paola Bonacina, Stéphane Graham-Lengrand, and Natarajan Shankar. Satisfiability modulo theories and assignments. In *International Conference on Automated Deduction*, pages 42–59. Springer, 2017.

[24] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. In *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings*, pages 151–165, 2007.

[25] Thomas Bouton, Diego Caminha B de Oliveira, David Déharbe, and Pascal Fontaine. verit: an open, trustable and efficient smt-solver. In *International Conference on Automated Deduction*, pages 151–156. Springer, 2009.

[26] Angelo Brillout, Daniel Kroening, Philipp Rümmer, and Thomas Wahl. An interpolating sequent calculus for quantifier-free presburger arithmetic. *Journal of Automated Reasoning*, 47(4):341–367, 2011.

[27] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer, 2009.

[28] Guillaume Burel. Experimenting with deduction modulo. In *International Conference on Automated Deduction*, pages 162–176. Springer, 2011.

[29] Guillaume Burel. A shallow embedding of resolution and superposition proofs into the $\lambda\pi$-calculus modulo. In *PxTP-Third International Workshop on Proof Exchange for Theorem Proving-2013*, volume 14, pages 43–57. EasyChair, 2013.

[30] Guillaume Bury. mSAT: An OCaml SAT Solver. In *OCaml Users and Developers Workshop*, September 2017.

[31] Guillaume Bury, Raphaël Cauderlier, and Pierre Halmagrand. Implementing Polymorphism in Zenon. In *11th International Workshop on the Implementation of Logics (IWIL)*, Suva, Fiji, November 2015.

[32] Guillaume Bury, Simon Cruanes, David Delahaye, and Pierre-Louis Euvrard. An automation-friendly set theory for the b method. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 409–414. Springer, 2018.

[33] Guillaume Bury and David Delahaye. Integrating Simplex with Tableaux. In *Automated Reasoning with Analytic Tableaux and Related Methods*, 24th International Conference, TABLEAUX 2015, Wroclaw, Poland, September 21–24, 2015. Proceedings, pages 86–101, Wrowlaw, Poland, September 2015.

[34] Guillaume Bury, David Delahaye, Damien Doligez, Pierre Halmagrand, and Olivier Hermant. Automated Deduction in the B Set Theory using Typed Proof Search and Deduction Modulo. In *LPAR 20 : 20th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, Suva, Fiji, November 2015.

[35] Guillaume Bury, David Delahaye, Damien Doligez, Pierre Halmagrand, and Olivier Hermant. Automated Deduction in the B Set Theory using Typed Proof Search and Deduction Modulo. In *Logic for Programming, Artificial Intelligence and Reasoning (LPAR) – Short Presentations*, volume 35, pages 42–58, Suva (Fiji), November 2015. EasyChair.

[36] Raphaël Cauderlier and Pierre Halmagrand. Checking zenon modulo proofs in dedukti. *arXiv preprint arXiv:1507.08719*, 2015.

[37] Thierry Coquand and Gérard Huet. *The calculus of constructions.* PhD thesis, INRIA, 1986.

[38] Simon Cruanes. *Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond. (Extensions de la Superposition pour l'Arithmétique Linéaire Entière, l'Induction Structurelle, et bien plus encore).* PhD thesis, École Polytechnique, Palaiseau, France, 2015.

[39] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[40] Leonardo De Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In *International Conference on Automated Deduction*, pages 183–198. Springer, 2007.

[41] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[42] Leonardo De Moura and Dejan Jovanović. A model-constructing satisfiability calculus. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 1–12. Springer, 2013.

[43] Anatoli Degtyarev and Andrei Voronkov. Simultaneous rigid e-unification is undecidable. In *International Workshop on Computer Science Logic*, pages 178–190. Springer, 1995.

[44] Anatoli Degtyarev and Andrei Voronkov. The undecidability of simultaneous rigid e-unification. *Theoretical Computer Science*, 166(1-2):291–300, 1996.

[45] Anatoli Degtyarev and Andrei Voronkov. What You Always Wanted to Know About Rigid E-Unification. In *Logics in Artificial Intelligence (JELIA)*, volume 1126 of *LNCS*, pages 50–69, Évora (Portugal), September 1996. Springer.

[46] David Delahaye, Damien Doligez, Frédéric Gilbert, Pierre Halmagrand, and Olivier Hermant. Zenon modulo: When achilles outruns the tortoise using deduction modulo. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 274–290. Springer, 2013.

[47] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem Proving Modulo. *Journal of Automated Reasoning (JAR)*, 31(1):33–72, September 2003.

[48] Claire Dross. *Generic decision procedures for axiomatic first-order theories.* PhD thesis, Université Paris Sud-Paris XI, 2014.

[49] Claire Dross, Sylvain Conchon, Johannes Kanig, and Andrei Paskevich. Adding decision procedures to smt solvers using axioms with triggers. *Journal of Automated Reasoning*, 56(4):387–457, 2016.

[50] Claire Dross, Sylvain Conchon, and Andrei Paskevich. *Reasoning with triggers.* PhD thesis, INRIA, 2012.

[51] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2):1–2, 2006.

[52] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. Smtcoq: A plug-in for integrating smt solvers into coq. In *International Conference on Computer Aided Verification*, pages 126–133. Springer, 2017.

[53] Michael Franssen. Implementing rigid e-unification. In *22nd International Conference on Automated Deduction McGill University, Montreal, Canada August 2009*, page 32. Citeseer, 2009.

[54] Jean H Gallier and Wayne Snyder. A general complete e-unification procedure. In *International Conference on Rewriting Techniques and Applications*, pages 216–227. Springer, 1987.

[55] Jean H Gallier and Wayne Snyder. Complete sets of transformations for general e-unification. *Theoretical Computer Science*, 67(2-3):203–260, 1989.

[56] Jean Goubault. A rule-based algorithm for rigid e-unification. In *Kurt Gödel Colloquium on Computational Logic and Proof Theory*, pages 202–210. Springer, 1993.

[57] Gérard Huet. A complete proof of correctness of the knuth-bendix completion algorithm. *Journal of Computer and System Sciences*, 23(1):11–21, 1981.

[58] Dejan Jovanovic, Clark Barrett, and Leonardo De Moura. The design and implementation of the model constructing satisfiability calculus. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 173–180. IEEE, 2013.

[59] Donald E Knuth and Peter B Bendix. Simple word problems in universal algebras. In *Automation of Reasoning*, pages 342–376. Springer, 1983.

[60] Konstantin Korovin. iprover–an instantiation-based theorem prover for first-order logic (system description). In *International Joint Conference on Automated Reasoning*, pages 292–298. Springer, 2008.

[61] Konstantin Korovin and Christoph Sticksel. Labelled Unit Superposition Calculi for Instantiation-Based Reasoning. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 6397 of *LNCS*, pages 459–473, Yogyakarta (Indonesia), October 2010. Springer.

[62] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer, 2013.

[63] Stéphane Lescuyer and Sylvain Conchon. A reflexive formalization of a sat solver in coq. In *21 st International Conference on Theorem Proving in Higher Order Logics*, page 64. Citeseer, 2008.

[64] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982.

[65] Alain Mebsout and Cesare Tinelli. Proof certificates for smt-based model checkers for infinite-state systems. In *Formal Methods in Computer-Aided Design (FMCAD), 2016*, pages 117–124. IEEE, 2016.

[66] Michał Moskal, Jakub Łopuszański, and Joseph R Kiniry. E-matching for fun and profit. *Electronic Notes in Theoretical Computer Science*, 198(2):19–35, 2008.

[67] Greg Nelson and Derek C Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

[68] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In *International Conference on Rewriting Techniques and Applications*, pages 453–468. Springer, 2005.

[69] Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Information and Computation*, 205(4):557–580, 2007.

[70] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. *Handbook of automated reasoning*, 1:371–443, 2001.

[71] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[72] Werner Nutt, Pierre Réty, and Gert Smolka. Basic narrowing revisited. *Journal of Symbolic Computation*, 7(3-4):295–317, 1989.

[73] David A. Plaisted. Special Cases and Substitutes for Rigid E-Unification. *Applicable Algebra in Engineering, Communication and Computing (AAECC)*, 10(2):97–152, January 2000.

[74] S. V. Reeves. Adding equality to semantic tableaux. *Journal of Automated Reasoning*, 3:225–246, 1987.

[75] Pierre Réty. Improving basic narrowing techniques. In *International Conference on Rewriting Techniques and Applications*, pages 228–241. Springer, 1987.

[76] Camilo Rocha, José Meseguer, and César Muñoz. Rewriting modulo smt and open system analysis. In *International Workshop on Rewriting Logic and its Applications*, pages 247–262. Springer, 2014.

[77] Philipp Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 274–289. Springer, 2008.

[78] Michaël Rusinowitch. Theorem-proving with resolution and superposition. *Journal of Symbolic Computation*, 11(1-2):21–49, 1991.

[79] Stephan Schulz. E – A Brainiac Theorem Prover. *AI Communications*, 15(2-3):111–126, August 2002.

[80] Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013.

[81] Aix-en-Provence Steria. France. atelier b, user and reference manuals, 1996.

[82] Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean, and Cesare Tinelli. Smt proof checking using a logical framework. *Formal Methods in System Design*, 42(1):91–118, 2013.

[83] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[84] Geoff Sutcliffe. The cade atp system competition—casc. *AI Magazine*, 37(2):99–101, 2016.

[85] Robert E Tarjan and Jan Van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM (JACM)*, 31(2):245–281, 1984.

[86] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.

[87] Andrei Voronkov. Avatar: The architecture for first-order theorem provers. In *International Conference on Computer Aided Verification*, pages 696–710. Springer, 2014.

[88] Andreas Weiermann. Termination proofs for term rewriting systems by lexicographic path orderings imply multiply recursive derivation lengths. *Theoretical Computer Science*, 139(1-2):355–362, 1995.

[89] Lintao Zhang, Conor F Madigan, Matthew H Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285. IEEE Press, 2001.

Index

Appendices

Appendix A

mSAT poster

mSAT was presented at the OCaml 2017 workshop, during the Internation Conference on Functional programming [30], during the poster sessions. See next page for the full size poster, and Section 1.4.1 for a more detailed explanation of what mSAT is.

mSAT : An OCaml SAT Solver

Guillaume Bury

DEDUC⊢EAM (INRIA) - LSV / CNRS
guillaume.bury@inria.fr

Introduction

mSAT : a SAT solving library in OCaml. It solves the **satisfiability** of propositional clauses. It is **Modular** : the user provides the theory. And it **produces formal proofs**.

Conflict Driven Clause Learning

Propagation If there exists a clause $C = C' \vee a$, where C' is false in the partial model, then add $a \mapsto \top$ to the partial model, and record C as the reason for a.

Decision Take an atom a that is not yet in the partial model, and add $a \mapsto \top$ to the model.

Conflict A conflict is a clause C that is false in the current partial model.

Analyze Perform resolution between the analyzed clause and the reason behind the propagation of its most recently assigned litteral, until the analyzed clause is suitable for backumping.

Backjump A clause is suitable for backjumping if its most recently assigned litteral a is a decision. We can then backtrack to before the decision, and add the analyzed clause to the solver, which will then enable to propagate $a \mapsto \bot$.

SMT Formulas using first-order theories can be handled using a theory. Each formula propagated or decided is sent to the theory, which then has the duty to check whether the conjunction of all formulas seen so far is satisfiable, if not, it should return a theory tautology (as a clause), that is not satisfied in the current partial model.

