Automating Interactive Theorem Provers and Certifying Automated Theorem Provers

by

Arjun Viswanathan

A thesis submitted in partial fulfillment
of the requirements for the Doctor of Philosophy
degree in Computer Science in
the
Graduate College of
The University of Iowa

December 2024

Thesis Committee: Cesare Tinelli, Thesis Supervisor
Omar Chowdhury
Chantal Keller
Garrett Morris
Alberto Maria Segre

Acknowledgements

This thesis and the PhD that it attests to is a culmination of an enormous amount of effort over a long period of time. This would not have been possible without all the support I have had over the years. It's likely that, I will accidentally omit some names that deserve to be here, due to a recency bias, if not by general oversight. Regardless, I am sincerely grateful to everybody who has helped me directly or indirectly through the years of my PhD.

All of the work described in this document is either built on or makes use of established tools. I thank all the authors of citations as well as all developers of the tools mentioned in this section, for providing the foundation for my work. All three contributions have some connection to the SMTCoq tool [5]. cvc5's abduction solver [84] is used for contribution 1. The alethe proof format [88] used in contribution 2 is still evolving and the developers have been open to feedback during its evolution. The invertibility conditions [72] that we verify in contribution 3 were synthesized by members of the cvc5 group.

I thank Haniel Barbosa, Clark Barrett, Burak Ekici, Chantal Keller, Andrew Reynolds, Cesare Tinelli, and Yoni Zohar — my co-authors from the following submissions: (1) the FroCoS (International Symposium on Frontiers of Combining Systems) 2023 paper submission [52] on verifying invertibility conditions, a previous version of which was submitted as an extended abstract to PxTP (Workshop on Proof eXchange for Theorem Proving) 2019 [51], and (2) the submission to LPAR (International Conference on Logic for Programming, Artificial Intelligence and Reasoning) 2023 on extending SMTCoq with abductive reasoning [9]. Large parts of the text from these submissions are used in this thesis. Some images have been borrowed from previous presentations of SMTCoq, and I thank the original authors.

Thanks to Kathleen Shaughnessy, Danielle Land, and everybody else who ran the thesis writing workshops at the University of Iowa. Thanks to Cesare Tinelli, Chantal Keller, Omar Chowdhury, Yoni Zohar and Alex Ozdemir for their suggested improvements to the thesis.

Thanks to Chantal Keller for being an excellent collaborator and mentor, and for hosting me in France. Thanks also to Valentin Blot, Louise Dubois de Prisque, Andrew Samokish from the Laboratory of Formal Methods at Université Paris-Saclay.

The CS Department at the University of Iowa has professionally backed me throughout my time in this program, and I am grateful to the staff, faculty and students for their assistance, mentorships and friendships. Special mention to Tanmay Inamdar, Shreyas Pai, Tejaswi Rohit, Jamil Gafur, Matthieu Biger, Sheryl Semler, Catherine Till, Alli Rockwell, Tina Kimbrell, Alberto Segre, Denise Szecsei, Sriram Pemmaraju, and Kasturi Varadarajan, among the numerous people from this department who deserve my gratitude.

Having to deal with immigration is difficult and stressful. Thanks to the International

Student and Scholar Services at the University of Iowa for their continuous assistance in dealing with the same. Special thanks to Sushant Kaura and Rachel Tobe for opening their home to me during my travels through Chicago.

All members of the CLC group, both past and present, have ensured that my research field remains alive and well, and it has been very enjoyable sharing the same space with these friends and colleagues over the years: Anthony Cantor, Mudathir Mohamed, Christa Jenkins, Andrew Marmaduke, Moosa Yahyazadeh, Daniel Larraz, Apoorv Ingle, Rob Lorch, Alexander Hubers, Hans-Jörg Schurr (whose thesis served to inspire this one!), Kartik Sabharwal, Shweta Rajiv, Muhammad Mazhar, Baoluo Meng, Abdalrhman Mohamed, Joyanta Debnath, Fareed Arif, Ananda Guneratne, Richard Blair, Alain Mebsout, Ruoyu Zhang, and Mitziu Echeverria.

The ever-expanding cvc5 group has given me some of the best colleague I've ever had. Yoni Zohar, Haniel Barbosa, Andrew Reynolds, Aina Niemetz, Mathias Preiner, Hanna Lachnitt, Alex Ozdemir, Andres Nötzli — thank you for serving as a motivation to do better research, and long may our friendships continue.

Participating in a doctoral committee in itself entails a significant amount of work. I owe the members of my committee special thanks for sticking through it with me and guiding me through the rough patches — Omar Chowdhury, Juan Pablo Hourcade, Chantal Keller, Garrett Morris, Andrew Reynold, Alberto Segre, Aaron Stump, and Cesare Tinelli.

The Men's Soccer Club at the University of Iowa played a huge role in keeping my mental health intact during the trials and tribulations of academic research. In addition to camaraderie and valuable friendships, it gave me the opportunity to play the sport I love at a level that satisfied my appetite for competition.

Thanks to Malissa, Jeff, Mary-Beth and Nolan Schroeder for teaching me the value of family and for never letting me miss mine too much.

Iowa City has been a perfect fit for me, I've loved my years here. People smile at you like they mean it, there is always enough room to watch the gorgeous sunset from the stairs of the Old Capitol Building, the stars shine down on you at night, and the snow eventually melts away. I've built so many relationships here that I hope will stand the test of time and distance. Thank you Haniel Barbosa for always being open to one more drink and Samantha Nagle for the friendship and hospitality; Dan Crouch for being a big brother and Mikayla, Ella, and Charlotte Crouch for being family; Joe Rattenni and Julia Bobinet for all the love and kindness; Sada Shiva for always making the time to visit and lend a helping hand. Thanks also to Aaron Junho Yoon, Anel Dozo, Salah Ibrahim, Tejas Mallela, Tom Walch, Mike Schumacher, Peter Rodd, and the rest of the 2 Dogs gang among many others locals.

My friends from back home in Hyderabad and from Bangalore who have not only managed to remain friends despite the distance but have found a way to back me from whichever part of the world they have ended up in — Kopal Sharraf, Kaushik Varma, Sachin Sanil, Akash Elichipuram, Anvesh Reddy, Prudhvi Kolaventy, Sada Shiva, Sai Kameshwar Rao, Srinivas Varma, Ashutosh Billa, Nithin Paidikalva, CK, Arushi Koul, Rishi Raj Sakya, Rakendu Jois, & family, Srikanth Vasudevan & family, Anirudh Varma & Amna Aunty & family, Rishmeet Singh, Vinayak Bhatt, Ratankumar Takhellambam, and the list goes on.

All these people have become like family to me over the past decade, but I am fortunate to also have family away from home — Janhavi Viswanathan, Ashwin Sampath, Janani Thyagarajan, Kaushik Sridhar, Ganesh Ramachandran, Smitha Radhakrishnan, Abhi & Giri, Raghuram Thyagarajan; Kaushik, Poornima & family; Vishu, Kavita and family; Nagu Mama and family; and Sriram Jayaram, to name few. I would be remiss if I didn't give special mention to my Bagya Periamma for always looking out for me; it is her ability to find the right words for thoughts that in many ways got me started on this adventure.

Thanks to my best friend, little Tim. He continues to live his best life, perhaps unaware at the impact he's had on so many of us.

I owe a lot to Taanya Malhotra for giving me strength to endure the hard parts of this journey and for her love, wisdom, and inspiration. Thanks also to her family for accepting me as one of theirs.

I truly believe that Cesare Tinelli was the advisor I needed — nothing more and nothing less. Arguably, it was his idea which he shared with me at a gas station somewhere in Kansas that proved to be the genesis of this work. Apart from being a distinguished Professor to look up to, his experience and wisdom, his network of leaders in industry and academia, his expansive knowledge of formal methods and of computer science, have been invaluable for my development as a doctoral candidate and have prepared me for life beyond my doctorate. If not for his liberal attitude, I would never have attempted a PhD to start with; the same attitude ensured that I didn't quit when things got difficult.

My parents, Amma and Appa, have been staunch admirers of academia. Funnily enough, they never brought up the possibility of me pursuing a PhD. I have no doubt that their appreciation of scholarship made its way to me by more implicit means. Thanks to them for nurturing, providing, guiding, and motivating me to become someone who decided to embark on this journey and for their encouragement through it.

Abstract

As software grows increasingly pervasive in our everyday lives, it is important to ensure that the software we rely on, especially in safety-critical systems, behaves as expected. Whereas software testing is a useful approach for detecting the presence of bugs, formal methods offer tools and techniques to prove the absence of bugs. One class of such tools is theorem provers — computer programs capable of proving mathematical theorems. Among other things, theorem provers are used to prove the correctness of software with respect to a specification, in an attempt to prevent buggy software.

Theorem provers are commonly classified as automated or interactive. Automated theorem provers (ATPs) such as satisfiability modulo theories (SMT) solvers aim to prove logical formulas quickly and without human intervention. To this end, they rely on various heuristics, decision procedures, and optimizations. Consequentially, ATPs are typically large software systems and therefore prone to bugs themselves. On the other hand, interactive theorem provers (ITPs), or proof assistants, restrict themselves to a (relatively) small trusted computing base (TCB), giving strong guarantees of the proofs performed. They do so at the cost of automation, and require elaborate proofs and higher user involvement.

Given ATPs' and ITPs' relative pros and cons, there are multiple avenues for leveraging the strengths of one to address the weaknesses of the other. We discuss three such possibilities, and our contributions to each kind, in the following.

1. *External ATPs can be used to automate sub-goals within ITPs.* SMTCoq is one such tool that is able to dispatch subgoals to an external SMT solver without extending the Coq proof assistant's TCB. In a traditional interaction, SMTCoq relies on the SMT solver's ability to do deductive reasoning — a call to a solver either succeeds with a proof in Coq, or fails with a possible counterexample. We enhance this interaction with an SMT solver capable of performing abductive reasoning so that in cases of failure, the solver may ask for additional facts that can convert a failure to a success.

2. *An ATP's result can be certified by checking it in an ITP.* We adapt SMTCoq to check more refined proofs from SMT solvers. We do this through the alethe proof format which is supported by both the cvc5 and veriT SMT solvers. This has the added benefit of increasing goal coverage by external SMT solvers in Coq, which also categorizes it as a contribution previously discussed in 1.

3. *An ATP can be certified by checking its algorithm (or modular parts thereof) in an ITP.* In this direction, we verify results called invertibility conditions, that are critical to the

operation of some SMT solvers, in Coq. Such solvers use a set of these invertibility conditions during solving in the theory of quantified bit-vectors. We prove a previously unverified subset of these conditions, increasing confidence in the results of bit-vector solvers that use invertibility conditions for quantified formulas.

Public Abstract

Computers are ubiquitous in today's world and have been integrated into every aspect of our lives and work. They have wide-ranging applications such as in defense systems, healthcare systems, banking, and infrastructure. A computer is operated using interfaces called software. It is important to ensure that a software behaves as it is instructed to. When a software produces undesirable behavior, we call the source of this behavior a bug. Avoiding software bugs is especially important in *safety-critical systems* — computer systems whose misbehavior could result in serious injury or loss of life. Examples of these include software that runs airplane systems, drones, cars, medical devices, and power plants.

Given (*i*) a specification of how a software should behave, as a set of logical formulas, and (*ii*) either a model or an implementation of the software, a theorem prover is a tool that can be used to ensure that the software behaves as it's supposed to. Theorem provers can be broadly classified as automated or interactive. Automated theorem provers (ATPs) aim to prove logical formulas quickly without external human help. Achieving this takes a substantial amount of code, which makes ATPs typically large computer software that are themselves prone to bugs. On the other hand, interactive theorem provers (ITPs) are smaller pieces of software that strictly follow certain principles that prevent bugs in their code. However, they provide a limited amount of automation, and hence, proving logical formulas in an ITP requires significantly more effort from a user. An ideal tool for software verification would offer both proof automation and strict guarantees of software correctness. This thesis presents three integrations between ATPs and ITPs that leverage the higher level of automation in ATPs and the higher level of trust in the proofs produced by ITPs to offer faster and more reliable provers for software systems. Provers integrated in these ways would prevent bugs in our software.

1. External ATPs can be used within ITPs to automate proofs. Such an integration can preserve the high level of trust in the ITP by internally following the steps taken by the ATP to prove the formula. Often, an external ATP can fail in such an integration because the ATP does not have sufficient information from the ITP to prove a formula. The first contribution of this thesis enhances a typical ATP-ITP integration so that the ATP can ask for extra information to prove a currently failing goal. Our work's experimental evaluation suggests that such an enhancement reduces the number of failures that can occur when an ITP calls an external ATP to automatically prove formulas for it.

2. An ATP's result can be externally verified by checking it in an ITP. This increases trust

in the results produced by an ATP. ATPs that produce additional certificates can have their results verified in an ITP. The second contribution of this thesis generates an ITP checker for certificates produced by two distinct ATPs. Our experimental evaluation argues that this checker is more complete in its coverage than previous ITP checkers for ATPs.

3. An ATP can be verified within an ITP. Although the size of modern ATPs makes this a very challenging goal, ATPs can be separated into independent modules. The final contribution of this thesis is to verify a set of results, called invertibility conditions in an ITP. For ATPs that use them, the correct operation of one of their modules depends on the correctness of these invertibility conditions. The verification of these conditions in an ITP increases the reliability of the corresponding module of the ATP that relies on them.

Contents

List of Figures

Chapter 1

Introduction

In the twenty-first century, software has become central to most things that we do and care about. Examples include, but are not limited to, automobiles, military, banking, retail, education, and infrastructure. Software, however, is susceptible to bugs, and the consequences of buggy software can be drastic, especially in safety-critical systems such as planes, cars, medical systems, and weapon systems. Although software testing is a common measure against such bugs, testing is non-exhaustive. Additionally, while testing is a useful apparatus to detect bugs, in most cases, it is not capable of proving the absence of bugs. To this end, formal methods have been developed to provide techniques and tools that can model hardware and software systems, and argue that they perform correctly with respect to a specification. Theorem provers are one such utility used to prove mathematical properties about systems and provide guarantees about their functionalities. However, theorem provers are themselves pieces of software, and are therefore subject to similar scrutiny. Classifying theorem provers as automated or interactive can help us better understand this problem and some state-of-the-art solutions.

Automated theorem provers (ATPs), such as SMT (satisfiability modulo theories) solvers, are able to (dis)prove formulas in an increasing number of logical fragments. While they were initially created to be tools capable of performing deductive reasoning in quantifier-free logics, where new facts are derived from a set of known logical premises, SMT solvers have rapidly grown in capability. One particular functionality of interest is their ability to do abductive reasoning, or hypotheses finding with respect to a goal that needs to be true. Other developments to SMT solvers include a steady increase in the number of supported theories, and smart instantiation techniques to support reasoning over quantifiers. Due to the accelerated expansion of their abilities, and since they often implement many elaborate decision procedures over possibly undecidable problem spaces, SMT solvers use many heuristics that make them efficient but also result in an extremely large codebase, which is hard to check and might itself be susceptible to bugs.

On the other hand, interactive theorem provers (ITPs), such as Coq, rely on a small, trustworthy kernel of code. One must write tedious machine-checkable proofs in them as one would on pencil and paper. As a consequence, it is harder to prove properties in ITPs, and moreover, automation is limited. Their results, however, are highly reliable since the user must stay faithful to a relatively small trusted computing base (TCB) while going through proofs.

Software certified by ITPs, then, are — while burdensome to verify — the most desirable in safety-critical applications. Due to this loose hierarchy of verification standards imposed on ITPs and ATPs, much recent research has focused on integrating these two kinds of provers to achieve a best-of-both-worlds scenario. An integration can benefit both the ATP, by validating its results in a proof assistant, and the ITP by automating proofs to increase efficiency. SMT solvers are evolving to emit, in addition to their results, proofs of the results that can be externally verified in an ITP. SMTCoq [50] is a certified checker for such proofs in Coq, invoking (among other proof-producing solvers) CVC4 [14] and veriT [27] to dispatch goals automatically. SMTCoq has some general limitations. First, although complete automation of Coq goals by an external SMT solver is very useful, there are practical restrictions on the number of goals that are amenable to such automation, and on the completeness of the automation. Often the solver may fail to prove a goal due to a lack of information about terms in the goal, rather than the goal being invalid in Coq. Moreover, even when the solver does succeed, it's proof might be incomplete. For example, some fine-grained steps, such as the rewriting of input formulas, may be left unjustified in the proof. Second, SMTCoq's representation of SMT formulas in Coq does not permit quantified formulas, and thus its checker is restricted in its ability to certify reasoning in quantified logical fragments. Towards addressing these limitations and in furthering the integration between ATPs and ITPs, this thesis makes the following three contributions:

1. Towards increasing goal coverage of SMT solvers in ITPs, we extend SMTCoq to leverage an SMT solver's abductive capabilities. In cases where the solver finds the Coq goal to be invalid, this feature allows the SMT solver to request the Coq user to provide more information about terms in the goal that would allow the solver to prove the goal, thus increasing interaction between the Coq user and the SMT solver.

2. Towards completely automating proofs of goals in ITPs, we implement a certified checker for refined SMT proofs in Coq. The implementation takes the form of a checker for alethe — a new proof format for SMT solvers that aims to unify multiple proof-producing solvers. Importantly, we provide support for checking fine-grained proofs from solvers including justifications of formula rewrites. Our checker works by reducing alethe proofs by a sequence of sound proof transformations, so that they are checkable by SMTCoq. It has the dual benefit of expanding SMT-driven automation in Coq, as well as increasing trust in the solvers that produce alethe proofs.

3. Towards certifying SMT solvers for quantified reasoning, we formalize and prove in Coq certain properties called invertibility conditions, which are used by SMT solvers for quantified reasoning over bit-vectors. By proving the correctness of these conditions in Coq, we certify a quantifier instantiation technique for bit-vectors in Coq, increasing the reliability of the solvers that use this technique.

Chapter 2 establishes the background necessary for the rest of the document. Chapter 3 formally describes our contributions using terms introduced in the Background section.

Chapters 4, 5, and 6 detail the work done towards contributions 1, 2, and 3; and Chapter 7 summarizes the contributions of the thesis and presents avenues for future research.

Chapter 2

Background

2.1 Preliminaries

Our logical setting is that of classical many-sorted first-order logic with equality, the base logic of SMT [12]. We define set S of sort symbols containing a distinguished symbol called *Bool*, and set X of variable symbols, each associated with a sort in S. A *signature* Σ is composed of:

- $\Sigma_S \subseteq S$, the *sort symbols*
- set Σ_F, the *function symbols*
- total mapping $ra : \Sigma_F \rightarrow (\Sigma_S)^+$, where $+$ is the regular expression operator that indicates one or more occurrences of the preceding element.

Each function symbol f in Σ_F has *arity* n and *rank* $ra(f) = \sigma_1 \dots \sigma_n \sigma$, with $n \geq 0$. Function symbols with arity 0 are called *constant* symbols. Σ-terms and Σ-formulas are defined as t and ϕ respectively in the following grammar.

$$t := x \mid f(t_1, ..., t_n)$$
$$\phi := x^{Bool} \mid False \mid t_1 = t_2 \mid \neg\phi \mid p(t_1, \dots, t_n) \mid \phi_1 \vee \phi_2 \mid \exists x.\phi$$

A Σ-*term* of sort σ is either a sorted variable x, or an application of $f \in \Sigma_F$ with rank $\sigma_1 \dots \sigma_n \sigma$ to terms $t_1, \dots, t_n$ such that the sort of each t_i is σ_i for $i = 1, \dots, n$. A Σ-*formula* — a Σ-term of sort *Bool* — is either a variable of sort *Bool* (distinguished by specifying the sort as a superscript), $False$ (the expression representing falsity); the equality between two terms ($t_1 = t_2$); the negation of a formula ($\neg\phi$); an application of $p \in \Sigma_F$ with rank $\sigma_1 \dots \sigma_n Bool$ (also called a *predicate symbol*) to terms $t_1, \dots, t_n$ such that the sort of each t_i is σ_i for $i = 1, \dots, n$; the disjunction of two formulas ($\phi_1 \vee \phi_2$); or an existentially quantified formula $\exists x.\phi$ where x is a variable with sort in Σ_S. We write $\psi[x_1, \dots, x_n]$ to represent a formula whose free variables are from the set $\{x_1, \dots, x_n\}$ and $\psi[x_1 \mapsto c_1, \dots, x_n \mapsto c_n]$ to represent the substitution in ψ of terms $c_1, \dots, c_n$ respectively for variables $x_1, \dots, x_n$.

For convenience, we naturally extend formulas to include $True$ for logical truth; and conjunctive ($\phi_1 \wedge \phi_2$), implicative ($\phi_1 \rightarrow \phi_2$) and universally quantified formulas ($\forall x.\phi$).

These extensions are defined in terms of basic formulas as usual. Often, we distinguishe equalities between formulas ($\phi_1 =^{Bool} \phi_2$ where we drop the sort when clear from context, or unnecessary) — as equivalences or if-and-only-ifs — from equalities over other types. We also allow for quantifiers to bind multiple variables to a formula, conjunctions and disjunctions to be naturally extended to any non-zero arity, and introduce the shorthand $\neq$ for negation of an equality. As we have done for variables in the grammar rule for formulas above, we use a superscript to denote the sort of a term, when necessary. An *atomic Σ-formula* is a Σ-formula that has no proper subterms of sort *Bool*. A *Σ-literal* is an atomic Σ-formula or the negation of one. A *clause* is a disjunction $l_1 \vee \cdots \vee l_n$ of literals. We often represent clauses as sets of their constituent literals ($\{l_1, \cdots, l_n\}$, where sometimes we omit the braces), and denote the empty clause by $\langle\,\rangle$. A formula is in *conjunction normal form* (or CNF) if it is a conjunction $c_1 \wedge \cdots \wedge c_n$ of zero or more clauses.
For each signature Σ and set $Y \subseteq X$ of sorted variables, a *Σ-interpretation* $\mathcal{I}$ over Y maps

- each sort $\sigma \in \Sigma^S$ to non-empty set $\mathcal{I}_\sigma$, the *domain* of σ in $\mathcal{I}$, such that the domain of *Bool* is $\{\top, \bot\}$;
- each variable $x \in Y$ of sort σ to an element $x^\mathcal{I} \in \mathcal{I}_\sigma$ (we call this mapping a *valuation* $V_\mathcal{I}$);
- each function symbol $f \in \Sigma_F$ of rank $\sigma_1 \dots \sigma_n \sigma$ to a total function $f^\mathcal{I} : \mathcal{I}_{\sigma_1} \times \ldots \times \mathcal{I}_{\sigma_n} \to \mathcal{I}_\sigma$

The notion of substitutions introduced above naturally extends to interpretations and valuations as well. We use notation $\{x_1 \mapsto c_1, \ldots, x_n \mapsto c_n\}$ for a valuation mapping $x_1, \ldots, , x_n$ to $c_1, \ldots, c_n$ respectively. An *evaluation* of a Σ-term with respect to a Σ-interpretation $\mathcal{I}$ is recursively defined as the function $[\![\ ]\!]_\mathcal{I}$:

- For variable x, $[\![x]\!]_\mathcal{I} = V_\mathcal{I}(x)$
- $[\![f(t_1, \ldots, t_n)]\!]_\mathcal{I} = f^\mathcal{I}([\![t_1]\!]_\mathcal{I}, \ldots, [\![t_n]\!]_\mathcal{I})$
- $[\![True]\!]_\mathcal{I} = \top$ and $[\![False]\!]_\mathcal{I} = \bot$
- $[\![t_1 = t_2]\!]_\mathcal{I} = \top$ if $[\![t_1]\!]_\mathcal{I} = [\![t_2]\!]_\mathcal{I}$; otherwise, $[\![t_1 = t_2]\!]_\mathcal{I} = \bot$
- $[\![\neg\phi]\!]_\mathcal{I} = \top$ if $[\![\phi]\!]_\mathcal{I} = \bot$; otherwise, $[\![\neg\phi]\!]_\mathcal{I} = \bot$
- $[\![\phi_1 \vee \phi_2]\!]_\mathcal{I} = \top$ if $[\![\phi_1]\!]_\mathcal{I} = \top$ or $[\![\phi_2]\!]_\mathcal{I} = \top$; otherwise, $[\![\phi_1 \vee \phi_2]\!]_\mathcal{I} = \bot$
- $[\![\exists x^\sigma.\phi]\!]_\mathcal{I} = \top$ if there exists $v \in \mathcal{I}_\sigma$ such that $[\![\phi]\!]_{\mathcal{I}[x \mapsto v]} = \top$; otherwise, $[\![\exists x^\sigma.\phi]\!]_\mathcal{I} = \bot$

A Σ-interpretation $\mathcal{I}$ *satisfies* a Σ-formula ϕ — denoted $\mathcal{I} \models \phi$ — if $[\![\phi]\!]_\mathcal{I} = \top$. Then, $\mathcal{I}$ is called a *model* of ϕ. Often, we only care about the valuation of variables and take the rest of the interpretation to be standard when talking about satisfying models. A *theory* is a pair $T = (\Sigma, I)$ where Σ is a signature and I is a non-empty class of Σ-interpretations called the models of T or *T-models*. A Σ-formula is *T-satisfiable* or *satisfiable modulo T* if it has a T-model, and T-*unsatisfiable* or *unsatisfiable modulo* T otherwise. Two formulas are $T-$*equisatisfiable* if they are both $T-$satisfiable or both $T-$unsatisfiable. A set Γ of formulas T-*entails* a formula ψ, written $\Gamma \models_T \psi$, if every model of T that satisfies all the formulas in Γ is also a model of ψ. A Σ-formula ϕ is *weaker (in T)* than a formula ψ if $\{\psi\} \models_T \phi$. Some theories from the SMT-LIB 2 [11] standard for SMT solvers that we will reference (often using the parenthesized abbreviations) are the theories of equality over uninterpreted functions (EUF), linear integer arithmetic (LIA), bit-vectors (BV), and arrays with extensionality (AX).

2.2 SMT Solvers

Propositional satisfiability (SAT) is the problem of determining whether a propositional formula is satisfiable. Satisfiability modulo theories (SMT) is concerned with the satisfiability of formulas with respect to some background theory [13].

Example 2.2.1. For propositional variables P, Q, and R (variables of sort *Bool*), the propositional formula $P \wedge Q \wedge \neg R$ is satisfiable, and a satisfying assignment is $\{P \mapsto \top, Q \mapsto \top, R \mapsto \bot\}$. A formula with a similar propositional structure $(a = b) \wedge (b = c) \wedge \neg(a = c)$ is unsatisfiable modulo the theory of equality (EUF). The analogous assignment $\{(a = b) \mapsto \top, (b = c) \mapsto \top, (a = c) \mapsto \bot\}$ is inconsistent by transitivity of equality, which requires $a = c$ to hold given that $a = b$ and $b = c$ hold.

SMT solvers are commonly used to drive various software and hardware verification tools and techniques such as model checking [7], symbolic execution [8], program synthesis [26], and interpolant generation [54]. They can also be used as provers, since a formula is valid if its negation is unsatisfiable. In fact, the unsatisfiability of the formula from Example 2.2.1 is an acceptable proof of the transitivity of equality over literals a, b and c: $(a = b) \to (b = c) \to (a = c)$, since the negation of this formula is logically equivalent to $(a = b) \wedge (b = c) \wedge \neg(a = c)$. Conversely, satisfiability of the negation of a formula is a *disproof*, or a proof of its invalidity, and a satisfying model is a counterexample witnessing the invalidity.

Conceptually, it is useful to think of formulas as entailments over some theory T between a (possibly empty) set ($\{H_1, \ldots, H_n\}$) of hypotheses H and a goal G:

$$H \models_T G$$

An SMT solver is able to deductively prove (resp. disprove) this entailment if it is able to prove the negation of the formula $H_1 \wedge \cdots \wedge H_n \to G$ unsatisfiable (resp. satisfiable). When such an entailment does not hold ($H \not\models_T G$), *abduction* is the problem of finding some formula ϕ — an *abduct* — such that:

- ϕ is consistent with H in T, that is, $H \wedge \phi$ is T-satisfiable.
- $H \wedge \phi \models_T G$

Syntax-restricted abduction is the problem of finding an abduct that is in the language generated by a given context-free grammar R.

For quantifier-free reasoning, a typical SMT solver composes a SAT solver with multiple theory solvers in an abstraction-refinement cycle, where the SAT engine tries to find a satisfying model of the propositional abstraction of a given set of constraints, and the theory solvers find a refutation of the refinement of the model, if one exists. This cycle is guided by the DPLL(T) [55] algorithm, which is an extension of the DPLL [69] (Davis-Putnam-Logemann-Loveland) algorithm with theory-level reasoning. An SMT solver converts its input constraints into conjunction normal form (CNF); the DPLL(T) algorithm then tries to find a satisfying assignment for these clauses, and otherwise, by exhaustion concludes their unsatisfiability. The steps taken to conclude unsatisfiability can be translated (roughly) into a chain of resolutions of the input clauses to conclude the empty clause from them, which is the most basic form of unsatisfiability.

2.2.1 Quantifiers

For quantified logics, quantifier handling methods are overlaid on the DPLL(T) architecture. *Skolemization* is a technique used to eliminate existential quantifiers, and the most popular method that SMT solvers use to deal with universal quantifiers is *quantifier instantiation* where ground (variable-free) terms are substituted for universal variables repeatedly, until either an unsatisfiable set of instances is found (implying that the original formula is unsatisfiable), or a model for the original formula is found. This process is not necessarily terminating, and its efficiency depends on problem-specific factors such as the theories and ground terms involved, and technique-specific factors, particularly, the quantifier instantiation technique employed. Some popular instantiation methods are E-matching [42, 37, 56] conflict-based instantiation [86], enumerative instantiation [83], counterexample guided instantiation [85], and model-based instantiation [57, 87].

Our contributions are relevant to SMT-LIB 2's theory of bit-vectors which we briefly describe here. A more expansive description is presented in Section 6.1. The signature Σ_{BV} of the SMT-LIB 2 theory of fixed-width bit-vectors includes a unique sort for each positive integer n, denoted $\sigma_{[n]}$, representing the bit-vectors of length n. In the following, we look at examples of 32-bit bit-vectors from $\sigma_{[32]}$ and describe the functions we use inline.

Example 2.2.2. This example is borrowed from Jonáš et al. [67]. Consider the following formula ϕ where x and y are 32-bit bit-vector variables and 2 and 3 are 32-bit bit-vector constants, $<_u$ is the less-than predicate over bit-vectors that interprets its arguments as unsigned, and $\cdot$ is bit-vector multiplication.

$$3 <_u x \wedge \forall y(x \neq 2 \cdot y)$$

Let G abbreviate the quantifier-free part — $3 <_u x$, and $\forall y$. Q the quantified part — $\forall y(x \neq 2 \cdot y)$. An SMT solver using model-based instantiation works as follows.

1. It checks the quantifier-free part of the formula. Finding it unsatisfiable is an easy way to conclude the unsatisfiability of the entire formula. Here, it finds G to be satisfiable with model $\mathcal{M} = \{x \mapsto 4\}$.

2. Next, it checks whether $\mathcal{M}$ is a model of $\forall y.\ Q$ by checking whether $\neg Q_{\mathcal{M}}$ is unsatisfiable, where $Q_{\mathcal{M}}$ is the quantifier-free formula obtained by substituting for free variables and uninterpreted function symbols from $\mathcal{M}$. If $\mathcal{M}$ is a model of $\forall y.\ Q$ ($\neg Q_{\mathcal{M}}$ is unsatisfiable), the solver can conclude that ϕ is satisfiable. However, this is not the case, since $4 = 2 \cdot y$ is satisfiable with model $\mathcal{N} = \{y \mapsto 2\}$.

3. Now the solver has model $\mathcal{N}$ (of $\neg Q_{\mathcal{M}}$) which it uses to rule out $\mathcal{M}$ as a model. This is done by conjoining the original formula with *instance* $Q[y \mapsto 2]$, the formula obtained by substituting for the variables in Q based on $\mathcal{N}$, and then repeating.

So, after one iteration, we have equisatisfiable formula $3 <_u x \wedge x \neq 2 \cdot 2 \wedge \forall y (x \neq 2 \cdot y)$ and its quantifier-free component is satisfiable with model $\mathcal{M}' = \{x \mapsto 5\}$ which is also a model of $\forall y.\ Q$, so ϕ is satisfiable with model $\mathcal{M}'$.

In Example 2.2.2, the efficiency of the solver depends on the models found for x. If the solver tried all even numbers greater than 3 (modulo 32) for possible values for x before trying any odd ones, it would take much longer to find a model of ϕ. Recent approaches have leveraged syntax-guided synthesis [3] to do quantifier instantiation [82]. In Chapter 6, we describe one such technique used by cvc5 for quantifier instantiation in the theory of bit-vectors, that helps overcome the dependence on the model-finder from Example 2.2.2. This technique is based on the validity of certain properties called invertibility conditions expressed over bit-vectors.

2.2.2 Proof Certificates

To increase reliability in their results, many SMT solvers are able to justify these results. A satisfying model is used to justify a satisfiable formula, and for unsatisfiable formulas, we define *proof certificates* in Section 5.1. Roughly, a proof certificate applies proof rules to (the negation of) the input formulas of an SMT solver and reduces them to $\langle\ \rangle$ (the empty clause), a standard form of unsatisfiability. Although all SMT solvers agree on the high-level aspects of what proof certificates should look like, there hasn't been a convergence on a proof certificate format for SMT solvers. One reason for this is that SMT solvers differ in their solving approaches. More importantly, they differ on the parts of solving that they justify (for example, CNF conversion and term-rewriting steps may be unjustified by some solvers), and on the granularity of detail in their proofs. Proofs generated by the z3 SMT solver [38], for instance, are more coarse-grained than the ones produced by cvc5 and veriT. Thus, various SMT proof formats exist, supported by different SMT solvers. CVC4 [14] emits resolution proofs via LFSC [94] (Logical Framework with Side Conditions), a framework that offers a dependent type theory as a language for describing proof rules, as well as a checker for these proofs. z3 uses a set of coarse-grained rules and an internal format over which it produces proofs [25]. The veriT solver's first proof format [18] was inspired by the SMT-LIB 2 standard for SMT solvers. Subsequent formats supporting more expressive rules have remained faithful to this motivation. The most recent iteration of veriT's proof format is the alethe proof format [88]. alethe implements a natural deduction style calculus for proofs of unsatisfiability. It contains extended rules that cover term rewriting steps done by SMT solvers and also provides support for quantified reasoning. Recently, cvc5 has also made advances in producing fine-grained proofs over multiple formats [10]. These include an internal proof format, a proof format checkable in the Lean theorem prover [39], an improved version of the LFSC format, and the alethe proof format. This makes alethe the first proof format supported by multiple SMT solvers.

2.3 Resolution Provers

Another set of popular automatic theorem provers (ATPs) are called superposition provers, or resolution provers [6]. These differ from SMT solvers in that they focus on proving conjectures rather than finding a satisfying model for a set of formulas. Although both these problems are duals of each other, picking one over the other makes a difference to the kinds of instances that become solvable, owing to the complexity of the problem space — the SAT problem is NP-complete and quantification and theory reasoning often lead to undecidability. The input problem to a superposition prover is formulated as a set of axioms that relate to the problem space, a set of assumptions, and a conjecture to prove. Whereas theories are built into SMT solvers, they need to be externally axiomatized for most resolution provers. As such, superposition provers are better suited for quantified formulas and minimal theory reasoning, whereas SMT solvers do well on problems that contain constraints in theories and quantified formulas slow them down. Within ATPs, our focus in this work is almost exclusively on SMT solvers.