Implementation

- ▶ Imperative design
  - ✓ 2-watch litteral
  - ✓ Backtrackable theories (less demanding than immutable theories)
- ▶ Features
  - ✓ Functorized design, using generative functors
  - ✓ Local assumptions
  - ✓ Model output and proof output (Coq, dot)

Solver Interface

```
module Make(Th: Theory_intf.S)() : sig
  type 'f sat_state = { eval : 'f -> bool; ... }
  type ('c,'p) unsat_state =
    { conflict: unit -> 'c; proof : unit -> 'p }
  type res = Sat of formula sat_state
           | Unsat of (clause, proof) unsat_state
  val assume : ?tag:int -> atom list list -> unit
  val solve : ?assumptions:atom list -> unit -> res
end
```

Other Solvers

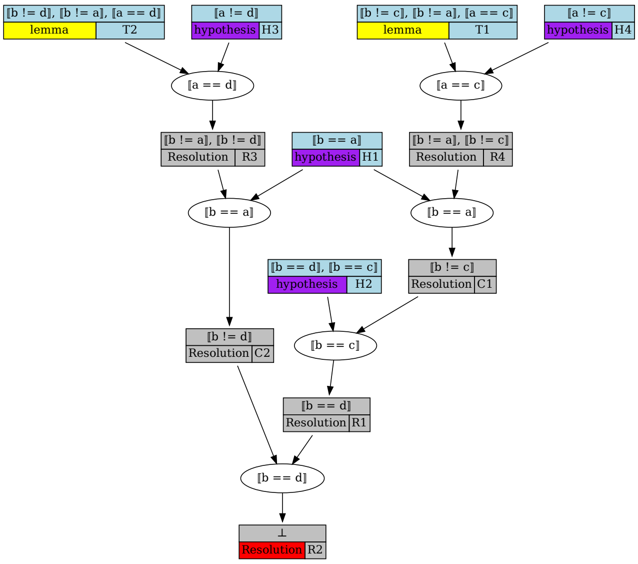
regstab	SAT	binary only	only pure SAT
minisat **sattools** ocaml-sat-solvers	SAT	C bindings	only pure SAT
Alt-ergo	SMT	binary only	Fixed theory
Alt-ergo-zero	SMT	OCaml lib	Fixed theory
ocamlyices yices2	SMT	C bindings	Fixed theory

Problem Example

Are the following hypotheses satisfiable ?

$H1 : \ a = b$ $\quad H2 : b = c \vee b = d$

$H3 : a <> d$ $\quad H4 : \ a <> c$



Theory Interface

```
type ('f, 'p) res = Sat | Unsat of 'f list * 'p
type 'f slice = { start:int; length:int; get:int -> 'f }
module type S = sig
  val backtrack : level -> unit
  val current_level : unit -> level
  val assume : formula slice -> (formula, proof) res
end
```

Proof Generation

- ✓ Each clause records its "history" which is the clauses used during analyzing
- ✓ Minimal impact on proof search (already done to compute unsat-core)
- ✓ Sufficient to rebuild the whole resolution tree
- ✓ A proof is a clause and proof nodes are expanded on demand
  → no memory issue
- ✓ Enables various proof outputs :
  - Dot/Graphviz (see example above)
  - Coq (and soon Dedukti) formal proofs

Performances

solver	Alt-ergo-zero	mSAT	minisat	cryptominisat
(package)	aez	msat	(minisat/sattools)	(sattools)
uuf100 (1000 pbs)	0.125	0.012	0.004	0.006
uuf125 (100 pbs)	2.217	0.030	0.006	0.013
uuf150 (100 pbs)	67.563	0.087	0.017	0.045
pigeon/hole6	0.120	0.018	0.006	0.006
pigeon/hole7	4.257	0.213	0.015	0.073
pigeon/hole8	31.450	0.941	0.096	2.488
pigeon/hole9	timeout (600)	8.886	0.634	4.075
pigeon/hole10	timeout (600)	161.478	9.579 (minisat) 160.376 (sattools)	72.050

Appendix B

Excerpts of Code for Proof Strucures

B.1 Proof Terms

Proof terms in ArchSAT are defined with the following OCaml code (extracted from the `src/proof/term.ml` file of the ArchSAT code repository) :

```
type descr = private
  | Type
  (** The Universe at the root of everything *)
  | Id of id
  (** Identifiers (i.e variables, constants) *)
  | App of t * t
  (** Curried appliation *)
  | Let of id * t * t
  (** Local let binding, as (var, expr, body). *)
  | Binder of binder * id * t
  (** Variable binding, without argument/value *)
(** Term descriptors. *)

and t = private {
  ty : t Lazy.t;          (* Type of the term (lazy). *)
  hash : int;             (* Term hash (invariant modulo alpha-renaming) *)
  index : int;            (* Unique integer *)
  term : descr;           (* Term descriptor *)
  reduced : t Lazy.t;     (* Lazy reduced form *)
  free : (id, unit) S.t;  (* Set of free variables, as a substitution *)
}
```

Terms are hashconsed modulo α-renaming to ease with comparison. Types are computed lazily to increase performances in cases where some terms are generated, and then printed, but the type of the term is not used. Similarly, the reduced (or normalized) form of the term is computed lazily, to avoid computing it for large terms that will not be compared to other terms.

B.2 Proof Trees

Proof steps are defined using the following type declarations (extracted from the `src/proof/proof.ml` file) :

```
type sequent = {
  env : Env.t;
  goal : Term.t;
} (** The type of sequents *)

type ('input, 'state) step = {

  (* step name *)
  name : string;
  stat : Stats.t;

  (* Printing information *)
  print : lang ->
    pretty * (Format.formatter -> 'state -> unit);

  (* Semantics *)
  compute   : sequent -> 'input -> 'state * sequent array;
  prelude   : 'state -> Prelude.t list;
  elaborate : 'state -> Term.t array -> Term.t;
} (** The type of proof steps. Proof steps are very generic, and will be
      used to build the proof tree. *)
```

Proof nodes are then defined using the following type declarations (again, extracted from the `src/proof/proof.ml` file) :

```
type node = {
  id : int;
  pos : pos;
  proof : proof_node;
  mutable term : Term.t option;
} (** A node of the proof tree. Identified by a unique id.
      Has a link to its position within the proof, and stores the actual
      reasoning step used in the [proof] field. *)

and pos = {
  i : int;
  t : node array;
  section : Section.t;
} (** Positions within a proof. *)

and proof_node =
  | Open  : sequent -> proof_node
  | Proof : (_, 'state) step * 'state * node array -> proof_node (**)
(** The different reasoning used to build proof nodes. Currently either
    no reasoning is done and the node is open, or a proof step has been applied.
    This application uses a GADT to wrap the existential type of the internal
    state carried by the proof step. *)
```

Lastly, Proof Trees are defined using the following type alias:

```
(* Alias for proof *)
type proof = sequent * node array
(** A proof is basically a pair of a sequent and an array of length 1.
    An array is used rather than a reference to share more code with functions
    that deal with positions. *)
```

The basic function to build a proof tree have the following signatures (extracted from the `src/proof/proof.mli` file) :

```
val mk : sequent -> proof
(** Create an empty proof with the given goal and environent. *)

val root : proof -> node
(** Returns the root of a proof
    @raise Open_proof if there is no step applied to the root of the proof. *)

val pos : node -> pos
(** Return the position of a node. *)

val apply_step : pos -> ('a, 'b) step -> 'a -> 'b * pos array
(** Apply a reasoning step at a position in the proof. Returns the computed
    internal state, as well as an array of positions corresponding to the
    branches to prove. These positions are in the same order as the branches
    computed by the reasoning step. *)
```

B.3 Proof Tactics

A proof tactic in ArchSAT is an OCaml function that takes as argument a proof position. The return type, however, depend on the tactic. Some tactics may close a branch, and thus have the type `pos -> unit`, while some may create exactly one branch and thus have type `pos -> pos`. Some may have even more explicit types, such as `pos -> bool`, in order to indicate whether the branch has been closed for instance. Here are some of the tactics used in ArchSAT, with their respective types:

```
type pos
(** The type of proof positions *)

type ('a, 'b) tactic = 'a -> 'b
(** An alias type to use to identify proof tactics. *)

val exact  : Prelude.t list -> Term.t -> (pos, unit) tactic
val apply1 : Prelude.t list -> Term.t -> (pos, pos) tactic
val apply2 : Prelude.t list -> Term.t -> (pos, pos * pos) tactic
val apply3 : Prelude.t list -> Term.t -> (pos, pos * pos * pos) tactic
(** Fixed arity applications. *)

val split :
  left:((pos, unit) tactic) ->
  right:((pos, unit) tactic) ->
  (pos * pos, unit) tactic
(** Convenience operator for operating on a pair of positions. *)

val trivial : (pos, bool) tactic
(** Try and find the goal in the environment.
    Returns true if the branch has been closed, else returns false. *)

val exfalso : (pos, pos) tactic
(** Apply exfalso, if needed, in order to get a sequent of the form
    Gamm |- False. *)

val absurd : Term.t -> (pos, unit) tactic
(** Given a term [t], find [t] and its negation in the environment
    in order to close the branch, possibly applying exfalso if needed. *)

val or_elim :
  f:(Term.t -> (pos, unit) tactic) ->
  Term.t -> (pos, unit) tactic
(** Eliminate a disjunction present in the environment. *)
```

Appendix C

Dedukti theory files

This Appendix presents the files used by ArchSAT to encode first-order logic in Dedukti[2]. The encoding used is taken from [36], and is relatively straight-forward.

C.1 Calculus of construction encoding

A first file cc.dk encodes the basic calculus of constructions into Dedukti:

```
(; This file is free software, part of Archsat. See file "LICENSE" for more details. ;)

#NAME cc.
(; Calculus of Construction embedded into Lambda-Pi Modulo ;)

uT : Type.
def eT : uT -> Type.

Pi : X : uT -> ((eT X) -> uT) -> uT.
PiT : (uT -> uT) -> uT.

[X : uT, Y : (eT X) -> uT]
    eT (Pi X Y) --> x : (eT X) -> (eT (Y x))
[Y : uT -> uT]
    eT (PiT Y) --> x : uT -> eT (Y x).

def Arrow : uT -> uT -> uT.
[ t1 : uT, t2 : uT ]
    Arrow t1 t2 --> Pi t1 (x : eT t1 => t2).
```

C.2 Polymorphic first-orderlogic encoding

Then in dk_logic.dk, the basis of polymorphic first-order logic are expressed:

```
(; This file is free software, part of Archsat. See file "LICENSE" for more details. ;)

#NAME dk_logic.

(; Impredicative prop ;)
```

[2] Note that the Coq outputs do not need files such as these since there already is an axiomatisation of first-order in its standard library

```

prop : cc.uT.
Prop : Type.
[] cc.eT prop --> Prop.
(; def ebP : cc.eT dk_bool.bool -> Prop. ;)

imp : Prop -> Prop -> Prop.
forall_type : (cc.uT -> Prop) -> Prop.
forall : A : cc.uT -> (cc.eT A -> Prop) -> Prop.

def eeP : Prop -> cc.uT.
def eP : Prop -> Type
   := f : Prop => cc.eT (eeP f).
[ f1 : Prop, f2 : Prop ]
    eeP (imp f1 f2)
      -->
    cc.Arrow (eeP f1) (eeP f2)
[ A : cc.uT, f : cc.eT A -> Prop ]
    eeP (forall A f)
      -->
    cc.Pi A (x : cc.eT A => eeP (f x)).

[ f : cc.uT -> Prop ]
    eeP (forall_type f)
      -->
    cc.PiT (x : cc.uT => eeP (f x)).

def True : Prop := forall prop (P : Prop => imp P P).
def False : Prop := forall prop (P : Prop => P).
def not (f : Prop) : Prop := imp f False.
def and (A : Prop) (B : Prop) : Prop :=
  forall prop (P : Prop => imp (imp A (imp B P)) P).
def or  (A : Prop) (B : Prop) : Prop :=
  forall prop (P : Prop => imp (imp A P) (imp (imp B P) P)).
def eqv (A : Prop) (B : Prop) : Prop :=
  and (imp A B) (imp B A).

def exists (A : cc.uT) (f : cc.eT A -> Prop) : Prop :=
  forall prop (P : Prop => imp (forall A (x : cc.eT A => imp (f x) P)) P).
def forallc (A : cc.uT) (f : cc.eT A -> Prop) : Prop :=
  not (not (forall A (x : cc.eT A => not (not (f x))))).
def existsc (A : cc.uT) (f : cc.eT A -> Prop) : Prop :=
  not (not (exists A (x : cc.eT A => not (not (f x))))).

def exists_type (f : cc.uT -> Prop) : Prop
:= forall prop (z : Prop =>
                (imp (forall_type (a : cc.uT =>
                                   imp (f a) z))
                     z)).


def TrueT : Type := eP True.
def FalseT : Type := eP False.
I : TrueT.
False_elim : A : cc.uT -> FalseT -> cc.eT A.

```

```
(;
def Istrue : dk_bool.Bool -> Type.
[ b : dk_bool.Bool ] Istrue b --> eP (ebP b).
;)

def and_intro (f1 : Prop)
              (f2 : Prop)
              (H1 : eP f1)
              (H2 : eP f2)
              : eP (and f1 f2)
    := f3 : Prop =>
       H3 : (eP f1 -> eP f2 -> eP f3) =>
       H3 H1 H2.

def and_elim1 (f1 : Prop)
              (f2 : Prop)
              (H3 : eP (and f1 f2))
              : eP f1
    := H3 f1 (H1 : eP f1 => H2 : eP f2 => H1).

def and_elim2 (f1 : Prop)
              (f2 : Prop)
              (H3 : eP (and f1 f2))
              : eP f2
    := H3 f2 (H1 : eP f1 => H2 : eP f2 => H2).

def or_intro1 (f1 : Prop)
              (f2 : Prop)
              (H1 : eP f1)
              : eP (or f1 f2)
    := f3 : Prop =>
       H13 : (eP f1 -> eP f3) =>
       H23 : (eP f2 -> eP f3) =>
       H13 H1.

def or_intro2 (f1 : Prop)
              (f2 : Prop)
              (H2 : eP f2)
              : eP (or f1 f2)
    := f3 : Prop =>
       H13 : (eP f1 -> eP f3) =>
       H23 : (eP f2 -> eP f3) =>
       H23 H2.

def or_elim (f1 : Prop)
            (f2 : Prop)
            (f3 : Prop)
            (H3 : eP (or f1 f2))
            (H13 : eP (imp f1 f3))
            (H23 : eP (imp f2 f3))
            : eP f3
    := H3 f3 H13 H23.

def eqv_intro := f1 : Prop =>
                 f2 : Prop =>
                 and_intro (imp f1 f2) (imp f2 f1).
```

```
def eqv_elim1 := f1 : Prop =>
                 f2 : Prop =>
                 and_elim1 (imp f1 f2) (imp f2 f1).
def eqv_elim2 := f1 : Prop =>
                 f2 : Prop =>
                 and_elim2 (imp f1 f2) (imp f2 f1).

(;
[] ebP dk_bool.true --> True
[] ebP dk_bool.false --> False.
;)

(; equality ;)
def equal : A : cc.uT -> x : cc.eT A -> y : cc.eT A -> Prop
      := A : cc.uT => x : cc.eT A => y : cc.eT A =>
                                     forall (cc.Arrow A prop)
                                            (H : (cc.eT A -> Prop) =>
                                             imp (H x) (H y)).

def equalc (A : cc.uT) (x : cc.eT A) (y : cc.eT A) : Prop :=
  not (not (equal A x y)).

def refl : A : cc.uT -> x : cc.eT A -> eP (equal A x x)
     := A : cc.uT => x : cc.eT A =>
                     H : (cc.eT A -> Prop) =>
                     px : eP (H x) => px.

def equal_ind :
            A : cc.uT ->
            H : (cc.eT A -> Prop) ->
            x : cc.eT A ->
            y : cc.eT A ->
            eP (equal A x y) ->
            eP (H x) ->
            eP (H y)
          :=
            A : cc.uT =>
            P : (cc.eT A -> Prop) =>
            x : cc.eT A =>
            y : cc.eT A =>
            eq: eP (equal A x y) =>
            eq P.

def equal_sym : A : cc.uT ->
            x : cc.eT A ->
            y : cc.eT A ->
            eP (equal A x y) ->
            eP (equal A y x)
          :=
            A : cc.uT =>
            x : cc.eT A =>
            y : cc.eT A =>
            eq : eP (equal A x y) =>
            equal_ind
              A
              (z : cc.eT A => equal A z x)
              x
                y
                eq
                (refl A x).

def equal_congr :
  A : cc.uT ->
  B : cc.uT ->
  f : (cc.eT A -> cc.eT B) ->
  x : cc.eT A ->
  y : cc.eT A ->
  eP (equal A x y) ->
  eP (equal B (f x) (f y))
  :=
    A : cc.uT =>
    B : cc.uT =>
    f : (cc.eT A -> cc.eT B) =>
    x : cc.eT A =>
    y : cc.eT A =>
    H : eP (equal A x y) =>
    equal_ind A (z : cc.eT A => equal B (f x) (f z)) x y H (refl B (f x)).
```

These two files are directly taken from the encoding used by Zenon Modulo in [36] and make up the basis of the encoding. However, some of the definitions are not quite practical to use. Thus, in `logic.dk`, some more readable names are given to the encoding functions, and some useful proof wrappers are defined in order to minimize differences with the Coq proofs:

```
(; This file is free software, part of Archsat. See file "LICENSE" for more details. ;)

#NAME logic.

(; Polymorphic First-order logic for Archsat ;)

def prop  : Type.
def type  : Type.
def proof : prop -> Type.
def term  : type -> Type.
def arrow : type -> type -> type.


(; Constant symbols ;)

def True  : prop.
def False : prop.

def not : prop -> prop.
def and : prop -> prop -> prop.
def or  : prop -> prop -> prop.
def imp : prop -> prop -> prop.
def equiv : prop -> prop -> prop.

def forall : a : type -> (term a -> prop) -> prop.
def exists : a : type -> (term a -> prop) -> prop.

def foralltype : (type -> prop) -> prop.
def existstype : (type -> prop) -> prop.

def equal      : a : type -> term a -> term a -> prop.


(; Proofs of these rules in the encoding of the calculus of constructions ;)

[] type  --> cc.uT.
[] term  --> cc.eT.
[] arrow --> cc.Arrow.
[] prop  --> dk_logic.Prop.
[] proof --> dk_logic.eP.

[] True       --> dk_logic.True.
[] False      --> dk_logic.False.
[] not        --> dk_logic.not.
[] and        --> dk_logic.and.
[] or         --> dk_logic.or.
[] imp        --> dk_logic.imp.
[] equiv      --> dk_logic.eqv.
[] forall     --> dk_logic.forall.
[] exists     --> dk_logic.exists.
[] foralltype --> dk_logic.forall_type.
[] existstype --> dk_logic.exists_type.
[] equal      --> dk_logic.equal.


(; True ;)

def true_intro : proof True :=
  p : prop => x : proof p => x.


(; False ;)

def false_elim (p : prop)
  : proof False -> proof p :=
  H : proof False => H p.


(; Conjunction ;)

def and_intro (p : prop) (q: prop)
  : proof p -> proof q -> proof (and p q) := dk_logic.and_intro p q.

def and_ind (p : prop) (q : prop) (r : prop)
  : (proof p -> proof q -> proof r) -> proof (and p q) -> proof r :=
  f : (proof p -> proof q -> proof r) => H : proof (and p q) => H r f.

def and_elim (p : prop) (q : prop) (r : prop)
  : proof (and p q) -> (proof p -> proof q -> proof r) -> proof r :=
  H : proof (and p q) => f : (proof p -> proof q -> proof r) => H r f.


(; Disjunction ;)

```

```
def or_introl (p : prop) (q : prop)
 : proof p -> proof (or p q) :=
 H1 : proof p => z : prop =>
 H2 : (proof p -> proof z) =>
 H3 : (proof q -> proof z) => H2 H1.

def or_intror (p : prop) (q : prop)
 : proof q -> proof (or p q) :=
 H1 : proof q => z : prop =>
 H2 : (proof p -> proof z) =>
 H3 : (proof q -> proof z) => H3 H1.

def or_ind (p : prop) (q: prop) (r: prop)
 : (proof p -> proof r) -> (proof q -> proof r) -> proof (or p q) -> proof r :=
 f : (proof p -> proof r) => g : (proof q -> proof r) => H : proof (or p q) => H r f g.

def or_elim (p : prop) (q: prop) (r: prop)
 : proof (or p q) -> (proof p -> proof r) -> (proof q -> proof r) -> proof r :=
 H : proof (or p q) => f : (proof p -> proof r) => g : (proof q -> proof r) => H r f g.


(; Equivalence ;)

def equiv_refl (p : prop) : proof (equiv p p) :=
 and_intro (imp p p) (imp p p) (x : proof p => x) (x : proof p => x).

def equiv_trans (p : prop) (q : prop) (r : prop)
 : proof (equiv p q) -> proof (equiv q r) -> proof (equiv p r) :=
 H1 : proof (equiv p q) => H2 : proof (equiv q r) =>
 and_intro (imp p r) (imp r p)
  (x : proof p => and_elim (imp p q) (imp q p) r H1
   (pq : proof (imp p q) => _ : proof (imp q p) =>
    and_elim (imp q r) (imp r q) r H2
     (qr : proof (imp q r) => _ : proof (imp r q) =>
      qr (pq x) ) ) )
  (x : proof r => and_elim (imp p q) (imp q p) p H1
   (_ : proof (imp p q) => qp : proof (imp q p) =>
    and_elim (imp q r) (imp r q) p H2
     (_ : proof (imp q r) => rq : proof (imp r q) =>
      qp (rq x) ) ) ).

def equiv_not (p : prop) (q : prop)
 : proof (equiv p q) -> proof (equiv (not p) (not q)) :=
 H : proof (equiv p q) =>
  and_elim (imp p q) (imp q p) (equiv (not p) (not q)) H
   (pq : proof (imp p q) => qp : proof (imp q p) =>
    and_intro (imp (not p) (not q)) (imp (not q) (not p))
     (x : proof (not p) => y : proof q => x (qp y))
     (x : proof (not q) => y : proof p => x (pq y))
   ).


(; Equality ;)

def eq_subst (a : type) (x : term a) (y: term a) (p : term a -> prop)
 : proof (equal a x y) -> proof (p x) -> proof (p y) :=
 H1 : proof (equal a x y) => H2 : proof (p x) =>
 dk_logic.equal_ind a p x y H1 H2.

def eq_refl (a: type) (x: term a) : proof (equal a x x) := dk_logic.refl a x.

def eq_sym (a : type) (x : term a) (y : term a)
 : proof (equal a x y) -> proof (equal a y x) :=
 H1 : proof (equal a x y) => dk_logic.equal_sym a x y H1.

def not_eq_sym (a : type) (x : term a) (y : term a)
 : proof (not (equal a x y)) -> proof (not (equal a y x)) :=
 H1 : proof (not (equal a x y)) => H2 : proof (equal a y x) =>
 H1 (eq_sym a y x H2).

def eq_trans (a : type) (x : term a) (y : term a) (z : term a)
 : proof (equal a x y) -> proof (equal a y z) -> proof (equal a x z) :=
 H1 : proof (equal a x y) => H2 : proof (equal a y z) =>
 eq_subst a y z (s : term a => equal a x s) H2 H1.


(; Functions and equality ;)

def f_equal
 (a : type) (b : type) (f : term a -> term b) (x : term a) (y : term a)
 : proof (equal a x y) -> proof (equal b (f x) (f y)) :=
 H : proof (equal a x y) =>
  eq_subst a x y (z : term a => equal b (f x) (f z)) H (eq_refl b (f x)).

def f_equal2
 (a : type) (b : type) (c : type) (f : term a -> term b -> term c)
 (x1 : term a) (y1 : term a) (x2 : term b) (y2 : term b)
 : proof (equal a x1 y1) -> proof (equal b x2 y2) -> proof (equal c (f x1 x2) (f y1 y2)) :=
 H1 : proof (equal a x1 y1) => H2 : proof (equal b x2 y2) =>
  eq_subst a x1 y1 (z1 : term a => equal c (f x1 x2) (f z1 y2)) H1 (
   eq_subst b x2 y2 (z2 : term b => equal c (f x1 x2) (f x1 z2)) H2 (
    eq_refl c (f x1 x2)
   )
  ).


(; Type inhabitation ;)

def inhabited : type -> prop.

def inhabits : a : type -> term a -> proof (inhabited a).
```