A note on terminology: many consider resolution provers to be the only kind of automatic theorem provers and consequentially use the terms interchangeably. This thesis treats both SMT solvers and resolution provers as different types of ATPs and distinguishes them from ITPs (described below). To make matters worse, since we focus on SMT solvers, we often use the terms SMT solvers and ATPs interchangeably.

2.4 Interactive Theorem Provers

Interactive theorem provers (ITPs), or proof assistants are provers that have a small trusted computing base (TCB), especially in comparison to ATPs. They offer strong guarantees of properties proved within this TCB. A lineage of ITPs that can be traced back to Automath [36] implement the Curry-Howard isomorphism, where properties — stated as logical formulas — are also types and can be proven by constructing terms of the corresponding type. Some examples of such ITPs include Agda [77] and Lean [39]. The one that we use in this work is Coq [33]. Via so called *conversion rules* [45] a proof term in Coq can have multiple types as long as they are computationally equivalent. The Coq type-checker plays the role of its TCB's guarantor. While a user can provide a term of the right type to Coq as a proof, Coq offers an interface to construct proof terms via scripts called *tactics*. Tactics range from single, one-word invocations of previously proven theorems to complicated scripts involving nested case-splittings that may involve inductive reasoning. In this document, we use the `Goal` keyword to specify unnamed theorems and the `Theorem` keyword for named ones in Coq; a proof term is specified between the `Proof.` and `Qed.` keywords, and a failed proof is closed using the `Admitted.` keyword. Multi-line comments in Coq are enclosed within `(*` and `*)`.

When external tools are used for providing automation in an ITP, care must be taken so that the TCB is not extended. One way to ensure this is to re-implement and prove correct the external tool within the ITP [68, 53]; another is to use the external tool as a guide and reconstruct its proofs within the ITP [19, 35, 20]. Tools that perform such a reconstruction are called *hammers*. A third potential route is *computation reflection*, which allows the proof assistant to replace a proof by a computation. Such a computation can be driven by an external prover and requires: (*i*) a represention of the external prover's formulas in the ITP (in the case of Coq, one can use Coq's programming language, Gallina [64] to do this), and (*ii*) a proof of the correspondence between these represented formulas, and formulas in the language of the ITP. This method leverages Coq's conversion rules to replace a proof term by a computation over a certificate produced by an external prover. One of the earliest known tactics to use external SMT solvers in Coq via computational reflection is the `kettle` tactic [29] that is able to do reasoning over equality and linear integer arithmetic. In this work, we use a tool called SMTCoq [5] which also provides proof automation in Coq via computational reflection.

2.5 SMTCoq

SMTCoq is a Coq plug-in that enables a skeptical cooperation between Coq and external SAT and SMT solvers. SMTCoq requires the external solvers to produce proof certificates, so that their results can be validated via Coq's TCB. This is done using Coq's computational reflection capabilities to construct proof terms for goals using the certificates from the solver.

In contrast to the many-sorted first order logic used by SMT solvers, Coq is based on the Calculus of Inductive Constructions, a constructive higher-order logic with dependent types [80]. SMTCoq resolves this mismatch by considering only goals of the form $\forall l, b_1 = b_2$ where l is a list of sorted variables, and b_1 and b_2 are quantifier-free expressions of type `Bool` (a Boolean type defined in Gallina), as opposed to terms of type `Prop`, the designated type for formulas in Coq. It has a separate mechanism for lifting such `Bool` formulas to `Prop` formulas. SMTCoq offers multiple tactics that are described below.

Example 2.5.1. Coq's `Z.ltb`, representing the less than operator over the integers, has type `Z` $\rightarrow$ `Z` $\rightarrow$ `Bool` and infix notation `<?`. A formula in Coq stating that all integers are lesser than 100 can be stated using this operator as:

$$\forall \texttt{x : Z, (x <? 100) = true}.$$

We refer to a goal such as this one as using the less than predicate in its `Bool` *form*. SMTCoq's tactics can be invoked on such a goal (in this case, they will disprove the goal).

The majority of SMTCoq's machinery provides a way to computationally reflect a proof certificate from an external solver into a proof term in Coq of the correct type. In essence, this consists of a checker for these certificates, supported by many efficient data structures to improve scalability, and a proof of this checker's correctness in terms of Coq's logic. The correctness proof of the internal SMTCoq checker boils down to a theorem stating, intuitively, that if the checker accepts an external proof certificate as a proof of the validity of a Boolean term ϕ in first-order logic, then the proposition $\forall l, \phi = \texttt{true}$ (where `true` is $\top$ for the `Bool` type) is valid in Coq's logic.

The goals that SMTCoq can deal with are restricted to a subset of the first-order fragment of Coq's logic. The Sniper [23] tool relaxes this limitation. It is built on top of SMTCoq with the goal of proving more expressive goals. It achieves this via a modular set of transformations that can be applied to a Coq goal to make it suitable for an external solver to solve, and adding to Coq's computational reflection mechanism to prove these goals while staying true to Coq's TCB.

SMTCoq's tactics are considered push-button provers that can either succeed in proving the subgoal or fail. Therefore, interaction between the user and the external solver is limited. Currently, interaction comes only in the form of solver proofs with "holes" in them. When it encounters such holes in an external solver's proof, SMTCoq presents them as new subgoals to the user. In Section 4, we enhance SMTCoq with more interactive capabilities.

2.5.1 SMTCoq's Tactics

SMTCoq provides a suite of Coq commands and tactics. The commands, also called *vernacular commands* or *Coq vernacs*, can be invoked to use SMTCoq as a proof checker, and create theorems from proof files. SMTCoq's commands to invoke its proof checker are explained in Section 5.4. The tactics help in proof automation, and the relevant ones are described in what follows.

- The `verit_bool` and the `cvc4_bool` tactics respectively invoke the veriT and CVC4 SMT solvers on goals of the form $\forall l, b_1 = b_2$ where l is a list of sorted variables, and b_1 and b_2 are quantifier-free expressions of type `Bool`. We will call goals of this type *Boolean goals.*

- The `prop2bool` and `bool2prop` tactics can be used to change terms in the proof context between their `Bool` and `Prop` forms. The type of terms that can be changed this way are limited to the theories that SMTCoq supports and the predicates that the SMT solver supports within those theories. These tactics are implemented using the SSReflect library [58].

- The `verit` and `cvc4` tactics invoke the respective solvers on supported goals of type `Prop` (these include goals covered by `verit_bool` and `cvc4_bool`) by first converting them into their corresponding Boolean goals.

- The `smt` tactic, which is used most often in this document, combines the previous tactics to invoke a combination of CVC4 and veriT on either Boolean goals or other goals of type `Prop`.

Example 2.5.2. Example 2.5.1 introduces `Z.ltb`, the less than operator over integers that returns a `Bool`. Terms over `Z.ltb` are embedded in coq's `Prop` type by equating them to other terms of type `Bool` (usually `true` or `false`). Coq also offers the operator `Z.lt` with type `Z` $\rightarrow$ `Z` $\rightarrow$ `Prop` and infix notation `<`. We refer to a goal such as the one from Example 2.5.1 that uses `Z.ltb` can be stated as:

$$\forall \texttt{x : Z, x < 100.}$$

This goal is an example of the less than predicate being used in its `Prop` *form*. SMTCoq's `prop2bool` and `bool2prop` tactics allow the user to switch between such a goal and its corresponding `Bool` form, given in Example 2.5.1. In order to do this, the tactics use the following property:

$$\forall \texttt{(n m : Z), (n <? m) = true <-> (n < m).}$$

where `<->` is Coq's equivalence operator. In the case of the less than predicate, the Coq standard library provides this proof (lemma `Z.ltb_lt`). For other predicates such a property equating their `Bool` and `prop` forms is proved within SMTCoq.

In Chapter 4, we introduce a new addition to the SMTCoq tactics.

Chapter 3

Thesis Outline

Given that the lack of trustworthiness in ATPs and the lack of automation in ITPs can be addressed by each other's capabilities, this thesis proposes three integrations between ATPs and ITPs, each focusing on at least one of these shortcomings. The contributions in this document concern SMT solvers (the ATPs) and the Coq proof assistant (the ITP).

1. *External ATPs can be used to automate proofs of goals within ITPs.* The SMTCoq plug-in described in Section 2.5 is one such tool that is able to dispatch sub-goals from the Coq proof assistant to SAT and SMT solvers. We adapt SMTCoq to cvc5's abduction solver, offering an interactive tactic called `abduce` which allows the Coq user to query the SMT solver for missing hypotheses that might allow it to prove a currently failing goal. Chapter 4 introduces SMTCoq's `abduce` tactic, scenarios in Coq developments in which it may prove useful, and the underlying mechanism of abductive reasoning in cvc5.

2. *An ATP's result can be certified by checking it via an ITP.* To this end, SMT solvers produce proof certificates, as described in Section 2.2.2, whose correctness can be checked by an ITP. To reconcile the different certificate formats of different SMT solvers, SMTCoq has an internal proof certificate format, and a translator for each external format. We adapt SMTCoq to accept proofs in the `alethe` format, supported by both veriT and cvc5. We do this via a sequence of modular transformations of `alethe` proofs into SMTCoq proofs. The byproducts of this contribution are: (i) a checker for the `alethe` proof format in Coq and (ii) support for finer-grained SMT proofs in Coq including aspects such as rewrites and sub-proofs that weren't previously supported. Chapter 5 is dedicated to this contribution.

3. *An ATP can be certified by checking modular parts of its algorithm in an ITP.* Although proof production by SMT solvers has become common for quantified theories, SMT-Coq is limited in its ability to certify quantified reasoning by SMT solvers. Progress in quantified SMT reasoning is a recent phenomenon in the theory of fixed-width bit-vectors (which is supported by SMTCoq), owing to the development of theory-specific quantifier instantiation techniques specializing the general techniques discussed in Section 2.2.1. cvc5 uses instantiation by invertibility conditions, as described in Chapter 6,

and would benefit from the verification of these invertibility conditions, some of which are trusted without justification. We verify a subset of these previously unjustified invertibility conditions in Coq. This work increases trust in the correctness of the quantifier-instantiation techniques used by cvc5's bit-vector solver. Chapter 6 formalizes invertibility conditions, describes previous attempts at their verification, and presents our Coq proofs as a complement to these attempts.

Chapter 4

The `abduce` Tactic

This chapter introduces the `abduce` tactic, our extension to SMTCoq — a Coq plugin that invokes external SMT solvers to prove subgoals in Coq, without extending the ITP's trusted computing base (see Section 2.5 for more). SMTCoq offers a set of tactics that the user can invoke to call the external solvers on a goal. We add the `abduce` tactic which allows a Coq user to interact with the cvc5 SMT solver's abduction engine.

4.1 Premise Selection

Given a goal G in an ITP that we want to prove using an external ATP, *premise selection* is the problem of selecting the set $H = \{H_1, \ldots, H_n\}$ of hypotheses or premises from the ITP's environment. Naturally, the goal of premise selection is to optimize this set H for both success and time. In other words, we want to pick H such that the ATP can prove $H \models_T G$ as fast as possible. However, a balance must be struck between the two optimization goals. In theory, an obvious strategy to optimize for success would be to send all possible facts from the ITP's environment. For any decently sized proof development, this strategy would most likely fail by overwhelming the external ATP with constraints. Similarly, sending no facts to optimize for time would fail for a development of considerable complexity since provable goals would likely depend on hypotheses not known to the external prover.

Hammers — tools used by ITPs to integrate external provers — have three possible integrations with external ATPs.

1. The ATP is used as an oracle for automating proofs, which is undesirable since this would extend the TCB of the ITP significantly. Using an ATP to disprove theorems is generally more acceptable since it is incapable of introducing unsoundness. Counterexample generators such as Nunchaku [34] and Refute [95] operate in this fashion. Notice that even in this safer integration, the premise selection problem persists. A theorem can only be disproved by an external solver if it can be argued that it had all the premises necessary to prove the goal and still found a counter-example.

2. An external ATP is used as a *relevance filter* for an internal prover. The entire proof is produced by the internal prover that is within the TCB of the ITP. The only role

that the external ATP plays is to reduce the set H from which G can be proved so that the internal prover is not overwhelmed by facts. An SMT solver performs this reduction via an *unsatisfiability core* — a subset of all the facts from the input that are still unsatisfiable (recall that proving the validity of a formula is equivalent to proving the unsatisfiability of its encoding).

3. A proof-producing external ATP is used to automate proofs within the ITP. The proof produced by the ATP is *reconstructed* within the ITP. Each step in the proof is reproved within the ITP using internal tactics or decision procedures, thus preserving its TCB.

Note that the ATP does not completely replace the premise selection process in any of its three possible integrations with an ITP. An initial premise selection strategy is still necessary because libraries of typical ITPs are huge, and any ATP would fail on the entirety of such a library. Thus, a premise selection strategy is used to select a few hundred relevant hypotheses to send to the ATP along with the goal, and in a successful invocation, the ATP further reduces these hypotheses so that the entailment can be internally proved. CoqHammer [35], a hammer for Coq offers integrations with external provers both via integrations 2 and 3. It uses machine-learning algorithms to select its premises before calling the ATP. It offers two tactics: `sauto` that tries to automate the proof of a goal without invoking any external provers; and `hammer` which when triggered on some Coq goal, (i) submits the goal together with a few facts gathered by its premise selection strategies to external provers, (ii) attempts to reconstruct a returned proofs (if one exists) directly in the Coq tactic language Ltac [41], and (iii) outputs the set of tactics closing the goal in case of success.

The HOL family of theorem provers (HOL [59], HOL Light [63], Isabelle/HOL [76]) follow the LCF approach (originating from the LCF system [60]) to theorem proving where a theorem is implemented as an abstract data type (ADT) in the ML programming language [70], and the only way to create theorems within the system is through a set of functions over the ADT that correspond to axioms and inference rules. So, while the LCF systems also reduce proof checking to type checking, they differ from the Curry-Howard approach (used in Coq) by their usage of an ADT. Isabelle/HOL, one of the most popular variants of this family of theorem provers, is a proof assistant for higher-order logic offering multiple means of automation to the user. One can invoke Metis [65], Isabelle/HOL's internal theorem prover, which is capable of proving theorems without extending its TCB. Moreover, Metis is instrumented to be able to perform reconstruction of external proofs. External solvers can be invoked in Isabelle/HOL via the Sledgehammer [81] tool. Sledgehammer offers all three of the possible integrations mentioned above [81, 19]. Initially, it was built only to work with resolution provers (see Section 2.3), but was later integrated with SMT solvers due to their ability to solve from the set of problems that is complementary to those generally solvable by resolution provers [19]. Its standard mode of operation is to call both resolution provers and SMT solvers parallelly on a particular goal along with some premises, and then to use Metis to reconstruct these proofs (integration 3). So the output of Sledgehammer is essentially a list of calls to Metis to prove each step in the proof. As support for more complex proofs by external provers was added, reconstruction began relying on other internal provers in addition to Metis (for example, support for bit-vectors needed additional tactics [24]). For premise selection, Sledgehammer relies on heuristics and machine learning algorithms based on the symbols shared between the goal and the facts in Isabelle/HOL's libraries.

4.2 Abduction for Premise Suggestion

Current methods for premise selection are implemented on the ITP side of the integration so that a large set of facts within the ITP's library can be reduced to a smaller set of facts which can either be used by the external solver or further reduced by it. In other words, given some goal G, the problem of finding the set H of hypotheses needed to prove G such that $H \models_T G$ is recast as the problem of reducing some large set of facts L to a set H such that an ATP (internal or external) can prove $H \models_T G$. This reduction is often performed by hammers using heuristics and possibly machine-learning techniques.

Example 4.2.1. Suppose our Coq development contains a binary function `f` of type `Z` $\rightarrow$ `Z` $\rightarrow$ `Z` (where `Z` is Coq's integer type), and many facts about `f`. We can invoke SMTCoq through the `smt` tactic as follows. An external solver will consider `f` to be uninterpreted, but the solver can still successfully prove certain kinds of goals.

```
Goal forall (x y : Z), x = y -> f x 1 = f y (21 - 20).
Proof. smt. Qed.
```

In the previous example, the external solvers (veriT and CVC4) successfully manage to prove the goal by using equational and arithmetic reasoning, along with basic properties of functions.

Example 4.2.2. Now, consider a more interesting goal that depends on the specific behavior of `f`.

```
Goal forall (x y z : Z), x = y + 1 -> (f y z) = f z (x - 1).
Proof. smt.
```

The external solvers fail to prove this goal, and return a counterexample witnessing the invalidity of the goal, shown to the user as the assignment:

$$\{\mathtt{f} \mapsto \lambda\ \mathtt{x}, \mathtt{y} \rightarrow \mathtt{x}, \qquad \mathtt{x} \mapsto 1, \qquad \mathtt{y} \mapsto 0, \qquad \mathtt{z} \mapsto 1\}$$

Here, we use the λ notation to define a function with arguments x and y, that simply returns x.

It is possible that the solver failed because the goal is indeed invalid. However, considering that the solver does not have access to a definition of `f` or an axiomatization of its properties, it is more likely that the solver is missing one or more of those additional facts from Coq as hypotheses for the goal. The hammer approach would be to run an efficient premise selection strategy on all available lemmas in Coq (including the ones about `f`).

An alternative approach is to look at the problem as one of hypothesis finding: given a goal G, find an H such that $H \models_T G$. Solvers capable of performing abductive reasoning can thus be used to try and solve this problem symbolically. We use cvc5's abduction solver in this manner so that the solver can be more directly involved in the premise selection process. In other words, the solver can perform *premise suggestion* as an alternative to the integration tool's premise selection. Consider again the goal from Example 4.2.2 on which the external solver fails. Instead of either reducing the facts about `f` or trying to derive them through the given counterexample, the user may invoke cvc5's abduction capability to get a suggestion from the solver. This may be invoked via the `abduce` tactic as demonstrated next in Example 4.3.1.

A note on the usage of CVC4 and its successor cvc5: SMTCoq supports external SMT solvers CVC4 and veriT in their deductive capabilities. The proofs produced by cvc5 are not backwards compatible with those produced by CVC4 due to which SMTCoq cannot use cvc5 as one of its external solvers to automate proofs in Coq. The contributions presented in Chapter 5 are a step in the direction of integrating SMTCoq with cvc5 using a proof format supported by the latter. To use an external abduction solver, SMTCoq does not need it to be proof producing. Therefore, SMTCoq uses CVC4 deductively and cvc5 abductively.

4.3 The `abduce` Tactic

We now introduce our addition of the `abduce` tactic to SMTCoq which enables the Coq user to interact with the SMT solver when the latter fails to prove a goal. Consider Example 4.2.2 where this is the case. As mentioned in Section 4.2, the more likely cause of failure by the external solver is the underspecification of `f` from the solver's point of view. In such a situation, the user may invoke cvc5's abductive capability to get a suggested premise from the solver.

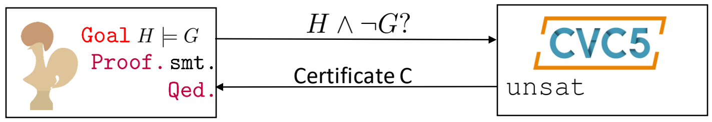
Given the goal of proving the entailment $H \models_T G$ (for some theory T), SMTCoq encodes H and G respectively as SMT formulas H' and G', phrases the entailment between them as an implication, and sends the negation of this implication to the SMT solver. Thus, the goal of showing that $H \models_T G$ holds is converted to that of showing that $H' \wedge \neg G'$ is T-unsatisfiable. For any particular Coq goal supported by SMTCoq, and sent to the SMT solver, there are three possible outcomes:

(i) the solver proves the goal, by finding $H' \wedge \neg G'$ to be T-unsatisfiable;

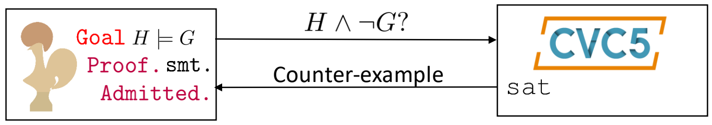
(ii) it disproves the goal, by finding $H' \wedge \neg G'$ to be T-satisfiable;

(iii) it produces an "unknown" answer for having run out of resources.

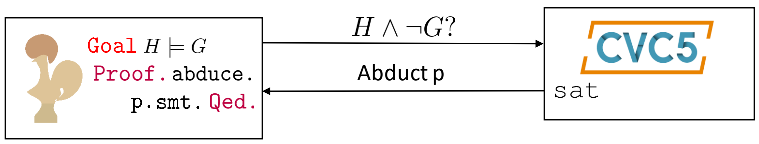
An acceptable certificate for outcome (i) is a proof of unsatisfiability — the SMT solver produces a formal proof that derives $\langle\,\rangle$ (the empty clause) from $H' \wedge \neg G'$ (explained in Chapter 5). An acceptable certificate for outcome (ii) is a *counterexample*, a valuation of the (free) variables of $H' \wedge \neg G'$ that satisfies H' and falsifies G'. Example 4.2.1 is an illustration of outcome (i), and Example 4.2.2 demonstrates outcome (ii) (where H would be just *True*). Figures 4.1a and 4.1b show the interaction between Coq and the SMT solver for both situations.



(a) cvc5 finds query to be unsatisfiable



(b) cvc5 finds query to be satisfiable



(c) cvc5 returns an abduct

Figure 4.1: Interaction of SMTCoq with the SMT solver. $H = \{H_1, \ldots, H_n\}$ is the set of hypotheses sent to the solver.

An important caveat to outcome (ii) from above is that the solver disproves the goal *in its current context (H')* by finding $H' \wedge \neg G'$ to be T-satisfiable. As suggested previously, it is a likely possibility that the solver's context is underspecified (in other words, H' is missing some relevant facts from the ITP). It is this possibility that our new `abduce` tactic in SMTCoq attempts to address. With the `abduce` tactic, a Coq user can ask cvc5 for abducts that would entail a currently failing goal. An integer argument allows the user to request a particular number of independent abducts — with each abduct separately entailing the goal (equivalently, with the disjunction of all abducts entailing the goal) along with the hypotheses. The tactic invokes cvc5's abduction solver which we will elaborate on in Section 4.4.

Example 4.3.1. We can use the `abduce` tactic on the goal from Example 4.2.2, since `smt` fails.

```
Goal forall (x y z : Z), x = y + 1 -> (f y z) = f z (x - 1).
Proof.
(* smt. *) (* Commented out because it fails with a counter-example *)
abduce 3. (* Temporarily added, to get candidate hypotheses *)
```

This presents three abducts to the user: `z = y`; `z + 1 = x`; and `f z y = f y z`. The third abduct tells the user that cvc5 would prove the goal if it was told that the function `f` is commutative over `z` and `y`. If one of the previously proven facts about `f` is

$$\texttt{comm\_f} \;:\; \forall\ \texttt{m n, f m n = f n m}$$

the user can easily instantiate it in Coq for the necessary variables. A subsequent call to the `smt` tactic, with this instantiated fact in scope would successfully close the proof.

```
Goal forall (x y z : Z), x = y + 1 -> (f y z) = f z (x - 1).
Proof. intros. assert (f z y = f y z). { apply comm_f. } smt. Qed.
```

The `intros` tactic introduces `x`, `y` and `z`, and the hypothesis `x = y + 1` into the scope of the proof. `assert` is a way to locally introduce a fact into scope, and we use it to state the chosen abduct. The abduct is easy to prove by an application of `comm_f`. At that point, the `smt` tactic can successfully prove the current goal `(f y z) = f z (x - 1)` from the (automatically collected) local hypotheses `x = y + 1` and `f z y = f y z`.

Internally, a call to `abduce` is composed of:

1. A call to CVC4 to confirm that the goal in its current context is disproved by the external solver. This call is in non-proof mode, meaning CVC4 isn't made to produce a proof certificate. If the goal is proved by the external solver then the tactic call ends by suggesting to the user that they use the `smt` tactic instead.

2. If CVC4 is able to disprove the goal (find its negation to be satisfiable) in the current context, the abduction solver from cvc5 is called, asking it for the specified number of abducts; these are then printed out to the user as Coq formulas which can be asserted verbatim and then proved locally.

The tactic always fails with an error message — if it is called on a goal that is provable by an external solver, the error message indicates to the user that it cannot find abducts and that the user should consider using a deductive SMTCoq tactic. Even in its successful invocation, the `abduce` tactic returns the abducts to the user via an error message. That is intentional as this is a tactic that does not change the state of the current proof in any way. And so the only difference it makes to a proof development is to add messages that will be printed out to the user. As a result, their invocations should be removed after they have served their purpose, so they don't clutter the output of the proof development.

We point out that in cases where SMTCoq disproves the goal (outcome (*ii*) above), the `abduce` tactic can provide a more general explanation of the failure than a counterexample. Counterexamples are single points over which the entailment $H \models_T G$ fails whereas an abduct represent a general sufficient condition for the provability of the goal that the user might be able to fulfill using the current Coq context. Since there are a large number of additional hypotheses that might help in proving a given goal, it is impractical to send all of them along with the goal. Abduction is then a way for the SMT solver to tell the user what else it needs. Figure 4.1c illustrates this case.

4.4 Abduction in cvc5

cvc5 performs syntax-restricted abduction [84] via syntax-guided synthesis (SyGuS) [3]. As defined in Section 2.2, given a background theory T, a context-free grammar R and a set of hypotheses H and a goal G such that $H \not\models_T G$, the problem of syntax-restricted abduction is to find a formula A such that (i) $A \wedge H$ is T-satisfiable; (ii) $A \wedge H \models_T G$; and (iii) A is generated by R. The grammar input is optional, and the solver defaults to the grammar that generates the entire language of T.

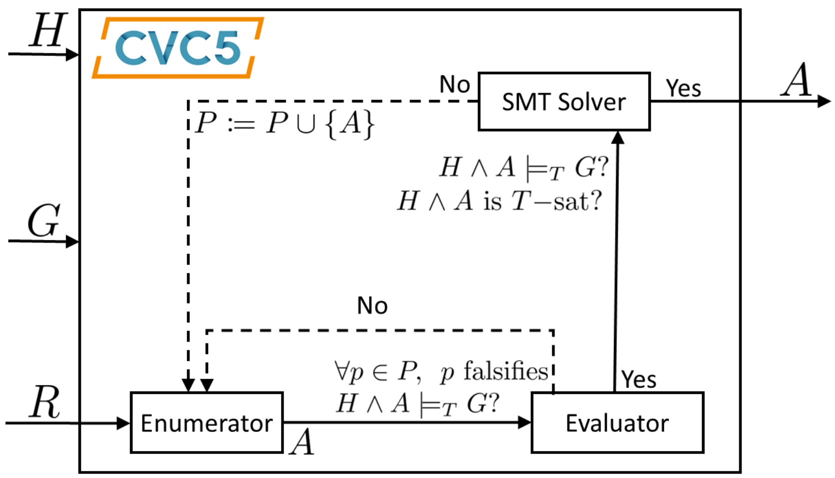


Figure 4.2: CEGIS procedure driving cvc5's abduction solver.

The solver is driven by a basic CEGIS [91] procedure (as depicted in Figure 4.2) where candidate abducts A, formulas generated from R that satisfy the consistency requirement (i) above, are validated by checking whether $H \wedge A \models_T G$. Valuations that invalidate this entailment (by satisfying $H \wedge A$ and falsifying G) are collected (in P) and used to guide the search of a solution: future candidates A that are satisfied by any of those valuations are immediately discarded as they are guaranteed to fail the entailment check. cvc5 refines this basic CEGIS procedure with various optimizations and symmetry breaking strategies that eliminate redundant solutions and aim at producing solutions quickly.

cvc5 generates a *sequence* of abducts for the same problem so that their disjunction is typically weaker in T than the individual abducts. This has the effect of producing an increasingly general (disjunctive) abduct at the cost of additional computation. This cost can be controlled by the user by specifying the length of the abduct sequence. Each separate disjunct is also a satisfactory solution to the abduction problem.

Related Work Several tools which perform abductive reasoning have also been developed over the years. Echenim et al. [49, 47] modify the superposition calculus to present an abductive algorithm for prime implicate generation in the theory of equality. In supporting all of cvc5's underlying theories, the cvc5 abduction solver comparatively supports a much wider range of applications. It is also different in that it is built on top of SMT technology. Other abduction solvers that use SMT solvers are GPiD [48] and EXPLAIN [43]. GPiD uses CVC4, Z3, and Alt-Ergo [30] and like cvc5's abduction engine, supports a wide range of theories. An important operational difference is that GPiD requires as input a set of literals over which abducts can be generated. Although analogous to the grammar parameter to the cvc5 solver, the latter can be omitted (in fact, the interface of the `abduce` tactic is such that the solver must always consider the default grammar). With `abduce`, we prioritized building a tactic whose invocation would require the minimal amount of effort from the Coq user. Future iterations of the tactic might offer to the user a means to customize the solution space. EXPLAIN uses the Mistral [44] SMT solver which only supports the EUF and LIA theories. Another similar tool is CAPI [62] which uses abduction in descriptive logics to provide explanations for observations that do not hold.

4.5 Evaluation

We performed multiple experiments to evaluate the `abduce` tactic's usefulness as a companion to the `smt` tactic. We used lemmas proven by standard Coq tactics as a baseline and tried to recreate these proofs, either in parts or in their entirety, using SMTCoq's automation tactics. We used a selection of proven lemmas from the Coq standard library [32] to evaluate `abduce` on. The Coq standard library has well-documented proofs over a breadth of mathematical and logical areas. Since it is included with Coq distributions, it is well maintained by the Coq community, and its proofs have been improved over multiple iterations by Coq developers, experts and researchers. We searched these libraries for steps within proofs that could be replaced by a call to SMTCoq's deductive tactics; in cases where these failed, we called the `abduce` tactic to test if it could help us recreate the proof step in question (in an interactive fashion, as illustrated in Section 4.3). In some of the experiments, the approach was reversed — we explicitly looked for lemmas used by the proof and treated them as candidates for abduction; removing such a lemma from a proof would most likely lead the deductive tactic to fail, and necessitate the use of the abductive one. In what follows, we present our evaluation of the `abduce` tactic on the proofs from three (sub-)libraries in the Coq standard library.

4.5.1 Experimental Setup

In the following, we describe three different sets of experiments we ran on the `abduce` tactic. While we emphasize the results of the `abduce` tactic from these experiments, we are testing it in conjunction with the `smt` tactic. In the first experiment we do this explicitly — a call to `smt` is followed by a call to `abduce` if the former fails. In the subsequent experiments, although we call only the `abduce` tactic, a deductive call to the SMT solver is implicit. We expect this call to fail (the solver to disprove the goal), but if it is able to prove the goal the `abduce` tactic lets us know, so we can add an explicit call to `smt`.

We ran all experiments on CoqIDE version 8.13.2 in a system with 16 GB RAM, running Ubuntu 20.04.[1] To call either of SMTCoq's tactics from within a library file, a command importing SMTCoq as a plug-in is added to the file. All three experiments identify goals (also called test units) to test SMTCoq's tactics on. These could be entire proofs of lemmas,

[1]Instructions and resources needed to reproduce our experiments can be found at `https://github.com/arjunvish/smtcoq/blob/thesis24/INSTALL.md`

or smaller parts of larger proofs. The result of running SMTCoq's tactics on a goal is an `smt` *success* if the goal can be fully solved by the SMT solver (with no additional hypotheses). Such a success may be achieved through either one of SMTCoq's deductive tactics (`smt`, `verit`, `cvc4`, etc.), but we count them towards the success of `smt` for convenience. In cases where the solver (through `smt`) finds a goal to be invalid, we repeatedly call `abduce` n with $n = 1, 2, 3 \ldots$ and a 20 second timeout per call, stopping as soon as we find a suitable abduct or when the solver times out for some n. Recall that `abduce` n asks for n abducts, each of which independently entail the goal. We classify the goal as an `abduce` *success*, if a call to `abduce` produces (within the given timeout) an *easily provable* abduct which, once added locally, allows `smt` to prove the goal. Otherwise, we classify the goal as a *timeout*. The criteria for classifying an abduct as easily provable differs for each experiment.

Our first experiment set is described in Section 4.5.2 and it employs an automated experimental set-up, where we replace the entire body of the proof to try and abduce it. This is an ambitious goal, as proved by the fact that we searched the entire standard library and a large collection of (around 71,000) Coq proofs called CoqGym [96] for candidate tests (within our supported theories). However, we ended up finding only one library where the `abduce` tactic was useful. This is not surprising as we expect the tactic to be more useful in automating smaller parts of large proofs, rather than proofs in their entirety. So we switched our experimental setup to reflect this expectation in our other two experiment sets. Section 4.5.3 describes our experiments on Coq's standard list library, and Section 4.5.4 details the experiments involving lemmas about multiplication over integers used in the Coq standard libraries.

4.5.2 `Zorder`

Here, we present a case study on applying the `smt` and `abduce` tactics in the Coq library `Zorder` [66], with the goal of automating the proofs in it. The library contains theorems about order predicates over Coq's `Z` (integer) type. While this library is deprecated, its lemmas are still available in the Coq core libraries. The `Zorder` library stands out in the Coq standard libraries due to its short proofs, as a consequence of which SMTCoq's tactics could in principle automate entire proofs from it.

Our study demonstrates the utility of the `smt` tactic and provides a proof-of-concept use case for interacting with the SMT solver via the `abduce` tactic in an IDE for Coq. Our experimental setup is as follows. Within the `Zorder` Coq file, we import SMTCoq as a plug-in, and for each goal, we first try the `smt` tactic, which attempts to solve the goal using a combination of the SMT solver CVC4 [14] and veriT[27], both of which are well integrated in SMTCoq. `smt` classifies the goal as an SMT success, or an invalid goal. In case of the latter, we call `abduce` to find a suitable abduct within a 20 second timeout. We classify the goal as an `abduce` *success*, if a call to `abduce` produces (within the given timeout) an *easily provable* abduct which, once added locally, allows `smt` to prove the goal. Otherwise, we classify the goal as a *timeout*. For the purposes of this experiment, we consider an abduct to be easily provable if it is provable from the Coq context by just unfolding once any applications of the

```
Lemma Znot_le_succ n : ~ Z.succ n <= n.
Proof. (* time abduce 1. *)
(* The solver finds the goal to be invalid; the abduce call runs
for 0.278 secs and returns the abduct n + 1 <= (Z.succ n) *)
assert (n + 1 <= (Z.succ n)). { unfold Z.succ. smt. } smt.
Qed.
```

(a) Example goal proven using `smt` and `abduce` where ∼ represents logical negation in Coq.

```
Lemma Znot_le_succ n : ~ Z.succ n <= n.
Proof. (* time abduce 1. *) unfold Z.succ. smt. Qed.
```

(b) An alternative interaction with abduction

Figure 4.3: Interactions with SMTCoq using the `abduce` tactic.

integer successor or predecessor functions (`Z.succ` or `Z.pred`, respectively) in the abduct.[2]

The experimental results are presented in Figure 4.4. From the 93 goals in the file, 30 goals contain non-linear arithmetic, a theory currently unsupported by SMTCoq; 3 goals are inexpressible in any known SMT theory; and 1 contains predicates unrecognized by SMTCoq. From the remaining 59 goals, we found 33 (55.9%) `smt` successes, and 26 candidates for abduction, over half of which were `abduce` successes.

Most goals found invalid by the SMT solver were so because they contained either Coq's integer successor or integer predecessor functions, `Z.succ` and `Z.pred`, which SMTCoq encodes as uninterpreted functions in the translation to SMT. When successful, the abduction solver was able to suggest either (correct) definitions of `Z.succ` and `Z.pred`, or properties satisfied by them in Coq. Both forms of abducts can be proven locally by unfolding the definitions of those functions, and applying some basic properties of inequalities over integers. We further automate this process by calling `smt` on the unfolded sub-goal. For example, consider goal `Znot_le_succ` in Figure 4.3a. `time` is used to output the duration of the tactic being run, along with its regular output. The tactic `abduce` is designed to fail when it successfully finds the abducts and to print the abducts as part of its error message. The call to the tactic is commented out in the figure. We report its output in a comment as well. An alternative way to view this tactic is presented in Figure 4.3b. The SMT solver fails to prove the goal as given, but the abduct returned by the `abduce` tactic suggests that all the user needs to do in this case is to unfold the definition of `Z.succ`.

Admittedly, this simple example may not seem very compelling since the user might have guessed from the start that the definition of `Z.succ` is needed for the SMT solver to prove the goal. Moreover, there is an alternative automated solution provided by the Sniper plug-in [23] whose `snipe` tactic is able to identify function definitions relevant to the goal and send them to the SMT solver. However, for more complicated functions, providing hypotheses capturing relevant properties of the function, as in the case of function `f` from Example 4.3.1, may be more effective than providing their definitions since proving such properties in the

[2] A more principled experiment would use a less stringent notion of easily provable formula.

Goals	smt **successes**	**Invalid goals**	abduce **successes**	**Timeouts**
59	33	26	16	12

Figure 4.4: Summary of results of using `abduce` in `Zorder`.

external prover may require inductive reasoning, something SMT solvers are not generally capable of. The following subsections present examples of such functions that the abduction solver has to reason about. So the `abduce` tactic can be seen as a complement to `snipe` in helping the user prove goals. Although we allowed the tactic 20 seconds to find a useful abduct, all 16 successful calls were made within 4 seconds. In fact, over half of them took less than 1 second. Note that there were 14 successful goals but 16 invocations of `abduce` because two goals required multiple calls, one for `Z.succ` and one for `Z.pred`.

Using the same test set, we also confirmed some of our hypotheses about the default grammar to provide, and the configuration with which to call the abduction solver. The first was to remove logical disjunction and the if-than-else (ITE) operator from the grammar. Such operators are not crucial since the user can recover disjunctive information by asking for more than one abduct. We found that eliminating these operators yielded more successful abducts. Second, we tested the ability of cvc5's abduction solver to generate conjunctive solutions quickly through unsat-core learning [84]. We found that, although the solver was much faster in generating solutions with this configuration, in almost all cases, at least one of the conjuncts was too specific, rendering the entire solution useless as it was not entailed by the Coq context. For instance, with this option enabled, one of the abducts for `Znot_le_succ` from Figure 4.3a is (`&&` denotes conjunction):

```
n <= (Z.succ n) && (not (Z.succ n) = n) && (Z.succ -2) = n && n = -1
```

We can see that the first conjunct is a useful abduct in isolation, whereas the full conjunction clearly does not hold for the successor function.

4.5.3 `List`

In the experiments over `Zorder` from Section 4.5.2, we regard the entire proof of each lemma as a possible *test unit* — a proof that can be completed by the SMT solver after the abduction solver finds a suitable abduct. However, due to Coq's logic — the calculus of inductive constructions — libraries often have inductive types and theorems over inductive terms. An inductive type is defined using constructors – functions that produce terms of the type – and is conducive to proofs by induction, where a theorem over an inductive term is proven by proving it for multiple cases, one for each constructor of the corresponding type, and by using proofs over smaller (in a particular ordering) subterms in the proving of larger ones. Although cvc5 does have support for the theory of inductive types [15], it is not proof producing for it and is therefore not integrated with SMTCoq for this theory. As a consequence, SMTCoq's tactics will most likely fail on proofs that would require induction (unless the inductive type is natively supported by the solver and integrated with SMTCoq, as is the case with integers). So large inductive proofs cannot be fully automated by an external SMT solver; they still show room for automation, however. Particularly, within the case for each constructor, these proofs often perform deductive steps. We try to identify such deductive steps as candidate test units for `abduce`. With these goals in mind, we perform our experiments on the Coq standard `List` library [1].

In these experiments over `List`, the `smt` tactic often does not work straight out of the box for the following reasons:

1. As mentioned above, external SMT solvers aren't integrated with SMTCoq to natively support proofs over lists (which are inductively defined in Coq). Instead, SMTCoq converts lists into uninterpreted types, and functions over lists into uninterpreted functions. The SMT solver often does not have enough information about these types and functions to reason about them.

2. While sub-goals within proofs about lists often involve arithmetic, especially the ones over the length of the list, the arithmetic is over Coq's natural number type rather than the integer type. Natural numbers are currently not supported by SMTCoq (although such support is being developed by others) and so the SMT solver must reason considering natural numbers and functions over them to be uninterpreted.

These factors make the library a good candidate to test the `abduce` tactic on. One way to account for the fact that most types and function symbols are going to be uninterpreted for the SMT solver is to axiomatize them within the SMT solver. Axioms would take the shape of quantified formulas, which SMTCoq has limited support for. Instead, we try to get the SMT solver to suggest premises that it needs to prove lemmas about the uninterpreted symbols. Moreover, abducts are quantifier-free. Thus, when successful, the solver would be suggesting instantiations of axioms, which are well supported by SMTCoq.

To find test units for the `abduce` tactic, we considered all sites inside proofs in `List` where a lemma was invoked to prove a step using either the `rewrite` or the `apply` tactic. A lemma is a named formula from either within the local context of the current proof (in which case, it is a local lemma); or, from the proofs in the current file that were closed before the current proof; or, from one of the imported libraries/files (the latter two constituting its global context). Such lemmas are most often invoked in Coq using the `rewrite` tactic that rewrites terms in the proof context, or the `apply` tactic that is used to prove a (sub-) goal using a lemma.

Example 4.5.1. Consider the lemma `app_nil_r` from Figure 4.5. `list` is the parametric type of inductive lists in Coq, and `l` has type `list A`, where `A` is a type variable representing an arbitrary type.

A `list` is an inductive type that can be constructed either by (*i*) `nil` (`[]`), the constructor representing an empty list, or (*ii*) `cons` (`::`), the constructor representing a non-empty list constructed from some element `h` (of type `A`) and some list `t` (`h::t` or `cons h t` is the constructed list).

The `++` or `append` operator appends two lists, and `app_nil_r` states that appending the empty list to an arbitrary list `l` results in `l`. The local proof context in the proof at the

```
1   Theorem app_nil_r : forall (l : list A), l ++ [ ]= l.
2   Proof.
3     induction l.
4     + simpl. reflexivity.
5     + simpl. rewrite IHl. reflexivity.
6   Qed.
7
8   Theorem app_nil_r_2 : forall (l : list A), l = l ++ [ ].
9   Proof.
10    symmetry. apply app_nil_r.
11  Qed.
```

Figure 4.5: Proof of lemmas `app_nil_r` and `app_nil_r_2` demonstrating inductive proofs and the usage of the `rewrite` and `apply` tactics.

beginning of line 3 contains the entire goal `forall l, l ++ [] = l` with no hypotheses. The `induction` tactic invokes inductive reasoning on `l` splitting the goal into two inductive cases, one for each constructor of `list`. The `+` symbol indicates the beginning of the proof of each case:

- In the *base case* (line 4) of the proof where `l` is constructed using the `nil` constructor, the goal is to prove `[] ++ [] = []`. The `simpl` tactic can simplify `[] ++ []` to `[]` leaving a goal that can be proved using `reflexivity`.

- In the *inductive case* (line 5) of the proof corresponding to the `cons` constructor, the following represents the local proof context:

  a: `A` l: `list A`   `IH:` l `++ []` $= l$

  `Goal:` $(a :: l)$ `++ []` $= a :: l$

  The way to read this is that given proof term of type l `++ []` $= l$ (the *inductive hypothesis* `IH`), the goal is to prove $(a :: l)$ `++ []` $= a :: l$. The application of `simpl` further simplifies the goal to $a :: l$ `++ []` $= a :: l$. The `rewrite` tactic takes the proof of an equality as its argument; pattern-matches for the left-hand-side of the equality, and replaces it with its right-hands-side if it can find a match (it fails otherwise). Here, its application further reduces the goal to $a :: l = a :: l$, which can now be closed using `reflexivity`.

The proof of `app_nil_r_2` simply uses `app_nil_r`. The `apply` tactic takes a lemma as argument and applies it to reduce the goal if it can pattern-match the two. `symmetry` changes the goal from $l = l$ `++ []` to l `++ []` $= l$ so that `app_nil_r` can be applied.

We treat every occurrence of `apply` and `rewrite` as a potential test unit for the `abduce` tactic. That is, we consider the proof before this step and see if the abduction solver can reproduce either the step or a different step that is available to use from the global context

Goals	`smt` **successes**	**Invalid Goals**	`abduce` **successes**	**Timeouts**
122	25	97	28	69

Figure 4.6: Summary of results of using `abduce` in `List`.

of the proof. We assume that removing the step would cause SMTCoq's deductive tactics to fail, but this need not be the case as can be seen from the results below. We found that successes often occurred when the lemma in consideration was being invoked at the end of a proof or of a case within a proof. This makes sense — if the call to a lemma is followed by a call to another lemma, an invocation of `abduce` before the first lemma is asking the solver to abduce both lemmas conjunctively. Section 4.5.2 discusses the cvc5 abduction solver's limitations in producing conjunctive solutions. In light of this, we modify the experimental setup to make each test site independent of any lemma invocations that follow it: suppose x is the lemma that we are testing for; we collect all lemmas Y whose invocations follow that of x until the end of the proof or case that the call to x is a part of; we assert all lemmas in Y and then call `abduce` to see if it can find x. Now we are testing the ability of the abduction solver to find only x rather than the formula conjoining x with all formulas in Y. This modified experimental set-up increased the number of successes by 10. The results from the experiment set (using this modified experimental setup) are summarized in Figure 4.6.

From the 122 goals that represented possibly automatable calls to a lemma via the `rewrite` or `apply` tactic, 25 (20.5%) are deductively solvable by the SMT solver, leaving 97 candidates for testing `abduce`, over which the tactic succeeds on 28 (28.9%). The `abduce` tactic took longer than 2 seconds to find a useful abduct in only a fourth of these 28 successful invocations. We define a successful invocation of the `abduce` tactic as one that can suggest to the user a formula that is entailed by the Coq (global) context at the location in the proof from where the tactic is being invoked and which once available to it, will allow the SMT solver to solve the goal. The proof term for the formula (if it exists) can be found using Coq's `Search` command, which allows the user to look up lemmas from the global context. The `Search` command allows the user to generalize the formula being searched for by using wildcard entries for variables. So when the `abduce` tactic prints a successful abduct, the user can automate the goal by:

1. locally asserting the goal verbatim using the `assert` command;

2. searching for a generalized version of the abduct using the `Search` command within the local sub-proof; and

3. if found, applying the lemma to close the local goal.

Once the abduct is in the local context, the `smt` tactic can be called to prove the goal.

Example 4.5.2. Consider the proof of `firstn_rev` from Figure 4.7a where `firstn n l` is a list that stores the first `n` elements of list `l` (if the list does not have at least `n` elements, it stores the entire list); `rev l` is the reverse of list `l`; `skipn n l` stores all elements in the list after the first `n` ones; `length l` is the number of elements in list `l`. After line 3 that

```
Lemma firstn_rev: forall x l, firstn x (rev l) = rev (skipn (length l -x) l).
Proof.
  intros x l.
  rewrite firstn_skipn_rev.
  rewrite rev_involutive.
  rewrite rev_length.
  reflexivity.
Qed.
```

(a) Proof of lemma `firstn_rev` from the Coq `List` library

```
Lemma firstn_skipn_rev: forall x l,
firstn x l = rev (skipn (length l -x) (rev l)).

Lemma rev_involutive : forall l:list A, rev (rev l) = l.

Lemma rev_length : forall l, length (rev l) = length l.
```

(b) Lemmas used in proof of `firstn_rev`

Figure 4.7: An example from the `List` library of `firstn_rev`.

introduces the variables into the local context, the local proof context is:

$$\frac{x:\ \texttt{nat}\quad l:\ \texttt{list A}}{\texttt{Goal:}\quad \texttt{firstn}\ (x)\ (\texttt{rev}(l)) = \texttt{rev}(\texttt{skipn}\ (\texttt{len}(l) - x)\ (l))}$$

where each function represents the Coq function of the same name except `len` for `length`. Rewriting `firstn_skipn_rev` (which is stated along with the other lemmas used by the proof in Figure 4.7b) changes the context to:

$$\frac{x:\ \texttt{nat}\quad l:\ \texttt{list A}}{\texttt{Goal:}\quad \texttt{rev}(\texttt{skipn}\ (\texttt{len}(\texttt{rev}(l)) - x)\ (\texttt{rev}(\texttt{rev}(l)))) = \texttt{rev}(\texttt{skipn}\ (\texttt{len}(l) - x)\ (l))}$$

`rev_involutive` further removes the double application of `rev` from the goal:

$$\frac{x:\ \texttt{nat}\quad l:\ \texttt{list A}}{\texttt{Goal:}\quad \texttt{rev}(\texttt{skipn}\ (\texttt{len}(\texttt{rev}(l)) - x)\ (l)) = \texttt{rev}(\texttt{skipn}\ (\texttt{len}(l) - x)\ (l))}$$

Finally, `rev_length` that asserts that the length of list is equal to the length of its reverse transforms the goal to:

$$\frac{x:\ \texttt{nat}\quad l:\ \texttt{list A}}{\texttt{Goal:}\quad \texttt{rev}(\texttt{skipn}\ (\texttt{len}(l) - x)\ (l)) = \texttt{rev}(\texttt{skipn}\ (\texttt{len}(l) - x)\ (l))}$$

which can be proved using the `reflexivity` tactic since the left and right-hand sides of the equality are identical.

```
1 Lemma firstn_rev: forall x l, firstn x (rev l) = rev (skipn (length l -x) l).
2 Proof.
3 intros x l. rewrite firstn_skipn_rev. rewrite rev_involutive.
4 (* time abduce 2. *)
5 (* The solver finds the goal to be invalid; the abduce call runs
6 for 1.847 seconds and returns two abducts:
7 1. rev l = l
8 2. length (rev l) = length l *)
9 assert (length (rev l) = length l).
10 { Search (length (rev _) = length _).
11 (* rev_length: forall l : list A, length (rev l) = length l *)
12 apply rev_length. } smt.
13 Qed.
```

Figure 4.8: Using `abduce` to find `rev_length`.

Figure 4.8 shows the result of treating the rewrite in line 6 (from Figure 4.7a) as a test unit. The `abduce` tactic is called before this line (the call and its output are commented out) and it takes about 2 seconds to come up with an instance of the `rev_length` lemma. The user can locally assert this lemma by copying the abduct into the parameter of an `assert` command. This lemma needs to now be proven inside braces that represent the body of a sub-proof. The user can use the `Search` command as in line 10 where the variable l is generalized using the _ character. The output of the command (also shown in a comment) returns the proof term that we need — `rev_length`, and this can be applied to close the sub-proof. Calling `smt` after this would result in a successful call (assuming that the type `A` that the lists in this proof are parameterized over have decidable equality) since the lemma, which is in the local context of the proof is now sent to the SMT solver as a hypothesis.

To understand our modified experimental setup, from the same proof of `firstn_rev`, consider as a test unit the rewrite of lemma `rev_involutive` (line 5 in Figure 4.7a). Since it is not the last rewrite in the proof, calling `abduce` from the beginning of this line is asking the abduction solver to suggest the application of both `rev_involutive` and `rev_length` before the goal can be solved deductively. Doing this in the unmodified iteration of the experiment caused the tactic to timeout. Figure 4.9 illustrates the extra step taken to separate `rev_involutive` from the conjunct — the instantiation of `rev_length` is locally asserted before the abduction solver is called. Given this extra fact, `abduce` succeeds in suggesting the correct instance of `rev_involutive` to the user. Following this, the user can use a similar approach as in Figure 4.8 to complete the proof. Notice that line 4 in the proof in Figure 4.9 introduces a hole in the proof since it proves a local lemma using `admit`. This assertion only serves the abduction solver and is, in fact, commented out along with the call to `abduce`. Therefore, it doesn't break the proof.

Although the `rewrite` tactic was invoked 241 times and `apply` 243 times within `List`, we had to eliminate most of these invocations from consideration for one the following reasons.

```
Lemma firstn_rev: forall x l, firstn x (rev l) = rev (skipn (length l -x) l).
Proof.
intros x l. rewrite firstn_skipn_rev.
assert (length (rev l) = length l) by admit. (* rev_length *)
(* time abduce 2. *)
(* The solver finds the goal to be invalid; the abduce call runs
for 0.433 seconds and returns two abducts:
1. rev l = l
2. rev (rev l) = l *)
assert (rev (rev l) = l).
{ Search (rev (rev _) = _).
(* rev_involutive: forall [A : Type] (l : list A), rev (rev l) = l *)
apply rev_involutive. } smt.
rewrite rev_length. reflexivity.
Qed.
```

Figure 4.9: Locally asserting future rewrites to avoid abducing conjunctive solutions.

- The test unit is nested within other automation tactics and isn't easy to isolate. This was an especially rare occurrence (only two such cases).
- Most often, the form of the current goal was unsupported by SMTCoq. This could be due to quantifiers, non-linear arithmetic, higher-order logic, or unsupported predicates (explained below).

As described in Section 2.5, SMTCoq's checker supports only goals containing predicates in their `Bool` form. A goal that contains a predicate in its `Prop` form must first be reduced to its corresponding `Bool` form. Examples 2.5.1 and 2.5.2 illustrate this using the integer less than predicate: SMTCoq's checker is built to support goals over `Z.ltb` (`<?`) of type `Z` → `Z` → `Bool`. To support `Z.lt` (`<`) of type `Z` → `Z` → `Prop`, SMTCoq uses the following property:

$$\forall\texttt{(n m : Z), (n <? m) = true <-> (n < m).}$$

This property proves an equivalence between `Z.lt` and `Z.ltb` so that the former can be reduced to the latter. A predicate in its `Prop` form is *unsupported* by SMTCoq if such a reduction cannot be performed by SMTCoq's tactics. We had to remove a large number of invocations of `rewrite` and `apply` from our test set because they were over a Coq predicate that could not be reduced to its `Bool` form. For example, the following goal is unsupported by SMTCoq due its usage of the less-than-or-equal-to predicate over the natural number type:

$$\frac{x:\ \texttt{A} \quad l,\ l1:\ \texttt{list A} \quad \texttt{H}\ :\ x :: (l1 \mathbin{++} l) = l}{\texttt{Goal:}\ \ \texttt{len}(x :: (l1 \mathbin{++} l)) <= \texttt{len}(l)}$$

where the `len` returns the length of a list as a natural number. If the `<=` predicate over natural numbers were supported by SMTCoq, such a goal would be easily solvable. Similarly, plenty of our discarded test units contained predicates over natural numbers. We

```
Goal forall (x y : Z), x = y + 1 -> x * x = (y + 1) * x.
Proof. smt.
(* Solver error: (error A non-linear fact was asserted
  to arithmetic in a linear logic. ). *)
```

(a) Goal with non-linear integer arithmetic.

```
Definition mul' := Z.mul.
Notation "x *' y" := (mul' x y) (at level 1).
Goal forall (x y : Z), x = y + 1 -> x *' x = (y + 1) *' x.
Proof. smt. Qed.
```

(b) A workaround.

Figure 4.10: A workaround to prove some NIA (but effectively linear) goals using SMTCoq.

expect that some of these might turn into either `smt` or `abduce` successes were support for natural numbers added to SMTCoq.

4.5.4 Multiplication over `Z`

SMTCoq supports linear integer arithmetic (LIA) using external SMT solvers, but not non-linear integer arithmetic (NIA). This is partly because at the time of SMTCoq's inception, proof production in NIA from SMT solvers had limited support. And so when the `smt` tactic is called on a non-linear goal or sees a non-linear hypothesis in context, it will fail, letting the user know that non-linear arithmetic isn't supported. Figure 4.10a shows an instance of such a failure (with the error message shown as a Coq comment). Notice though, that in this case, the SMT solver does not need to know anything about multiplication to solve the goal. It only needs to substitute `y+1` for `x` in (the implicative consequence of) the goal. Figure 4.10b demonstrates this by making multiplication an uninterpreted function — `mul'` (respectively `*'`) is alternate syntax for `mul` (respectively `*`) which is Coq's multiplication operator over the `Z` type. The effect of this alternate syntax is that whereas Coq does not differentiate the two, SMTCoq encodes the latter as an uninterpreted function, thus avoiding the NIA error.

Such a workaround is limited in its usefulness. Often the SMT solver needs to know some axioms about multiplication (to reason in non-linear arithmetic, in other words) to be able to solve the goal. In such a situation, the EUF solver (the sub-solver of the SMT solver responsible for the theory of the same name) will fail. Figure 4.11a shows a modification of the goal from Figure 4.10 for which this workaround fails. The SMT solver returns the following (fairly unhelpful) counterexample:

$$\{\texttt{mul'} \mapsto \lambda\, \texttt{x}, \texttt{y} \rightarrow \texttt{ite}(\texttt{x} = -1, \texttt{ite}(\texttt{y} = 1, 2, \texttt{-2}), \texttt{-2}), \quad \texttt{x} \mapsto 0, \quad \texttt{y} \mapsto -1, \quad \texttt{z} \mapsto 1\}$$

Although SMT solvers have made advances in NIA solving and proving, an integration with ITPs via tools like SMTCoq is yet to be implemented. From the example in Figure 4.11a, it is evident that, while the SMT solver needs to reason about multiplication, it only needs to

```
Goal forall (x y z: Z), x = y + 1 -> y *' z = z *' (x - 1).
Proof. smt.
(* Fails with counterexample *)
```

(a) Limits of uninterpreted multiplication.

```
Goal forall (x y z: Z), x = y + 1 -> y *' z = z *' (x - 1).
Proof. abduce 3.
(* cvc5 returned SAT.
The solver cannot prove the goal, but one of the following hypotheses
would make it provable:
z = y
x - 1 = z
(mul' y z) = (mul' z y) *)
  intros. assert ((mul' y z) = (mul' z y)).
  { apply Z.mul_comm. } smt.
Qed.
```

(b) Yet another workaround using `abduce`.

Figure 4.11: A workaround for proving some (effectively linear) NIA goals using `abduce`.

know limited facts about it. In fact, it only needs to know that multiplication is commutative. Figure 4.11b shows how the `abduce` tactic can be used in such situations (the abducts returned by the solver are commented as usual). The third abduct confirms that the solver only needs to know that the multiplication function is commutative to be able to prove the goal. This is easy to do for the Coq user, since Coq provides efficient ways to search for lemmas about symbols from the current global context (such as the `Search` command). Notice that `mul'` from this example is simply a more specific version of `f` from Example 4.3.1 (except for this, the examples are identical).

In our third and final set of experiments we explore an alternative way of solving NIA goals in Coq using cvc5's abduction solver and an external SMT solver that supports the theories of LIA and EUF.

Two files from the Coq standard library contain most of the definition/axiomatization of multiplication over the `Z` type in Coq— `BinIntDef` contains the definition of the multiplication operator, and `BinInt` contains proofs of various properties over it. To find test units for this experimental setup, we used properties of multiplication, defined in `BinInt`, that were most often invoked within other proofs in the standard library. As with the `List` experiments from Section 4.5.3, these properties were invoked either using the `rewrite` or `apply` tactic. Furthermore, lemmas of the same name are used for the axiomatization of multiplication over other types (such as natural numbers, positive numbers, etc.) as well. From all invocations of lemmas with these names, we filtered out the invocations of the `Z` types. Finally, we considered 8 lemmas related to `Z.mul` from `BinInt` that were invoked 100 times from other library files in the Coq standard library. Figure 4.12 lists the lemmas, the property they prove, and the number of times they are called from within the Coq standard

Lemma Name	Lemma Property	No. of Invocations
mul_1_l	1 * n = n	48
mul_add_distr_r	(n + m) * p = n * p + m * p	20
mul_0_r	n * 0 = 0	8
mul_opp_r	n * - m = - (n * m)	8
mul_0_l	0 * n = 0	8
mul_reg_l	p <> 0 -> p * n = p * m -> n = m	4
mul_reg_r	p <> 0 -> n * p = m * p -> n = m	2
opp_eq_mul_m1	- n = n * -1	2

Figure 4.12: Commonly occurring lemmas about multiplication used for testing `abduce`.

Goals	`smt` successes	Invalid Goals	`abduce` successes	Timeouts
84	11	73	17	56

Figure 4.13: Summary of results of using `abduce` to solve NIA goals in Coq.

library. From the 100 potential test units, we disregard 16 that are unsupported by SMTCoq (for the same reasons that resulted in unsupported goals in Section 4.5.3). For each of the remaining 84 test units, we made all occurrences of multiplication in the proof context uninterpreted (similar to Figure 4.10b), and called `abduce` before the invocation of the lemma in question. The test was considered a success if cvc5 could abduce a formula entailed by the current Coq context. As with the `List` experiments, lemmas could easily be found from the environment using the `Search` command. Our results are summarized in Figure 4.13. In our original experimental setup (for `mul`) where we simply replaced an invocation to one of the lemmas with `abduce`, our tactic was successful for 11 of the 84 goals. In a similar manner to the experiments from Section 4.5.3, we modified our experimental setup to separate a conjunctive test unit into test units for each of its conjuncts. This modification increased the number of `abduce` successes to 17 which is the number displayed in Figure 4.13, yielding a 23% success rate for the `abduce` tactic. All but two `abduce` successes took less than 2 seconds to find a successful abduct, with over half taking less than 1 second. Failures can be attributed to one of the following reasons:

- Goals were often expressed in terms of other types such as positive numbers and rational numbers, which caused the generation of additional uninterpreted symbols that the abduction solver had to deal with.
- Even when the only type used in the proof context was `Z` (Coq's integer type), the formulas in the context contained other symbols such as those for division, and exponentiation, which were also given to the SMT solver as uninterpreted symbols. These often proved to be too much for the abduction solver.

4.5.5 Conclusion and Future Work

Figure 4.14 summarizes the results from each of the three experiments of the `abduce` tactic. The tactic does especially well in the `ZOrder` library owing to the size of the proofs in

Experiment Set	Goals	`smt` successes		Invalid goals	`abduce` successes		Timeouts
		#	%		#	%	
`ZOrder`	59	33	55.93	26	16	61.53	12
`List`	122	25	20.49	97	28	28.86	69
`Z.mul`	84	11	13.09	73	17	23.28	56

Figure 4.14: Summary of results from all 3 experiments over `abduce`. Percentage of SMT success is over the total number of goals, and percentage of `abduce` successes is over the number of invalid goals.

the library and the definition of success unique to that experiment. In both the subsequent experiments, it is able to automate around one-quarter of the available goals when used along with Coq's `Search` command. We noticed that when the abduction engine is unsuccessful, it is usually because it either has to reason about too many uninterpreted symbols, or that it has too many facts to abduce.

Furthermore, many of the suggested abducts in the case of `abduce` successes were a minor modification of the goal (for example, a symmetric version of the goal), which arguably could easily be guessed by a human user. Nevertheless, such a feature could be useful, especially if we could automate the entire pipeline: the call to `smt`; a call to `abduce` when `smt` fails; the search for a generalized version of each abduct; the local assertion of an abduct that could be found via `Search` and the application of the lemma to close the sub-proof; and another (now successful) call to `smt`. We propose the development of such an improved tactic for future work.

With the increase of SMTCoq's supported theories, we expect the `abduce` tactic to be applicable in more settings where it can be useful in providing external automation. For example, if SMTCoq natively supported the Coq natural number type, it would provide many more interesting test units for `abduce`. With the development of SMT SyGuS technology, and the consequent improvement of syntax-restricted abduction, we expect the quality of abducts to also increase. An improvement in the quality of conjunctive abducts, for example, would allow for the `abduce` tactic to be used earlier in proofs, whereas now all successes come from invocations at the end of a proof or of a case within a proof.

Chapter 5

The alethe Checker

The ability of an SMT solver to produce a proof certificate in addition to a result increases its trustworthiness. Proof certificates (or just proofs) — specified in a proof certificate format — detail the steps taken by the solver in determining the validity of its input. The solver proves its input to be valid by reducing the input's negation to a form of falsity. Proofs vary in their level of detail. For example, a fine-grained proof provides justifications for steps that rewrite terms to equivalent ones, whereas a coarse-grained proof might avoid such details. The steps in a proof can be verified to confirm the solver's result. A *proof checker* can automate this task.

Proof certificates are also integral to incorporating an SMT solver into an ITP so that the solver can automate proofs of sub-goals within the ITP. An SMT solver whose steps are not justified to the ITP would increase the trusted computing base (TCB) of an ITP that uses it, which is undesirable. Proof certificates preserve the ITP's TCB by guiding the creation of a proof within the ITP's framework. This process can be incomplete in that unjustified steps in a proof can be returned to the user of the ITP as a sub-goal. Such a step is called a proof *hole*. SMTCoq is a tool that offers both the above mentioned utilities — a checker for SMT proofs in Coq that is certified so that it can be used to provide automation for Coq goals. An SMT solver produces a proof certificate in a particular proof certificate format. SMTCoq supports the CVC4 and (an older version of) veriT SMT solvers via the LFSC (Logical Framework with Side Conditions) and the verit2016 proof certificate formats, respectively. Towards supporting these proof formats and other SMT solver formats, SMTCoq uses an internal proof certificate format called smtcoq-certif. The goal of integrating an SMT solver with SMTCoq is then reduced to that of soundly converting a proof in the solver's proof certificate format to smtcoq-certif.

alethe is a new proof certificate format that is supported by both the cvc5 SMT solver and a modern version of veriT. Besides supporting multiple SMT solvers, alethe permits the generation of fine-grained proofs that justify steps such as term rewrites. Carcara [4] is a standalone checker written in Rust for alethe proofs. This chapter presents the description and evaluation of a Coq-certified proof checker for the alethe proof format. We implement the checker by reducing alethe to smtcoq-certif, the internal proof format of SMTCoq. We argue that our checker supports more fine-grained SMT proofs than were previously supported by SMTCoq owing to (i) the alethe proof format's ability to specify such steps in many cases, and (*ii*) our efforts to elaborate other steps in terms of ones that are checkable by SMTCoq. Consequentially, SMTCoq is able to offer more complete automation to Coq users through our proof checker (through fewer proof holes). Our proof checker also offers an ITP-certified alternative to Carcara.

Section 5.1 formally defines proofs and proof certificate formats, Section 5.2 and 5.3 specify smtcoq-certif and alethe, and Section 5.4 presents the proof certificate transformations used to implement the alethe checker. Section 5.5 details an evaluation of the checker on a set of benchmarks.

5.1 Proof Certificate Formats

A *proof rule* or a rule of inference takes the form

$$\frac{P_1 \quad \dots \quad P_n}{C} \; RuleName(arguments)$$

where $P_1, \dots, P_n$ are the premises ($n \geq 0$); C is the conclusion; *RuleName* is the name of the rule and *arguments* is a possibly-empty list of arguments to the rule. Both premises and conclusion are formulas. The rule specifies that if the premises $P_1, \dots, P_n$ hold, then the conclusion C holds. A rule that takes at least one premise is a *conversion* rule. We classify *non-conversion* rules (rules that take no premise) as *assumptions* — with rule name `assume`; *subproofs* — defined below; and, *lemmas* — all other non-conversion rules. A proof rule is applied by an instantiation of its meta-variables to formulas. The application of a proof rule to zero or more formulas to derive a formula is called a *(proof) step*. Each step in a proof is uniquely identified by an ID. We specify steps either using the notation for a proof rule above (with the ID on the left side of the rule), or as a tuple

$$(ID,\ RuleName[\texttt{id(}P_1\texttt{)}; \dots; \texttt{id(}P_n\texttt{)}],\ C,\ (arguments))$$

where $\texttt{id(}P\texttt{)}$ is the ID of the step that derives P.

Example 5.1.1. The proof rule for *resolution* is

$$\frac{\phi_1 \lor \dots \lor \chi \lor \dots \lor \phi_n \qquad \psi_1 \lor \dots \lor \neg\chi \lor \dots \lor \psi_m}{\phi_1 \lor \dots \lor \phi_n \lor \psi_1 \lor \dots \lor \psi_m} \; \texttt{res}$$

and the following is an example of a step that applies resolution:

$$3\ \frac{1\ \dfrac{}{a \lor \neg b}\ \texttt{assume} \qquad 2\ \dfrac{}{b \lor c}\ \texttt{assume}}{a \lor c}\ \texttt{res}$$

Equivalently, this derivation can be represented in tuple form as follows:

$$\begin{array}{llll} (1, & \texttt{assume}, & a \lor \neg b, & (\,)) \\ (2, & \texttt{assume}, & b \lor c, & (\,)) \\ (3, & \texttt{res}[1;2], & a \lor c, & (\,)) \end{array}$$

Given that two clauses hold, and a pivot — a literal that occurs with opposite polarities in each clause (b in the above example) — the resolution rule concludes that the combination of the two clauses without either occurrence of the pivot holds. While implementing a checker for this rule, one must consider issues such as: whether the pivot can occur anywhere in each clause; whether the pivot can occur exactly once within each clause; whether disjunction is a binary or n-ary operator. If the proof certificate format doesn't specify how such issues must be addressed, then the burden falls on the checker to accept the most general form of the rule. For instance, if the format doesn't specify where in each clause the pivot must occur, then the checker must search each clause to find a pair of literals with opposite polarities.

A *proof* or *proof certificate* $\mathcal{P}$ of $H \models_T G$ is a derivation of a formula G from formulas $H_1, \ldots, H_n$ (set H) by the application of one or more proof rules. Figure 5.1 presents an inductive definition of proof certificates using recursive function `getAssumptions` (specified in Figure 5.2) that given a step, returns a set of formulas — the assumptions that the step depends on. We use either the *tree* notation or the *tuple* notation to represent proofs and proof certificates in this document, as illustrated in Example 5.1.2. In our presentation of proofs, we sometimes omit parts of a step such as the ID, rule name, premises, or arguments either when it is clear from context, or unnecessary. We also often refer to clauses by their IDs.

Example 5.1.2. This example shows the proof certificate for the unsatisfiability of $a = b$ and $\neg(f\ a = f\ b)$ (conversely, the validity of $a = b \rightarrow f\ a = f\ b$). In the tree form:

$$\dfrac{\dfrac{\dfrac{}{a = b}\,\texttt{assume}}{f\ a = f\ b}\,\texttt{cong} \qquad \dfrac{}{\neg(f\ a = f\ b)}\,\texttt{assume}}{\langle\ \rangle}\,\texttt{res}$$

and in the tuple form:

$$\begin{array}{llll}
(1, & \texttt{assume}, & a = b &) \\
(2, & \texttt{assume}, & \neg(f\ a = f\ b) &) \\
(3, & \texttt{cong}[1], & f\ a = f\ b &) \\
(4, & \texttt{res}[3;2], & \langle\ \rangle &)
\end{array}$$

The `cong` rule states that if two (or more) terms are equal, then the equality between terms obtained by applying the same function to them also holds.