C.3 Classical logic axioms

The classical part of the reasoning is separated in a final file `classical.dk`, so that classical proofs can easily be distinguished from intuitinistic proofs by checking dependencies of proof files:

```
(; This file is free software, part of Archsat. See file "LICENSE" for more details. ;)


(; Law of excluded middle,
   defined directly as the elimination of the (p \/ ~ p) disjunction ;)

classic : p : logic.prop -> z : logic.prop ->
          (logic.proof p -> logic.proof z) ->
          (logic.proof (logic.not p) -> logic.proof z) ->
          logic.proof z.

(; Proof by contradiction using the exlcuded middle ;)

def nnpp (p : logic.prop)
  : logic.proof (logic.not (logic.not p)) -> logic.proof p :=
  H1 : logic.proof (logic.not (logic.not p)) =>
  classic p p (H2 : logic.proof p => H2)
              (H3 : logic.proof (logic.not p) => H1 H3 p).


(; de Morgan Laws for quantifiers ;)

def not_all_not_ex
  (u : logic.type) (p : logic.term u -> logic.prop) :
  logic.proof (logic.not (logic.forall u (x : logic.term u => logic.not (p x)))) ->
  logic.proof (logic.exists u p) :=
    notall : logic.proof (logic.not (logic.forall u (x : logic.term u => logic.not (p x)))) =>
      nnpp (logic.exists u p) (abs : logic.proof (logic.not (logic.exists u p)) =>
        notall (n : logic.term u => H : logic.proof (p n) =>
          abs (z : logic.prop => p0 : (x : logic.term u -> logic.proof (p x) -> logic.proof z) =>
            p0 n H
          )
        )
      ).

def not_all_ex_not
  (u : logic.type) (p : logic.term u -> logic.prop) :
  logic.proof (logic.not (logic.forall u p)) ->
  logic.proof (logic.exists u (x : logic.term u => logic.not (p x))) :=
    notall : logic.proof (logic.not (logic.forall u p)) =>
      not_all_not_ex u (x : logic.term u => logic.not (p x)) (
        (all : logic.proof (logic.forall u (x : logic.term u => logic.not (logic.not (p x)))) =>
          notall (n : logic.term u =>
            nnpp (p n) (all n)
          )
        )
      ).

def not_ex_all_not
  (u : logic.type) (p : logic.term u -> logic.prop) :
  logic.proof (logic.not (logic.exists u p)) ->
  logic.proof (logic.forall u (x : logic.term u => logic.not (p x))) :=
    notex : logic.proof (logic.not (logic.exists u p)) =>
      n : logic.term u => abs : logic.proof (p n) =>
        notex (z : logic.prop =>
          p0 : (x : logic.term u -> logic.proof (p x) -> logic.proof z) =>
            p0 n abs
          ).
```

```
def not_ex_not_all
  (u : logic.type) (p : logic.term u -> logic.prop) :
  logic.proof (logic.not (logic.exists u (x : logic.term u => logic.not (p x)))) ->
  logic.proof (logic.forall u p) :=
    H : logic.proof (logic.not (logic.exists u (x : logic.term u => logic.not (p x)))) =>
      n : logic.term u =>
        nnpp (p n) (k : logic.proof (logic.not (p n)) =>
          H (z : logic.prop =>
            p0 : (x : logic.term u -> logic.proof (logic.not (p x)) -> logic.proof z) =>
              p0 n k
          )
        ).


(; de Morgan Laws for type quantifiers ;)

def not_all_not_ex_type
  (p : logic.type -> logic.prop) :
  logic.proof (logic.not (logic.foralltype (x : logic.type => logic.not (p x)))) ->
  logic.proof (logic.existstype p) :=
    notall : logic.proof (logic.not (logic.foralltype (x : logic.type => logic.not (p x)))) =>
      nnpp (logic.existstype p) (abs : logic.proof (logic.not (logic.existstype p)) =>
        notall (n : logic.type => H : logic.proof (p n) =>
          abs (z : logic.prop => p0 : (x : logic.type -> logic.proof (p x) -> logic.proof z) =>
            p0 n H
          )
        )
      ).

def not_all_ex_not_type
  (p : logic.type -> logic.prop) :
  logic.proof (logic.not (logic.foralltype p)) ->
  logic.proof (logic.existstype (x : logic.type => logic.not (p x))) :=
    notall : logic.proof (logic.not (logic.foralltype p)) =>
      not_all_not_ex_type (x : logic.type => logic.not (p x)) (
        (all : logic.proof (logic.foralltype (x : logic.type => logic.not (logic.not (p x)))) =>
          notall (n : logic.type =>
            nnpp (p n) (all n)
          )
        )
      ).

def not_ex_all_not_type
  (p : logic.type -> logic.prop) :
  logic.proof (logic.not (logic.existstype p)) ->
  logic.proof (logic.foralltype (x : logic.type => logic.not (p x))) :=
    notex : logic.proof (logic.not (logic.existstype p)) =>
      n : logic.type => abs : logic.proof (p n) =>
        notex (z : logic.prop =>
          p0 : (x : logic.type -> logic.proof (p x) -> logic.proof z) =>
            p0 n abs
          ).

def not_ex_not_all_type
  (p : logic.type -> logic.prop) :
  logic.proof (logic.not (logic.existstype (x : logic.type => logic.not (p x)))) ->
  logic.proof (logic.foralltype p) :=
    H : logic.proof (logic.not (logic.existstype (x : logic.type => logic.not (p x)))) =>
      n : logic.type =>
        nnpp (p n) (k : logic.proof (logic.not (p n)) =>
          H (z : logic.prop =>
            p0 : (x : logic.type -> logic.proof (logic.not (p x)) -> logic.proof z) =>
              p0 n k
          )
        ).
```

C.4 Hilbert's epsilons encoding

Finally, definition and specification of epsilon terms (aka Hilbert's choice operator) are defined in a file `epsilon.dk`:

```
(; This file is free software, part of Archsat. See file "LICENSE" for more details. ;)


(; Axiomatisation for Hilbert's epsilon operator ;)

def epsilon :
  a : logic.type ->
  logic.proof (logic.inhabited a) ->
  (logic.term a -> logic.prop) -> logic.term a.

def epsilon_spec :
  a : logic.type ->
  i : logic.proof (logic.inhabited a) ->
  p :(logic.term a -> logic.prop) ->
  logic.proof (logic.exists a p) ->
  logic.proof (p (epsilon a i p)).

(; Axiomatisation for Hilbert's epsilon operator for type existencials ;)

def epsilon_type : (logic.type -> logic.prop) -> logic.type.

def epsilon_type_spec :
  p :(logic.type -> logic.prop) ->
  logic.proof (logic.existstype p) ->
  logic.proof (p (epsilon_type p)).
```

Appendix D

Proofs for a simple SAT problem

D.1 Input problem

Considering the problem first presented in Section 1.1.3, which can be expressed in tptp syntax as:

```
% #expect: unsat

cnf(c1, axiom, p | q).

cnf(c2, axiom, ~ p | r).

cnf(c3, axiom, ~ q | r).

cnf(c4, axiom, s | t).

cnf(c5, axiom, ~ s | ~ r).

cnf(c6, axiom, ~ t | ~ r).

fof(goal, conjecture, $false).
```

We will show on the next pages the different outputs of ArchSAT, namely:

- The Coq proof script output
- The Coq and Dedukti term output generated from the proof script
- The reduced (i.e. normalized) term output for Coq and Dedukti

For each output, the goal is then to prove ⊥. Which, for the different outputs mean:

- For the Coq proof script, proving the theorem `Theorem goal : False.`.
- For the Coq proof terms (including the normalized one), it means providing a term of type ⊥: `Definition goal : False := <term_here>.`
- For the Dedukti proof terms (including the normalized one), it means providing a term which has the type of proofs of the encoded type: `def goal : logic.proof logic.False := <term_here>.`

D.2 Coq headers

We will first show the Coq header defining all parameters, which is common to all three proofs outputs:

```
(**
   Proof automatically generated by Archsat
   Input file: sat_ex1.p
**)

(* Implicitly declared *)
Parameter p : Prop.

(* Implicitly declared *)
Parameter q : Prop.

(* File './sat_ex1.p', line 3, character 15-20 *)
Axiom c1 : (p \/ q).

(* Implicitly declared *)
Parameter r : Prop.

(* File './sat_ex1.p', line 5, character 15-22 *)
Axiom c2 : ((~ p) \/ r).

(* File './sat_ex1.p', line 7, character 15-22 *)
Axiom c3 : ((~ q) \/ r).

(* Implicitly declared *)
Parameter s : Prop.

(* Implicitly declared *)
Parameter t : Prop.

(* File './sat_ex1.p', line 9, character 15-20 *)
Axiom c4 : (s \/ t).

(* File './sat_ex1.p', line 11, character 15-24 *)
Axiom c5 : ((~ s) \/ (~ r)).

(* File './sat_ex1.p', line 13, character 15-24 *)
Axiom c6 : ((~ t) \/ (~ r)).
```

D.3 Coq proof script

We can produce the following proof script for Coq:

```
(**
   Proof automatically generated by Archsat
   Input file: sat_ex1.p
**)

(* Prelude: Alias *)
pose ( or_elim :=
(fun (A B P: Prop) (o: (A \/ B)) (f: (A -> P)) (g: (B -> P)) =>
   (or_ind f g o)) ).

(* PROOF START *)
assert (C0: (~ p -> ~ q -> False)).
{ intro Ax0.
  intro Ax1.
  apply (or_elim p q False c1).
  - intro O0.
    exact (Ax0 O0).
  - intro O0.
    exact (Ax1 O0). }
assert (C1: (~ ~ q -> ~ r -> False)).
{ intro Ax0.
  intro Ax1.
  apply (or_elim (~ q) r False c3).
  - intro O0.
    exact (Ax0 O0).
  - intro O0.
    exact (Ax1 O0). }
assert (C2: (~ s -> ~ t -> False)).
{ intro Ax0.
  intro Ax1.
  apply (or_elim s t False c4).
  - intro O0.
    exact (Ax0 O0).
  - intro O0.
    exact (Ax1 O0). }
assert (C3: (~ ~ r -> ~ ~ s -> False)).
{ intro Ax0.
  intro Ax1.
  apply (or_elim (~ s) (~ r) False c5).
  - intro O0.
    exact (Ax1 O0).
  - intro O0.
    exact (Ax0 O0). }
(* Resolution C2/C3 -> R0 *)
  pose proof (
    (fun (l0: ~ ~ r) (l1: ~ t) =>
       (C2 (fun (r0: s) => (C3 l0 (fun (r1: ~ s) => (r1 r0)))) l1))
  ) as R0.
assert (C4: (~ ~ r -> ~ ~ t -> False)).
{ intro Ax0.
  intro Ax1.
  apply (or_elim (~ t) (~ r) False c6).
  - intro O0.
    exact (Ax1 O0).
  - intro O0.
    exact (Ax0 O0). }
(* Resolution R0/C4 -> R1 *)
  pose proof (
    (fun (l0: ~ ~ r) =>
       (R0 l0 (fun (r0: t) => (C4 l0 (fun (r1: ~ t) => (r1 r0))))))
  ) as R1.
(* Resolution C1/R1 -> R2 *)
  pose proof (
    (fun (l0: ~ ~ q) =>
       (C1 l0 (fun (r0: r) => (R1 (fun (r1: ~ r) => (r1 r0))))))
  ) as R2.
(* Resolution C0/R2 -> R3 *)
  pose proof (
    (fun (l0: ~ p) =>
       (C0 l0 (fun (r0: q) => (R2 (fun (r1: ~ q) => (r1 r0))))))
  ) as R3.
assert (C5: (~ ~ p -> ~ r -> False)).
{ intro Ax0.
  intro Ax1.
  apply (or_elim (~ p) r False c2).
  - intro O0.
    exact (Ax0 O0).
  - intro O0.
    exact (Ax1 O0). }
(* Resolution C5/R1 -> R4 *)
  pose proof (
    (fun (l0: ~ ~ p) =>
       (C5 l0 (fun (r0: r) => (R1 (fun (r1: ~ r) => (r1 r0))))))
  ) as R4.
(* Resolution R3/R4 -> R5 *)
  pose proof ((R3 (fun (r0: p) => (R4 R3)))) as R5.
exact R5.
(* PROOF END *)
```