A *proof* (or proof certificate) $\mathcal{P}$ can be inductively defined as follows:

- For non-conversion rule R (such that R is not `subproof`),

$$\frac{}{P}\ R$$

is a proof *by* R of P, or of

$$\models_T P$$

- A *subproof* is used within a proof to prove a lemma by discharging locally introduced hypotheses. A subproof that introduces hypotheses $H_1, \ldots, H_n$ and discharges them to prove G

$$\cfrac{\cfrac{}{H_1}\ \texttt{assume} \\ \vdots \\ \cfrac{}{H_n}\ \texttt{assume} \\ \vdots \\ G}{\neg H_1 \vee \cdots \vee \neg H_n \vee G}\ \texttt{subproof}$$

is a proof *by* `subproof` of

$$\models_T \neg H_1 \vee \cdots \vee \neg H_n \vee G$$

- For conversion rule R, and proofs $P_1, \ldots, P_n$ with $\texttt{getAssumptions}(P_1) \cup \cdots \cup \texttt{getAssumptions}(P_n) = \{H_1, \ldots, H_m\}$,

$$\frac{P_1 \quad \cdots \quad P_n}{C}\ R$$

is a proof of

$$H_1, \ldots, H_m \models_T C$$

Figure 5.1: The inductive definition of a proof.

Sometimes, we fragment a proof into multiple sequences of steps to make ordering constraints explicit. For example, for sequences of steps Π_1 and Π_2, the proof

$$\Pi_1$$
$$\Pi_2$$

```
getAssumptions(P) := match P with
  | ⊨_T H  by assume,                 ⟶  H
  | ⊨_T S  by subproof,               ⟶  {}
  | ⊨_T H  by a lemma,                ⟶  {}
  | H_1,...,H_1 ⊨_T G by some R,      ⟶  getAssumptions(H_1) ∪ ···
                                          ∪ getAssumptions(H_n)
```

Figure 5.2: The recursive definition of the `getAssumptions` function, using the pattern matching syntax from functional programs, necessary for the definition of proofs.

imposes an ordering on the steps so that no step in Π_1 can have a step from Π_2 as a premise.

As described in Section 2.2.2, many SMT solvers emit proof certificates, in addition to unsatisfiability results, that can be externally checked to increase trust in the solver. Such an SMT solver produces its proofs in a *proof certificate format* F. The proof certificate format specifies the proof rules that can be used in a proof certificate. We denote a proof $\mathcal{P}$ in proof certificate format F as $\mathcal{P}_F$. A proof is *correct* in a proof certificate format F if it derives a proof by correctly applying only rules from F. We do not fully formalize the *correct application* of a rule in F but such an application is specified by F, and generally refers to sound instantiations of the meta-variables in the rule to terms or formulas. All our examples show correct proofs, unless specified otherwise. A proof checker is able to check whether a proof certificate is correct with respect to a proof certificate format. In the rest of this section, we reference (and partially specify) three proof formats alethe, verit2016, and smtcoq-certif; and refer to a proof $\mathcal{P}$ in each format as $\mathcal{P}_\mathcal{A}$, $\mathcal{P}_\mathcal{V}$, and $\mathcal{P}_\mathcal{S}$ respectively.

Given a proof $\mathcal{P}_F$ of $H \models_T G$ in format F, we define a *transformation* $\mathcal{T}$ of $\mathcal{P}_F$ to some format F' as $\mathcal{P}_{F'}$.

$$\mathcal{T}(\mathcal{P}_F) = \mathcal{P}_{F'}$$

This transformation is *sound* if whenever $\mathcal{P}_F$ is a correct proof of $H \models_T G$ in F, $\mathcal{P}_{F'}$ is a correct proof of $H \models_T G$ in F'. In what follows, we propose a sound transformation (for a restricted logical fragment) from proofs in alethe to proofs in smtcoq-certif.

Clauses vs Disjunctions As mentioned in Section 2.1, a clause is a disjunction of literals, and we sometimes represent a clause as a set of its constituent literals. While we have been using these notations interchangeably so far, SMTCoq differentiates clauses from disjunctions in its representation. Furthermore, it provides rules to convert between the two. Moving forward, we will also make this differentiation explicit — a clause is represented as a list of its constituent literals (we will use a comma separated sequence of literals sometimes enclosed in square brackets), whereas a disjunction of literals separates the terms by the $\vee$ operator. Note also that SMTCoq normalizes the ordering of literals in a clause based on its internal representation, and so the order in which we present the literals of a clause are inconsequential to the checker.

To understand the rules, it might be useful to think of a disjunction as giving implicative information. This is evident from the fact that for two *Bool* terms x and y, $\neg x \vee y$ is equivalent to $x \rightarrow y$.

Section 5.2 specifies smtcoq-certif, Section 5.3 specifies alethe, and Section 5.4 proposes the transformation from alethe to smtcoq-certif, and a certified checker for alethe in Coq.

5.2 smtcoq-certif

Each proof producing SMT solver generally produces proof certificates in its distinctly defined format: CVC4 uses LFSC, a meta-format or framework that allows one to specify the proof rules over which proofs are defined; veriT (until recently) produced proofs in the verit2016 proof certificate format whose syntax is similar to the tuple notation defined in Section 5.1; z3 has its own internal format for coarse-grained proofs. In order to be compatible with these and other possible formats, SMTCoq fixes its own internal proof certificate format which we call smtcoq-certif, and preprocesses certificates from solvers into this format. In smtcoq-certif, a certificate is a sequence of uniquely identifiable steps that eventually derive the empty clause $\langle\ \rangle$ from a set of assumptions.

The smtcoq-certif format can be traced back to the proof certificate format of the veriT SMT solver, the first solver to be integrated with SMTCoq. Together with the ideas proposed in Deharbe et al. [40], veriT was able to produce unsatisfiability proof certificates for the theories of equality over uninterpreted functions (EUF) and linear integer arithmetic (LIA), both with and without quantifiers. The format evolved to a format used by veriT version v2016 which we call verit2016. smtcoq-certif is very similar to the verit2016 format.

Currently, smtcoq-certif has rules for propositional logic, reasoning in the theories of equality over uninterpreted functions (EUF), linear integer arithmetic (LIA), bit-vectors (BV), and arrays with extensionality (AX), and limited support for quantified reasoning. We specify a representative set of rules of the smtcoq-certif proof certificate format in Section 5.2.1 for propositional logic and the theories of EUF and LIA. The `assume` and `res` rules are used to introduce hypotheses and to perform propositional resolution, respectively. Propositional logic rules include standard connective introduction and elimination rules. smtcoq-certif provides these rules in the form of lemmas specifying valid clauses as well as conversion rules on clauses. The EUF rules occur only as lemmas and allow an implementation of the congruence closure algorithm [71]. SMTCoq uses a verified decision procedure in Coq called `Micromega` [17] to check LIA rules: `lia_micromega` is used to prove arithmetic lemmas and `spl_arith` is used to prove arithmetic transformations from premise to conclusion. The checkers for both rules adapt the clause derived by the rule and the premises (if any) to a form that can be sent to `Micromega`.

5.2.1 smtcoq-certif Proof Rules

Here, we present the rules from smtcoq-certif for propositional logic, the theories of EUF and LIA, and the rule used for instantiating universally quantified formulas. Furthermore, the rules are restricted by the connectives specified in Section 2.1. The rules are as follows.

- *Assumption*

  $$\frac{}{\phi}\ \texttt{assume}$$

  An assumption represents a leaf of the proof-tree and as such, introduces a hypotheses into the proof.

- *Resolution*

  $$\frac{\phi_1 \quad \dots \quad \phi_n}{\phi_1', \dots, \phi_m'}\ \texttt{res}$$

  where $n \geq 1, m \geq 0$.
  Resolution is the central rule for clause reduction. We define it for three separate cases:

  1. When $n = 1$, `res` simply returns the premise unchanged.
  2. When $n = 2$, `res` performs a single resolution step:

     $$\frac{\phi_1, \cdots, \chi, \cdots \phi_n \qquad \psi_1, \cdots, \neg\chi, \cdots, \psi_m}{\phi_1, \cdots, \phi_n, \psi_1, \cdots, \psi_m}\ \texttt{res}$$

     It takes 2 clauses with at least one *pivot* literal, a literal that occurs with opposite polarities in each clause, and returns a new clause that contains all the literals from both clauses except all occurrences in either polarity of an aribitrarily chosen pivot (typically solvers perform resolution so that there is only one possible pivot).
  3. When $n > 1$, `res` chains a sequence of pairwise resolution steps such that each step returns a clause resolvable with the next.

- *Weaken*

  $$\frac{\phi_1, \dots, \phi_n}{\phi_1, \dots, \phi_n, \psi_1, \dots, \psi_m}\ \texttt{weaken}$$

  A clause can be weakened by adding arbitrary literals to it.

Propositional Rules

In the following, for rules that take arguments, we specify the expected argument given the specified form of the rule; any other argument would make it an unsound invocation of the rule. All rules derive clauses.

- *Lemmas*

  $$\frac{}{\top}\ \texttt{true} \qquad\qquad \frac{}{\neg\bot}\ \texttt{false}$$

$$\frac{}{(x_1 \wedge \cdots \wedge x_n), \neg x_1, \cdots, \neg x_n}\ \texttt{andn} \qquad \frac{}{\neg(x_1 \vee \cdots \vee x_n), x_1, \cdots, x_n}\ \texttt{orp}$$

$$\frac{}{\neg(x \rightarrow y), \neg x, y}\ \texttt{impp}$$

$$\frac{}{\neg(x =^{Bool} y), x, \neg y}\ \texttt{eqvp1} \qquad \frac{}{(x =^{Bool} y), \neg x, \neg y}\ \texttt{eqvn1}$$

$$\frac{}{\neg(x =^{Bool} y), \neg x, y}\ \texttt{eqvp2} \qquad \frac{}{(x =^{Bool} y), x, y}\ \texttt{eqvn2}$$

`andn`, `orp`, `impp`, `eqvp1`, `eqvp2`, `eqvn1`, and `eqvn2` are elimination rules. They enable the indirect elimination of either a positive (`p`) or negative (`n`) occurrence of the corresponding connective. A derivation of such an occurrence can be removed by resolving it with the derivation of the corresponding rule.

$$\frac{}{(x_1 \vee \cdots \vee x_n), \neg x_i}\ \texttt{orn}\ \text{(i)} \qquad \frac{}{\neg(x_1 \wedge \cdots \wedge x_n), x_i}\ \texttt{andp}\ \text{(i)}$$

$$\frac{}{(x \rightarrow y), x}\ \texttt{impn1}\ (1) \qquad \frac{}{(x \rightarrow y), \neg y}\ \texttt{impn2}\ (2)$$

`orn`, `andp`, `impn1`, and `impn2` project an operand from their corresponding connective and take as argument the index of the operand to project.

- *Conversion Rules*

$$\frac{\neg(x_1 \wedge \cdots \wedge x_n)}{\neg x_1, \cdots, \neg x_n}\ \texttt{nand} \qquad \frac{x \rightarrow y}{\neg x, y}\ \texttt{imp}$$

$$\frac{x =^{Bool} y}{\neg x, y}\ \texttt{eqv1} \qquad \frac{x \neq^{Bool} y}{x, y}\ \texttt{neqv1}$$

$$\frac{x =^{Bool} y}{x, \neg y}\ \texttt{eqv2} \qquad \frac{x \neq^{Bool} y}{\neg x, \neg y}\ \texttt{neqv2}$$

`nand`, `imp`, `eqv1`, `eqv2`, `neqv1`, and `neqv2` are conversion forms of the elimination rules for either the positive or negative (`n`) form of the corresponding connective.

$$\frac{x_1 \wedge \cdots \wedge x_n}{x_i}\ \texttt{and}\ \text{(i)} \qquad \frac{\neg(x_1 \vee \cdots \vee x_n)}{x_i}\ \texttt{nor}\ \text{(i)}$$

$$\frac{\neg(x \rightarrow y)}{x}\ \texttt{nimp1}\ \text{(i)} \qquad \frac{\neg(x \rightarrow y)}{\neg y}\ \texttt{nimp2}\ \text{(i)}$$

and, nor, nimp1, and nimp2 are conversion forms of the projection rules of the respective operators. They also take an argument specifying the index of the projection.

EUF Rules

$$\frac{}{x =^{\sigma} x}\ \texttt{eqrefl}$$

eqrefl proves the equality of some term x of sort σ — such that σ is not *Bool* — with a syntactically equivalent term.

$$\frac{}{(x_1 \neq^{\sigma} x_2), \cdots, (x_{n-1} \neq^{\sigma} x_n), (x_1 \neq^{\sigma} x_n)}\ \texttt{eqtrans}$$

eqtrans proves the transitivity of equality over terms of any non-Boolean sort σ in the lemma form.

$$\frac{}{(x_1 \neq^{\sigma} y_1), \cdots, (x_n \neq^{\sigma} y_n), (f\ x_1\ \cdots\ x_n =^{\sigma} f\ y_1\ \cdots\ y_n)}\ \texttt{eqcong}$$

eqcong proves the congruence of equality for non-Boolean terms. Given that $x_1, \ldots, x_n$ are equal to $y_1, \ldots, y_n$ respectively, eqcong proves that an application of n-ary function f to the former is equal to its application to the latter.

$$\frac{}{(x_1 \neq^{Bool} y_1), \cdots, (x_n \neq^{Bool} y_n), \neg(p\ x_1\ \cdots\ x_n), (p\ y_1\ \cdots\ y_n)}\ \texttt{eqcongp}$$

eqcongp proves the congruence of equivalence for Boolean formulas. Given that $x_1, \ldots, x_n$ are equal to $y_1, \ldots, y_n$ respectively, eqcongp proves that an application of n-ary predicate p to the former implies its application to the latter.

LIA Rules

$$\frac{}{x_1, \cdots, x_n}\ \texttt{lia\_micromega} \qquad \frac{x_1, \cdots, x_n}{y_1, \cdots, y_m}\ \texttt{spl\_arith}$$

lia_micromega proves lemmas in the LIA theory and spl_arith proves conversions of clauses that are valid in LIA.

Quantifier Rules

$$\frac{}{\neg\forall x_1, \ldots x_n.P \vee P[x_1 \mapsto t_1] \ldots [x_n \mapsto t_n]}\ \texttt{forall\_inst}$$

The forall_inst rule allows instantiation of quantified variables $x_1, \ldots, x_n$ to terms $t_1, \ldots, t_n$ in a universally quantified formula. $t_1, \ldots, t_n$ are inferred by the checker from the instantiated form of P.

5.3 alethe

alethe is a proof certificate format for SMT solvers, aiming to provide an easy-to-use and uniform set of natural-deduction style proof rules for proofs of unsatisfiability of first-order formulas. Its syntax is an extension of the SMT-LIB 2 language, and its logic is the many-sorted first-order logic used by SMT solvers. The format is fully supported by veriT and partially supported by cvc5. Through alethe proofs, these solvers are being integrated with the Isabelle/HOL proof assistant; our goal is to do the same with the Coq proof assistant. alethe can be considered an extension of verit2016, with support for linear real arithmetic and a richer set of rules enabling both fine- and coarse-grained proofs. Since the proof of unsatisfiability from an SMT solver generally tries to reflect its internal reasoning, it will have to account for various term rewrites performed by the SMT solver. The larger set of rules in alethe are also inspired by the goal of covering such rewrite reasoning in the proofs. The most practical advantage of alethe is that it is supported natively by both veriT and cvc5, two state-of-the-art SMT solvers. Section 5.3.1 lists the rules in alethe that are not already in smtcoq-certif. Most of these are rewrite rules (for example, `andsimp`). The rules from the LIA theory in alethe are omitted from this section, since they are still evolving. As mentioned in Section 5.2, SMTCoq uses a Coq decision procedure called `Micromega` [17] to check LIA rules, so the transformation of the existing LIA rules in alethe to smtcoq-certif is fairly straightforward: lemmas (including LIA rewrites) are converted to an application of the `lia_micromega` rule; and rules that derive from external premises are converted to an application of `spl_arith`. Another interesting difference from smtcoq-certif is the ability to introduce lemmas via subproofs in alethe, as introduced in Section 5.1.

5.3.1 alethe Proof Rules

Since alethe is an extension of verit2016, which is the format that smtcoq-certif is based on, alethe is effectively a superset of smtcoq-certif. The following is a presentation of the rules from alethe that do not exist in smtcoq-certif.

Propositional Rules

Lemmas

$$\frac{}{\neg\neg\neg x, x}\ \texttt{notnot} \qquad\qquad \frac{\phi_1, \cdots, \psi, \cdots \neg\psi, \cdots \phi_n}{\top}\ \texttt{tautology}$$

The `notnot` rule eliminates double negations, and occurs in the lemma form. The `tautology` rule simplifies a trivial clause — one that contains some literal such that its negation also exists in the clause — to the singleton clause representing truth.

Rewrite Rules

All rewrite rule take the form of an equality/equivalence. They are indirectly applied to terms in a proof in order to replace a term that takes the form from one side of the equality/equivalence by one that takes the form from the other side.

$$\frac{}{(\varphi_1 \wedge \cdots \wedge \varphi_n) =^{Bool} \psi} \; \texttt{andsimp}$$

where the possible rewrites are:

- $\top \wedge \cdots \wedge \top =^{Bool} \top$
- $x_1 \wedge \cdots \wedge x_n =^{Bool} x_1 \wedge \cdots \wedge x_{n'}$ where the RHS has all $\top$ literals removed.
- $x_1 \wedge \cdots \wedge x_n =^{Bool} x_1 \wedge \cdots \wedge x_{n'}$ where the RHS has all repeated literals removed.
- $x_1 \wedge \cdots \wedge \bot \wedge \cdots \wedge x_n =^{Bool} \bot$
- $x_1 \wedge \cdots \wedge x_i \wedge \cdots \wedge x_j \wedge \cdots \wedge x_n =^{Bool} \bot$ where $x_i =^{Bool} \neg x_j$

$$\frac{}{(\varphi_1 \vee \cdots \vee \varphi_n) =^{Bool} \psi} \; \texttt{orsimp}$$

where the possible rewrites are:

- $\bot \vee \cdots \vee \bot =^{Bool} \bot$
- $x_1 \vee \cdots \vee x_n =^{Bool} x_1 \vee \cdots \vee x_{n'}$ where the RHS has all $\bot$ literals removed.
- $x_1 \vee \cdots \vee x_n =^{Bool} x_1 \vee \cdots \vee x_{n'}$ where the RHS has all repeated literals removed.
- $x_1 \vee \cdots \vee \top \vee \cdots \vee x_n =^{Bool} \top$
- $x_1 \vee \cdots \vee x_i \vee \cdots \vee x_j \vee \cdots \vee x_n = \top$ where $x_i =^{Bool} \neg x_j$

$$\frac{}{\varphi =^{Bool} \psi} \; \texttt{notsimp}$$

where the possible rewrites are:

- $\neg(\neg x) =^{Bool} x$
- $\neg\bot =^{Bool} \top$
- $\neg\top =^{Bool} \bot$

$$\frac{}{\varphi_1 \rightarrow \varphi_2 =^{Bool} \psi} \; \texttt{impsimp}$$

where the possible rewrites are:

- $\neg x_1 \to \neg x_2 =^{Bool} x_2 \to x_1$
- $\bot \to x =^{Bool} \top$
- $x \to \top =^{Bool} \top$
- $\top \to x =^{Bool} x$
- $x \to \bot =^{Bool} \neg x$
- $x \to x =^{Bool} \top$
- $\neg x \to x =^{Bool} x$
- $x \to \neg x =^{Bool} \neg x$

$$\frac{}{(\varphi_1 =^{Bool} \varphi_2) =^{Bool} \psi}\ \texttt{eqvsimp}$$

where the possible rewrites are:

- $(\neg x_1 =^{Bool} \neg x_2) =^{Bool} (x_1 = x_2)$
- $(x =^{Bool} x) =^{Bool} \top$
- $(x =^{Bool} \neg x) =^{Bool} \bot$
- $(\neg x =^{Bool} x) =^{Bool} \bot$
- $(\top =^{Bool} x) =^{Bool} x$
- $(x =^{Bool} \top) =^{Bool} x$
- $(\bot =^{Bool} x) =^{Bool} \neg x$
- $(x =^{Bool} \bot) =^{Bool} \neg x$

$$\frac{}{\varphi =^{Bool} \psi}\ \texttt{boolsimp}$$

where the possible rewrites are:

- $\neg(x_1 \to x_2) =^{Bool} (x_1 \wedge \neg x_2)$
- $\neg(x_1 \vee x_2) =^{Bool} (\neg x_1 \wedge \neg x_2)$
- $\neg(x_1 \wedge x_2) =^{Bool} (\neg x_1 \vee \neg x_2)$
- $(x_1 \to (x_2 \to x_3)) =^{Bool} (x_1 \wedge x_2) \to x_3$

- $((x_1 \to x_2) \to x_2) =^{Bool} (x_1 \lor x_2)$
- $(x_1 \land (x_1 \to x_2)) =^{Bool} (x_1 \land x_2)$
- $((x_1 \to x_2) \land x_1) =^{Bool} (x_1 \land x_2)$

$$\frac{}{(\varphi_1 =^{\sigma} \varphi_2) =^{Bool} \psi}\ \texttt{eqsimp}$$

where the possible rewrites are:

- $x =^{\sigma} x = \top$, for any non-Boolean sort σ.
- $(x_1 = x_2) =^{\sigma} \bot$ if x_1 and x_2 are different numeric constants, for a numeric sort σ.
- $\neg(x =^{\sigma} x) = \bot$ if x is a numeric constant, for a numeric sort σ.

`andsimp`, `orsimp`, `notsimp`, `impsimp`, `eqvsimp`, and `eqsimp` specify possible rewrites over the corresponding connectives. `boolsimp` specifies some useful equivalences about formulas.

EUF Rules

$$\frac{}{x =^{\sigma} x}\ \texttt{refl} \qquad \frac{x_1 =^{\sigma} x_2 \quad \cdots \quad x_n =^{\sigma} x_{n+1}}{x_1 =^{\sigma} x_{n+1}}\ \texttt{trans} \qquad \frac{x_1 =^{\sigma} y_1 \quad \cdots \quad x_n =^{\sigma} y_n}{f\ x_1 \cdots x_n =^{\sigma} f\ y_1 \cdots y_n}\ \texttt{cong}$$

These rules are similar to `eqrefl`, `eqtrans`, `eqcong`, and `eqcongp` except: (i) they take the premise-conclusion form, that is, they are conversion rules (ii) they are more expressive since σ can be any sort, including *Bool* (for instance, `cong` is as expressive as both `eqcong` and `eqcongp`).

Subproof Rules

$$\frac{\begin{array}{c} \frac{}{H_1}\ \texttt{assume} \\ \vdots \\ \frac{}{H_n}\ \texttt{assume} \\ \vdots \\ G \end{array}}{\neg H_1, \cdots, \neg H_n, G}\ \texttt{subproof}$$

The `subproof` rule can derive an implicative proof using a separate proof context: one can prove $H \to G$ by proving G from the proof of H (this naturally extends to multiple implicative premises). The assumptions and steps in the subproof constitute its local context (since they cannot be accessed from outside the proof); the assumptions and steps outside the *box* generated by the subproof, including the implication derived by the `subproof` rule, constitute the global context.

5.4 Coq Checker for alethe

Currently, SMTCoq provides two Coq commands that invoke its internal checker:

- The `Verit_Checker` command takes two arguments — the path of the SMT file and the path of the proof file. It expects the proof to be in the verit2016 proof certificate format. It converts this into a proof certificate in the smtcoq-certif format; and invokes SMTCoq's checker on the SMT file and the corresponding smtcoq-certif proof certificate. It returns `true` if the checker is able to check that the proof proves the assertions in the SMT file, and `false` otherwise.

- Similarly, the `Lfsc_Checker` command takes two arguments — the path of the SMT file and that of the LFSC proof file; and invokes the SMTCoq checker on them after converting the LFSC proof into an smtcoq-certif proof. It also returns a Boolean value reflecting the success of the checker.

Both these commands call SMTCoq's internal checker, which is proven correct in Coq. We implement an integration between alethe-producing SMT solvers and Coq via a certified checker for alethe in Coq, called `Alethe_Checker`. This checker is principally implemented in the same way as `Verit_Checker` and `Lfsc_Checker` — internally, we use SMTCoq's certified checker by reducing the alethe proof to a smtcoq-certif one. Since the rules in alethe are a proper superset of those in smtcoq-certif, this reduction is done via a sequence of proof transformations that gradually eliminate rules unique to alethe.

The goal of this integration is twofold. First, we want to implement a certified checker for alethe in Coq. Carcara is the only alternative for checking alethe proofs; it is a stand-alone proof checker for the alethe proof format implemented in Rust. A checker certified in Coq carries with it the endorsement of Coq's TCB, thus increasing the authenticity of the solvers producing alethe proofs. Second, we want to address the issue of incomplete automation of goals by SMT solvers due to rewrites of formulas performed by them. Because SMT solvers often rewrite input formulas before they use them in a proof of unsatisfiability, the proof is contingent upon the rewrites being valid, often given as subgoals to the Coq user, or causing SMTCoq to fail. alethe has extensive rules documenting the rewrites of formulas by the veriT SMT solver, and one of our proof transformations eliminates such rewrites. `Alethe_Checker` also handles rewrites produced by cvc5 by representing the cvc5 rewrite rules in terms of the veriT ones.

We propose a sequence of proof transformations, most of them focusing on a particular kind of proof rule from Section 5.3.1 (rules in alethe that are not in smtcoq-certif). We will present all transformations necessary to reduce an alethe proof soundly to an smtcoq-certif proof. This is a moving target since alethe is an evolving format. Currently, it supports propositional logic, quantified reasoning, and the theories of equality over uninterpreted functions (EUF), bit-vectors (BV), and linear arithmetic over integers and reals (LIA and LRA), with a planned extension to the theory of arrays with extensionality (AX). We integrate SMTCoq with alethe for propositional logic and EUF, and for universal quantifier instantiation.

We also add some support for alethe's LIA rules but these extensions are incomplete and not fully tested. One reason for the incompleteness is that alethe's LIA rules are currently evolving. For the stable LIA rules in alethe, we use the `Micromega` solver's decision procedure to check steps that use these rules. This approach is generally successful, except for rewrite rules that combine the LIA theory with other theories. The `Micromega` solver fails on such steps since it can only prove purely arithmetic steps. Due to its incompleteness, we omit the formalization of the alethe checker for the theory of linear integer arithmetic.

5.4.1 Correctness of Checking by Transformations

As mentioned in Section 2.5, SMTCoq uses Coq's computational reflection capabilities to replace a proof term by a computation over a certificate produced by an external SMT solver. To do so, it provides:

1. A representation of formulas used by external SMT solvers using an internal type defined in Gallina. This type, `form`, represents a *deep embedding* of SMT formulas into Coq. In contrast, representing an SMT formula directly as a term of Coq's `Bool` type constitutes a *shallow embedding*.

2. A Boolean checker, `checker`, for a deeply-embedded formula and a certificate (in the smtcoq-certif format) that claims to prove it. The checker returns `true` if the certificate indeed proves the formula, and `false` otherwise.

3. A proof of correctness of the checker, `checker_correct`, that says that if the checker returns `true` for a formula and a certificate that proves it, the corresponding formula in the shallow embedding holds.

Given some formula f of type `form` (in the deep embedding), a certificate specifies how to derive $\langle\,\rangle$ from its negation, thus proving the validity of f. The certificate is composed of steps, each of which corresponds to the application of a proof rule to a formula. `checker` is modularly implemented as illustrated in Figure 5.3. It is composed of multiple *small checkers* each of which checks a particular kind of step (for example, steps involved in the conversion of a formula to CNF, and steps in the LIA theory). The *main checker* goes through the entire certificate and calls the corresponding small checker for each step. After applying the last step, it checks whether $\neg f$ has been reduced to $\langle\,\rangle$ and returns `true` if that is the case, and `false` otherwise. Similarly, `checker_correct` is composed of the correctness lemmas for each small checker.

Given that alethe is a (proper) superset of smtcoq-certif, one way of extending SMTCoq to check alethe proofs is by extending `checker` with steps that correspond to the new rules (those that exist in alethe but not in smtcoq-certif). This would break `checker_correct`, which would also need to be adapted for the new rules. However, at least for the chosen theories (propositional logic, EUF, LIA, and quantifier instantiation), while alethe is larger than smtcoq-certif in terms of the number of rules, it is not more expressive. In other words, the application of rules unique to alethe can be expressed in terms of the application of

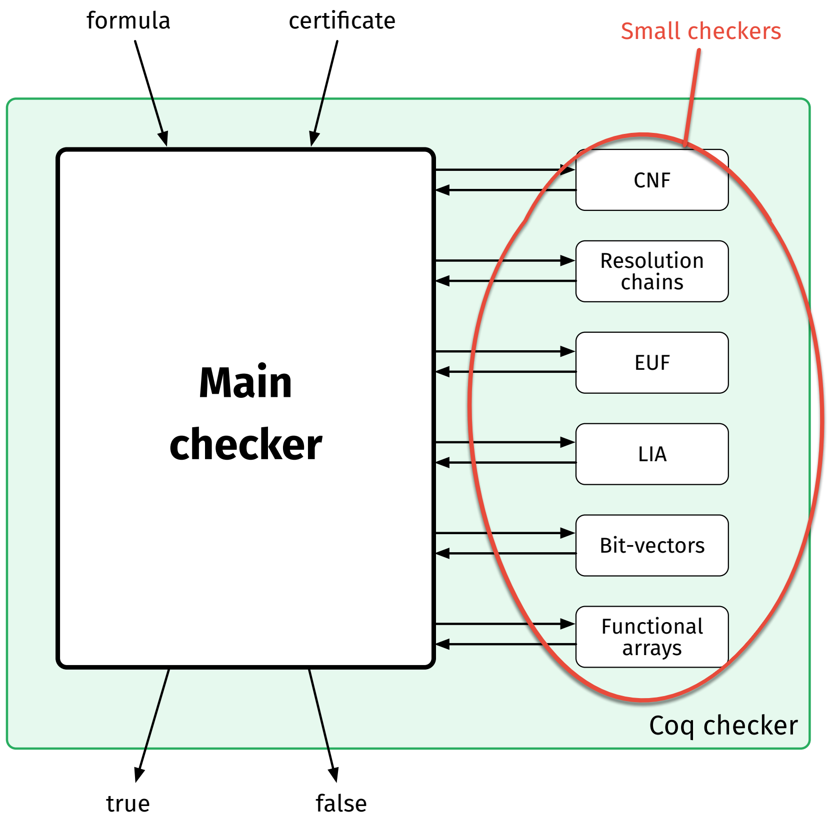


Figure 5.3: Architecture of the SMTCoq checker.

the rules in smtcoq-certif. Consequentially, a second approach for extending SMTCoq to support alethe rules is by transforming applications of new rules into applications of rules that already exist in smtcoq-certif. In our implementation of `Alethe_Checker`, when possible, we choose this second approach. That is, we apply transformations to alethe certificates to reduce them to smtcoq-certif certificates. We find transforming certificates to be less complicated than extending `checker_correct`, a complex proof with many intricacies. Our transformations are performed in a pre-processing phase over the ASTs that represent the alethe certificates in OCaml code. Although this is untrusted code, it does not effect the soundness of `Alethe_Checker` because a transformed alethe proof certificate simply tells `checker` how to reduce the negation of the input formula into $\langle\,\rangle$ via steps that it already understands (in smtcoq-certif). The correctness of the checker in Coq guarantees that if a certificate in the smtcoq-certif format can prove a formula in the deep embedding, then the corresponding formula in the shallow embedding holds. It does not matter how the certificate proves the formula; multiple certificates can do this as long as they are faithful to the smtcoq-certif format. This is a significant advantage of computational reflection.

5.4.2 Transformations

We now present our certificate transformations. As already mentioned, we classify rules as conversion and non-conversion rules, with the latter further classified into assumptions, subproofs and lemmas. All proofs that we consider are aimed at reducing a set of assumptions used to derive $\langle\ \rangle$, and so each of our rule types can be seen as playing a part in this reduction. Conversion rules directly modify clauses in some way, whereas non-conversion rules do this indirectly. Assumptions are usually subject to (in)direct modification. Subproofs and lemmas participate in clause modification via resolution steps. A clause that has more than one literal is indirectly modified since all the direct modification rules expect a clause of one literal.

Example 5.4.1. Whereas the `and` rule directly projects a conjunct from a conjunction as follows:

$$\dfrac{\overline{a \wedge b}}{a}\ \texttt{and}(1)$$

the `andp` rule is used to do this indirectly:

$$\dfrac{\overline{a \wedge b} \qquad \overline{\neg(a \wedge b), a}\ \texttt{andp}(1)}{a}\ \texttt{res}$$

We now present some useful definitions for the specification of the proof transformations. Consider some proof P that proves $C_1, \ldots C_n \models G$. For any clause C derived within P such that $s_C = (i_C, r_C[p_1, \ldots, p_n], C, a_c)$ is the step that derives C, we define its *path*, $\texttt{path}(C)$ inductively as follows.

- $s_C \in \texttt{path}(C)$
- Any step $s_i = (i, r[p_1, \ldots, p_m], C_i, a) \in \texttt{path}(C)$ if the steps pointed to by any of $p_1, \ldots, p_m$ is also in $\texttt{path}(C)$.

In other words, the path of a clause C starts at step s_C that derives C and contains all steps that use C directly or indirectly. Notice that only the first step of $\texttt{path}(C)$ (the step that derives C) can be a non-conversion rule. All other rules must be conversion rules.

A proof transformation might add a *clause residue* — some formula R — to a clause C, derived by step s_C in P. Assuming that the addition of R to C is sound, to maintain soundness in the rest of the proof, the residue must be propagated down the path of C in P. This is done using a function called `extend_cl` that given a proof P and an ID i, such that some residue R is added to clause C derived at i, replaces all conversion rules except `res` (resolution) in $\texttt{path}(C)$ to indirect modifications via `res`, as in Example 5.4.1; and then adds the residue to all `res` derivations.

Example 5.4.2. Consider the following proof P:

$$\begin{array}{l}1\\2\end{array}\quad \dfrac{\dfrac{\dfrac{}{\neg(a\wedge b)}\,\texttt{assume}}{\neg a,\neg b}\,\texttt{nand} \qquad \dfrac{\dfrac{}{c\to b}\,\texttt{assume}}{\neg c, b}\,\texttt{imp}}{\neg a,\neg c}\,\texttt{res}$$

Consider proof P' that adds residue R to the assumption at ID 1.

$$\begin{array}{l}1\\2\end{array}\quad \dfrac{\dfrac{\dfrac{}{\neg(a\wedge b), R}\,\texttt{assume}}{\neg a,\neg b}\,\texttt{nand} \qquad \dfrac{\dfrac{}{c\to b}\,\texttt{assume}}{\neg c, b}\,\texttt{imp}}{\neg a,\neg c}\,\texttt{res}$$

This makes the proof unsound. Moreover, simply propagating this residue down $\texttt{path}(1)$ in P will not suffice for soundness, because the `nand` rule expects a singleton clause with a negated conjunction as its premise. $\texttt{extend\_cl}(P', 1)$ corrects for the added residue:

$$1\quad \dfrac{\dfrac{\dfrac{}{\neg(a\wedge b), R}\,\texttt{assume} \qquad \dfrac{}{a\wedge b,\neg a,\neg b}\,\texttt{andn}}{\neg a,\neg b, R}\,\texttt{res} \qquad \dfrac{\dfrac{}{c\to b}\,\texttt{assume}}{\neg c, b}\,\texttt{imp}}{\neg a,\neg c, R}\,\texttt{res}$$

`extend_cl` replaces the `nand` derivation by a derivation via its corresponding non-conversion rule `andn` and `res`, and also adds the residue to all the `res` steps in $\texttt{path}(1)$.

Similarly, all conversion rules (except resolution) have their respective non-conversion rules that `extend_cl` uses.

Terms vs Formulas An important difference between alethe and smtcoq-certif, especially relevant to some of the transformations, comes from the internal representation of formulas in SMTCoq. Particularly, the internal representation of SMTCoq differentiates formulas (terms of type *Bool*) from terms of other types (which we will refer to as just terms when comparing them with formulas). This differentiation comes from the rules in verit2016, which treat them distinctly, and influenced the rules in smtcoq-certif, which in turn affected SMTCoq's deep embedding. For example, smtcoq-certif has separate rules for congruence over predicates (of return type *Bool*) and functions (of other return types). As a consequence, SMTCoq's internal representation of the equivalence operator (that is concerned with equating predicate applications) is independent from that of equality (for equating non-Boolean function applications). alethe does not treat these differently — this is evident from the fact that it provides a single congruence rule to account for both forms of congruence discussed above. Therefore, a reduction from alethe to smtcoq-certif must often make the difference between terms and formulas (equalities and equivalences) explicit so that the correct rules can be applied from the smtcoq-certif format.

5.4.2.1 $\mathcal{T}_s$: Subproof Flattening

To reiterate from Section 5.1, subproofs are used to prove lemmas locally inside a proof. One can see a subproof as introducing a *box* inside the proof, one that is opened using local assumptions and closed using the `subproof` rule. The following subproof introduces hypotheses $H_1, \ldots, H_n$ and discharges them to prove G inside the box, and the box derives a proof of the clause $\neg H_1, \cdots, \neg H_n, G$.

$$\dfrac{\dfrac{}{H_1}\ \texttt{assume} \quad \vdots \quad \dfrac{}{H_n}\ \texttt{assume} \quad \vdots \quad G}{\neg H_1, \cdots, \neg H_n, G}\ \texttt{subproof}$$

A subproof is a conceptualization of an implicative proof in natural deduction. The steps in the subproof (including its assumptions) constitute its local context, and steps outside the box (including the implication derived by the `subproof` rule) constitute the global context.

Example 5.4.3. Consider the following proof P:

$$\dfrac{\dfrac{}{x \wedge y}\ \texttt{assume} \qquad \dfrac{}{\neg x}\ \texttt{assume} \qquad 3\ \dfrac{2\ \dfrac{1\ \dfrac{}{x \wedge y}\ \texttt{assume}}{x}\ \texttt{andp}}{\neg(x \wedge y), x}\ \texttt{subproof}}{\langle\ \rangle}\ \texttt{res}$$

Steps 1, 2, and 3 in P constitute a subproof that derives x form $x \wedge y$, generating a derivation of the lemma $\neg(x \wedge y), x$. Steps 1 and 2 are in the local context of the subproof, whereas 3 is in the global context of P.

SMTCoq does not support the opening and closing of additional contexts within the context of a proof, and so $\mathcal{T}_s$ *flattens* subproofs so that the resultant proof has one global context. Consider the following proof, P, representing an arbitrary proof containing a single subproof with a single hypothesis. The flattening can be naturally extended to multiple subproofs, possibly containing multiple assumptions.

$$\begin{array}{c} \Pi_1 \\ \dfrac{}{H}\ \texttt{assume} \\ \Pi_2 \\ \dfrac{G}{\neg H, G}\ \texttt{subproof} \\ \Pi_3 \\ \langle\ \rangle \end{array}$$

Π_1 is the sequence of steps before the subproof; the subproof assumes H and proves G via the sequence of steps Π_2; and Π_3 follows the `subproof` step, to derive $\langle\ \rangle$. Note that Π_2 and Π_3 would have at least one step, the former deriving G from H and the latter deriving $\langle\ \rangle$ from $\neg H, G$. $\mathcal{T}_s(P)$, the flattened version of P is presented as follows:

$$
\dfrac{
  \dfrac{
    \begin{array}{c}
      \dfrac{\Pi_1}{H \land \neg G, \neg H, G}\,\texttt{andn} \\
      \Pi_3' \\
      H \land \neg G \quad (\mathbf{1})
    \end{array}
    \qquad
    \dfrac{}{\neg(H \land \neg G), H}\,\texttt{andp}
  }{
    \begin{array}{c} H \\ \Pi_2 \\ G \end{array}
  }\,\texttt{res}
  \qquad
  \dfrac{
    \dfrac{}{H \land \neg G}\,(1)
    \qquad
    \dfrac{}{\neg(H \land G), \neg G}\,\texttt{andp}
  }{\neg G}\,\texttt{res}
}{\langle\ \rangle}\,\texttt{res}
$$

Given that P derives $\langle\ \rangle$ from $H_1, H_2, \ldots, H_n$, $\mathcal{T}_s(P)$ performs the same derivation while soundly eliminating the local context of the subproof. To understand why this derivation is sound:

- First, notice that Π_1 remains unchanged, as does its relative ordering with the rest of the proof.
- The local assumption of H from the subproof is replaced by a sound derivation of H. $\neg H, G$ — previously derived by the subproof — is now independently derived using `andn` along with a residual $H \land \neg G$. Since Π_3 derives $\langle\ \rangle$ (the empty clause) from $\neg H, G$, it will derive the residue $H \land \neg G$ from $H \land \neg G, \neg H, G$ after some modifications to its steps. Specifically,

  $$
  \begin{array}{c} H \land \neg G, \neg H, G \\ \Pi_3' \end{array}
  \quad = \quad
  \texttt{extend\_cl}\left(\begin{array}{c} H \land \neg G, \neg H, G \\ \Pi_3 \end{array} \quad , \mathrm{i} \right)
  $$

  where i is the ID of the step that derives $H \land \neg G, \neg H, G$. So, `extend_cl` propagates the residue through Π_3, deriving Π_3'. Finally, H is projected from this residue (using `andp`).
- Π_2 derives G from H, that is now in the global context.
- To derive $\langle\ \rangle$, $\neg G$ is projected from $H \land \neg G$ and resolved with G.

Whereas Π_2 precedes Π_3 in P, $\mathcal{T}_s(P)$ reverses this order. This does not effect soundness, since Π_2 is within the subproof's context, due to which no step from after the subproof accesses any step from Π_2. However, Π_2 might access steps from the global context that occur before the subproof, and so the the ordering between Π_1 and Π_2 is still maintained by $\mathcal{T}_s$.

Example 5.4.4. The flattening of P from Example 5.4.3 is given by $\mathcal{T}_s(P)$:

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{}{(1)}
      \qquad
      \dfrac{}{\neg((x \land y) \land \neg x), x \land y}\,\texttt{andp}(0)
    }{x \land y}\,\texttt{res}
  }{x}\,\texttt{and}(0)
  \qquad
  \dfrac{
    \dfrac{}{(1)}
    \qquad
    \dfrac{}{\neg((x \land y) \land \neg x), \neg x}\,\texttt{andp}(1)
  }{\neg x}\,\texttt{res}
}{\langle\ \rangle}\,\texttt{res}
$$

where (1) is derived as follows:

$$\dfrac{\dfrac{}{x \land y}\ \texttt{assume} \qquad \dfrac{}{\neg x}\ \texttt{assume} \qquad \dfrac{}{(x \land y) \land \neg x, \neg(x \land y), x}\ \texttt{andn}}{(x \land y) \land \neg x \quad (\mathbf{1})}\ \texttt{res}$$

5.4.2.2 $\mathcal{T}_n$: notnot Elimination

$\mathcal{T}_n$ soundly eliminates all `notnot` steps in a certificate:

$$\dfrac{}{\neg\neg\neg x, x}\ \texttt{notnot}$$

alethe uses such steps to eliminate double negations in terms. For example, consider the following sequence of steps that eliminates the double negation from $\neg\neg x$, where C denotes the remainder of the clause.

$$\dfrac{\dfrac{}{\neg\neg x, C} \qquad \dfrac{}{\neg\neg\neg x, x}\ \texttt{notnot}}{x, C}\ \texttt{res}$$

Since the term representation of SMTCoq implicitly simplifies double negations, such a rule is unnecessary.

$\mathcal{T}_n$ removes each occurrence of the `notnot` rule, and also removes any call to a `notnot` step from the premise list of any resolution in the rest of the certificate. The sequence of steps previously mentioned, for example, is transformed to the following by $\mathcal{T}_n$:

$$\dfrac{\dfrac{}{\neg\neg x, C}}{x, C}\ \texttt{res}$$

and SMTCoq's internal representation (that we are not explicitly formalizing here) eliminates the double negation.

5.4.2.3 $\mathcal{T}_c$ and $\mathcal{T}_t$: Encoding Conversion Versions of Congruence, Transitivity, and Reflexivity

The `cong` and `trans` rules from the alethe proof format encapsulate the congruence of equality/equivalence over function/predicate applications and the transitivity of equality/equivalence, respectively:

$$\dfrac{x_1 =^\sigma x_2 \quad \cdots \quad x_n =^\sigma x_{n+1}}{x_1 =^\sigma x_{n+1}}\ \texttt{trans} \qquad \dfrac{x_1 =^\sigma y_1 \quad \cdots \quad x_n =^\sigma y_n}{f\ x_1 \cdots x_n =^\sigma f\ y_1 \cdots y_n}\ \texttt{cong}$$

for any sort σ.

smtcoq-certif has similar (non-conversion) rules:

$$\frac{}{(x_1 \neq^\sigma x_2), \cdots, (x_{n-1} \neq^\sigma x_n), (x_1 \neq^\sigma x_n)}\ \texttt{eqtrans}$$

$$\frac{}{(x_1 \neq^\sigma y_1), \cdots, (x_n \neq^\sigma y_n), (f\ x_1\ \cdots\ x_n =^\sigma f\ y_1\ \cdots\ y_n)}\ \texttt{eqcong}$$

$$\frac{}{(x_1 \neq^{Bool} y_1), \cdots, (x_n \neq^{Bool} y_n), \neg(p\ x_1\ \cdots\ x_n), (p\ y_1\ \cdots\ y_n)}\ \texttt{eqcongp}$$

where σ is a non-*Bool* sort. The goal of $\mathcal{T}_c$ (resp. $\mathcal{T}_t$) is to encode `cong` (`trans`) in terms of `eqcong` and `eqcongp` (`eqtrans`). This presents a few challenges since `cong` and `trans` are significantly more expressive than their lemma counterparts.

Equality vs Equivalence While `cong` supports congruence over equality/equivalence of both function and predicate applications, `eqcong` supports only the function case, and `eqcongp`, the predicate case. Given a `cong` step, $\mathcal{T}_c$ case-splits on whether the congruence is over an equality of function applications (case 1 below), or an equivalence of predicate applications (case 2). Similarly, for transitivity, $\mathcal{T}_t$ case-splits on whether the conclusion is an equality (case 1) or an equivalence (case 2). `eqtrans`, that supports only transitivity over equality, is used to encode the former, while the latter is done using the equivalence rules and resolution.

- **Case 1**
  This case deals with encoding applications of `cong` and `trans` when the literals in the premises and conclusion are equalities (between non-*Bool* terms).
  For congruence, convert a step of the form:

$$\frac{\overline{x = a} \quad \overline{y = b}}{f(x, y) = f(a, b)}\ \texttt{cong}$$

  to one of the form:

$$\frac{\dfrac{}{x \neq a, y \neq b, f(x, y) = f(a, b)}\ \texttt{eqcong} \qquad \overline{x = a} \quad \overline{y = b}}{f(x, y) = f(a, b)}\ \texttt{res}$$

  For transitivity, convert a step of the form:

$$\frac{\overline{x_1 = x_2} \quad \dots \quad \overline{x_{n-1} = x_n}}{x_1 = x_n}\ \texttt{trans}$$

  to one of the form:

$$\frac{\dfrac{}{x_1 \neq x_2, \dots, x_{n-2} \neq x_{n-1}, x_{n-1} = x_n}\ \texttt{eqtrans} \qquad \overline{x_1 = x_2} \quad \dots \quad \overline{x_{n-1} = x_n}}{x_1 = x_n}\ \texttt{res}$$

- **Case 2**
  Here, we are concerned with encoding applications of `cong` and `trans` when the literals in the premises and conclusion of the step are equivalences (between formulas).
  For congruence, convert a step of the form:

$$\dfrac{\overline{x = a} \qquad \overline{y = b}}{P(x,y) = P(a,b)}\ \texttt{cong}$$

to one of the form:

$$\dfrac{\overline{(1)} \qquad \overline{(2)} \qquad \overline{x = a} \qquad \overline{y = b}}{P(x,y) = P(a,b)}\ \texttt{res}$$

where (1) and (2) are derived as follows:

$$\dfrac{\dfrac{}{x \neq a, y \neq b, \neg P(x,y), P(a,b)}\ \texttt{eqcongp} \qquad \dfrac{}{P(x,y) = P(a,b), P(x,y), P(a,b)}\ \texttt{eqvn2}}{x \neq a, y \neq b, P(a,b), P(x,y) = P(a,b) \quad (\mathbf{1})}\ \texttt{res}$$

$$\dfrac{\dfrac{}{x \neq a, y \neq b, \neg P(a,b), P(x,y)}\ \texttt{eqcongp} \qquad \dfrac{}{P(x,y) = P(a,b), \neg P(x,y), \neg P(a,b)}\ \texttt{eqvn1}}{x \neq a, y \neq b, \neg P(a,b), P(x,y) = P(a,b) \quad (\mathbf{2})}\ \texttt{res}$$

For transitivity, convert a step of the form (we present transitivity over 2 premises, but this is easily generalized to n premises):

$$\dfrac{\overline{a = b} \qquad \overline{b = c}}{a = c}\ \texttt{trans}$$

to one of the form:

$$\dfrac{\overline{(1)} \qquad \overline{(2)}}{a = c}\ \texttt{res}$$

where (1) and (2) are derived as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a \neq b, a, \neg b}\ \texttt{eqvp1} \qquad \overline{a = b}}{a, \neg b}\ \texttt{res} \qquad \dfrac{\dfrac{}{b \neq c, b, \neg c}\ \texttt{eqvp1} \qquad \overline{b = c}}{b, \neg c}\ \texttt{res}}{a, \neg c}\ \texttt{res} \qquad \dfrac{}{a = c, a, c}\ \texttt{eqvn2}}{a = c, a \quad (\mathbf{1})}\ \texttt{res}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{b \neq c, \neg b, c}\ \texttt{eqvp2} \qquad \overline{b = c}}{\neg b, c}\ \texttt{res} \qquad \dfrac{\dfrac{}{a \neq b, \neg a, b}\ \texttt{eqvp2} \qquad \overline{a = b}}{\neg a, b}\ \texttt{res}}{\neg a, c}\ \texttt{res} \qquad \dfrac{}{a = c, \neg a, \neg c}\ \texttt{eqvn1}}{a = c, \neg a \quad (\mathbf{2})}\ \texttt{res}$$

Reflexivity SMTCoq implements reflexivity using transitivity. As with `eqtrans`, `eqrefl` works only for equalities, so reflexive equalities derived by `refl` can simply be derived by `eqrefl` instead. However, reflexive equivalences derived by the `refl` rule are encoded using a simpler version of the transitivity encoding from Case 2. We encode the following (for formula x):

$$\dfrac{}{x =^{Bool} x}\ \texttt{refl}$$

as the following:

$$\dfrac{\dfrac{}{x =^{Bool} x, \neg x}\ \texttt{eqvn1} \qquad \dfrac{}{x =^{Bool} x, x}\ \texttt{eqvn2}}{x =^{Bool} x}\ \texttt{res}$$

Logical Operators `eqcongp` does not support congruence over logical operators (that is, cases where the predicate applied by congruence is a logical operator) whereas `cong` does, warranting a separate encoding of congruence over each of the logical operators ($\wedge$, $\vee$, $\neg$, $\rightarrow$, and $=^{Bool}$) using their corresponding introduction and elimination rules. For instance, to encode congruence over the $\neg$ predicate, a step of the form:

$$\dfrac{\overline{x = a}}{\neg x = \neg a}\ \texttt{cong}$$

is encoded as:

$$\dfrac{\overline{(1)} \qquad \overline{(2)}}{\neg x = \neg a}\ \texttt{res}$$

where (1) and (2) are derived as follows:

$$\dfrac{\dfrac{\overline{x = a} \qquad \dfrac{}{x \neq a, \neg a, x}\ \texttt{eqvp2}}{\neg a, x}\ \texttt{res} \qquad \dfrac{}{\neg x = \neg a, \neg x, \neg a}\ \texttt{eqvn2}}{\neg a, \neg x = \neg a \quad (\mathbf{1})}\ \texttt{res}$$

$$\dfrac{\dfrac{\overline{x = a} \qquad \dfrac{}{x \neq a, \neg x, a}\ \texttt{eqvp1}}{\neg x, a}\ \texttt{res} \qquad \dfrac{}{\neg x = \neg a, x, a}\ \texttt{eqvn1}}{a, \neg x = \neg a \quad (\mathbf{2})}\ \texttt{res}$$

The encodings for the other logical operators, although more elaborate, do not pose any interesting challenges and so they are omitted. The logical equivalence operator is an exception that is addressed in what follows.

Congruence over Equality/Equivalence Consider congruence over equality/equivalence, where the predicate applied to literals from the premises is itself equality/equivalence.

Example 5.4.5. Suppose a, b, x, and y are formulas. Then, the following is a valid application of the `cong` rule.

$$\dfrac{\overline{x = y} \qquad \overline{a = b}}{(x = a) = (y = b)}\ \texttt{cong}$$

Such an application is also valid if a, b, x, and y are non-Boolean terms.

As mentioned earlier, SMTCoq stores equalities and equivalences separately in its internal format. So, congruence over $=^{Bool}$ is encoded as done for the other logical operators, and $=^{\phi}$ for all other sorts ϕ can be processed using `eqcongp` (Case 2 from the Equality vs Equivalence paragraph of this section). Such a differentiation is also necessary for processing the `trans` rule that combines reasoning for equivalence and equality.

Implicit Arguments Another difference between `cong` (`trans`) and its previous counterparts is that it supports implicit reflexive arguments. For example, in derivation

$$\dfrac{\overline{a = b}}{f\ a\ x = f\ b\ x}\ \texttt{cong}$$

$x = x$ is an implicit argument. The rule as currently specified by alethe does not allow implicit arguments for the `cong` rule (although this has been reported to the maintainers of the alethe specification so that such an accommodation can be made), but solvers producing alethe proofs generate such derivations. To be able to support such solvers, our encoding permits implicit arguments. In our encoding, we search for implicit arguments and explicitly prove them using `refl` for terms, or using `eqvn1` and `eqvn2` for formulas (see the Reflexivity paragraph).

5.4.2.4 $\mathcal{T}_r$: Encoding Rewrites

Rewrite rules specify rewrites that take the form of an equality/equivalence and are applied (indirectly) to terms in the proof. For each possible rewrite specified by alethe's rewrite rules (specified in Section 5.3.1), $\mathcal{T}_r$ replaces the equality/equivalence by a derivation using rules in the smtcoq-certif format. For each formula rewrite from alethe that takes the form of an equivalence $a =^{Bool} b$, $\mathcal{T}_r$ takes the following approach:

1. Derive $\neg a, b$ by `subproof`, that is, by assuming a and deriving b.
2. Derive $\neg b, a$ by `subproof`, that is, by assuming b and deriving a.
3. Use these to derive $a =^{Bool} b$ as follows:

$$\dfrac{\dfrac{\dfrac{}{a =^{Bool} b, \neg a, \neg b}\ \texttt{eqvn1} \quad \dfrac{}{\neg a, b}\ \texttt{subproof}}{a =^{Bool} b, \neg a}\ \texttt{res} \qquad \dfrac{\dfrac{}{a =^{Bool} b, a, b}\ \texttt{eqvn2} \quad \dfrac{}{\neg b, a}\ \texttt{subproof}}{a =^{Bool} b, a}\ \texttt{res}}{a =^{Bool} b}\ \texttt{res}$$

The subproofs are handled using $\mathcal{T}_s$ (Section 5.4.2.1). For each rewrite $a =^{Bool} b$, the derivation of a from b for Step 1 and of b from a for Step 2 are specified in Appendix A.

The cvc5 solver's rewrite rules are different from those used by veriT, and consequentially from those specified in alethe. Instead the cvc5 rewrite rules are derived using the `allsimp` rule. Since this one rule covers many possible equalities/equivalences, supporting it is a non-trivial task, and is described in its own section below.

5.4.2.5 $\mathcal{T}_f$: Handling Forall Instantiation

For our alethe integration with SMTCoq, we focus primarily on the quantifier-free theory of EUF. This is compatible with Coq's `Bool` type which serves as the deep embedding of formulas for SMTCoq. Note, however, that quantified formulas aren't expressible within this type. Despite this restriction, SMTCoq supports certain proofs containing quantified reasoning thanks to an extension by Blot et al. [22]. Particularly, it supports instantiations of universally quantified hypotheses (in prenex normal form, where all the quantifiers are in the prefix of the formula). Here, we summarize the extension by Blot et al. and describe our adaptation of it for alethe's quantifier instantiation rules.

Quantified hypotheses cannot be deeply embedded, but are represented in Coq as `Prop`s. A quantified formula is eliminated from the proof by instantiating it to a quantifier-free formula via the `forall_inst` rule:

$$3\ \dfrac{1\ \dfrac{}{\forall x_1, \ldots x_n.\ P}\ \texttt{assume} \qquad 2\ \dfrac{}{\neg(\forall x_1, \ldots x_n.\ P) \vee P[x_1 \mapsto t_1] \ldots [x_n \mapsto t_n]}\ \texttt{forall\_inst}}{P[t_1 \ldots t_n]}\ \texttt{res}$$

SMTCoq is unable to represent the clauses derived at IDs 1 and 2 in its deep embedding. It performs the derivation of $P[t_1 \ldots t_n]$ outside of its computatational reflection mechanism (while still maintaining its soundness guarantees). Specifically, it replaces the derivation at ID 3 by a derivation of $P[t_1 \ldots t_n]$ via an application of the *modus ponens* rule (supported by Coq's logic, so it is applied at the meta level):

$$\dfrac{\overline{P \to Q} \qquad \overline{P}}{Q}\ \textit{modus ponens}$$

Thus, to derive a proof of $P[t_1 \ldots t_n]$, SMTCoq needs a proof of $\forall x_1, \ldots x_n.\ P \to P[t_1 \ldots t_n]$ and proof of $\forall x_1, \ldots x_n.\ P$, both in Coq's `Prop` type. The latter is available from the assumption (ID 1). To prove the implication between the quantified formula and its instance, SMTCoq uses a variant of the `auto` tactic [31]. In other words, `auto` proves the universal quantifier instantiation, given the instance from the SMT solver. In this way, SMTCoq uses the SMT solver to search for the instance, employing external help from Coq to complete the proof. This constrains SMTCoq so that it can support only a very restricted form of quantifier instantiation.

The checker for `forall_inst` takes both the quantified hypothesis and its instance, and invokes `auto` to prove the implication between them. However, both these pieces are not always readily available within the same immediate derivation as in steps 1, 2, and 3 above. The SMT solver often performs modifications of quantified formulas that include α-renamings of its bound variables. $\mathcal{T}_f$ processes these modifications and eliminates them when possible (unnecessary α-renamings, for example). It also finds the original hypothesis that is being instantiatied and pairs it with its instance for the `forall_inst` checker.

5.4.2.6 $\mathcal{T}_{tr}$: Eliminating Trivial Clauses

A trivial clause is one that contains x as well as $\neg x$ for some literal x. These arise in some of our proofs, sometimes introduced by other transformations.

Example 5.4.6. The `eqvp1` rule would introduce a trivial clause when applied to an equality between the same literal. Consider the following proof that contains such a trivial clause introduced by `eqvp1`:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x = x, x, \neg x}\,\texttt{eqvp1} \quad \dfrac{}{x \neq x}\,\texttt{assume}}{1\ \ x, \neg x}\,\texttt{res} \quad \dfrac{}{x}\,\texttt{assume}}{x}\,\texttt{res} \quad \dfrac{}{\neg x}\,\texttt{assume}}{\langle\ \rangle}\,\texttt{res}$$

It is straightforward to see that this proof is sound. However, the resolution checker for SMTCoq will fail on this proof. Recursive function `res_checker`(C_1, C_2) defines the operation of the checker for resolution steps given two clauses C_1 and C_2:

```
res_checker(C1, C2)
1: for each x in C1 do
2:     if x ∈ C2 then
3:         return x :: res_checker(C1 \ x, C2 \ x)
4:     else if ¬x ∈ C2 then
5:         return (C1 \ x) ++ (C2 \ ¬x)
6:     end if
7: end for
8: Fail
```

where $::$ is the list *cons* operator that given an element x of type A and a list l of elements of type A, creates a new list with the element x added to l (here we treat clauses as lists of literals which aligns with SMTCoq's internal representation); for clause C and element x, $C \backslash x$ is the clause identical to C except that it does not contain x.

`res_checker` makes an optimization on line 3 assuming that a literal will never appear in both polarities within the same clause. While this assumption might have been practical to make in the past, solvers using alethe produce certificates that render this assumption to be too strong. In Example 5.4.6, for instance, the optimization is applied at step 1 so that the resolution between clauses $[x, \neg x]$ and $[x]$ returns x`::res_checker`$([\neg x], \langle\ \rangle)$, making the recursive call fail.

To prevent the checker from failing on certificates like this, we implement certificate transformation $\mathcal{T}_{tr}$ that soundly removes trivial clauses. Particularly, $\mathcal{T}_{tr}$ soundly removes any clause C such that $x \in C, y \in C$ for some literals x and y that are negations of each other. This includes syntactic negation (x and $\neg x$), negation modulo double negation elimination ($\neg x$ and $\neg\neg\neg x$), and negation modulo symmetry of equality ($x = y$ and $y \neq x$, when x and

y are not Boolean literals). Reasoning modulo double negation elimination and symmetry of equality is necessary since SMTCoq reasons this way.

Recall that a clause with more than one literal is only indirectly converted by resolving it with other clauses. And so given a trivial clause $C_1, x, \neg x$ at ID t_1, where C_1 represents the rest of the clause, we have some clause at t_3 that resolves t_1 with a clause at some ID t_2. t_2 and t_3 can take three possible forms based on three possible pivots:

1. t_2 contains x, so that the certificate P can be generalized as:

   $$\begin{array}{llll} (t_1, & \_\_, & C_1, x, \neg x &) \\ & \ldots & & \\ (t_2, & \_\_, & C_2, x &) \\ & \ldots & & \\ (t_3, & \texttt{res}[L, t_1, M, t_2, N], & C_3, x &) \end{array}$$

   where C_1, C_2, and C_3 are placeholders for the irrelevant parts of the clauses; similarly, L, M, and N represents irrelevant parts of the resolution chain, and __ denotes the name of the rule that derives the clause.

2. t_2 contains $\neg x$; the derivation of t_3 is similar to the one above, except that it derives $C_3, \neg x$ instead.

3. t_2 contains some y such that its negation is in C_1. In this case, t_3 will contain x and $\neg x$ making it a trivial clause.

We specify $\mathcal{T}_{tr}$ for case 1 above; the transformation for case 2 is very similar, and for case 3 $\mathcal{T}_{tr}$ is recursively applied to the trivial clause created at t_3. Given certificate P (from case 1), $\mathcal{T}_{tr}$ (P) soundly removes the step at t_1:

$$\begin{array}{llll} & \ldots & & \\ (t_2, & \_\_, & C_2, x &) \\ & \ldots & & \\ (t_{3a}, & \texttt{weaken}[t_2], & C_1, C_2, x &) \\ (t_3, & \texttt{res}[L, t_{3a}, M, t_2, N], & C_3, x &) \end{array}$$

Recall that the `weaken` rule permits the weakening of a clause by adding arbitrary literals to it. So the non-trivial part of t_1 which is necessary for the soundness of the proof is preserved using `weaken`, so that the trivial part can be eliminated from the proof.

Example 5.4.7. $\mathcal{T}_{tr}$ would transform the proof from Example 5.4.6 to the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x}\,\texttt{assume}}{x = x, x}\,\texttt{weaken} \qquad \dfrac{}{x \neq x}\,\texttt{assume}}{x}\,\texttt{res} \qquad \dfrac{}{\neg x}\,\texttt{assume}}{\langle\ \rangle}\,\texttt{res}$$

$\mathcal{T}_{tr}$ removes the `eqvp1` step that introduces the trivial clause and maintains soundness using `weaken`.

Recall from Section 5.4.1 that there are two ways to adapt SMTCoq for a new rule: add a checker for the rule and prove its correctness, or pre-process the rule to recast it in terms of rules that SMTCoq already supports. While we follow the latter approach for all the new rules from alethe, the problem with trivial clauses is a good candidate for modifying the SMTCoq checker. This modification requires removing the optimization from line 3 in `res_checker` and fixing its proof of correctness. Since `res` is a central rule to SMTCoq's proof calculus, and since it occurs quite often (even more so due to the transformations specified in this chapter), a change to its checker must be made only after ensuring that such a change does not reduce efficiency. We leave as future work, a comparison of both approaches to handling trivial clauses and the consequent implementation of the more efficient one.

5.4.3 cvc5 Rules and Rewrites

One goal of the alethe proof format is to allow SMT solvers to completely justify the steps that they take in proving the validity of a set of formulas. This includes not only the reduction of their negation to the empty clause, but any lemmas that solver might use in the proof. A particular kind of lemma of concern is one that rewrites terms within a proof. Rewrites take the form of equalities (or equivalences) and are used to indirectly modify terms using resolution and rules for equivalence introduction and elimination.

Example 5.4.8. The following proof uses a rewrite rule (`andsimp`) to reduce a conjunction by removing a redundant $\top$ from its conjuncts.

$$\dfrac{\dfrac{\dfrac{}{\neg(x \land \top = x), \neg(x \land \top), x}\ \texttt{eqvp2} \qquad \dfrac{}{x \land \top = x}\ \texttt{andsimp}}{\neg(x \land \top), x}\ \texttt{res} \qquad \dfrac{}{x \land \top}\ \texttt{assume}}{x}\ \texttt{res}$$

Rewrites pose one of the main barriers in unifying proof certificate formats between SMT solvers — since different solvers use similar algorithms to solve problems in particular theories, they agree on the general steps of proof reduction, but how they choose to simplify formulas before or during the reduction process is unique to each solver. Although alethe specifies elaborate rules for term rewrites, these are influenced by the veriT SMT solver. cvc5, on the other hand, almost never rewrites terms using these rules (even when it does, it doesn't use the same rule name for the rewrite). Instead, it produces almost all its rewrites using the `allsimp` rule. Our support for alethe rewrite rules `andsimp`, `orsimp`, `notsimp`, `impsimp`, `eqvsimp`, `boolsimp`, and `eqsimp` only cover rewrites of veriT. Additonally, we need to support `allsimp`, which is a general rule covering multiple possible rewrites emitted by cvc5. We use cvc5's ability to reconstruct its rewrites in terms of a particular set of rewrite rules that can be declared to it using a domain specific language called RARE [78]. We use the RARE language to describe the alethe rewrite rules mentioned above, and cvc5's DSL compiler then reconstructs its rewrites in terms of these rules. A reconstructed rule is still derived using `allsimp`, but specifies as its argument the RARE rule to use to derive it. The implementation of `Alethe_Checker` then simply parses these arguments and uses

SMT Solver	Proof Certificate Format	Checker	# Benchmarks	# Successful Checks	# Failed Checks
CVC4	LFSC	`Lfsc_Checker`	138	131	7
cvc5	alethe	`Alethe_Checker`	138	128	10
veriT v2016	verit2016	`Verit_Checker`	138	138	0
veriT	alethe	`Alethe_Checker`	138	138	0

Figure 5.4: Summary of results of checking proofs produced by CVC4, cvc5, veriT v2016, and veriT on the set of reduced benchmarks.

SMT Solver	#Benchmarks	# Successful Checks	# Successful Checks with Holes	# Failed Checks	# Holes	# Files with Holes
CVC4	138	54	77	7	153	82
cvc5	138	84	44	10	66	44

Figure 5.5: Comparison of `Lfsc_Checker` and `Alethe_Checker`'s performance with CVC4 and cvc5 respectively

the respective rewrite rule for the derivation of the equality. As the following experimental results show, this process helps reduce the number of holes significantly.

5.5 Evaluation

In this section, we detail our experiments on `Alethe_Checker`, comparing it with `Verit_Checker` and `Lfsc_Checker`, for SMT files and their proofs in propositional logic and the theory of equality over uninterpreted functions (EUF). We use a set of 4260 benchmarks (SMT files) [89] generated by the Sledgehammer [81] tool when it calls external SMT solvers to prove goals in the Isabelle/HOL ITP. All benchmarks are unsatisfiable, and can be solved by either veriT, CVC4, or z3 within 12 seconds. These benchmarks express queries over SMT-LIB 2's core propositional logic and theories of linear integer arithmetic (LIA), arrays (AX), uninterpreted functions (EUF), using both quantified and quantifier-free formulas. We filter this benchmark set so that we are considering only those SMT files that are restricted to propositional logic and the EUF theory. Furthermore, we remove any files whose proofs (in alethe) include the `acsimp` rule (that simplifies nested occurrences of $\wedge$ and $\vee$), since we have not added support for it yet. This leaves us with 138 benchmarks on which we perform our comparison. We ran all experiments on CoqIDE version 8.13.2 in a system with 16 GB RAM, running Ubuntu 20.04.

First, we call CVC4 and veriT v2016 on these 138 SMT files and have them produce proof certificates in the LFSC and verit2016 proof certificate formats, respectively. We then call `Lfsc_Checker` (for CVC4) and `Verit_Checker` (for veriT v2016) on the SMT files with the corresponding proof files. These results are in the first and third rows of the table in Figure 5.4. The last column (# Failed Checks) includes files for which the checker does not return `True` (it either returns `False` or raises an error). For CVC4 all 7 failures are caused by exceptions raised in SMTCoq's code. The `Lfsc_Checker` has a high success rate on these benchmarks; however, many of the successful checks produce holes — proof steps that cannot be checked. On the other hand, `Verit_Checker` is complete for these benchmarks — all files are successfully checked without any holes.

As a comparison, we call cvc5 and veriT on the same 138 SMT files to produce alethe proof certificates. We then call `Alethe_Checker` on the SMT files and proof files from the two solvers. The results from these experiments are in the second and fourth rows of the table in Figure 5.4. With veriT, we match the 100% success rate of veriT v2016, and with cvc5 we come close to the success rate of CVC4, since the checker only fails on 10 benchmarks, and all of them are cases where the same exception is raised by SMTCoq's code. We do not expect the fixing of these 10 failures to be particularly challenging and leave it as future work.

This table only shows a partial picture of the comparison. In checking the proof certificate of an external SMT solver, SMTCoq can leave unjustified steps. Such a step — called a proof *hole* — is returned to the Coq user as a sub-goal. The table in Figure 5.5 presents the experimental results while taking proof holes into consideration. Since both versions of veriT produce no holes in their proofs over the benchmarks, this table omits veriT results. For CVC4, we found 153 holes from the LFSC proofs of the benchmark files, and 82 files with at least one hole. All holes (from both solvers) appeared due to rewrite steps that SMTCoq couldn't justify. Considering successful checks now to be only those cases where the checker succeeds without finding any holes in the proof, we note that there are 54 of these. Notice from the second row, that this number has increased to 84 using cvc5 and our alethe checker; and that the number of holes in proofs have been halved owing to the added support for rewrite rules in `Alethe_Checker`.