D.4 Coq proof terms

As well as the proof term:

```
(**
   Proof automatically generated by Archsat
   Input file: sat_ex1.p
**)

(* PROOF START *)
let C0 := (fun (Ax0: ~ p) (Ax1: ~ q) =>
             (or_elim p q False c1 (fun (O0: p) => (Ax0 O0))
             (fun (O0: q) => (Ax1 O0))))
in
let C1 := (fun (Ax0: ~ ~ q) (Ax1: ~ r) =>
             (or_elim (~ q) r False c3 (fun (O0: ~ q) => (Ax0 O0))
             (fun (O0: r) => (Ax1 O0))))
in
let C2 := (fun (Ax0: ~ s) (Ax1: ~ t) =>
             (or_elim s t False c4 (fun (O0: s) => (Ax0 O0))
             (fun (O0: t) => (Ax1 O0))))
in
let C3 := (fun (Ax0: ~ ~ r) (Ax1: ~ ~ s) =>
             (or_elim (~ s) (~ r) False c5 (fun (O0: ~ s) => (Ax1 O0))
             (fun (O0: ~ r) => (Ax0 O0))))
in
let R0 := (fun (l0: ~ ~ r) (l1: ~ t) =>
             (C2 (fun (r0: s) => (C3 l0 (fun (r1: ~ s) => (r1 r0)))) l1))
in
let C4 := (fun (Ax0: ~ ~ r) (Ax1: ~ ~ t) =>
             (or_elim (~ t) (~ r) False c6 (fun (O0: ~ t) => (Ax1 O0))
             (fun (O0: ~ r) => (Ax0 O0))))
in
let R1 := (fun (l0: ~ ~ r) =>
             (R0 l0 (fun (r0: t) => (C4 l0 (fun (r1: ~ t) => (r1 r0))))))
in
let R2 := (fun (l0: ~ ~ q) =>
             (C1 l0 (fun (r0: r) => (R1 (fun (r1: ~ r) => (r1 r0))))))
in
let R3 := (fun (l0: ~ p) =>
             (C0 l0 (fun (r0: q) => (R2 (fun (r1: ~ q) => (r1 r0))))))
in
let C5 := (fun (Ax0: ~ ~ p) (Ax1: ~ r) =>
             (or_elim (~ p) r False c2 (fun (O0: ~ p) => (Ax0 O0))
             (fun (O0: r) => (Ax1 O0))))
in
let R4 := (fun (l0: ~ ~ p) =>
             (C5 l0 (fun (r0: r) => (R1 (fun (r1: ~ r) => (r1 r0))))))
in
let R5 := (R3 (fun (r0: p) => (R4 R3))) in
R5
(* PROOF END *)
```

And finally the reduced proof term:

```
(**
   Proof automatically generated by Archsat
   Input file: sat_ex1.p
**)

(* PROOF START *)
(or_ind (fun (r0: p) =>
           (or_ind (fun (l0: ~ p) =>
                      (or_ind (fun (O0: p) => (l0 O0))
                      (fun (r0_773: q) =>
                         (or_ind (fun (r1: ~ q) => (r1 r0_773))
                         (fun (r0_763: r) =>
                            (or_ind (fun (r0_750: s) =>
                                       (or_ind (fun (r1: ~ s) => (r1 r0_750))
                                       (fun (r1: ~ r) => (r1 r0_763)) c5))
                            (fun (r0_730: t) =>
                               (or_ind (fun (r1: ~ t) => (r1 r0_730))
                               (fun (r1: ~ r) => (r1 r0_763)) c6)) c4)) c3))
                      c1))
           (fun (r0_763: r) =>
              (or_ind (fun (r0_750: s) =>
                         (or_ind (fun (r1: ~ s) => (r1 r0_750))
                         (fun (r1: ~ r) => (r1 r0_763)) c5))
              (fun (r0_730: t) =>
                 (or_ind (fun (r1: ~ t) => (r1 r0_730))
                 (fun (r1: ~ r) => (r1 r0_763)) c6)) c4)) c2))
(fun (r0_773: q) =>
   (or_ind (fun (r1: ~ q) => (r1 r0_773))
   (fun (r0_763: r) =>
      (or_ind (fun (r0_750: s) =>
                 (or_ind (fun (r1: ~ s) => (r1 r0_750))
                 (fun (r1: ~ r) => (r1 r0_763)) c5))
      (fun (r0_730: t) =>
         (or_ind (fun (r1: ~ t) => (r1 r0_730))
         (fun (r1: ~ r) => (r1 r0_763)) c6)) c4)) c3)) c1)
(* PROOF END *)
```

D.5 Dedukti headers

This is the Dedukti header generated for the term and normalized term outputs:

```
(;
  Proof automatically generated by Archsat
  Input file: sat_ex1.p
;)

(; Implicitly declared ;)
p: logic.prop.

(; Implicitly declared ;)
q: logic.prop.

(; File './sat_ex1.p', line 3, character 15-20 ;)
c1: logic.proof (logic.or p q).

(; Implicitly declared ;)
r: logic.prop.

(; File './sat_ex1.p', line 5, character 15-22 ;)
c2: logic.proof (logic.or (logic.not p) r).

(; File './sat_ex1.p', line 7, character 15-22 ;)
c3: logic.proof (logic.or (logic.not q) r).

(; Implicitly declared ;)
s: logic.prop.

(; Implicitly declared ;)
t: logic.prop.

(; File './sat_ex1.p', line 9, character 15-20 ;)
c4: logic.proof (logic.or s t).

(; File './sat_ex1.p', line 11, character 15-24 ;)
c5: logic.proof (logic.or (logic.not s) (logic.not r)).

(; File './sat_ex1.p', line 13, character 15-24 ;)
c6: logic.proof (logic.or (logic.not t) (logic.not r)).
```

D.6 Dedukti proof terms

The Dedukti proof term:

```
(;
  Proof automatically generated by Archsat
  Input file: sat_ex1.p
;)

(; PROOF START ;)
(C0: logic.proof (logic.imp (logic.not p) (logic.not (logic.not q))) =>
(C1: logic.proof (logic.imp (logic.not (logic.not q))
                 (logic.not (logic.not r))) =>
(C2: logic.proof (logic.imp (logic.not s) (logic.not (logic.not t))) =>
(C3: logic.proof (logic.imp (logic.not (logic.not r))
                 (logic.not (logic.not (logic.not s)))) =>
(R0: logic.proof (logic.imp (logic.not (logic.not r))
                 (logic.not (logic.not t))) =>
(C4: logic.proof (logic.imp (logic.not (logic.not r))
                 (logic.not (logic.not (logic.not t)))) =>
(R1: logic.proof (logic.not (logic.not (logic.not r))) =>
(R2: logic.proof (logic.not (logic.not (logic.not q))) =>
(R3: logic.proof (logic.not (logic.not p)) =>
(C5: logic.proof (logic.imp (logic.not (logic.not p))
                 (logic.not (logic.not r))) =>
(R4: logic.proof (logic.not (logic.not (logic.not p))) =>
(R5: logic.proof logic.False => R5 ) (R3 (r0: logic.proof p => (R4 R3)))
) (l0: logic.proof (logic.not (logic.not p)) =>
  (C5 l0
  (r0: logic.proof r => (R1 (r1: logic.proof (logic.not r) => (r1 r0))))))
) (Ax0: logic.proof (logic.not (logic.not p)) =>
   Ax1: logic.proof (logic.not r) =>
  (logic.or_elim (logic.not p) r logic.False c2
  (O0: logic.proof (logic.not p) => (Ax0 O0))
  (O0: logic.proof r => (Ax1 O0))))
) (l0: logic.proof (logic.not p) =>
  (C0 l0
  (r0: logic.proof q => (R2 (r1: logic.proof (logic.not q) => (r1 r0))))))
) (l0: logic.proof (logic.not (logic.not q)) =>
  (C1 l0
  (r0: logic.proof r => (R1 (r1: logic.proof (logic.not r) => (r1 r0))))))
) (l0: logic.proof (logic.not (logic.not r)) =>
  (R0 l0
  (r0: logic.proof t => (C4 l0 (r1: logic.proof (logic.not t) => (r1 r0))))))
) (Ax0: logic.proof (logic.not (logic.not r)) =>
   Ax1: logic.proof (logic.not (logic.not t)) =>
  (logic.or_elim (logic.not t) (logic.not r) logic.False c6
  (O0: logic.proof (logic.not t) => (Ax1 O0))
  (O0: logic.proof (logic.not r) => (Ax0 O0))))
) (l0: logic.proof (logic.not (logic.not r)) => l1: logic.proof (logic.not t) =>
  (C2 (r0: logic.proof s =>
      (C3 l0 (r1: logic.proof (logic.not s) => (r1 r0)))) l1))
) (Ax0: logic.proof (logic.not (logic.not r)) =>
   Ax1: logic.proof (logic.not (logic.not s)) =>
  (logic.or_elim (logic.not s) (logic.not r) logic.False c5
  (O0: logic.proof (logic.not s) => (Ax1 O0))
  (O0: logic.proof (logic.not r) => (Ax0 O0))))
) (Ax0: logic.proof (logic.not s) => Ax1: logic.proof (logic.not t) =>
  (logic.or_elim s t logic.False c4 (O0: logic.proof s => (Ax0 O0))
  (O0: logic.proof t => (Ax1 O0))))
) (Ax0: logic.proof (logic.not (logic.not q)) =>
   Ax1: logic.proof (logic.not r) =>
  (logic.or_elim (logic.not q) r logic.False c3
  (O0: logic.proof (logic.not q) => (Ax0 O0))
  (O0: logic.proof r => (Ax1 O0))))
) (Ax0: logic.proof (logic.not p) => Ax1: logic.proof (logic.not q) =>
  (logic.or_elim p q logic.False c1 (O0: logic.proof p => (Ax0 O0))
  (O0: logic.proof q => (Ax1 O0))))
(; PROOF END ;)
```

And the normalized Dedukti term:

```
(;
  Proof automatically generated by Archsat
  Input file: sat_ex1.p
;)

(; PROOF START ;)
(logic.or_ind p q logic.False
(r0: logic.proof p =>
(logic.or_ind (logic.not p) r logic.False
(l0: logic.proof (logic.not p) =>
(logic.or_ind p q logic.False (O0: logic.proof p => (l0 O0))
(r0_773: logic.proof q =>
(logic.or_ind (logic.not q) r logic.False
(r1: logic.proof (logic.not q) => (r1 r0_773))
(r0_763: logic.proof r =>
(logic.or_ind s t logic.False
(r0_750: logic.proof s =>
(logic.or_ind (logic.not s) (logic.not r) logic.False
(r1: logic.proof (logic.not s) => (r1 r0_750))
(r1: logic.proof (logic.not r) => (r1 r0_763)) c5))
(r0_730: logic.proof t =>
(logic.or_ind (logic.not t) (logic.not r) logic.False
(r1: logic.proof (logic.not t) => (r1 r0_730))
(r1: logic.proof (logic.not r) => (r1 r0_763)) c6)) c4)) c3)) c1))
(r0_763: logic.proof r =>
(logic.or_ind s t logic.False
(r0_750: logic.proof s =>
(logic.or_ind (logic.not s) (logic.not r) logic.False
(r1: logic.proof (logic.not s) => (r1 r0_750))
(r1: logic.proof (logic.not r) => (r1 r0_763)) c5))
(r0_730: logic.proof t =>
(logic.or_ind (logic.not t) (logic.not r) logic.False
(r1: logic.proof (logic.not t) => (r1 r0_730))
(r1: logic.proof (logic.not r) => (r1 r0_763)) c6)) c4)) c2))
(r0_773: logic.proof q =>
(logic.or_ind (logic.not q) r logic.False
(r1: logic.proof (logic.not q) => (r1 r0_773))
(r0_763: logic.proof r =>
(logic.or_ind s t logic.False
(r0_750: logic.proof s =>
(logic.or_ind (logic.not s) (logic.not r) logic.False
(r1: logic.proof (logic.not s) => (r1 r0_750))
(r1: logic.proof (logic.not r) => (r1 r0_763)) c5))
(r0_730: logic.proof t =>
(logic.or_ind (logic.not t) (logic.not r) logic.False
(r1: logic.proof (logic.not t) => (r1 r0_730))
(r1: logic.proof (logic.not r) => (r1 r0_763)) c6)) c4)) c3)) c1)
(; PROOF END ;)
```