In our future work, we propose to bring down the number of holes in alethe proofs produced by cvc5 down to 0 (from 66). All 66 holes can be encoded using rules currently supported by alethe fairly easily. For example, over half of these rewrites take one of the following forms:

- $\neg\top =^{Bool} \bot$
- $(\bot =^{Bool} \top) =^{Bool} \bot$
- $(\top =^{Bool} \bot) =^{Bool} \bot$
- $(x = x) =^{Bool} \top$ for some x.

However, hard-coding a transformation for each form of rewrite rule would require a large amount of code with little potential for reuse. Instead, we propose to use veriT version v2016 as an *elaborator* for these rewrite rules. Since the proofs produced by that version of veriT are well-supported by SMTCoq we can use it to replace the derivation of a rewrite rule by a derivation using rules that are already supported by SMTCoq. We see such a solution as being general enough to cover not only the 66 rewrites from this experiment set, but any rewrite (for the relevant theories) produced by cvc5.

Chapter 6

Proving Invertibility Conditions

Many applications in hardware and software verification rely on bit-precise reasoning, which can be modeled using the SMT-LIB 2 theory of fixed-width bit-vectors. While satisfiability modulo theories (SMT) solvers are able to reason about bit-vectors of fixed width, they currently require all widths to be expressed concretely (by a numeral) in their input formulas. For this reason, they cannot be used to prove properties of bit-vector operators that are parametric in the bit-width, such as the associativity of bit-vector concatenation. Interactive theorem provers (ITPs) such as Coq, which have direct support for dependent types, are better suited for such tasks. Bit-vector formulas that are parametric in the bit-width arise in the verification of parametric Boolean functions and circuits [61]). In our case, we are mainly interested in parametric lemmas that are relevant to internal techniques of SMT solvers for the theory of fixed-width bit-vectors. These include, for example, rewrite rules, refinement schemes, and preprocessing passes. Such techniques are developed a priori for every possible bit-width. Meta-reasoning about the correctness of such solvers then requires bit-width independent reasoning.

In this chapter, we focus on parametric lemmas that originate from a quantifier-instantiation technique implemented in the SMT solver cvc5. This technique is based on *invertibility conditions* [73] (previously introduced in Section 2.2.1). For a trivial case of an invertibility condition, consider the equation $x + s = t$ where x, s and t are variables of the same bit-vector sort. In the terminology of Niemetz et al. [73], this equation is "invertible for x." A general inverse, or "solution," is given by the term $t - s$. Since there is always such an inverse, the invertibility condition for $x+s = t$ is simply the universally true formula $\top$. The formula stating this fact, referred to here as an *invertibility equivalence*, is $\top \Leftrightarrow \exists x.\, x + s = t$, which is valid in the theory of fixed-width bit-vectors, for any bit-width. In contrast, the equation $x \cdot s = t$ is not always invertible for x. A necessary and sufficient condition for invertibility in this case was found in [73] to be $(-s \mid s)\ \&\ t = t$. So, the invertibility equivalence $(-s \mid s)\ \&\ t = t \Leftrightarrow \exists x.\, x \cdot s = t$ is valid for any bit-width. Notice that the invertibility condition does not contain x. Hence, invertibility conditions can be seen as a technique for quantifier elimination. In [73], a total of 160 invertibility conditions were provided. However, they were verified only for bit-widths up to 65, due to the reasoning limitations of SMT solvers mentioned earlier. Recent work [75, 74] addresses this challenge by translating the invertibility equivalences to the combined theory of non-linear integer arithmetic and

Symbol	SMT-LIB Syntax	Sort
$=, \neq$	=, distinct	$\sigma_{[n]} \times \sigma_{[n]} \rightarrow Bool$
$<_u, >_u, \leq_u, \geq_u$	bvult, bvugt, bvule, bvuge	$\sigma_{[n]} \times \sigma_{[n]} \rightarrow Bool$
$\sim, -$	bvnot, bvneg	$\sigma_{[n]} \rightarrow \sigma_{[n]}$
$\&, \mid, \ll, \gg, \gg_a$	bvand, bvor, bvshl, bvlshr, bvashr	$\sigma_{[n]} \times \sigma_{[n]} \rightarrow \sigma_{[n]}$
$+$	bvadd	$\sigma_{[n]} \times \sigma_{[n]} \rightarrow \sigma_{[n]}$
$<_s, >_s, \leq_s, \geq_s$	bvslt, bvsgt, bvsle, bvsge	$\sigma_{[n]} \times \sigma_{[n]} \rightarrow Bool$
$\cdot$, mod, $\div$	bvmul, bvurem, bvudiv	$\sigma_{[n]} \times \sigma_{[n]} \rightarrow \sigma_{[n]}$
$\circ$	concat	$\sigma_{[n]} \times \sigma_{[m]} \rightarrow \sigma_{[n+m]}$
$[u : l]$	extract	$\sigma_{[n]} \rightarrow \sigma_{[u-l+1]}$

Figure 6.1: The signatures Σ_1 and Σ_0 with SMT-LIB 2 syntax. Σ_1 consists of the operators in the entire table. Σ_0 consists of the operators in the upper part.

uninterpreted functions. This approach was partially successful, but failed to verify over a quarter of the equivalences.

We verify invertibility equivalences proposed in [73] by proving them interactively in Coq. From a representative subset of the invertibility equivalences, we prove 19 equivalences, 12 of which were not proven in [75, 74]. For the remaining 7, that were already proved there, our Coq proofs provide more confidence. Our results offer evidence that ITPs can support ATPs in meta-verification tasks. To facilitate the verification of invertibility equivalences, we use a rich Coq library for bit-vectors, which is a part of SMTCoq [50]. The remainder of this chapter is organized as follows. Section 6.1 introduces the theory of bit-vectors that is relevant to this work followed by an introduction invertibility conditions in Section 6.2. The Coq library that we use and our extensions to it are specified in Section 6.3; Section 6.4 discusses the Coq proofs using detailed examples and Section 6.5 summarizes the results.

6.1 Theory of Fixed-Size Bit-Vectors

We introduced signature Σ_{BV} of the SMT-LIB 2 theory of fixed-width bit-vectors in Section 2.2.1. For every positive integer n and a bit-vector of width n, the signature includes a constant of sort $\sigma_{[n]}$ in Σ_{BV} representing that bit-vector, which we denote as a binary string of length n. The function and predicate symbols of Σ_{BV} are fully described in the SMT-LIB 2 standard [11]. Formulas of Σ_{BV} are built from variables (sorted by the sorts $\sigma_{[n]}$), bit-vector constants, and the function and predicate symbols of Σ_{BV}, along with the usual logical connectives and quantifiers.

Figure 6.1 contains the operators from Σ_{BV} for which invertibility conditions were defined in [73]. We define Σ_1 to be the signature that contains only these symbols. Σ_0 is the sub-signature obtained by only taking the operators from the upper part of the table. We use the (overloaded) constant 0 to represent the bit-vectors composed of all 0-bits.

6.2 Invertibility Conditions

Recall that the problem with the model-based instantiation technique used in Example 2.2.2 is that the efficiency of the SMT solver depends on the models found for x. Particularly, the solver would take longer to find a model for ϕ if it tried all even numbers greater than 3 for possible values for x before trying any odd ones. To address such issues, Niemetz et al. [72] present a technique to solve quantified bit-vector formulas, which is based on *invertibility conditions*. An invertibility condition for a variable x in a Σ_{BV}-literal $\ell[x, s, t]$ is a formula $IC[s, t]$ such that $\forall s. \forall t. IC[s, t] \Leftrightarrow \exists x. \ell[x, s, t]$ is valid in the theory of fixed-width bit-vectors. For example, the invertibility condition for x in the bit-vector literal $x \mathbin{\&} s = t$ (where x, s and t are distinct variables of the same sort, and & is the bit-wise conjunction operation) is $t \mathbin{\&} s = t$.

Example 6.2.1. This example is borrowed from Jonáš et al. [67]. Consider ϕ from Example 2.2.2 after the first iteration. Instead of adding instance $x \neq 2 \cdot 2$ to the formula, which prohibits only 4, invertibility conditions-based instantiation prevents all values of x satisfying this disequality. The invertibility condition for y is a formula that specifies the conditions under which $x = 2 \cdot y$ holds: $((-2 \mid 2) \mathbin{\&} x) = x$. In other words, values for y satisfying $x = 2 \cdot y$ exist only when $((-2 \mid 2) \mathbin{\&} x) = x$ holds (notice that this is true only for even values of x). So the solver adds its negation to the formula:

$$3 <_u x \wedge \neg((-2 \mid 2) \mathbin{\&} x = x) \wedge \forall y(x \neq 2 \cdot y)$$

which prevents all even values of x, forcing the solver to come up with a value for x which is not even and is greater than 3, such as 5.

As described in Section 2.2.1, cvc5 performs quantifier instantiation via invertibility conditions for the theory of bit-vectors.

Niemetz et al. [72] define invertibility conditions for a representative set of literals ℓ having a single occurrence of x, that involve the bit-vector operators of Σ_1. The soundness of the technique proposed in that work relies on the correctness of the invertibility conditions. Every literal $\ell[x, s, t]$ and its corresponding invertibility condition $IC[s, t]$ induce the *invertibility equivalence*

$$IC[s, t] \Leftrightarrow \exists x. \ell[x, s, t] \tag{6.1}$$

The correctness of invertibility equivalences should be verified for all possible sorts for the variables x, s, t for which the condition is well sorted. More concretely, for the case where x, s, t are all of sort $\sigma_{[n]}$, say, this means that one needs to prove, for *all* $n > 0$, the validity of

$$\forall s : \sigma_{[n]}. \forall t : \sigma_{[n]}. IC[s, t] \Leftrightarrow \exists x : \sigma_{[n]}. \ell[x, s, t] \ .$$

This was done in Niemetz et al. [72] using SMT solvers but only for concrete values of n from 1 to 65. A proof of Equation (6.1) that is parametric in the bit-width n cannot be done with SMT solvers, since they currently only support the theory of *fixed-width* bit-vectors, where Equation (6.1) cannot even be expressed. To overcome this limitation, a

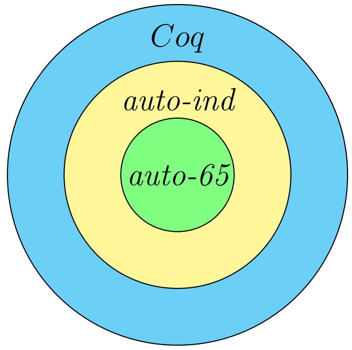


Figure 6.2: The level of confidence achieved by the different approaches.

later paper by Niemetz et al. [74] suggested a translation from bit-vector formulas with *parametric* bit-widths to the theory of (non-linear) integer arithmetic with uninterpreted functions. Thanks to this translation, the authors were able to verify, with the aid of SMT solvers for the theory of integer arithmetic with uninterpreted functions, the correctness of 110 out of 160 invertibility equivalences. None of the solvers used in that work were able to prove the remaining equivalences. For those, it then seems appropriate to use an ITP, as this allows for more intervention by the user who can provide crucial intermediate steps. It goes without saying that even for the 110 invertibility equivalences that were proved, the level of confidence achieved by proving them in a proof-assistant such as Coq would be greater than a verification (without a verified formal proof) by an SMT solver.

Figure 6.2 depicts the level of confidence achieved by the various approaches to verify invertibility equivalences. The smallest circle, labelled *auto-65*, represents the approach taken by [73], where invertibility equivalences were verified automatically up to 65 bits. While a step in the right direction, this approach is insufficient, because invertibility conditions are used for arbitrary bit-widths. The next circle, labeled *auto-ind*, depicts the approach of [74], which addresses the restrictions of auto-65 by providing bit-width independent proofs of the invertibility equivalences. However, both auto-65 and auto-ind provide proofs by SMT solvers, which are less trusted than ITPs. The largest circle (*Coq*) corresponds to work presented in this chapter which, while addressing the limitations of auto-65 via bit-width independent proofs, also provides stronger verification guarantees by proving the equivalences in an interactive theorem prover. Moreover, with this approach, we were able to prove equivalences that couldn't be fully verified (for arbitrary bit-widths) by either auto-65 or auto-ind.

6.3 The `BVList` Library

In this section, we describe the Coq library we use and the extensions we developed with the goal of formalizing and proving invertibility equivalences. Various formalizations of bit-vectors in Coq exist. The internal Coq library of bit-vectors [46] is one, but it has only definitions and no lemmas. The Bedrock Bit Vectors Library [28] treats bit-vectors as words (machine integers). The SSRBit Library [21] represents bit-vectors as finite bit-sets in Coq and extracts them to OCaml machine integers. Our library is more suited to the SMT-LIB 2 bit-vectors, and includes operators that are not fully covered by any of the previously mentioned libraries. More recently, Shi et al. [90] developed a library called CoqQFBV that presents a bit-vector type as a sequence of Booleans, defines operators over it, and proves the correctness of these operations with respect to a (machine integer) semantics. [90] uses this library to define a bit-blasting algorithm in Coq, that is extracted into an OCaml program to perform certified bit-blasting. Since CoqQFBV covers the entire SMT-LIB 2 bit-vector signature, it would be a good alternative to ours in formalizing and proving invertibility conditions. Our library offers a rich set of lemmas over bit-vector operations that makes it suitable for proofs of invertibility conditions and other bit-vector properties. Bit-vectors have also been formalized in other proof assistants. Within the Isabelle/HOL framework, one can utilize the library developed by Beeren et al. [16] to align with SMT-LIB 2 bit-vector operations. Furthermore, Harrison [2] presents a formalization of finite-dimensional Euclidean space within HOL light, accompanied by an implementation of vectors.

6.3.1 `BVList` Without Extensions

`BVList` was developed for SMTCoq [50], a Coq plugin that enables Coq to dispatch proofs to external proof-producing solvers. While the library was only briefly mentioned in [50], here we provide more details.

The library adopts the little-endian notation for bit-vectors, following the internal representation of bit-vectors in SMT solvers such as cvc5, and corresponding to lists in Coq. This makes arithmetic operations easier to perform since the least significant bit of a bit-vector is the head of the Boolean list that represents it.

For formalizing the bit-vector type, a dependently-typed definition is natural, allowing the type of a bit-vector to be parameterized by its length. However, such a representation leads to some difficulties in proofs. Dependent pattern-matching or case-analysis with dependent types is cumbersome and unduly complex (see, e.g., [92]), because of the complications brought by unification in Coq (which is inherently undecidable [93]). A simply-typed definition, on the other hand, does not provide such obstacles for proofs, but is less natural, as the length becomes external to the type. The `BVList` library defines for convenience both the dependently and the simply typed version of bit-vectors. It uses the Coq module system to separate them, and a functor that connects them, avoiding redundancy. The relationship between the two definitions is depicted in Figure 6.3.

In `BVList`, a dependently-typed bit-vector is a record parameterized by its size n and consisting of two fields: a Boolean list and a condition to ensure that the list has length n. This type, and the corresponding lemmas and properties over it, are encapsulated by the `BITVECTOR_LIST` module of type `BITVECTOR`. A simply-typed or *raw* bit-vector representation is simply a Boolean list which, along with its associated operators and lemmas is specified by module signature `RAWBITVECTOR` and implemented in module `RAWBITVECTOR_LIST`. In other words, the interface of `BVList` offers dependently-typed bit-vectors, while the underlying operators are defined and proofs are performed using raw bit-vectors.

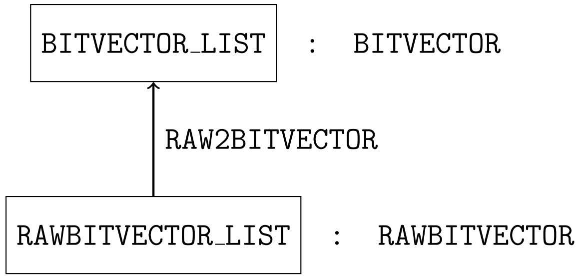


Figure 6.3: Modular separation of `BVList`

A functor called `RAW2BITVECTOR` derives corresponding definitions and proofs over dependently-typed bit-vectors within the module for dependent-types, when it is applied to `RAWBITVECTOR_LIST`. The functor establishes a correspondence between the two theories so that one can first prove a bit-vector property in the context of the simply-typed theory and then map it to its corresponding dependently-typed one via the functor module. Otherwise put, users of the library can encode theorem statements more naturally, and in a more expressive environment employing dependent types. For proofs, one can unlift them (by the functor) to the equivalent encodings with simple types, and prove them there.

6.3.2 Extending `BVList`

Out of the 13 bit-vector functions and 10 predicates contained in Σ_1, `BVList` had direct support for 10 functions and 6 predicates. The predicate symbols that were not directly supported were the weak inequalities $\leq_u$, $\geq_u$, $\leq_s$, $\geq_s$ and the unsupported function symbols were $\gg_a$, $\div$, and mod. We extended `BVList` with the operator $\gg_a$ and the predicates $\leq_u$ and $\geq_u$ in order to support the corresponding invertibility conditions. Additionally, we redefined $\ll$ and $\gg$ in order to simplify the proofs of invertibility conditions over them.[1]

We focused on invertibility conditions for literals of the form $x \diamond s \bowtie t$ and $s \diamond x \bowtie t$, where $\diamond$ and $\bowtie$ are respectively function and predicate symbols in Σ_0. Σ_0 was chosen as a representative set because it is both expressive enough (in the sense that other operators can be easily translated to this fragment), and feasible for proofs in Coq using the library. In particular, it was chosen as one that would require the minimal amount of changes to `BVList`. As a result, such literals, as well as their invertibility conditions, contain only operators supported by `BVList` (after its extension with $\gg_a$, $\leq_u$, and $\geq_u$). Supporting the full set of operators in Σ_1, both in the library and the proofs is left for future work.

In what follows, we describe our extensions to `BVList` with weak unsigned inequalities, alternative definitions for logical shifts, and the arithmetic right shift operator.

[1]Both the extended library and the proofs of invertibility equivalences can be found at https://github.com/ekiciburak/bitvector/tree/frocos23.

```
Fixpoint ule_list_big_endian (x y : list bool) :=
  match x, y with
   | [ ], [ ] => true
   | [ ], _  => false
   | _, [ ] => false
   | xi :: x', yi :: y' => ((eqb xi yi)
                          && (ule_list_big_endian x' y'))
                          || ((negb xi) && yi)
  end.

Definition ule_list (x y: list bool) :=
  (ule_list_big_endian (rev x) (rev y)).

Definition bv_ule (a b : bitvector) :=
  if @size a =? @size b then
    ule_list a b
  else
    false.

Definition bv_ule n (bv1 bv2:bitvector n) : bool :=
  M.bv_ule bv1 bv2.
```

Figure 6.4: Definitions of $\leq_u$ in Coq.

6.3.2.1 Weak Unsigned Inequalities

We added both weak inequalities for unsigned bit-vectors, $\leq_u$ and $\geq_u$. We illustrate this extension via that of the $\leq_u$ operator (the extension of $\geq_u$ is similar). The relevant Coq definitions are provided in Figure 6.4. The top three definitions (including the fixpoint) cover the simply-typed representation, and the fourth, `bv_ule` is the dependently-typed representation that invokes the definition with the same name from module `M` of type `RAWBITVECTOR`. Like most other operators, $\leq_u$ (over raw bit-vectors) is defined over a few *layers*. The function `bv_ule`, at the highest layer, ensures that comparisons are between bit-vectors of the same size and then calls `ule_list`. Since we want to compare bit-vectors starting from their most significant bits and the input lists start instead with the least significant bits, `ule_list` first reverses the two lists. Then it calls `ule_list_big_endian`, which we consider to be at the lowest layer of the definition. This function does a lexicographic comparison of the two lists, starting from the most significant bits.

To see why the addition of $\leq_u$ to the library is useful, consider, for example, the following parametric lemma, stating that ~ 0 is the largest unsigned bit-vector of its type:

$$\forall x : \sigma_{[n]}.\, x \leq_u \sim 0 \tag{6.2}$$

Without an operator for the weak inequality, we would write it as:

$$\forall x : \sigma_{[n]}.\, x <_u \sim\!0 \vee x = \sim\!0 \tag{6.3}$$

In such cases, since the definitions of $<_u$ and $=$ have a similar structure to that of $\leq_u$, we strip down the layers of $<_u$ and $=$ separately, whereas using $\leq_u$, we only do this once.

6.3.2.2 Left and Right Logical Shifts

We have redefined the shift operators $\ll$ and $\gg$ in `BVList`. Figure 6.5 shows both the original and new definitions of $\ll$. Those of $\gg$ are similar. Originally, $\ll$ was defined using the `shl_one_bit` and `shl_n_bits`. The function `shl_one_bit` shifts the bit-vector to the left by one bit and is called by `shl_n_bits` as many times as necessary. The new definition `shl_n_bits_a` uses `mk_list_false` which constructs the necessary list of 0 bits and appends (`++` in Coq) to it the bits to be shifted from the original bit-vector, which are retrieved using the `firstn` function, from the Coq standard library for lists. The `nat` type used in Figure 6.5 is the Coq representation of Peano natural numbers that has `O` and `S` as its two constructors — as depicted in the cases rendered by pattern matching n (lines 10-11). The theorem at the bottom of Figure 6.5 asserts the equivalence of the two representations, allowing us to switch between them, when needed. In the extended library, `bv_shl` defines the left shift operation using `shl_n_bits` whereas `bv_shl_a` does it using `shl_n_bits_a`. This new representation was useful in proving some of the invertibility equivalences over shift operators. (see, e.g., Example 6.4.3 below).

6.3.2.3 Arithmetic Right Shift

Unlike logical shifts that were already defined in `BVList` and for which we have added alternative definitions, arithmetic right shift was not defined at all. We provided two alternative definitions for it, very similar to the definitions of logical shifts — `bv_ashr` and `bv_ashr_a`. Both definitions are conditional on the sign of the bit-vector (its most-significant bit). Apart from this detail, the definitions take the same approach taken by `shl_n_bits` and `shl_n_bits_a` from Figure 6.5. `bv_ashr` uses the definition of an independent shift and repeats it as many number of times as necessary, and `bv_ashr_a` uses either `mk_list_false` or `mk_list_true` to append the necessary number of sign bits to the shifted bits.

6.4 Proving Invertibility Equivalences in Coq

In this section we provide specific details about proving invertibility equivalences in Coq. We start by outlining the general approach for proving invertibility equivalences in Section 6.4.1. Then, Section 6.4.2 presents detailed examples of such proofs.

```
Definition shl_one_bit (a: list bool) :=
  match a with
   | [ ] ⇒ [ ]
   | _ ⇒ false :: removelast a
  end.

Fixpoint shl_n_bits (a: list bool) (n: nat) :=
  match n with
   | O ⇒ a
   | S n' ⇒ shl_n_bits (shl_one_bit a) n'
  end.

Definition shl_n_bits_a (a: list bool) (n: nat) :=
  if (n <? length a)%nat then
    mk_list_false n ++ firstn (length a -n) a
  else
    mk_list_false (length a).

Theorem bv_shl_eq: forall (a b : bitvector),
  bv_shl a b = bv_shl_a a b.
```

Figure 6.5: Various definitions of $\ll$.

6.4.1 General Approach

The natural representation of bit-vectors in Coq is the dependently-typed representation, and therefore the invertibility equivalences are formulated using this representation. In keeping with the modular approach described in Section 6.3, however, proofs in this representation are composed of proofs over simply-typed bit-vectors, which are easier to reason about. Most of the work is on proving an equivalence over raw bit-vectors. Then, we derive the proof of the corresponding equivalence over dependently-typed bit-vectors using a smaller, boilerplate set of tactics. Since this derivation process is mostly the same across many equivalences, these tactics are a good candidate for automation in the future.

When proving an invertibility equivalence $IC[s,t] \Leftrightarrow \exists x.\ \ell[x,s,t]$, we first split it into two sub-goals: the left-to-right and right-to-left implications. For proving the left-to-right implication, since Coq implements a constructive logic, the only way to prove an existentially quantified formula is to construct the literal witnessing it. Thus, in addition to being able to prove the equivalence, a positive side-effect of our proofs are actual inverses for x in literals of the form $\ell[x,s,t]$. In Niemetz et al. [75], these are called *conditional inverses*, as the fact that they are inverses is conditional on the correctness of the invertibility condition. There, such inverses were synthesized automatically for a subset of the literals. In each of our Coq proofs, such an inverse is found, even when the proof is done by case-splitting. This provides a more general solution than the one in [75], which did not consider case-splitting.

Example 6.4.1. Consider the literal $s \gg_a x \geq_u t$. Its invertibility condition is $(s \geq_u \sim s) \vee (s \geq_u t)$. The left-to-right implication of the invertibility equivalence is:

$$\forall s, t : \sigma_{[n]}.\ (s \geq_u \sim s) \vee (s \geq_u t) \Rightarrow \exists x : \sigma_{[n]}.\ s \gg_a x \geq_u t$$

Here, case splitting is done on the disjunction in the invertibility condition. When $s \geq_u \sim s$ is true, the inverse for x is the bit-vector constant that correspond to the length of the s, namely n; when $s \geq_u t$ is true, the inverse is 0. □

In addition to `BVList`, several proofs of invertibility equivalences benefited from CoqHammer [35], a plug-in that aims at extending the level of automation in Coq by combining machine learning and automated reasoning techniques in a similar fashion to what is done by Sledgehammer [81] in Isabelle/HOL [76]. CoqHammer, when triggered on some Coq goal, (*i*) submits the goal together with potentially useful terms to external solvers/automated-provers, (*ii*) attempts to reconstruct returned proofs (if any) directly in the Coq tactic language Ltac [41], and (*iii*) outputs the set of tactics closing the goal in case of success. As we directly employ these tactics inside `BVList`, one does not need to install CoqHammer in order to build the library, although it would be beneficial for further extensions.

6.4.2 Detailed Examples

In this section we provide specific examples for proofs of invertibility equivalences. The first example illustrates the two-theories approach of the library.

Example 6.4.2. Consider the literal $s \gg_a x <_u t$. Its invertibility condition is $((s <_u t \vee \neg(s <_s 0)) \wedge t \neq 0)$. Figure 6.6 shows the proof of the following direction of the corresponding invertibility equivalence:

$$\forall s, t : \sigma_{[n]}.\ (\exists x : \sigma_{[n]}.\ s \gg_a x <_u t) \Rightarrow ((s <_u t \vee \neg(s <_s 0)) \wedge t \neq 0)$$

In the proof, lines 8–11 transform the dependent bit-vectors from the goal and the hypotheses into simply-typed bit-vectors. Then, lines 12–14 invoke the corresponding lemma for simply-typed bit-vectors (called `InvCond.bvashr_ult2_rtl`) along with some simplifications. □

Most of the effort in this project went into proving equivalences over raw bit-vectors, as the following example illustrates.

Example 6.4.3. Consider the literal $x \ll s >_u t$. Its invertibility condition is $(t <_u \sim 0 \ll s)$. The corresponding invertibility equivalence is:

$$\forall s, t : \sigma_{[n]}.\ (t <_u \sim 0 \ll s) \Leftrightarrow (\exists x : \sigma_{[n]}.\ x \ll s >_u t) \qquad (6.4)$$

The left-to-right implication is easy to prove using ~ 0 itself as the witness of the existential proof goal and considering the symmetry between $>_u$ and $<_u$. The proof of the right-to-left implication relies on the following lemma:

```
Theorem bvashr_ult2_rtl :
forall (n : N), forall (s t : bitvector n),
(exists (x : bitvector n), (bv_ult (bv_ashr_a s x) t = true)) ->
((( bv_ult s t = true) ∨ (bv_slt s (zeros n)) = false) ∧
(bv_eq t (zeros n)) = false).
Proof.
intros n s t H.
destruct H as ((x, Hx), H).
destruct s as (s, Hs).
destruct t as (t, Ht).
unfold bv_ult, bv_slt, bv_ashr_a, bv_eq, bv in *. cbn in *.
specialize (InvCond.bvashr_ult2_rtl n s t Hs Ht); intro STIC.
rewrite Hs, Ht in STIC. apply STIC.
now exists x.
Qed.
```

Figure 6.6: A proof of one direction of the invertibility equivalence for $\gg_a$ and $<_u$ using dependent types.

$$\forall x, s : \sigma_{[n]}.\ (x \ll s) \leq_u (\sim 0 \ll s) \tag{6.5}$$

From the right side of the equivalence in Equation (6.4), we get some skolem x for which $x \ll s >_u t$ holds. Flipping the inequality, we have that $t <_u x \ll s$; using this, and transitivity over $<_u$ and $\leq_u$, the lemma given by Equation (6.5) gives us the left side of the equivalence in Equation (6.4).

As mentioned in Section 6.3, we have redefined the shift operators $\ll$ and $\gg$ in the library. This was instrumental, for example, in the proof of Equation (6.5). The new definition uses `firstn` and `++`, over which many useful properties are already proven in the standard library. This benefits us in manual proofs, and in calls to CoqHammer, since the latter is able to use lemmas from the imported libraries to prove the goals that are given to it. Using this representation, proving Equation (6.5) reduces to proving Lemmas `bv_ule_1_firstn` and `bv_ule_pre_append`, shown in Figure 6.7. The proof of `bv_ule_pre_append` benefited from the property `app_comm_cons` from the standard list library of Coq, whereas `firstn_length_le` was useful in reducing the goal of `bv_ule_1_firstn` to the Coq equivalent of Equation (6.2). The statements of the properties mentioned from the standard library are also shown in Figure 6.7. □

Finally, we examine what was considered a challenge problem in the previous version of this work [51]. The next example details how we completed the proof.

Example 6.4.4. Consider the literal $(x \gg s) >_u t$. Its invertibility condition is $t <_u (\sim s \gg s)$. Now consider the following direction of the corresponding invertibility equivalence:

$$\forall s, t : \sigma_{[n]}.\ t <_u (\sim s \gg s) \Rightarrow \exists x : \sigma_{[n]}.\ (x \gg s) >_u t \tag{6.6}$$

```
Lemma bv_ule_1_firstn : forall (n : nat) (x : bitvector),
(n < length x)%nat ->
bv_ule (firstn n x) firstn n (mk_list_true (length x))) = true.

Lemma bv_ule_pre_append : forall (x y z : bitvector),
bv_ule x y  = true -> bv_ule (z ++ x) (z ++ y) = true.

Theorem app_comm_cons : forall (x y:list A) (a:A),
a ::  (x ++ y) = (a :: x) ++ y

Lemma firstn_length_le: forall l:list A, forall n:nat,
n <= length l -> length (firstn n l) = n.
```

Figure 6.7: Examples of lemmas used in proofs of invertibility equivalences.

Figure 6.8 contains the theorem stating the equivalence, and some lemmas used within its proof. A crucial step in the proof of the implication is to rewrite the definition of the right shift operator `bv_shr` to its alternate definition `bv_shr_a` (see Section 6.3.2.2). Unfolding the alternative definition leads to a case-analysis on the following condition:

$$\texttt{toNat}(s) < \texttt{len}(x)$$

where `toNat` casts a bit-vector to its natural number representation, and `len` returns the length of a bit-vector as a natural number.

The challenge in the proof arises in the positive case of the condition, which reduces to a proof of `first_bits_zero` (see Figure 6.8). `first_bits_zero` says that given $\texttt{toNat}(s) < \texttt{len}(s)$, the most-significant $\texttt{len}(s) - \texttt{toNat}(s)$ bits of s are 0. As seen in Figure 6.5, the second argument to the top-most layer of the shift (called from `bv_shl_eq`) is a bit-vector that specifies the number of times to shift the bit-vector in the first argument. This second argument is converted to a natural number by the abstract `toNat` function invoked above, the concrete definitions of which are specified in Figure 6.8 as `list2nat_be_a` and `list2N`. At the same level of abstraction, we use `rev` for the list reversal function corresponding to the Coq function of the same name, and `firstn` also for its Coq namesake (`firstn` n l returns the n most significant bits of l), so that `first_bits_zero` can be specified as follows:

$$\texttt{toNat}(s) < \texttt{len}(s) \Rightarrow \texttt{firstn}\ (\texttt{len}(s) - \texttt{toNat}(s))\ (\texttt{rev}(s)) = 0$$

The intuition behind its validity is that if the most-significant $\texttt{len}(s) - \texttt{toNat}(s)$ bits were not 0 then they would contribute to the value of $\texttt{toNat}(s)$, making it greater than or equal to $\texttt{len}(s)$ and thus falsifying the condition. However, it is challenging to convert this intuition into a proof using induction over lists, as explained in what follows.

To prove `first_bits_zero`, we redefined `list2N` as a tail-recursive function `list2NTR`. This step was proven to be sound by a lemma of equivalence between the two definitions (`list2N_eq`). Since `list2N` is not tail recursive, it only begins computation at the end of

```
Theorem bvshr_ugt_ltr : forall (n : N), forall (s t : bitvector n),
(bv_ult t (bv_shr (bv_not s) s) = true) ->
(exists (x : bitvector n), bv_ugt (bv_shr x s) t = true).

Lemma first_bits_zero : forall (s : bitvector),
(N.to_nat (list2N s) < length s)%nat ->
firstn (length s -N.to_nat (list2N s)) (rev s) =
mk_list_false (length s -N.to_nat (list2N s)).

Lemma first_bits_zeroA : forall (s : bitvector),
(length s >= (list2NTR s))%nat ->
firstn (length s -(list2NTR s)) s =
mk_list_false (length s -(list2NTR s)).

Fixpoint list2N (a: list bool) :=
match a with
| [ ] => 0
| x :: xs => if x then N.succ_double (list2N xs) else
N.double (list2N xs)
end.

Definition list2nat_be_a (a: list bool) := N.to_nat (list2N a).

Fixpoint list2NR (a: list bool) (n: nat) :=
match a with
| [ ]     => n
| x :: xs => if x then list2NR xs (2 * n + 1) else
list2NR xs (2 * n)
end.

Definition list2NTR (a: list bool) := list2NR a 0.

Lemma list2N_eq: forall (s: bitvector),
list2NTR (rev s) = N.to_nat (list2N s).
```

Figure 6.8: Invertibility equivalence for $\gg$ and $>_u$ and some lemmas used by its proof.

the input list representing a bit-vector. Such a definition further complicates the proof of `first_bits_zero` when based on the typical induction principle over the structure of the Boolean list underlying the bit-vector `s`. This is because it does not easily reduce (via ι-reduction for inductive definitions [79]), into a useful expression in the step case of the intended induction.

The advantage of tail recursion in this context is best illustrated by Figure 6.9 where `x` is a Boolean variable and `xs` represents an arbitrary Boolean list. The derivation of the goal from the inductive hypothesis (`IH`) in derivation (6.7) from Figure 6.9 is complicated in Coq because the functions `firstn` and `rev` are not well-matched with `list2N`, if not incompatible. For instance, observe that the in the inductive step (`Goal`), as the first argument to `firstn` increases, the number of bits fetched from the list increases towards the *right*. However, due to the little-endian notation of bit-vectors and the fact that the list `cons` function (`::`) can be seen as incrementing its argument list to its *left*, the `rev` function must be used to corrects the direction of increase of the second argument to `firstn`. Despite this correction, an induction over s must deal with two structurally different lists.