Appendix E

Proofs for a simple SMT problem

E.1 Input problem

Considering the following simple SMT problem (which is already in clausal normal form):

```
% #expect: unsat

cnf(c1, axiom, a = b).

cnf(c2, axiom, b = c | b = d).

cnf(c3, axiom, ~ a = d).

cnf(c4, axiom, ~ a = c).

fof(goal, conjecture, $false).
```

We will show on the next pages the different outputs of ArchSAT, namely:

- The Coq proof script output
- The Coq and Dedukti term output generated from the proof script
- The reduced (i.e. normalized) term output for Coq and Dedukti

For each output, the goal is then to prove $\bot$. Which, for the different outputs mean:

- For the Coq proof script, proving the theorem `Theorem goal : False.`.
- For the Coq proof terms (including the normalized one), it means providing a term of type $\bot$: `Definition goal : False := <term_here>.`
- For the Dedukti proof terms (including the normalized one), it means providing a term which has the type of proofs of the encoded type: `def goal : logic.proof logic.False := <term_here>.`

E.2 Coq headers

We will first show the Coq header defining all parameters, which is common to all three proofs outputs:

```
(**
   Proof automatically generated by Archsat
   Input file: smt_ex1.p
**)

(* Implicitly declared *)
Parameter _i : Type.

(* Implicitly declared *)
Parameter a : _i.

(* Implicitly declared *)
Parameter b : _i.

(* File './smt_ex1.p', line 3, character 15-20 *)
Axiom c1 : (a = b).

(* Implicitly declared *)
Parameter c : _i.

(* Implicitly declared *)
Parameter d : _i.

(* File './smt_ex1.p', line 5, character 15-28 *)
Axiom c2 : ((b = c) \/ (b = d)).

(* File './smt_ex1.p', line 7, character 15-22 *)
Axiom c3 : ~ (a = d).

(* File './smt_ex1.p', line 9, character 15-22 *)
Axiom c4 : ~ (a = c).
```

E.3 Coq proof script

We can produce the following proof script for Coq:

```
(**
   Proof automatically generated by Archsat
   Input file: smt_ex1.p
**)

(* Prelude: Alias *)
pose ( or_elim :=
(fun (A B P: Prop) (o: (A \/ B)) (f: (A -> P)) (g: (B -> P)) =>
   (or_ind f g o)) ).

(* PROOF START *)
assert (L0: (~ ~ (a = b) -> ~ ~ (b = c) -> ~ (a = c) -> False)).
{ intro E0.
  intro E1.
  intro E2.
  apply E1.
  intro E3.
  apply E2.
  exact (eq_trans c1 E3). }
assert (C0: ~ ~ ~ (a = c)).
{ intro Ax0.
  exact (Ax0 c4). }
(* Resolution L0/C0 -> R0 *)
  pose proof (
    (fun (l0: ~ ~ (a = b)) (l1: ~ ~ (b = c)) =>
       (L0 l0 l1
       (fun (r0: (a = c)) => (C0 (fun (r1: ~ (a = c)) => (r1 r0))))))
  ) as R0.
assert (C1: ~ ~ (a = b)).
{ intro Ax0.
  exact (Ax0 c1). }
(* Resolution R0/C1 -> R1 *)
  pose proof ((fun (l0: ~ ~ (b = c)) => (R0 C1 l0))) as R1.
assert (C2: (~ (b = c) -> ~ (b = d) -> False)).
{ intro Ax0.
  intro Ax1.
  apply (or_elim (b = c) (b = d) False c2).
  - intro O0.
    exact (Ax0 O0).
  - intro O0.
    exact (Ax1 O0). }
assert (L1: (~ ~ (a = b) -> ~ ~ (b = d) -> ~ (a = d) -> False)).
{ intro E0.
  intro E1.
  intro E2.
  apply E1.
  intro E3.
  apply E2.
  exact (eq_trans c1 E3). }
assert (C3: ~ ~ ~ (a = d)).
{ intro Ax0.
  exact (Ax0 c3). }
```

```
(* Resolution L1/C3 -> R2 *)
  pose proof (
    (fun (l0: ~ ~ (a = b)) (l1: ~ ~ (b = d)) =>
       (L1 l0 l1
       (fun (r0: (a = d)) => (C3 (fun (r1: ~ (a = d)) => (r1 r0))))))
  ) as R2.
(* Resolution R2/C1 -> R3 *)
  pose proof ((fun (l0: ~ ~ (b = d)) => (R2 C1 l0))) as R3.
(* Resolution C2/R3 -> R4 *)
  pose proof (
    (fun (l0: ~ (b = c)) =>
       (C2 l0 (fun (r0: (b = d)) => (R3 (fun (r1: ~ (b = d)) => (r1 r0))))))
  ) as R4.
(* Resolution R1/R4 -> R5 *)
  pose proof ((R1 R4)) as R5.
exact R5.
(* PROOF END *)
```

E.4 Coq proof terms

As well as the proof term:

```
(**
   Proof automatically generated by Archsat
   Input file: smt_ex1.p
**)

(* PROOF START *)
let L0 := (fun (E0: ~ ~ (a = b)) (E1: ~ ~ (b = c)) (E2: ~ (a = c)) =>
             (E1 (fun (E3: (b = c)) => (E2 (eq_trans c1 E3)))))
in
let C0 := (fun (Ax0: ~ ~ (a = c)) => (Ax0 c4)) in
let R0 := (fun (l0: ~ ~ (a = b)) (l1: ~ ~ (b = c)) =>
             (L0 l0 l1
             (fun (r0: (a = c)) => (C0 (fun (r1: ~ (a = c)) => (r1 r0))))))
in
let C1 := (fun (Ax0: ~ (a = b)) => (Ax0 c1)) in
let R1 := (fun (l0: ~ ~ (b = c)) => (R0 C1 l0)) in
let C2 := (fun (Ax0: ~ (b = c)) (Ax1: ~ (b = d)) =>
             (or_elim (b = c) (b = d) False c2
             (fun (O0: (b = c)) => (Ax0 O0)) (fun (O0: (b = d)) => (Ax1 O0))))
in
let L1 := (fun (E0: ~ ~ (a = b)) (E1: ~ ~ (b = d)) (E2: ~ (a = d)) =>
             (E1 (fun (E3: (b = d)) => (E2 (eq_trans c1 E3)))))
in
let C3 := (fun (Ax0: ~ ~ (a = d)) => (Ax0 c3)) in
let R2 := (fun (l0: ~ ~ (a = b)) (l1: ~ ~ (b = d)) =>
             (L1 l0 l1
             (fun (r0: (a = d)) => (C3 (fun (r1: ~ (a = d)) => (r1 r0))))))
in
let R3 := (fun (l0: ~ ~ (b = d)) => (R2 C1 l0)) in
let R4 := (fun (l0: ~ (b = c)) =>
             (C2 l0
             (fun (r0: (b = d)) => (R3 (fun (r1: ~ (b = d)) => (r1 r0))))))
in
let R5 := (R1 R4) in
R5
(* PROOF END *)
```

And finally the reduced proof term:

```
(**
   Proof automatically generated by Archsat
   Input file: smt_ex1.p
**)

(* PROOF START *)
(or_ind (fun (E3: (b = c)) => (c4 (eq_trans c1 E3)))
(fun (r0: (b = d)) => (c3 (eq_trans c1 r0))) c2)
(* PROOF END *)
```

E.5 Dedukti headers

This is the Dedukti header generated for the term and normalized term outputs:

```
(;
  Proof automatically generated by Archsat
  Input file: smt_ex1.p
;)

(; Implicitly declared ;)
_i: logic.type.

(; Implicitly declared ;)
a: logic.term _i.

(; Implicitly declared ;)
b: logic.term _i.

(; File './smt_ex1.p', line 3, character 15-20 ;)
c1: logic.proof (logic.equal _i a b).

(; Implicitly declared ;)
c: logic.term _i.

(; Implicitly declared ;)
d: logic.term _i.

(; File './smt_ex1.p', line 5, character 15-28 ;)
c2: logic.proof (logic.or (logic.equal _i b c) (logic.equal _i b d)).

(; File './smt_ex1.p', line 7, character 15-22 ;)
c3: logic.proof (logic.not (logic.equal _i a d)).

(; File './smt_ex1.p', line 9, character 15-22 ;)
c4: logic.proof (logic.not (logic.equal _i a c)).
```

E.6 Dedukti proof terms

The Dedukti proof term:

```
(;
  Proof automatically generated by Archsat
  Input file: smt_ex1.p
;)

(; PROOF START ;)
(L0: logic.proof (logic.imp (logic.not (logic.not (logic.equal _i a b)))
                 (logic.imp (logic.not (logic.not (logic.equal _i b c)))
                 (logic.not (logic.not (logic.equal _i a c))))) =>
(C0: logic.proof (logic.not (logic.not (logic.not (logic.equal _i a c)))) =>
(R0: logic.proof (logic.imp (logic.not (logic.not (logic.equal _i a b)))
                 (logic.not (logic.not (logic.not (logic.equal _i b c))))) =>
(C1: logic.proof (logic.not (logic.not (logic.equal _i a b))) =>
(R1: logic.proof (logic.not (logic.not (logic.not (logic.equal _i b c)))) =>
(C2: logic.proof (logic.imp (logic.not (logic.equal _i b c))
                 (logic.not (logic.not (logic.equal _i b d)))) =>
(L1: logic.proof (logic.imp (logic.not (logic.not (logic.equal _i a b)))
                 (logic.imp (logic.not (logic.not (logic.equal _i b d)))
                 (logic.not (logic.not (logic.equal _i a d))))) =>
(C3: logic.proof (logic.not (logic.not (logic.not (logic.equal _i a d)))) =>
(R2: logic.proof (logic.imp (logic.not (logic.not (logic.equal _i a b)))
                 (logic.not (logic.not (logic.not (logic.equal _i b d))))) =>
(R3: logic.proof (logic.not (logic.not (logic.not (logic.equal _i b d)))) =>
(R4: logic.proof (logic.not (logic.not (logic.equal _i b c))) =>
(R5: logic.proof logic.False => R5 ) (R1 R4)
) (l0: logic.proof (logic.not (logic.equal _i b c)) =>
  (C2 l0
  (r0: logic.proof (logic.equal _i b d) =>
  (R3 (r1: logic.proof (logic.not (logic.equal _i b d)) => (r1 r0))))))
) (l0: logic.proof (logic.not (logic.not (logic.equal _i b d))) =>
  (R2 C1 l0))
) (l0: logic.proof (logic.not (logic.not (logic.equal _i a b))) =>
   l1: logic.proof (logic.not (logic.not (logic.equal _i b d))) =>
  (L1 l0 l1
  (r0: logic.proof (logic.equal _i a d) =>
  (C3 (r1: logic.proof (logic.not (logic.equal _i a d)) => (r1 r0))))))
) (Ax0: logic.proof (logic.not (logic.not (logic.equal _i a d))) => (Ax0 c3))
) (E0: logic.proof (logic.not (logic.not (logic.equal _i a b))) =>
   E1: logic.proof (logic.not (logic.not (logic.equal _i b d))) =>
   E2: logic.proof (logic.not (logic.equal _i a d)) =>
  (E1 (E3: logic.proof (logic.equal _i b d) =>
      (E2 (logic.eq_trans _i a b d c1 E3)))))
) (Ax0: logic.proof (logic.not (logic.equal _i b c)) =>
   Ax1: logic.proof (logic.not (logic.equal _i b d)) =>
  (logic.or_elim (logic.equal _i b c) (logic.equal _i b d) logic.False c2
  (O0: logic.proof (logic.equal _i b c) => (Ax0 O0))
  (O0: logic.proof (logic.equal _i b d) => (Ax1 O0))))
) (l0: logic.proof (logic.not (logic.not (logic.equal _i b c))) =>
  (R0 C1 l0))
) (Ax0: logic.proof (logic.not (logic.equal _i a b)) => (Ax0 c1))
) (l0: logic.proof (logic.not (logic.not (logic.equal _i a b))) =>
   l1: logic.proof (logic.not (logic.not (logic.equal _i b c))) =>
  (L0 l0 l1
  (r0: logic.proof (logic.equal _i a c) =>
  (C0 (r1: logic.proof (logic.not (logic.equal _i a c)) => (r1 r0))))))
) (Ax0: logic.proof (logic.not (logic.not (logic.equal _i a c))) => (Ax0 c4))
) (E0: logic.proof (logic.not (logic.not (logic.equal _i a b))) =>
   E1: logic.proof (logic.not (logic.not (logic.equal _i b c))) =>
   E2: logic.proof (logic.not (logic.equal _i a c)) =>
  (E1 (E3: logic.proof (logic.equal _i b c) =>
      (E2 (logic.eq_trans _i a b c c1 E3)))))
(; PROOF END ;)
```