In contrast, the tail-recursive definition of `list2NTR` hides the `rev` function. This is illustrated in derivation (6.8) in Figure 6.9, where `toNatTR` corresponds to `list2NTR`. Furthermore, such an induction over lists using `append` (`++`) to the right, rather than `cons` to the left is possible thanks to the *reverse induction principle*[2]. Closing such a goal allowed us to prove the `list2NTR`-variant of `first_bits_zero`, specified as `first_bits_zeroA` in Figure 6.8, and the proof of equivalence between the two definitions (`list2N_eq`) allowed us to use this in closing the original goal (6.6). □

$$\frac{x:\ \ \texttt{bool}\quad xs:\ \ \texttt{list bool}\quad \texttt{IH:}\ \texttt{firstn}\ (\texttt{len}(xs) - \texttt{toNat}(xs))\ (\texttt{rev}(xs)) = 0}{\texttt{Goal:}\quad \texttt{firstn}\ (\texttt{len}(xs) + 1 - \texttt{toNat}(x :: xs))\ (\texttt{rev}(x :: xs)) = 0} \tag{6.7}$$

$$\frac{x:\ \ \texttt{bool}\quad xs:\ \ \texttt{list bool}\quad \texttt{IH:}\ \texttt{firstn}\ (\texttt{len}(xs) - \texttt{toNatTR}(xs))\ (xs) = 0}{\texttt{Goal:}\quad \texttt{firstn}\ (\texttt{len}(xs) + 1 - \texttt{toNatTR}(xs \texttt{ ++ } [x]))\ (xs \texttt{ ++ } [x]) = 0} \tag{6.8}$$

Figure 6.9: Sub-goals generated in the proof of `first_bits_zero`. Note that 0 is a bit-vector constant of the appropriate length (list of `false`s).

6.5 Results

Figure 6.10 summarizes the results of proving invertibility equivalences for invertibility conditions in the signature Σ_0. In the table, ✔ means that the invertibility equivalence was successfully verified in Coq but not by Niemetz et al. [74], while ✓ means the opposite; ✔✓

[2]see `rev_ind` in https://coq.inria.fr/library/Coq.Lists.List.html

$\ell[x]$	$=$	$\neq$	$<_u$	$>_u$	$\leq_u$	$\geq_u$
$-x \bowtie t$	**✓**✓	✓	✓	✓	✓	✓
$\sim x \bowtie t$	**✓**✓	✓	✓	✓	✓	✓
$x \,\&\, s \bowtie t$	**✓**	✓	✓	✓	✓	✓
$x \mid s \bowtie t$	**✓**	✓	✓	✓	✓	✓
$x \ll s \bowtie t$	**✓**	**✓**	✓	**✓**	✓	**✓**
$s \ll x \bowtie t$	**✓**✓	✓	✓	✓	✓	✓
$x \gg s \bowtie t$	**✓**✓	✓	✓	**✓**	✓	✓
$s \gg x \bowtie t$	**✓**✓	✓	✓	✓	✓	✓
$x \gg_a s \bowtie t$	**✓**	✓	✓	✓	✓	✓
$s \gg_a x \bowtie t$	**✓**✓	✓	**✓**	**✓**	**✓**	**✓**
$x + s \bowtie t$	**✓**✓	✓	✓	✓	✓	✓

Figure 6.10: Proofs of invertibility equivalences in Σ_0. $\bowtie$ is a placeholder for the predicate symbol indicated by the column headers.

means that the invertibility equivalence was verified using both approaches. Notice that all invertibility equivalences in this table are verified by at least one of the two approaches. We successfully proved all invertibility equivalences over $=$ that are expressible in Σ_0, including 4 that were not proved by Niemetz et al. [74]. For the rest of the predicates, we focused only on the 8 invertibility equivalences that were not proved by Niemetz et al. [74], and succeeded in proving all of them.

Our work thus complements [74] in verifying all invertibility conditions in Σ_0 for arbitrary bit-widths, by proving all 12 equivalences that were previously unverified, and corroborating 7 others that were verified by SMT solvers. It also complements [73], which verified all invertibility conditions in Σ_1, but only up to bit-width of 65

Chapter 7

Conclusion and Future Work

The increasing pervasiveness of software in our world underscores the need for formal methods. We need ways to ensure that all the software in our life, whether generated by humans or AI, behave as we expect them to. Theorem provers have been advancing in efficiency and expressive power for decades, and since the turn of the century, automatic theorem provers (ATPs) have been rapidly developing to meet the demand for software verification. This development has caused them to grow substantially in size and call into question their own correctness. Parallelly, interactive theorem provers (ITPs) — tools that have stronger correctness guarantees — have also been growing while prioritizing the preservation of their guarantees over growth. Due to this preservation, ITPs still have limited support for automation. The work done in this thesis makes progress towards the goals of (1) addressing correctness concerns for ATPs and (2) enhancing automation within ITPs without losing their correctness guarantees. To these dual ends, we have demonstrated three ATP-ITP integrations.

The `abduce` Tactic Our first integration is implemented in SMTCoq — a tool that uses external SMT solvers (ATPs) in a deductive capacity to prove goals inside the Coq ITP via the `smt` tactic. We complement `smt` with a means to use an external solver abductively. The result is a Coq tactic called `abduce` that can call an external SMT solver on a failing goal so that the solver can ask the Coq user for more information in order to solve the goal. We evaluate the `abduce` tactic on three sets of benchmarks to show that when used in conjunction with the `smt` tactic, it can increase the number goals discharged by external automated tools without compromising the correctness of the ITP results. A previously failing goal is converted to a provable one by (*i*) calling the `abduce` tactic to get an abduct from the SMT solver, (*ii*) searching for (a generalization of) the abduct inside the Coq environment using Coq's `Search` command, (*iii*) locally asserting a matching lemma (if found), and finally (*iv*) calling the `smt` tactic on this renewed local context. This manual series of events can be automated using Coq's automation and tactic programming tools to save the user time and effort. We leave the implementation of such an automated tactic for future work.

In current integrations, it is common for ITPs to treat external solvers as push-button provers that either succeed or fail in proving the goal. Since ATPs often operate on a restricted set of the ITP's language, the ATP is bound to fail often. With the `abduce` tactic, we propose a more interactive approach to split this success-failure binary. Failing full automation, an ATP can still be useful to the ITP user. In this case, we leverage the SMT solver's ability to perform abductive reasoning to convert a typically failing instance into one where the external solver can be of assistance.

The alethe Checker Second, we add to SMTCoq a proof checker called `Alethe_Checker`, for the alethe proof certificate format. This allows SMTCoq to check a large class of proofs from the cvc5 and veriT SMT solvers. Previously, SMTCoq supported CVC4 with LFSC proofs and an older version of veriT. We support alethe proofs by preprocessing them into proofs in SMTCoq's internal proof certificate format. Since alethe (when considered as a set of its proof rules) is a superset of the internal format, our reduction supports more low-level proof steps and also offers a higher coverage of *rewrite rules* produced by SMT solvers, many of which were previously left unjustified. To deal with the sizable number of distinct rewrite rules, our implementation employs various methods of elaboration — checking a rule that isn't supported by reconstructing it in terms of other rules that are supported. We evaluate these claims on a benchmark set generated from calls to external SMT solvers from an ITP. Our implementation is restricted to propositional logic and the (quantifier-free) theory of equality over uninterpreted functions (EUF). However, the implementation provides all the general infrastructure necessary to extend the integration to other theories used in SMT. Ultimately, we hope to have support for alethe in at least all theories that SMTCoq currently supports cumulatively over its multiple solvers: propositional logic, equality over uninterpreted functions (EUF), linear integer arithmetic (LIA), arrays with extensionality (AX), bit-vector arithmetic (BV), and universal quantifier instantiation. Such an integration would allow SMTCoq to automatically prove a wide range of verification goals in Coq using the latest versions of the state-of-the-art SMT solvers cvc5 and veriT. The work done in this thesis will serve as a foundation for this large-scale project. The efforts put into `Alethe_Checker` serve as a useful use-case in proof engineering in a programming language. Thanks to the soundness guarantees of SMTCoq's checker, we are able to perform the heavy-lifting of proof checking in the OCaml programming language, rather than within Coq. The formalization of the transformations also offer an insight into the intricacies of fully checking the proofs produced by SMT solvers. To understand the complexities of the operation of the modern SMT solver, one must look at the fringes of its proof rules. The corner cases supported in our transformations, and the differences in rewrites produced and exceptions made by just two of the supported SMT solvers suggest that even a uniform proof format leaves room for plenty of idiosyncrasies that a particular solver can impose on its proofs. All of these must be accounted for while building a tool to check proofs in a proof format such as alethe. Owing to the variety of rewrite rules produced by an SMT solver, we will explore a change in strategy for supporting rewrite rules in `Alethe_Checker` moving forward. Instead of encoding each possible rewrite in SMTCoq's OCaml codebase, we propose using an external tool — an elaborator — that can convert a derivation of an unsupported lemma into a derivation using rules supported by `Alethe_Checker`. In Section 5.4.1, we discuss two approaches to implement a checker for alethe and our justification for choosing the approach that requires preprocessing proof certificates. Thus, the work done in implementing `Alethe_Checker` leaves SMTCoq's internal checker unchanged. Moving forward, we want to give equal consideration to the second possible method for supporting alethe rules — by extending the checker. The proposed modification to the resolution checker in Section 5.4.2.6 is a good start. Proof assistants are foundational to the project of guaranteeing software correctness. However, proving non-trivial software correct (in an ITP) is notoriously difficult. Tools like SMTCoq aim to assist in this undertaking by providing external help to the user. In order to leverage the capabilities of all the advancements in the world of ATPs, ITPs need to be integrated with multiple solvers, for multiple theories. `Alethe_Checker` is a step in this direction.

Coq Proofs of Invertibility Conditions Third, we explore recursively applying traditional formal methods to formal methods tools. Proving the correctness of an ATP within an ITP promises to be a massive project for a modern ATP. Instead, we verify a modular element of an ATP. Specifically, we verify the correctness of *invertibility conditions*, formulas used by the quantifier module of the cvc5 SMT solver over the theory of bit-vectors. The verification of these 166 equivalences was previously done using ATPs for a number of special cases. We complement that verification effort by providing proofs in Coq in the most general case for 19 equivalences. A total of 40 equivalences remain unverified in at least one direction. These could be verified by relying on the bit-vector library that we use to represent these proofs (after its extension with bit-vector multiplication and division), and on the lemmas that we have generated through our proof effort. Verification of ATPs within ITPs are uncommon owing to the size of modern ATPs. Once again, our work suggests that the success-failure binary can be extended. Being able to divide an SMT solver into modular parts and verifying some of these parts within an ITP are positive steps towards increasing its reliability.

The implementations discussed in this document are specific to the Coq ITP and the cvc5 and veriT SMT solvers, but all three contributions are general enough to be applied to integrations between other similar ATPs and ITPs. All three of the contributions presented in this thesis, though distinct from one other, play a small part in automating interactive theorem provers and certifying automated theorem provers, with the goal of offering faster, expressive, and trustworthy verification tools for software.

Appendix A

alethe Rewrite Encodings

Here, we present the proof of the left-to-right and right-to-left implications of 36 rewrite rules in alethe (5 from `andsimp`, 5 from `orsimp`, 2 from `notsimp`, 8 from `impsimp`, 8 from `eqvsimp`, 7 from `boolsimp`, and 1 from `eqsimp`) used by $\mathcal{T}_r$. The rewrites are specified in Section 5.3.1 and $\mathcal{T}_r$ in Section 5.4.2.4. We omit 17 rewrites — 1 from `notsimp` (handled instead by $\mathcal{T}_n$), 2 from `eqsimp` (handled by the `Micromega` solver), all 12 from `itesimp`, and 4 from `connective_def` (both because they contain rewrites over the xor and the if-then-else operators that we don't consider in our signature).

As shown in Section 5.3.1, each rewrite rule has a general equivalence form, and is specified by multiple possible transformations. Thus, for each rewrite rule application, $\mathcal{T}_r$ distinguishes the transformation by pattern-matching on the equivalence. The order in which the proofs of each transformation are presented in this section matches the order from the pattern-matching code rather than that of the alethe specification (which is what Section 5.3.1 follows, repeated here for convenience). Some notations and representations are simplified by SMTCoq's representation of SMT formulas. First, it implicitly removes double negations so $\neg\neg x$ is indistinguishable from x for SMTCoq. We use them interchangeably as a consequence. The clause representation of SMTCoq automatically removes duplicates and the proofs below often treat this duplicate-removal step as implicit. Note that duplicates are only automatically removed from clauses and not from disjunctions. For these, we use a function called `to_unique`, that given a collection of terms, eliminates duplicates in them:

$$\texttt{to\_unique}(x_1, \ldots x_n) = x_1, \ldots, x_{n'}$$

SMTCoq's resolution rule, as specified by case 3 in Section 5.2.1, is able to handle arbitrarily long resolution chains efficiently. Many of the proofs presented in the following take advantage of this feature; however, the chains shown here are sometimes split up due to space constraints.

A.1 Rewrites Over Conjunctions: `andsimp`

The `andsimp` rule specifies possible rewrites over conjunction terms.

$$\dfrac{}{(\varphi_1 \wedge \cdots \wedge \varphi_n) = \psi}\ \texttt{andsimp}$$

where the possible transformations are:

- $\top \wedge \cdots \wedge \top = \top$
- $x_1 \wedge \cdots \wedge x_n = x_1 \wedge \cdots \wedge x_{n'}$ where the RHS has all $\top$ literals removed.
- $x_1 \wedge \cdots \wedge x_n = x_1 \wedge \cdots \wedge x_{n'}$ where the RHS has all repeated literals removed.
- $x_1 \wedge \cdots \wedge \bot \wedge \cdots \wedge x_n = \bot$
- $x_1 \wedge \cdots \wedge x_i \wedge \cdots \wedge x_j \wedge \cdots \wedge x_n = \bot$ where $x_i = \neg x_j$

and the proofs for each transformation are:

1. $x_1 \wedge \cdots \wedge \bot \wedge \cdots \wedge x_n = \bot$

   LTR Proof:

$$\dfrac{\dfrac{}{x_1 \wedge \cdots \wedge \bot \wedge \cdots \wedge x_n}\ \texttt{assume} \qquad \dfrac{}{x_1 \wedge \cdots \wedge \bot \wedge \cdots \wedge x_n, \bot}\ \texttt{andp}}{\bot}\ \texttt{res}$$

   RTL Proof:

$$\dfrac{\dfrac{\dfrac{}{\bot}\ \texttt{assume}}{\bot, x_1 \wedge \cdots \wedge \bot \wedge \cdots \wedge x_n}\ \texttt{weaken} \qquad \dfrac{}{\neg\bot}\ \texttt{false}}{x_1 \wedge \cdots \wedge \bot \wedge \cdots \wedge x_n}\ \texttt{res}$$

2. $x_1 \wedge \cdots \wedge x_i \wedge \cdots \wedge x_j \wedge \cdots \wedge x_n = \bot$ where $x_i = \neg x_j$

   LTR Proof:

   (1) derives x_j:

$$\dfrac{\dfrac{}{x_1 \wedge \cdots \wedge x_j \wedge \cdots \wedge x_j \wedge \cdots \wedge x_n}\ \texttt{assume} \qquad \dfrac{}{x_1 \wedge \cdots \wedge \neg x_j \wedge \cdots \wedge x_j \wedge \cdots \wedge x_n, x_j}\ \texttt{andp}}{x_j \quad \mathbf{(1)}}\ \texttt{res}$$

   (2) derives $\neg x_j$:

$$\dfrac{\dfrac{}{x_1 \wedge \cdots \wedge \neg x_j \wedge \cdots \wedge x_j \wedge \cdots \wedge x_n}\ \texttt{assume} \qquad \dfrac{}{x_1 \wedge \cdots \wedge \neg x_j \wedge \cdots \wedge x_j \wedge \cdots \wedge x_n, \neg x_j}\ \texttt{andp}}{\neg x_j \quad \mathbf{(2)}}\ \texttt{res}$$

   (3) derives $\neg x, \bot$:

$$\dfrac{\dfrac{}{\neg x_j}\ (2) \qquad \dfrac{}{x_j \to \bot, x_j}\ \texttt{impn1}}{x_j \to \bot}\ \texttt{res} \qquad \dfrac{}{\neg(x_j \to \bot), \neg x_j, \bot}\ \texttt{impp}$$

$$\dfrac{x_j \to \bot \qquad \neg(x_j \to \bot), \neg x_j, \bot}{\neg x_j, \bot \quad (\mathbf{3})}\ \texttt{res}$$

and the final proof is:

$$\dfrac{\dfrac{}{x_j}\ (1) \qquad \dfrac{}{\neg x_j, \bot}\ (3)}{\bot}\ \texttt{res}$$

RTL Proof:

$$\dfrac{\dfrac{\dfrac{}{\bot}\ \texttt{assume}}{\bot, x_1 \wedge \cdots \wedge x_i \wedge \cdots \wedge x_j \wedge \cdots \wedge x_n}\ \texttt{weaken} \qquad \dfrac{}{\neg\bot}\ \texttt{false}}{x_1 \wedge \cdots \wedge x_i \wedge \cdots \wedge x_j \wedge \cdots \wedge x_n}\ \texttt{res}$$

3. $\top \wedge \cdots \wedge \top = \top$

   LTR Proof:

   $$\dfrac{}{\top}\ \texttt{true}$$

   RTL Proof:

   $$\dfrac{\dfrac{}{\top \wedge \cdots \wedge \top, \neg\top}\ \texttt{andn} \qquad \dfrac{}{\top}\ \texttt{assume}}{\top \wedge \cdots \wedge \top}\ \texttt{res}$$

   Note that `andn` would project all conjuncts from the conjunction $(\top \wedge \cdots \wedge \top, \neg\top, \ldots, \neg\top)$ but since SMTCoq automatically removes any repeated literals from a clause, they don't appear in the proof.

4. $x_1 \wedge \cdots \wedge \top \wedge \cdots \wedge x_n = x_1 \wedge \cdots \wedge x_n$ where the RHS has all $\top$ literals removed.

   LTR Proof:

   (1) derives x_1 from the conjunct:

   $$\dfrac{\dfrac{}{x_1 \wedge \cdots \wedge \top \wedge \cdots \wedge x_n}\ \texttt{assume} \qquad \dfrac{}{x_1 \wedge \cdots \wedge \top \wedge \cdots \wedge x_n, x_1}\ \texttt{andp}}{x_1 \quad (\mathbf{1})}\ \texttt{res}$$

   Similarly, (2) to (n) derive x_2 to x_n respectively. The final proof is:

   $$\dfrac{\dfrac{}{x_1}\ (1) \qquad \cdots \qquad \dfrac{}{x_n}\ (n) \qquad \dfrac{}{x_1 \wedge \cdots \wedge x_n, \neg x_1, \ldots, \neg x_n}\ \texttt{andn}}{x_1 \wedge \cdots \wedge x_n}\ \texttt{res}$$

   RTL Proof:

In the following derivation, $\texttt{to\_unique}(x_1, \ldots x_n) = x_1, \ldots, x_{n'}$.

(1) derives x_1:

$$\dfrac{\dfrac{}{x_1 \wedge \cdots \wedge x_n}\ \texttt{assume} \qquad \dfrac{}{\neg(x_1 \wedge \cdots \wedge x_n), x_1}\ \texttt{andp}}{x_1 \quad (\mathbf{1})}\ \texttt{res}$$

Similarly, (2) to (n') derive x_2 to $x_{n'}$ respectively. The final proof is:

$$\dfrac{\dfrac{}{x_1}\ (1) \quad \ldots \quad \dfrac{}{x_{n'}}\ (n') \quad \dfrac{}{\top}\ \texttt{true} \quad \dfrac{}{x_1 \wedge \cdots \wedge \top \wedge \cdots \wedge x_n, \neg x_1, \ldots, \top, \ldots, x_m}\ \texttt{andn}}{x_1 \wedge \cdots \wedge \top \wedge \cdots \wedge x_n}\ \texttt{res}$$

Notice that the `andn` in the final proof produces $x_1, \ldots, \top, \ldots, x_m$ because SMTCoq's clause representation automatically removes duplicates, which we need to account for using the `to_unique` function.

5. $x_1 \wedge \cdots \wedge x_n = x_1 \wedge \cdots \wedge x_{n'}$ where the RHS has all repeated literals removed.

In the following derivations, $\texttt{to\_unique}(x_1, \ldots x_n) = x_1, \ldots, x_{n'}$.

LTR Proof:

(1) derives x_1:

$$\dfrac{\dfrac{}{x_1 \wedge \cdots \wedge x_n}\ \texttt{assume} \qquad \dfrac{}{\neg(x_1 \wedge \cdots \wedge x_n), x_1}\ \texttt{andp}}{x_1 \quad (\mathbf{1})}\ \texttt{res}$$

Similarly, (2) to (n') derive x_2 to $x_{n'}$, respectively. The final proof is:

$$\dfrac{\dfrac{}{x_1}\ (1) \quad \ldots \quad \dfrac{}{x_{n'}}\ (n') \quad \dfrac{}{x_1 \wedge \cdots \wedge x_{n'}, \neg x_1, \ldots, \neg x_{n'}}\ \texttt{andn}}{x_1 \wedge \cdots \wedge x_{n'}}\ \texttt{res}$$

RTL Proof:

(1) derives x_1:

$$\dfrac{\dfrac{}{x_1 \wedge \cdots \wedge x_{n'}}\ \texttt{assume} \qquad \dfrac{}{\neg(x_1 \wedge \cdots \wedge x_{n'}), x_1}\ \texttt{andp}}{x_1 \quad (\mathbf{1})}\ \texttt{res}$$

Similarly, (2) to (n') derive x_2 to $x_{n'}$, respectively. The final proof is:

$$\dfrac{\dfrac{}{x_1}\ (1) \quad \ldots \quad \dfrac{}{x_{n'}}\ (n') \quad \dfrac{}{x_1 \wedge \cdots \wedge x_n, \neg x_1, \ldots, \neg x_{n'}}\ \texttt{andn}}{x_1 \wedge \cdots \wedge x_n}\ \texttt{res}$$

A.2 Rewrites Over Disjunctions: `orsimp`

$$\frac{}{(\varphi_1 \vee \cdots \vee \varphi_n) = \psi}\ \texttt{orsimp}$$

where the possible transformations are:

- $\bot \vee \cdots \vee \bot = \bot$
- $x_1 \vee \cdots \vee x_n = x_1 \vee \cdots \vee x_{n'}$ where the RHS has all $\bot$ literals removed.
- $x_1 \vee \cdots \vee x_n = x_1 \vee \cdots \vee x_{n'}$ where the RHS has all repeated literals removed.
- $x_1 \vee \cdots \vee \top \vee \cdots \vee x_n = \top$
- $x_1 \vee \cdots \vee x_i \vee \cdots \vee x_j \vee \cdots \vee x_n = \top$ where $x_i = \neg x_j$

and the proofs for each transformation are:

1. $x_1 \vee \cdots \vee \top \vee \cdots \vee x_n = \top$

   LTR Proof:

   $$\frac{}{\top}\ \texttt{true}$$

   RTL Proof:

   $$\frac{\dfrac{}{\top}\ \texttt{assume} \qquad \dfrac{}{x_1 \vee \cdots \vee \top \vee \cdots \vee x_n, \neg\top}\ \texttt{orn}}{x_1 \vee \cdots \vee \top \vee \cdots \vee x_n}\ \texttt{res}$$

2. $x_1 \vee \cdots \vee x_i \vee \cdots \vee x_j \vee \cdots \vee x_n = \top$ where $x_i = \neg x_j$

   LTR Proof:

   $$\frac{}{\top}\ \texttt{true}$$

   RTL Proof:

   $$\frac{\dfrac{}{x_1 \vee \cdots \vee \neg x_j \vee \cdots \vee x_j \vee \cdots \vee x_n, \neg x_j}\ \texttt{orn} \qquad \dfrac{}{x_1 \vee \cdots \vee \neg x_j \vee \cdots \vee x_j \vee \cdots \vee x_n, \neg\neg x_j}\ \texttt{orn}}{x_1 \vee \cdots \vee \neg x_j \vee \cdots \vee x_j \vee \cdots \vee x_n}\ \texttt{res}$$

3. $\bot \vee \cdots \vee \bot = \bot$

   LTR Proof:

   $$\frac{\dfrac{}{\bot \vee \cdots \vee \bot}\ \texttt{assume} \qquad \dfrac{}{\neg(\bot \vee \cdots \vee \bot), \bot}\ \texttt{orn}}{\bot}\ \texttt{res}$$

RTL Proof:

$$\dfrac{\dfrac{}{\bot}\ \texttt{assume} \qquad \dfrac{}{\bot \vee \cdots \vee \bot, \neg\bot}\ \texttt{orn}}{\bot \vee \cdots \vee \bot}\ \texttt{res}$$

4. $x_1 \vee \cdots \vee \bot \vee \cdots \vee x_n = x_1 \vee \cdots \vee x_n$ where the RHS has all $\bot$ literals removed.

   In the following derivations, $\texttt{to\_unique}(x_1, \ldots x_n) = x_1, \ldots, x_{n'}$.

   LTR Proof:

   (1) derives $x_1, \ldots, x_n$:

$$\dfrac{\dfrac{}{x_1 \vee \cdots \vee \bot \vee \cdots \vee x_n}\ \texttt{assume} \qquad \dfrac{}{\neg(x_1 \vee \cdots \vee \bot \vee \cdots \vee x_n), x_1, \ldots, x_n, \bot}\ \texttt{orp} \qquad \dfrac{}{\neg\bot}\ \texttt{false}}{x_1, \ldots, x_n \quad \textbf{(1)}}\ \texttt{res}$$

   The final proof is:

$$\dfrac{\dfrac{}{x_1, \ldots, x_n}\ (1) \qquad \dfrac{}{x_1 \vee \cdots \vee x_n, \neg x_1}\ \texttt{orn} \quad \cdots \quad \dfrac{}{x_1 \vee \cdots \vee x_n, \neg x_{n'}}\ \texttt{orn}}{x_1 \vee \cdots \vee x_n}\ \texttt{res}$$

   RTL Proof:

   (1) derives $x_1, \ldots, x_{n'}$:

$$\dfrac{\dfrac{}{x_1 \vee \cdots \vee x_n}\ \texttt{assume} \qquad \dfrac{}{\neg(x_1 \vee \cdots \vee x_n), x_1, \ldots, x_{n'}}\ \texttt{orp}}{x_1, \ldots, x_{n'} \quad \textbf{(1)}}\ \texttt{res}$$

   The final proof is:

$$\dfrac{\dfrac{}{x_1, \ldots, x_{n'}}\ (1) \qquad \dfrac{}{x_1 \vee \cdots \vee \bot \vee \cdots \vee x_n, \neg x_1}\ \texttt{orn} \quad \cdots \quad \dfrac{}{x_1 \vee \cdots \vee \bot \vee \cdots \vee x_n, \neg x_{n'}}\ \texttt{orn}}{x_1 \vee \cdots \vee \bot \vee \cdots \vee x_n}\ \texttt{res}$$

5. $x_1 \vee \cdots \vee x_n = x_1 \vee \cdots \vee x_{n'}$ where the RHS has all repeated literals removed.

   In the following derivations, $\texttt{to\_unique}(x_1, \ldots x_n) = x_1, \ldots, x_{n'}$.

   LTR Proof:

   (1) derives $x_1, \ldots, x_{n'}$, where the SMTCoq clause notation guarantees no duplicate literals are projected from the disjunction by the `orn` rule.

$$\dfrac{\dfrac{}{x_1 \lor \cdots \lor x_n}\ \texttt{assume} \qquad \dfrac{}{x_1 \lor \cdots \lor x_n, x_1, \ldots, x_{n'}}\ \texttt{orp}}{x_1, \ldots, x_{n'} \quad \textbf{(1)}}\ \texttt{res}$$

The final proof is:

$$\dfrac{\dfrac{}{x_1, \ldots, x_{n'}}\ (1) \qquad \dfrac{}{x_1 \lor \cdots \lor x_{n'}, \neg x_1}\ \texttt{orn} \qquad \cdots \qquad \dfrac{}{x_1 \lor \cdots \lor x_{n'}, \neg x_{n'}}\ \texttt{orn}}{x_1 \lor \cdots \lor x_{n'}}\ \texttt{res}$$

RTL Proof:

(1) derives $x_1, \ldots, x_{n'}$:

$$\dfrac{\dfrac{}{x_1, \ldots, x_{n'}}\ \texttt{assume} \qquad \dfrac{}{\neg(x_1, \ldots, x_{n'}), x_1, \ldots, x_{n'}}\ \texttt{orp}}{x_1, \ldots, x_{n'} \quad \textbf{(1)}}\ \texttt{res}$$

The final proof is:

$$\dfrac{\dfrac{}{x_1, \ldots, x_{n'}}\ (1) \qquad \dfrac{}{x_1 \lor \cdots \lor x_n, \neg x_1}\ \texttt{orn} \qquad \cdots \qquad \dfrac{}{x_1 \lor \cdots \lor x_n, \neg x_{n'}}\ \texttt{orn}}{x_1 \lor \cdots \lor x_n}\ \texttt{res}$$

A.3 Rewrites Over Negations: `notsimp`

$$\dfrac{}{\varphi = \psi}\ \texttt{notsimp}$$

where the possible transformations are:

- $\neg(\neg x) = x$
- $\neg\bot = \top$
- $\neg\top = \bot$

and the proofs for each transformation are:

1. $\neg\bot = \top$

   LTR Proof:

   $$\dfrac{}{\top}\ \texttt{true}$$

   RTL Proof:

   $$\dfrac{}{\neg\bot}\ \texttt{false}$$

2. $\neg\top = \bot$

LTR Proof:

$$\dfrac{\dfrac{\dfrac{}{\neg\top}\ \texttt{assume} \qquad \dfrac{}{\top \to \bot, \top}\ \texttt{impn1}}{\dfrac{\top \to \bot}{\neg\top, \bot}\ \texttt{imp}}\ \texttt{res} \qquad \dfrac{}{\top}\ \texttt{true}}{\bot}\ \texttt{res}$$

RTL Proof:

$$\dfrac{\dfrac{\dfrac{}{\bot}\ \texttt{assume}}{\bot, \neg\top}\ \texttt{weaken} \qquad \dfrac{}{\neg\bot}\ \texttt{false}}{\neg\top}\ \texttt{res}$$

3. $\neg(\neg x) = x$

   This proof is simple enough, that we don't need to separate the *LTR* and *RTL* proofs. The following is the proof of the entire equivalence that we use as our encoding for this variant of the `notsimp` rule.

$$\dfrac{\dfrac{}{\neg\neg x = x, \neg\neg\neg x, \neg x}\ \texttt{eqvn1} \qquad \dfrac{}{\neg\neg x = x, \neg\neg x, x}\ \texttt{eqvn2}}{\neg\neg x = x}\ \texttt{res}$$

   Although negations are mentioned, recall from Section 5.4.2.2 that they will be implicitly simplified by SMTCoq.

A.4 Rewrites Over Implications: `impsimp`

$$\dfrac{}{\varphi_1 \to \varphi_2 = \psi}\ \texttt{impsimp}$$

where the possible transformations are:

- $\neg x_1 \to \neg x_2 = x_2 \to x_1$
- $\bot \to x = \top$
- $x \to \top = \top$
- $\top \to x = x$
- $x \to \bot = \neg x$
- $x \to x = \top$
- $\neg x \to x = x$
- $x \to \neg x = \neg x$

and the proofs for each transformation are:

1. $\neg x_1 \to \neg x_2 = x_2 \to x_1$

   LTR Proof:

   (1) derives $\neg\neg x_1, x_2$:

   $$\dfrac{\dfrac{}{\neg x_1 \to \neg x_2}\,\texttt{assume} \qquad \dfrac{}{\neg(\neg x_1 \to \neg x_2), \neg\neg x_1, \neg x_2}\,\texttt{impp}}{\neg\neg x_1, x_2 \quad (\mathbf{1})}\,\texttt{res}$$

   The final proof is:

   $$\dfrac{\dfrac{}{\neg\neg x_1, x_2}\,(1) \qquad \dfrac{}{x_2 \to x_1, x_2}\,\texttt{impn1} \qquad \dfrac{}{x_2 \to x_1, \neg x_1}\,\texttt{impn2}}{x_2 \to x_1}\,\texttt{res}$$

   RTL Proof:

   (1) derives $\neg x_2, x_1$:

   $$\dfrac{\dfrac{}{x_2 \to x_1}\,\texttt{assume} \qquad \dfrac{}{\neg(x_2 \to x_1), \neg x_2, x_1}\,\texttt{impp}}{\neg x_2, x_1 \quad (\mathbf{1})}\,\texttt{res}$$

   The final proof is:

   $$\dfrac{\dfrac{}{\neg x_2, x_1}\,(1) \qquad \dfrac{}{\neg x_1 \to \neg x_2, \neg x_1}\,\texttt{impn1} \qquad \dfrac{}{\neg x_1 \to \neg x_2, \neg\neg x_2}\,\texttt{impn2}}{\neg x_1 \to \neg x_2}\,\texttt{res}$$

2. $\bot \to x = \top$

   LTR Proof:

   $$\dfrac{}{\top}\,\texttt{true}$$

   RTL Proof:

   $$\dfrac{\dfrac{}{\bot \to x, \bot}\,\texttt{impn1} \qquad \dfrac{}{\neg\bot}\,\texttt{false}}{\bot \to x}\,\texttt{res}$$

3. $x \to \top = \top$

   LTR Proof:

   $$\dfrac{}{\top}\,\texttt{true}$$

RTL Proof:

$$\dfrac{\dfrac{}{\top}\ \texttt{assume} \qquad \dfrac{}{x \to \top, \neg\top}\ \texttt{impn2}}{x \to \top}\ \texttt{res}$$

4. $\top \to x = x$

LTR Proof:

$$\dfrac{\dfrac{\dfrac{}{\top \to x}\ \texttt{assume} \qquad \dfrac{}{\neg(\top \to x), \neg\top, x}\ \texttt{impp}}{\neg\top, x}\ \texttt{res} \qquad \dfrac{}{\top}\ \texttt{true}}{x}\ \texttt{res}$$

RTL Proof:

$$\dfrac{\dfrac{}{x}\ \texttt{assume} \qquad \dfrac{}{\top \to x, \neg x}\ \texttt{impn2}}{\top \to x}\ \texttt{res}$$

5. $x \to \bot = \neg x$

LTR Proof:

$$\dfrac{\dfrac{\dfrac{}{x \to \bot}\ \texttt{assume} \qquad \dfrac{}{\neg(x \to \bot), \neg x, \bot}\ \texttt{impp}}{\neg x, \bot}\ \texttt{res} \qquad \dfrac{}{\neg\bot}\ \texttt{false}}{\neg x}\ \texttt{res}$$

RTL Proof:

$$\dfrac{\dfrac{}{\neg x}\ \texttt{assume} \qquad \dfrac{}{x \to \bot, x}\ \texttt{impn1}}{x \to \bot}\ \texttt{res}$$

6. $x \to x = \top$

LTR Proof:

$$\dfrac{}{\top}\ \texttt{true}$$

RTL Proof:

$$\dfrac{\dfrac{}{x \to x, x}\ \texttt{impn1} \qquad \dfrac{}{x \to x, \neg x}\ \texttt{impn2}}{x \to x}\ \texttt{res}$$

7. $\neg x \to x = x$

LTR Proof:

$$\dfrac{\dfrac{}{\neg x \to x}\ \texttt{assume} \qquad \dfrac{}{\neg(\neg x \to x), \neg\neg x, x}\ \texttt{impp}}{x}\ \texttt{res}$$

Note that SMTCoq implicitly simplifies the double negation and removes the resultant duplicate x in the `impp` derivation.