And the normalized Dedukti term:

```
(;
  Proof automatically generated by Archsat
  Input file: smt_ex1.p
;)

(; PROOF START ;)
(logic.or_ind (logic.equal _i b c) (logic.equal _i b d) logic.False
(E3: logic.proof (logic.equal _i b c) =>
(c4 (logic.eq_trans _i a b c c1 E3)))
(r0: logic.proof (logic.equal _i b d) =>
(c3 (logic.eq_trans _i a b d c1 r0))) c2)
(; PROOF END ;)
```

Appendix F

Proofs for a problem using rewriting

F.1 Input problem

Consider the following problem on polymorphic lists:

```
tff(list_type, type, ( list : $tType > $tType ) ).

tff(cons_type, type, ( cons : !> [A : $tType ] : ( ( A * list(A) ) > list(A) ) ) ).

tff(nil_type, type, ( nil : !> [A : $tType ] : list(A) ) ).

tff(car_type, type, ( car : !> [A : $tType ] : ( list(A) > A ) ) ).

tff(cdr_type, type, ( cdr : !> [A: $tType ] : ( list(A) > list(A) ) ) ).

tff(car_axiom, axiom,
    (! [A : $tType, X: A , L: list(A) ] : (car(A, cons(A, X, L)) = X))).

tff(cons_axiom, axiom,
    (! [A : $tType, X: A , L: list(A) ] : (cdr(A, cons(A, X, L)) = L))).

tff(hyp_3, conjecture,
    (! [V_x1: $i , V_y1: list($i) , V_x2: $i , V_y2: list($i) ] :
       ((cons($i, V_x1, V_y1) = cons($i, V_x2, V_y2))
       => ((V_x1 = V_x2) & (V_y1 = V_y2))))).
```

The goal is to show that equality of two non-empty lists implies equality of the heads and tails of the lists. Compared to the examples in Appendix D and E, this problem has en explicit goal (instead of trying to prove $\bot$ from a set of axioms). The proofs will thus aim at proving the goal, and will all be proof by contradiction. The next pages show the term output for Coq (proof outputs are rather big, the interested reader can download and build the ArchSAT tool[2] and inspect generated proofs).

[2] Available at: https://github.com/Gbury/archsat

F.2 Coq proof script

```
(**
   Proof automatically generated by Archsat
   Input file: rewr_ex1.p
**)

(* File './rewr_ex1.p', line 2, character 0-50 *)
Parameter list : (forall (_: Type), Type).

(* File './rewr_ex1.p', line 4, character 0-83 *)
Parameter cons : (forall (A: Type), (A -> (list A) -> (list A))).

(* File './rewr_ex1.p', line 6, character 0-59 *)
Parameter nil : (forall (A: Type), (list A)).

(* File './rewr_ex1.p', line 8, character 0-67 *)
Parameter car : (forall (A: Type), ((list A) -> A)).

(* File './rewr_ex1.p', line 10, character 0-72 *)
Parameter cdr : (forall (A: Type), ((list A) -> (list A))).

(* File './rewr_ex1.p', line 12, character 0 to line 13, character 72 *)
Axiom car_axiom : (forall (A: Type),
                     (forall (X: A) (L: (list A)), ((car A (cons A X L)) = X))).

(* File './rewr_ex1.p', line 15, character 0 to line 16, character 72 *)
Axiom cons_axiom : (forall (A: Type),
                      (forall (X: A) (L: (list A)),
                         ((cdr A (cons A X L)) = L))).

(* Implicitly declared *)
Parameter _i : Type.

(* Prelude: Module import *)
Require Import Coq.Logic.Classical.

(* Prelude: Module import *)
Require Import Coq.Logic.Epsilon.

(* Prelude: Alias *)
Definition and_elim :
(forall (A B P: Prop), ((A /\ B) -> (A -> B -> P) -> P)) :=
(fun (A B P: Prop) (o: (A /\ B)) (f: (A -> B -> P)) => (and_ind f o)).

(* Prelude: Alias *)
Definition eq_subst :
(forall (A: Type),
   (forall (x y: A) (P: (A -> Prop)), ((x = y) -> (P x) -> (P y)))) :=
(fun (A: Type) (x y: A) (P: (A -> Prop)) (e: (x = y)) (proof: (P x)) =>
   (eq_ind x P proof y e)).

(* Prelude: Alias *)
Definition e_1 :
_i :=
(epsilon (inhabits (car _i (nil _i)))
(fun (V_x1: _i) =>
   ~ (forall (V_y1: (list _i)) (V_x2: _i) (V_y2: (list _i)),
        (((cons _i V_x1 V_y1) = (cons _i V_x2 V_y2))
        -> ((V_x1 = V_x2) /\ (V_y1 = V_y2)))))).

(* Prelude: Alias *)
Definition e_2 :
(list _i) :=
(epsilon (inhabits (nil _i))
(fun (V_y1: (list _i)) =>
   ~ (forall (V_x2: _i) (V_y2: (list _i)),
        (((cons _i e_1 V_y1) = (cons _i V_x2 V_y2))
        -> ((e_1 = V_x2) /\ (V_y1 = V_y2)))))).

(* Prelude: Alias *)
Definition e_3 :
_i :=
(epsilon (inhabits (car _i (nil _i)))
(fun (V_x2: _i) =>
   ~ (forall (V_y2: (list _i)),
        (((cons _i e_1 e_2) = (cons _i V_x2 V_y2))
        -> ((e_1 = V_x2) /\ (e_2 = V_y2)))))).

(* Prelude: Alias *)
Definition e_4 :
(list _i) :=
(epsilon (inhabits (nil _i))
(fun (V_y2: (list _i)) =>
   ~ (((cons _i e_1 e_2) = (cons _i e_3 V_y2))
     -> ((e_1 = e_3) /\ (e_2 = V_y2))))).


(* File './rewr_ex1.p', line 18, character 0 to line 21, character 45 *)
Definition
  hyp_3 :
  (forall (V_x1: _i) (V_y1: (list _i)) (V_x2: _i) (V_y2: (list _i)),
     (((cons _i V_x1 V_y1) = (cons _i V_x2 V_y2))
     -> ((V_x1 = V_x2) /\ (V_y1 = V_y2))))
  :=
  (* PROOF START *)
  (NNPP (forall (V_x1: _i) (V_y1: (list _i)) (V_x2: _i) (V_y2: (list _i)),
           (((cons _i V_x1 V_y1) = (cons _i V_x2 V_y2))
           -> ((V_x1 = V_x2) /\ (V_y1 = V_y2))))
  (fun (G0:
       ~ (forall (V_x1: _i) (V_y1_503: (list _i)) (V_x2_501: _i) (V_y2_499:
                 (list _i)),
            (((cons _i V_x1 V_y1_503) = (cons _i V_x2_501 V_y2_499)) ->
             ((V_x1 = V_x2_501) /\ (V_y1_503 = V_y2_499))))) =>
     let L0 := (fun (E0: ~ (e_2 = e_4)) (E1:
                    ~ ~ ((cdr _i (cons _i e_3 e_4)) = e_4)) (E2:
                    ~ ~ ((cdr _i (cons _i e_1 e_2)) = e_2)) (E3:
                    ~ ~ ((cdr _i (cons _i e_1 e_2))
                        = (cdr _i (cons _i e_3 e_4)))) =>
                  (E3 (fun (E4:
                           ((cdr _i (cons _i e_1 e_2))
                           = (cdr _i (cons _i e_3 e_4)))) =>
                         (E2 (fun (E5: ((cdr _i (cons _i e_1 e_2)) = e_2)) =>
                                (E1 (fun (E6:
                                         ((cdr _i (cons _i e_3 e_4))
                                         = e_4)) =>
                                       (E0 (eq_trans (eq_trans (eq_sym E5)
                                                     E4) E6)))))))))
     in
     let L1 := (fun (Q0:
                    ~ ~ (forall (A: Type),
                           (forall (X: A) (L: (list A)),
                              ((cdr A (cons A X L)) = L)))) (Q1:
                    ~ ((cdr _i (cons _i e_1 e_2)) = e_2)) =>
                  (Q0 (fun (Q2:
                           (forall (A: Type),
                              (forall (X: A) (L: (list A)),
                                 ((cdr A (cons A X L)) = L)))) =>
                         (Q1 (Q2 _i e_1 e_2)))))
     in
     let C0 := (fun (Ax0:
                    ~ (forall (A: Type),
                         (forall (X: A) (L: (list A)),
                            ((cdr A (cons A X L)) = L)))) => (Ax0 cons_axiom))
     in
     let R0 := (fun (l0: ~ ((cdr _i (cons _i e_1 e_2)) = e_2)) =>
                  (L1 C0 l0))
     in
     let R1 := (fun (l0: ~ (e_2 = e_4)) (l1:
                    ~ ~ ((cdr _i (cons _i e_3 e_4)) = e_4)) (l2:
                    ~ ~ ((cdr _i (cons _i e_1 e_2))
                        = (cdr _i (cons _i e_3 e_4)))) => (L0 l0 l1 R0 l2))
     in
     let L2 := (fun (E0: ~ ~ ((cons _i e_1 e_2) = (cons _i e_3 e_4)))
                    (E1:
                    ~ ((cdr _i (cons _i e_1 e_2))
                      = (cdr _i (cons _i e_3 e_4)))) =>
                  (E0 (fun (E2: ((cons _i e_1 e_2) = (cons _i e_3 e_4))) =>
                         (E1 (f_equal (cdr _i) E2)))))
     in
     let L3 := (fun (Ax0:
                    ~ (((cons _i e_1 e_2) = (cons _i e_3 e_4)) ->
                       ((e_1 = e_3) /\ (e_2 = e_4)))) (Ax1:
                    ~ ((cons _i e_1 e_2) = (cons _i e_3 e_4))) =>
                  (Ax0 (fun (I0: ((cons _i e_1 e_2) = (cons _i e_3 e_4))) =>
                          (False_ind ((e_1 = e_3) /\ (e_2 = e_4))
                          (Ax1 I0)))))
     in
     let L4 := (fun (Q0:
                    ~ (forall (V_x1: _i) (V_y1_503: (list _i)) (V_x2_501: _i)
                              (V_y2_499: (list _i)),
                         (((cons _i V_x1 V_y1_503)
                          = (cons _i V_x2_501 V_y2_499)) ->
                          ((V_x1 = V_x2_501) /\ (V_y1_503 = V_y2_499)))))
                    (Q1:
                    ~ ~ (((cons _i e_1 e_2) = (cons _i e_3 e_4)) ->
                         ((e_1 = e_3) /\ (e_2 = e_4)))) =>
                  (Q1 (epsilon_spec (inhabits (nil _i))
                      (fun (V_y2: (list _i)) =>
                         ~ (((cons _i e_1 e_2) = (cons _i e_3 V_y2)) ->
                            ((e_1 = e_3) /\ (e_2 = V_y2))))
                      (not_all_ex_not _ _
                      (epsilon_spec (inhabits (car _i (nil _i)))
                      (fun (V_x2: _i) =>
                         ~ (forall (V_y2_582: (list _i)),
                              (((cons _i e_1 e_2)
                               = (cons _i V_x2 V_y2_582)) ->
                               ((e_1 = V_x2) /\ (e_2 = V_y2_582)))))
                      (not_all_ex_not _ _
                      (epsilon_spec (inhabits (nil _i))
                      (fun (V_y1: (list _i)) =>
                         ~ (forall (V_x2_567: _i) (V_y2_565: (list _i)),
                              (((cons _i e_1 V_y1)
                               = (cons _i V_x2_567 V_y2_565)) ->
                               ((e_1 = V_x2_567) /\ (V_y1 = V_y2_565)))))
                      (not_all_ex_not _ _
                      (epsilon_spec (inhabits (car _i (nil _i)))
                      (fun (V_x1: _i) =>
                         ~ (forall (V_y1_503: (list _i)) (V_x2_501: _i)
                                   (V_y2_499: (list _i)),
                              (((cons _i V_x1 V_y1_503)
                               = (cons _i V_x2_501 V_y2_499)) ->
                               ((V_x1 = V_x2_501) /\ (V_y1_503 = V_y2_499)))))
                      (not_all_ex_not _ _ Q0))))))))))
     in
     let C1 := (fun (Ax0:
                    ~ ~ (forall (V_x1: _i) (V_y1_503: (list _i)) (V_x2_501:
                                _i) (V_y2_499: (list _i)),
                           (((cons _i V_x1 V_y1_503)
                            = (cons _i V_x2_501 V_y2_499)) ->
                            ((V_x1 = V_x2_501) /\ (V_y1_503 = V_y2_499))))) =>
                  (Ax0 G0))
     in
     let R2 := (fun (l0:
                    ~ ~ (((cons _i e_1 e_2) = (cons _i e_3 e_4)) ->
                         ((e_1 = e_3) /\ (e_2 = e_4)))) =>
                  (L4 (fun (r0:
                           (forall (V_x1: _i) (V_y1_503: (list _i))
                                   (V_x2_501: _i) (V_y2_499: (list _i)),
                              (((cons _i V_x1 V_y1_503)
                               = (cons _i V_x2_501 V_y2_499)) ->
                               ((V_x1 = V_x2_501) /\ (V_y1_503 = V_y2_499))))) =>
                         (C1 (fun (r1:
                                  ~ (forall (V_x1: _i) (V_y1_503: (list _i))
                                            (V_x2_501: _i) (V_y2_499:
                                            (list _i)),
                                       (((cons _i V_x1 V_y1_503)
                                        = (cons _i V_x2_501 V_y2_499)) ->
                                        ((V_x1 = V_x2_501)
                                        /\ (V_y1_503 = V_y2_499))))) =>
                                (r1 r0)))) l0))
     in
     let R3 := (fun (l0: ~ ((cons _i e_1 e_2) = (cons _i e_3 e_4))) =>
                  (L3 (fun (r0:
                           (((cons _i e_1 e_2) = (cons _i e_3 e_4)) ->
                            ((e_1 = e_3) /\ (e_2 = e_4)))) =>
                         (R2 (fun (r1:
                                  ~ (((cons _i e_1 e_2)
                                     = (cons _i e_3 e_4)) ->
                                     ((e_1 = e_3) /\ (e_2 = e_4)))) =>
                                (r1 r0)))) l0))
     in
     let R4 := (fun (l0:
                    ~ ((cdr _i (cons _i e_1 e_2))
                      = (cdr _i (cons _i e_3 e_4)))) => (L2 R3 l0))
     in
     let R5 := (fun (l0: ~ (e_2 = e_4)) (l1:
                    ~ ~ ((cdr _i (cons _i e_3 e_4)) = e_4)) =>
                  (R1 l0 l1 R4))
     in
     let L5 := (fun (Q0:
                    ~ ~ (forall (A: Type),
                           (forall (X: A) (L: (list A)),
                              ((cdr A (cons A X L)) = L)))) (Q1:
                    ~ ((cdr _i (cons _i e_3 e_4)) = e_4)) =>
                  (Q0 (fun (Q2:
                           (forall (A: Type),
                              (forall (X: A) (L: (list A)),
                                 ((cdr A (cons A X L)) = L)))) =>
                         (Q1 (Q2 _i e_3 e_4)))))
     in
     let R6 := (fun (l0: ~ ((cdr _i (cons _i e_3 e_4)) = e_4)) =>
                  (L5 C0 l0))
     in
     let R7 := (fun (l0: ~ (e_2 = e_4)) => (R5 l0 R6)) in
     let L6 := (fun (Ax0: ~ ((e_1 = e_3) /\ (e_2 = e_4))) (Ax1:
                    ~ ~ (e_2 = e_4)) (Ax2: ~ ~ (e_1 = e_3)) =>
                  (Ax2 (fun (Ax3: (e_1 = e_3)) =>
                          (Ax1 (fun (Ax4: (e_2 = e_4)) =>
                                  (Ax0 (conj Ax3 Ax4)))))))
     in
     let L7 := (fun (Ax0:
                    ~ (((cons _i e_1 e_2) = (cons _i e_3 e_4)) ->
                       ((e_1 = e_3) /\ (e_2 = e_4)))) (Ax1:
                    ~ ~ ((e_1 = e_3) /\ (e_2 = e_4))) =>
                  (Ax1 (fun (Ax2: ((e_1 = e_3) /\ (e_2 = e_4))) =>
                          (Ax0 (fun (I0:
                                    ((cons _i e_1 e_2)
                                    = (cons _i e_3 e_4))) => Ax2)))))
     in
     let R8 := (fun (l0: ~ ~ ((e_1 = e_3) /\ (e_2 = e_4))) =>
                  (L7 (fun (r0:
                           (((cons _i e_1 e_2) = (cons _i e_3 e_4)) ->
                            ((e_1 = e_3) /\ (e_2 = e_4)))) =>
                         (R2 (fun (r1:
                                  ~ (((cons _i e_1 e_2)
                                     = (cons _i e_3 e_4)) ->
                                     ((e_1 = e_3) /\ (e_2 = e_4)))) =>
                                (r1 r0)))) l0))
```

```
    in
    let R9 := (fun (l0: ~ ~ (e_2 = e_4)) (l1: ~ ~ (e_1 = e_3)) =>
                 (L6 (fun (r0: ((e_1 = e_3) /\ (e_2 = e_4))) =>
                       (R8 (fun (r1: ~ ((e_1 = e_3) /\ (e_2 = e_4))) =>
                              (r1 r0)))) l0 l1))
    in
    let L8 := (fun (Ax0: ~ (e_1 = e_3)) (Ax1:
                   ~ ~ ((car _i (cons _i e_1 e_2))
                       = (car _i (cons _i e_3 e_4)))) (Ax2:
                   ~ ~ (((car _i (cons _i e_1 e_2))
                        = (car _i (cons _i e_3 e_4))) -> (e_1 = e_3))) =>
                 (Ax2 (fun (Ax3:
                           (((car _i (cons _i e_1 e_2))
                            = (car _i (cons _i e_3 e_4))) -> (e_1 = e_3))) =>
                         (Ax1 (fun (Ax4:
                                   ((car _i (cons _i e_1 e_2))
                                   = (car _i (cons _i e_3 e_4)))) =>
                                 let O0 := (Ax3 Ax4) in
                                 (Ax0 O0))))))
    in
    let L9 := (fun (Ax0:
                   ~ ~ ((((car _i (cons _i e_1 e_2))
                         = (car _i (cons _i e_3 e_4))) -> (e_1 = e_3))
                       /\ ((e_1 = e_3) ->
                           ((car _i (cons _i e_1 e_2))
                           = (car _i (cons _i e_3 e_4)))))) (Ax1:
                   ~ (((car _i (cons _i e_1 e_2))
                      = (car _i (cons _i e_3 e_4))) -> (e_1 = e_3))) =>
                 (Ax0 (fun (Ax2:
                           ((((car _i (cons _i e_1 e_2))
                             = (car _i (cons _i e_3 e_4))) ->
                             (e_1 = e_3))
                           /\ ((e_1 = e_3) ->
                               ((car _i (cons _i e_1 e_2))
                               = (car _i (cons _i e_3 e_4)))))) =>
                         (and_elim (((car _i (cons _i e_1 e_2))
                                    = (car _i (cons _i e_3 e_4))) ->
                                    (e_1 = e_3))
                         ((e_1 = e_3) ->
                          ((car _i (cons _i e_1 e_2))
                          = (car _i (cons _i e_3 e_4)))) False Ax2
                         (fun (A0:
                              (((car _i (cons _i e_1 e_2))
                               = (car _i (cons _i e_3 e_4))) ->
                               (e_1 = e_3))) (A1:
                              ((e_1 = e_3) ->
                               ((car _i (cons _i e_1 e_2))
                               = (car _i (cons _i e_3 e_4))))) =>
                            (Ax1 A0))))))
    in
    let L10 := (fun (Q0 Q1:
                    ~ ~ (forall (A: Type),
                           (forall (X: A) (L: (list A)),
                              ((car A (cons A X L)) = X)))) (Q2:
                    ~ ((((car _i (cons _i e_1 e_2))
                        = (car _i (cons _i e_3 e_4))) -> (e_1 = e_3))
```

```
                  /\ ((e_1 = e_3) ->
                      ((car _i (cons _i e_1 e_2))
                      = (car _i (cons _i e_3 e_4)))))) =>
              (Q1 (fun (Q3:
                       (forall (A: Type),
                          (forall (X: A) (L: (list A)),
                             ((car A (cons A X L)) = X)))) =>
                     (Q1 (fun (Q4:
                              (forall (A: Type),
                                 (forall (X: A) (L: (list A)),
                                    ((car A (cons A X L)) = X)))) =>
                            (Q2 (eq_subst _i e_3
                                (car _i (cons _i e_3 e_4))
                                (fun (_1: _i) =>
                                   (((car _i (cons _i e_1 e_2)) = _1)
                                   <-> (e_1 = e_3)))
                                (let I0 := (Q4 _i e_3 e_4) in
                                 (eq_sym I0))
                                (eq_subst _i e_1
                                (car _i (cons _i e_1 e_2))
                                (fun (_2: _i) =>
                                   ((_2 = e_3) <-> (e_1 = e_3)))
                                (let I0 := (Q4 _i e_1 e_2) in
                                 (eq_sym I0)) (iff_refl (e_1 = e_3))))))))))
in
let R10 := (fun (l0:
                ~ ~ (forall (A: Type),
                       (forall (X: A) (L: (list A)),
                          ((car A (cons A X L)) = X)))) (l1:
                ~ ((((car _i (cons _i e_1 e_2))
                    = (car _i (cons _i e_3 e_4))) -> (e_1 = e_3))
                  /\ ((e_1 = e_3) ->
                      ((car _i (cons _i e_1 e_2))
                      = (car _i (cons _i e_3 e_4)))))) =>
              (L10 l0 l0 l1))
in
let C2 := (fun (Ax0:
               ~ (forall (A: Type),
                    (forall (X: A) (L: (list A)),
                       ((car A (cons A X L)) = X)))) => (Ax0 car_axiom))
in
let R11 := (fun (l0:
                ~ ((((car _i (cons _i e_1 e_2))
                    = (car _i (cons _i e_3 e_4))) -> (e_1 = e_3))
                  /\ ((e_1 = e_3) ->
                      ((car _i (cons _i e_1 e_2))
                      = (car _i (cons _i e_3 e_4)))))) => (R10 C2 l0))
in
let R12 := (fun (l0:
                ~ (((car _i (cons _i e_1 e_2))
                   = (car _i (cons _i e_3 e_4))) -> (e_1 = e_3))) =>
              (L9 R11 l0))
in
let R13 := (fun (l0: ~ (e_1 = e_3)) (l1:
                ~ ~ ((car _i (cons _i e_1 e_2))
                    = (car _i (cons _i e_3 e_4)))) => (L8 l0 l1 R12))
```

```
     in
     let L11 := (fun (E0: ~ ~ ((cons _i e_1 e_2) = (cons _i e_3 e_4)))
                     (E1:
                     ~ ((car _i (cons _i e_1 e_2))
                       = (car _i (cons _i e_3 e_4)))) =>
                   (E0 (fun (E2: ((cons _i e_1 e_2) = (cons _i e_3 e_4))) =>
                          (E1 (f_equal (car _i) E2)))))
     in
     let R14 := (fun (l0:
                     ~ ((car _i (cons _i e_1 e_2))
                       = (car _i (cons _i e_3 e_4)))) => (L11 R3 l0))
     in
     let R15 := (fun (l0: ~ (e_1 = e_3)) => (R13 l0 R14)) in
     let R16 := (fun (l0: ~ ~ (e_2 = e_4)) => (R9 l0 R15)) in
     let R17 := (R7 (fun (r0: (e_2 = e_4)) => (R16 R7))) in
     R17))
  (* PROOF END *).
```