RTL Proof:

$$\dfrac{\dfrac{}{x}\ \texttt{assume} \qquad \dfrac{}{\neg x \to x, \neg x}\ \texttt{impn1}}{\neg x \to x}\ \texttt{res}$$

8. $x \to \neg x = \neg x$

LTR Proof:

$$\dfrac{\dfrac{}{x \to \neg x}\ \texttt{assume} \qquad \dfrac{}{\neg(x \to \neg x), \neg x, \neg x}\ \texttt{impp}}{\neg x}\ \texttt{res}$$

where the duplicate in the `impp` is removed by SMTCoq.

RTL Proof:

$$\dfrac{\dfrac{}{\neg x}\ \texttt{assume} \qquad \dfrac{}{x \to \neg x, x}\ \texttt{impn1}}{x \to \neg x}\ \texttt{res}$$

A.5 Rewrites Over Equivalences: `eqvsimp`

$$\dfrac{}{(\varphi_1 = \varphi_2) = \psi}\ \texttt{eqvsimp}$$

where the possible transformations are:

- $(\neg x_1 = \neg x_2) = (x_1 = x_2)$
- $(x = x) = \top$
- $(x = \neg x) = \bot$
- $(\neg x = x) = \bot$
- $(\top = x) = x$
- $(x = \top) = x$
- $(\bot = x) = \neg x$

- $(x = \bot) = \neg x$

and the proofs for each transformation are:

1. $(\neg x_1 = \neg x_2) = (x_1 = x_2)$

   LTR Proof:

   Derivation (1):

$$\dfrac{\dfrac{}{x_1 = x_2, \neg x_1, \neg x_2}\ \texttt{eqvn1} \qquad \dfrac{}{\neg(\neg x_1 = \neg x_2), \neg x_1, \neg\neg x_2}\ \texttt{eqvp1}}{x_1 = x_2, \neg x_1, \neg(\neg x_1 = \neg x_2) \quad (\mathbf{1})}\ \texttt{res}$$

   Derivation (2):

$$\dfrac{\dfrac{}{x_1 = x_2, x_1, x_2}\ \texttt{eqvn2} \qquad \dfrac{}{\neg(\neg x_1 = \neg x_2), \neg\neg x_1, \neg x_2}\ \texttt{eqvp2}}{x_1 = x_2, x_1, \neg(\neg x_1 = \neg x_2) \quad (\mathbf{2})}\ \texttt{res}$$

   The final proof:

$$\dfrac{\dfrac{}{x_1 = x_2, \neg x_1, \neg(\neg x_1 = \neg x_2)}\ (1) \qquad \dfrac{}{x_1 = x_2, x_1, \neg(\neg x_1 = \neg x_2)}\ (2) \qquad \dfrac{}{\neg x_1 = \neg x_2}\ \texttt{assume}}{x_1 = x_2}\ \texttt{res}$$

   RTL Proof:

   Derivation (1):

$$\dfrac{\dfrac{}{\neg x_1 = \neg x_2, \neg\neg x_1, \neg\neg x_2}\ \texttt{eqvn1} \qquad \dfrac{}{\neg(x_1 = x_2), x_1, \neg x_2}\ \texttt{eqvp1}}{\neg x_1 = \neg x_2, x_1, \neg(x_1 = x_2) \quad (\mathbf{1})}\ \texttt{res}$$

   Derivation (2):

$$\dfrac{\dfrac{}{\neg x_1 = \neg x_2, \neg x_1, \neg x_2}\ \texttt{eqvn2} \qquad \dfrac{}{\neg(x_1 = x_2), \neg x_1, x_2}\ \texttt{eqvp2}}{\neg x_1 = \neg x_1, \neg x_1, \neg(x_1 = x_2) \quad (\mathbf{2})}\ \texttt{res}$$

   The final proof:

$$\dfrac{\dfrac{}{\neg x_1 = \neg x_2, x_1, \neg(x_1 = x_2)}\ (1) \qquad \dfrac{}{\neg x_1 = \neg x_1, \neg x_1, \neg(x_1 = x_2)}\ (2) \qquad \dfrac{}{x_1 = x_2}\ \texttt{assume}}{\neg x_1 = \neg x_2}\ \texttt{res}$$

2. $(x = x) = \top$

   LTR Proof:

$$\dfrac{}{\top}\ \texttt{true}$$

RTL Proof:

$$\dfrac{\dfrac{}{x = x, \neg x, \neg x}\ \texttt{eqvn1} \qquad \dfrac{}{x = x, x, x}\ \texttt{eqvn2}}{x = x}\ \texttt{res}$$

3. $(x = \neg x) = \bot$

   LTR Proof:

   Derivation (1):

$$\dfrac{\dfrac{}{\neg(x = \neg x), \neg x, \neg x}\ \texttt{eqvp2} \qquad \dfrac{}{x \to \bot, x}\ \texttt{impn1} \qquad \dfrac{}{\neg(x \to \bot), \neg x, \bot}\ \texttt{impp}}{\neg(x = \neg x), \neg x, \bot \quad (\mathbf{1})}\ \texttt{res}$$

   The final proof:

$$\dfrac{\dfrac{}{\neg(x = \neg x), \neg x, \bot}\ (1) \qquad \dfrac{}{\neg(x = \neg x), x, \neg\neg x}\ \texttt{eqvp1} \qquad \dfrac{}{x = \neg x}\ \texttt{assume}}{\bot}\ \texttt{res}$$

   This proof can be simplified by deriving $\bot$ by directly resolving the 5 premises, but is split into 2 resolutions due to space constraints.

   RTL Proof:

$$\dfrac{\dfrac{\dfrac{}{\bot}\ \texttt{assume}}{\bot, x = \neg x}\ \texttt{weaken} \qquad \dfrac{}{\neg\bot}\ \texttt{false}}{x = \neg x}\ \texttt{res}$$

4. $(\neg x = x) = \bot$

   LTR Proof:

   Derivation (1):

$$\dfrac{\dfrac{}{\neg(\neg x = x), \neg x, \neg x}\ \texttt{eqvp1} \qquad \dfrac{}{x \to \bot, x}\ \texttt{impn1} \qquad \dfrac{}{\neg(x \to \bot), \neg x, \bot}\ \texttt{impp}}{\neg(\neg x = x), \neg x, \bot \quad (\mathbf{1})}\ \texttt{res}$$

   The final proof:

$$\dfrac{\dfrac{}{\neg(\neg x = x), \neg x, \bot}\ (1) \qquad \dfrac{}{\neg(\neg x = x), \neg\neg x, x}\ \texttt{eqvp2} \qquad \dfrac{}{x = \neg x}\ \texttt{assume}}{\bot}\ \texttt{res}$$

   RTL Proof:

$$\dfrac{\dfrac{\dfrac{}{\bot}\ \texttt{assume}}{\bot, \neg x = x}\ \texttt{weaken} \qquad \dfrac{}{\neg\bot}\ \texttt{false}}{\neg x = x}\ \texttt{res}$$

5. $(\top = x) = x$

   LTR Proof:

$$\dfrac{\dfrac{}{\top = x}\ \texttt{assume} \qquad \dfrac{}{\neg(\top = x), \neg\top, x}\ \texttt{eqvp2} \qquad \dfrac{}{\top}\ \texttt{true}}{x}\ \texttt{res}$$

   RTL Proof:

$$\dfrac{\dfrac{}{x}\ \texttt{assume} \qquad \dfrac{}{\top = x, \neg\top, \neg x}\ \texttt{eqvn1} \qquad \dfrac{}{\top}\ \texttt{true}}{\top = x}\ \texttt{res}$$

6. $(x = \top) = x$

   LTR Proof:

$$\dfrac{\dfrac{}{x = \top}\ \texttt{assume} \qquad \dfrac{}{\neg(x = \top), x, \neg\top}\ \texttt{eqvp1} \qquad \dfrac{}{\top}\ \texttt{true}}{x}\ \texttt{res}$$

   RTL Proof:

$$\dfrac{\dfrac{}{x}\ \texttt{assume} \qquad \dfrac{}{x = \top, \neg x, \neg\top}\ \texttt{eqvn1} \qquad \dfrac{}{\top}\ \texttt{true}}{x = \top}\ \texttt{res}$$

7. $(\bot = x) = \neg x$

   LTR Proof:

$$\dfrac{\dfrac{}{\bot = x}\ \texttt{assume} \qquad \dfrac{}{\neg(\bot = x), \bot, \neg x}\ \texttt{eqvp1} \qquad \dfrac{}{\neg\bot}\ \texttt{false}}{\neg x}\ \texttt{res}$$

   RTL Proof:

$$\dfrac{\dfrac{}{\neg x}\ \texttt{assume} \qquad \dfrac{}{\bot = x, \bot, x}\ \texttt{eqvn2} \qquad \dfrac{}{\neg\bot}\ \texttt{false}}{\bot = x}\ \texttt{res}$$

8. $(x = \bot) = \neg x$

   LTR Proof:

$$\dfrac{\dfrac{}{\neg x}\ \texttt{assume} \qquad \dfrac{}{x = \bot, x, \bot}\ \texttt{eqvn2} \qquad \dfrac{}{\neg\bot}\ \texttt{false}}{x = \bot}\ \texttt{res}$$

RTL Proof:

$$\dfrac{\dfrac{}{\neg x}\,\texttt{assume} \qquad \dfrac{}{x = \bot, x, \bot}\,\texttt{eqvn2} \qquad \dfrac{}{\neg\bot}\,\texttt{false}}{x = \bot}\,\texttt{res}$$

A.6 Other Boolean Rewrites: `boolsimp`

$$\dfrac{}{\varphi = \psi}\,\texttt{boolsimp}$$

where the possible transformations are:

- $\neg(x_1 \to x_2) = (x_1 \wedge \neg x_2)$
- $\neg(x_1 \vee x_2) = (\neg x_1 \wedge \neg x_2)$
- $\neg(x_1 \wedge x_2) = (\neg x_1 \vee \neg x_2)$
- $(x_1 \to (x_2 \to x_3)) = (x_1 \wedge x_2) \to x_3$
- $((x_1 \to x_2) \to x_2) = (x_1 \vee x_2)$
- $(x_1 \wedge (x_1 \to x_2)) = (x_1 \wedge x_2)$
- $((x_1 \to x_2) \wedge x_1) = (x_1 \wedge x_2)$

and the proofs for each transformation are:

1. $\neg(x_1 \to x_2) = (x_1 \wedge \neg x_2)$

   LTR Proof:

   (1) derives x_1:

   $$\dfrac{\dfrac{}{\neg(x_1 \to x_2)}\,\texttt{assume} \qquad \dfrac{}{x_1 \to x_2, x_1}\,\texttt{impn1}}{x_1 \quad (\mathbf{1})}\,\texttt{res}$$

   (2) derives $\neg x_2$:

   $$\dfrac{\dfrac{}{\neg(x_1 \to x_2)}\,\texttt{assume} \qquad \dfrac{}{x_1 \to x_2, \neg x_2}\,\texttt{impn2}}{\neg x_2 \quad (\mathbf{2})}\,\texttt{res}$$

   The final proof:

   $$\dfrac{\dfrac{}{x_1 \wedge \neg x_2, \neg x_1, \neg\neg x_2}\,\texttt{andn} \qquad \dfrac{}{x_1}\,(1) \qquad \dfrac{}{\neg x_2}\,(2)}{x_1 \wedge \neg x_2}\,\texttt{res}$$

RTL Proof:

(1) derives x_1:

$$\dfrac{\dfrac{}{x_1 \land \neg x_2} \qquad \dfrac{}{\neg(x_1 \land \neg x_2), x_1}\ \texttt{andp}}{x_1 \quad (\mathbf{1})}\ \texttt{res}$$

(2) derives $\neg x_2$:

$$\dfrac{\dfrac{}{x_1 \land \neg x_2} \qquad \dfrac{}{\neg(x_1 \land \neg x_2), \neg x_2}\ \texttt{andp}}{\neg x_2 \quad (\mathbf{2})}\ \texttt{res}$$

The final proof:

$$\dfrac{\dfrac{}{\neg(x_1 \to x_2), \neg x_1, x_2}\ \texttt{impp} \qquad \dfrac{}{x_1}\ (1) \qquad \dfrac{}{\neg x_2}\ (2)}{\neg(x_1 \to x_2)}\ \texttt{res}$$

2. $\neg(x_1 \lor x_2) = (\neg x_1 \land \neg x_2)$

LTR Proof:

(1) derives $\neg x_1$:

$$\dfrac{\dfrac{}{\neg x_1 \land \neg x_2}\ \texttt{assume} \qquad \dfrac{}{x_1 \lor x_2, \neg x_1}\ \texttt{orn}}{\neg x_1 \quad (\mathbf{1})}\ \texttt{res}$$

(2) derives $\neg x_2$:

$$\dfrac{\dfrac{}{\neg x_1 \land \neg x_2}\ \texttt{assume} \qquad \dfrac{}{x_1 \lor x_2, \neg x_2}\ \texttt{orn}}{\neg x_2 \quad (\mathbf{2})}\ \texttt{res}$$

The final proof:

$$\dfrac{\dfrac{}{\neg x_1 \land \neg x_2, \neg\neg x_1, \neg\neg x_2}\ \texttt{andn} \qquad \dfrac{}{\neg x_1}\ (1) \qquad \dfrac{}{\neg x_2}\ (2)}{\neg x_1 \land \neg x_2}\ \texttt{res}$$

RTL Proof:

(1) derives $\neg x_1$:

$$\dfrac{\dfrac{}{\neg x_1 \land \neg x_2}\ \texttt{assume} \qquad \dfrac{}{\neg(\neg x_1 \land \neg x_2), \neg x_1}\ \texttt{andp}}{\neg x_1 \quad (\mathbf{1})}\ \texttt{res}$$

(2) derives $\neg x_2$:

$$\dfrac{\dfrac{}{\neg x_1 \land \neg x_2}\ \texttt{assume} \qquad \dfrac{}{\neg(\neg x_1 \land \neg x_2), \neg x_2}\ \texttt{andp}}{\neg x_2 \quad (\mathbf{2})}\ \texttt{res}$$

The final proof:

$$\dfrac{\dfrac{}{\neg(x_1 \lor x_2), x_1, x_2}\ \texttt{orn} \qquad \dfrac{}{\neg x_1}\ (1) \qquad \dfrac{}{\neg x_2}\ (2)}{\neg(x_1 \lor x_2)}\ \texttt{res}$$

3. $\neg(x_1 \land x_2) = (\neg x_1 \lor \neg x_2)$

   LTR Proof:

   A single resolution chain is split up as follows. Derivation (1):

$$\dfrac{\dfrac{}{x_1 \land x_2, \neg x_1, \neg x_2}\ \texttt{andn} \qquad \dfrac{}{\neg(x_1 \land x_2)}\ \texttt{assume}}{\neg x_1, \neg x_2 \quad (\mathbf{1})}\ \texttt{res}$$

   The final proof:

$$\dfrac{\dfrac{}{\neg x_1, \neg x_2}\ (1) \qquad \dfrac{}{\neg x_1 \lor \neg x_2, \neg\neg x_1}\ \texttt{orn} \qquad \dfrac{}{\neg x_1 \lor \neg x_2, \neg\neg x_2}\ \texttt{orn}}{\neg x_1 \lor \neg x_2}\ \texttt{res}$$

   RTL Proof:

   A single resolution chain is split up as follows. Derivation (1):

$$\dfrac{\dfrac{}{\neg(\neg x_1 \lor \neg x_2), \neg x_1, \neg x_2}\ \texttt{orp} \qquad \dfrac{}{\neg x_1 \lor \neg x_2}\ \texttt{assume}}{\neg x_1, \neg x_2 \quad (\mathbf{1})}\ \texttt{res}$$

   The final proof:

$$\dfrac{\dfrac{}{\neg x_1, \neg x_2}\ (1) \qquad \dfrac{}{\neg(x_1 \land x_2), x_1}\ \texttt{andp} \qquad \dfrac{}{\neg(x_1 \land x_2), x_2}\ \texttt{andp}}{\neg(x_1 \land x_2)}\ \texttt{res}$$

4. $(x_1 \to (x_2 \to x_3)) = (x_1 \land x_2) \to x_3$

   LTR Proof:

   A single resolution chain is split up as follows. Derivation (1):

$$\dfrac{\dfrac{}{\neg(x_1 \to (x_2 \to x_3)), \neg x_1, x_2 \to x_3}\ \texttt{impp} \qquad \dfrac{}{x_1 \to (x_2 \to x_3)}\ \texttt{assume}}{\neg x_1, x_2 \to x_3 \quad (\mathbf{1})}\ \texttt{res}$$

Derivation (2):

$$\dfrac{\dfrac{}{\neg x_1, x_2 \to x_3}\ (1) \qquad \dfrac{}{\neg(x_2 \to x_3), \neg x_2, x_3}\ \texttt{impp} \qquad \dfrac{}{\neg(x_1 \land x_2), x_1}\ \texttt{andp}}{\neg x_2, x_3, \neg(x_1 \land x_2) \quad (\mathbf{2})}\ \texttt{res}$$

Derivation (3):

$$\dfrac{\dfrac{}{\neg x_2, x_3, \neg(x_1 \land x_2)}\ (2) \qquad \dfrac{}{\neg(x_1 \land x_2), x_2}\ \texttt{andp} \qquad \dfrac{}{(x_1 \land x_2) \to x_3, \neg x_3}\ \texttt{impn2}}{\neg(x_1 \land x_2), (x_1 \land x_2) \to x_3 \quad (\mathbf{3})}\ \texttt{res}$$

The final proof:

$$\dfrac{\dfrac{}{\neg(x_1 \land x_2), (x_1 \land x_2) \to x_3}\ (3) \qquad \dfrac{}{(x_1 \land x_2) \to x_3, x_1 \land x_2}\ \texttt{impn1}}{(x_1 \land x_2) \to x_3}\ \texttt{res}$$

RTL Proof:

A single resolution chain is split up as follows. Derivation (1):

$$\dfrac{\dfrac{}{\neg((x_1 \land x_2) \to x_3), \neg(x_1 \land x_2), x_3}\ \texttt{impp} \qquad \dfrac{}{(x_1 \land x_2) \to x_3}\ \texttt{assume}}{\neg(x_1 \land x_2), x_3 \quad (\mathbf{1})}\ \texttt{res}$$

Derivation (2):

$$\dfrac{\dfrac{}{\neg(x_1 \land x_2), x_3}\ (1) \qquad \dfrac{}{x_1 \land x_2, \neg x_1, \neg x_2}\ \texttt{andn} \qquad \dfrac{}{x_2 \to x_3, \neg x_3}\ \texttt{impn2}}{\neg x_1, \neg x_2, x_2 \to x_3 \quad (\mathbf{2})}\ \texttt{res}$$

Derivation (3):

$$\dfrac{\dfrac{}{\neg x_1, \neg x_2, x_2 \to x_3}\ (2) \qquad \dfrac{}{x_1 \to (x_2 \to x_3), x_1}\ \texttt{impn1} \qquad \dfrac{}{x_2 \to x_3, x_2}\ \texttt{impn1}}{x_2 \to x_3, x_1 \to (x_2 \to x_3) \quad (\mathbf{3})}\ \texttt{res}$$

The final proof:

$$\dfrac{\dfrac{}{x_2 \to x_3, x_1 \to (x_2 \to x_3)}\ (3) \qquad \dfrac{}{x_1 \to (x_2 \to x_3), \neg(x_2 \to x_3)}\ \texttt{impn2}}{x_1 \to (x_2 \to x_3)}\ \texttt{res}$$

5. $((x_1 \to x_2) \to x_2) = (x_1 \lor x_2)$

LTR Proof:

A single resolution chain is split up as follows. Derivation (1):

$$\dfrac{\dfrac{}{(x_1 \to x_2) \to x_2}\ \texttt{assume} \qquad \dfrac{}{\neg((x_1 \to x_2) \to x_2), \neg(x_1 \to x_2), x_2}\ \texttt{impp}}{\neg(x_1 \to x_2), x_2 \quad (\mathbf{1})}\ \texttt{res}$$

The final proof:

$$\dfrac{\dfrac{}{\neg(x_1 \to x_2), x_2}\ (1) \qquad \dfrac{}{x_1 \to x_2, x_1}\ \texttt{impn1} \qquad \dfrac{}{x_1 \lor x_2, \neg x_1}\ \texttt{orn} \qquad \dfrac{}{x_1 \lor x_2, \neg x_2}\ \texttt{orn}}{x_1 \lor x_2}\ \texttt{res}$$

RTL Proof:

A single resolution chain is split up as follows. Derivation (1):

$$\dfrac{\dfrac{}{x_1 \lor x_2}\ \texttt{assume} \qquad \dfrac{}{\neg(x_1 \lor x_2), x_1, x_2}\ \texttt{orp} \qquad \dfrac{}{\neg(x_1 \to x_2), \neg x_1, x_2}\ \texttt{impp}}{x_2, \neg(x_1 \to x_2) \quad (\mathbf{1})}\ \texttt{res}$$

The final proof:

$$\dfrac{\dfrac{}{x_2, \neg(x_1 \to x_2)}\ (1) \qquad \dfrac{}{(x_1 \to x_2) \to x_2, \neg x_2}\ \texttt{impn2} \qquad \dfrac{}{(x_1 \to x_2) \to x_2, x_1 \to x_2}\ \texttt{impn1}}{(x_1 \to x_2) \to x_2}\ \texttt{res}$$

6. $(x_1 \land (x_1 \to x_2)) = (x_1 \land x_2)$

LTR Proof:

A single resolution chain is split up as follows. Derivation (1):

$$\dfrac{\dfrac{}{\neg(x_1 \land (x_1 \to x_2)), x_1 \to x_2}\ \texttt{andp} \qquad \dfrac{}{\neg(x_1 \to x_2), \neg x_1, x_2}\ \texttt{impp} \qquad \dfrac{}{x_1 \land x_2, \neg x_1, \neg x_2}\ \texttt{andn}}{\neg(x_1 \land (x_1 \to x_2)), \neg x_1, x_1 \land x_2 \quad (\mathbf{1})}\ \texttt{res}$$

Derivation (2):

$$\dfrac{\dfrac{}{\neg(x_1 \land (x_1 \to x_2)), \neg x_1, x_1 \land x_2}\ (1) \qquad \dfrac{}{\neg(x_1 \land (x_1 \to x_2)), x_1}\ \texttt{andp}}{\neg(x_1 \land (x_1 \to x_2)), x_1 \land x_2 \quad (\mathbf{2})}\ \texttt{res}$$

The final proof:

$$\dfrac{\dfrac{}{\neg(x_1 \land (x_1 \to x_2)), x_1 \land x_2}\ (2) \qquad \dfrac{}{x_1 \land (x_1 \to x_2)}\ \texttt{assume}}{x_1 \land x_2}\ \texttt{res}$$

RTL Proof:

A single resolution chain is split up as follows. Derivation (1):

$$\dfrac{\dfrac{}{x_1 \land (x_1 \to x_2), \neg x_1, \neg(x_1 \to x_2)}\ \texttt{andn} \qquad \dfrac{}{x_1 \to x_2, \neg x_2}\ \texttt{impn2}}{x_1 \land (x_1 \to x_2), \neg x_1, \neg x_2 \quad (\mathbf{1})}\ \texttt{res}$$

Derivation (2):

$$\dfrac{\dfrac{}{x_1 \land (x_1 \to x_2), \neg x_1, \neg x_2}\ (1) \qquad \dfrac{}{\neg(x_1 \land x_2), x_1}\ \texttt{andp}}{x_1 \land (x_1 \to x_2), \neg x_2, \neg(x_1 \land x_2) \quad (\mathbf{2})}\ \texttt{res}$$

The final proof:

$$\dfrac{\dfrac{}{x_1 \land (x_1 \to x_2), \neg x_2, \neg(x_1 \land x_2)}\ (2) \qquad \dfrac{}{\neg(x_1 \land x_2), x_2}\ \texttt{andp} \qquad \dfrac{}{x_1 \land x_2}\ \texttt{assume}}{x_1 \land (x_1 \to x_2)}\ \texttt{res}$$

7. $((x_1 \to x_2) \land x_1) = (x_1 \land x_2)$

LTR Proof:

A single resolution chain is split up as follows. Derivation (1):

$$\dfrac{\dfrac{}{\neg((x_1 \to x_2) \land x_1), x_1 \to x_2}\ \texttt{andp} \qquad \dfrac{}{\neg(x_1 \to x_2), \neg x_1, x_2}\ \texttt{impp}}{\neg((x_1 \to x_2) \land x_1), \neg x_1, x_2 \quad (\mathbf{1})}\ \texttt{res}$$

Derivation (2):

$$\dfrac{\dfrac{}{\neg((x_1 \to x_2) \land x_1), \neg x_1, x_2}\ (1) \qquad \dfrac{}{x_1 \land x_2, \neg x_1, \neg x_2}\ \texttt{andn} \qquad \dfrac{}{\neg((x_1 \to x_2) \land x_1), x_1}\ \texttt{andp}}{\neg((x_1 \to x_2) \land x_1), x_1 \land x_2 \quad (\mathbf{2})}\ \texttt{res}$$

The final proof:

$$\dfrac{\dfrac{}{\neg((x_1 \to x_2) \land x_1), x_1 \land x_2}\ (2) \qquad \dfrac{}{(x_1 \to x_2) \land x_1}\ \texttt{assume}}{x_1 \land x_2}\ \texttt{res}$$

RTL Proof:

A single resolution chain is split up as follows. Derivation (1):

$$\dfrac{\dfrac{}{(x_1 \to x_2) \land x_1, \neg(x_1 \to x_2), \neg x_1}\ \texttt{andn} \qquad \dfrac{}{x_1 \to x_2, \neg x_2}\ \texttt{impn2}}{(x_1 \to x_2) \land x_1, \neg x_1, \neg x_2 \quad (\mathbf{1})}\ \texttt{res}$$

Derivation (2):

$$\dfrac{\dfrac{}{(x_1 \to x_2) \land x_1, \neg x_1, \neg x_2}\,(1) \qquad \dfrac{}{\neg(x_1 \land x_2), x_1}\,\texttt{andp}}{(x_1 \to x_2) \land x_1, \neg x_2, \neg(x_1 \land x_2) \quad \mathbf{(2)}}\,\texttt{res}$$

The final proof:

$$\dfrac{\dfrac{}{(x_1 \to x_2) \land x_1, \neg x_2, \neg(x_1 \land x_2)}\,(2) \qquad \dfrac{}{\neg(x_1 \land x_2), x_2}\,\texttt{andp} \qquad \dfrac{}{x_1 \land x_2}\,\texttt{assume}}{(x_1 \to x_2) \land x_1}\,\texttt{res}$$

A.7 Rewrites Over Equality: `eqsimp`

$$\dfrac{}{(\varphi_1 = \varphi_2) = \psi}\,\texttt{eqsimp}$$

where the possible transformations are:

- $x = x = \top$
- $(x_1 = x_2) = \bot$ if x_1 and x_2 are different numeric constants.
- $\neg(x = x) = \bot$ if x is a numeric constant.

and the proofs for each transformation are:

1. $x = x = \top$

   LTR Proof:

   $$\dfrac{}{\top}\,\texttt{true}$$

   RTL Proof:

   $$\dfrac{}{x = x}\,\texttt{eqrefl}$$

2. $(x_1 = x_2) = \bot$ if x_1 and x_2 are different numeric constants.

   LTR Proof:

   $$\dfrac{\dfrac{\dfrac{}{x_1 = x_2}\,\texttt{assume} \qquad \dfrac{}{\neg(x_1 = x_2)}\,\texttt{lia\_micromega}}{\langle\,\rangle}\,\texttt{res}}{\bot}\,\texttt{weaken}$$

   RTL Proof:

   $$\dfrac{\dfrac{\dfrac{}{\bot}\,\texttt{assume} \qquad \dfrac{}{\neg\bot}\,\texttt{false}}{\langle\,\rangle}\,\texttt{res}}{x_1 = x_2}\,\texttt{weaken}$$

3. $\neg(x = x) = \bot$ if x is a numeric constant.

LTR Proof:

$$\dfrac{\dfrac{\dfrac{}{\neg(x = x)}\ \texttt{assume} \qquad \dfrac{}{x = x}\ \texttt{lia\_micromega}}{\langle\ \rangle}\ \texttt{res}}{\bot}\ \texttt{weaken}$$

RTL Proof:

$$\dfrac{\dfrac{\dfrac{}{\bot}\ \texttt{assume} \qquad \dfrac{}{\neg\bot}\ \texttt{false}}{\langle\ \rangle}\ \texttt{res}}{x = x}\ \texttt{weaken}$$

Bibliography

[1] The Coq List Library. `https://github.com/coq/coq/blob/master/theories/Lists/List.v`.

[2] A HOL theory of Euclidean space. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*, Oxford, UK, 2005. Springer-Verlag.

[3] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-Guided Synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8, 2013.

[4] B. Andreotti, H. Lachnitt, and H. Barbosa. Carcara: An Efficient Proof Checker and Elaborator for SMT Proofs in the Alethe Format. In S. Sankaranarayanan and N. Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 367–386, Cham, 2023. Springer Nature Switzerland.

[5] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In J. Jouannaud and Z. Shao, editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2011.

[6] L. Bachmair and H. Ganzinger. Resolution Theorem Proving. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 19–99. Elsevier and MIT Press, 2001.

[7] C. Baier and J. Katoen. *Principles of model checking*. MIT Press, 2008.

[8] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, 2018.

[9] H. Barbosa, C. Keller, A. Reynolds, A. Viswanathan, C. Tinelli, and C. W. Barrett. An Interactive SMT Tactic in Coq using Abductive Reasoning. In R. Piskac and A. Voronkov, editors, *LPAR 2023: Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Manizales, Colombia, 4-9th June 2023*, volume 94 of *EPiC Series in Computing*, pages 11–22. EasyChair, 2023.

[10] H. Barbosa, A. Reynolds, G. Kremer, H. Lachnitt, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Viswanathan, S. Viteri, Y. Zohar, C. Tinelli, and C. Barrett. Flexible Proof Production in an Industrial-Strength SMT Solver. In J. Blanchette, L. Kovács, and D. Pattinson, editors, *Automated Reasoning*, pages 15–35, Cham, 2022. Springer International Publishing.

[11] C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[12] C. Barrett and C. Tinelli. Satisfiability Modulo Theories. In *Handbook of Model Checking.*, pages 305–343. 2018.

[13] C. Barrett and C. Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018.

[14] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[15] C. W. Barrett, I. Shikanian, and C. Tinelli. An Abstract Decision Procedure for a Theory of Inductive Data Types. *J. Satisf. Boolean Model. Comput.*, 3(1-2):21–46, 2007.

[16] J. Beeren, M. Fernandez, X. Gao, G. Klein, R. Kolanski, J. Lim, C. Lewis, D. Matichuk, and T. Sewell. Finite Machine Word Library. *Archive of Formal Proofs*, June 2016. https://isa-afp.org/entries/Word_Lib.html, Formal proof development.

[17] F. Besson. Fast Reflexive Arithmetic Tactics the Linear Case and Beyond. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs*, pages 48–62, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[18] F. Besson, P. Fontaine, and L. Théry. A Flexible Proof Format for SMT: a Proposal. In P. Fontaine and A. Stump, editors, *PxTP 2011: First International Workshop on Proof eXchange for Theorem Proving, Wrocław, Poland, August 1, 2011*, pages 15–26, 2011.

[19] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT Solvers. *J. Autom. Reason.*, 51(1):109–128, 2013.

[20] J. C. Blanchette, C. Kaliszyk, L. C. Paulson, and J. Urban. Hammering towards QED. *J. Formaliz. Reason.*, 9(1):101–148, 2016.

[21] A. Blot, P.-E. Dagand, , and J. Lawall. Bit Sequences and Bit Sets Library.

[22] V. Blot, A. Bousalem, Q. Garchery, and C. Keller. SMTCoq: automatisation expressive et extensible dans Coq. In *JFLA 2019 - Journées Francophones des Langages Applicatifs*, Les Rousses, France, Jan. 2019.

[23] V. Blot, D. Cousineau, E. Crance, L. D. de Prisque, C. Keller, A. Mahboubi, and P. Vial. Compositional Pre-processing for Automated Reasoning in Dependent Type Theory. In R. Krebbers, D. Traytel, B. Pientka, and S. Zdancewic, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, pages 63–77. ACM, 2023.

[24] S. Böhme, A. C. J. Fox, T. Sewell, and T. Weber. Reconstruction of Z3's Bit-Vector Proofs in HOL4 and Isabelle/HOL. In J.-P. Jouannaud and Z. Shao, editors, *Certified Programs and Proofs*, pages 183–198, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[25] S. Böhme and T. Weber. Fast LCF-Style Proof Reconstruction for Z3. In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving*, pages 179–194, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[26] J. Bornholt. Program Synthesis Explained. https://homes.cs.washington.edu/~bornholt/post/synthesis-explained.html, 2015. [Online; accessed 28-August-2018].

[27] T. Bouton, D. Caminha B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: An Open, Trustable and Efficient SMT-Solver. In R. A. Schmidt, editor, *Automated Deduction – CADE-22*, pages 151–156, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[28] T. Chajed, H. Chen, A. Chlipala, J. Choi, A. Erbsen, J. Gross, S. Gruetter, F. Kaashoek, A. Konradi, G. Malecha, D. Oe, M. Vijayaraghavan, N. Zeldovich, and D. Ziegler. Bedrock Bit Vectors Library.

[29] A. Chlipala and G. C. Necula. Cooperative Integration of an Interactive Proof Assistant and an Automated Prover. In *Proceedings of the 6th International Workshop on Strategies in Automated Deduction*, 2006.

[30] S. Conchon, A. Coquereau, M. Iguernlala, and A. Mebsout. Alt-Ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, Oxford, United Kingdom, July 2018.

[31] M. contributors. Programmable proof search - The auto Tactic. https://coq.inria.fr/doc/V8.18.0/refman/proofs/automatic-tactics/auto.html.

[32] M. contributors. The Coq Standard Library. https://coq.inria.fr/doc/V8.13.0/stdlib/.

[33] T. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[34] S. Cruanes and J. C. Blanchette. Extending Nunchaku to Dependent Type Theory. *Electronic Proceedings in Theoretical Computer Science*, 210:3–12, jun 2016.

[35] L. Czajka and C. Kaliszyk. Hammer for Coq: Automation for Dependent Type Theory. *Journal of Automated Reasoning*, 61, 06 2018.

[36] N. G. de Bruijn. *AUTOMATH, a Language for Mathematics*, pages 159–200. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.

[37] L. de Moura and N. Bjørner. Efficient E-Matching for SMT Solvers. In F. Pfenning, editor, *Automated Deduction – CADE-21*, pages 183–198, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[38] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[39] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean Theorem Prover (System Description). In A. P. Felty and A. Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing.

[40] D. Deharbe, P. Fontaine, and B. W. Paleo. Quantifier Inference Rules for SMT proofs. 2011.

[41] D. Delahaye. A Tactic Language for the System Coq. In M. Parigot and A. Voronkov, editors, *Logic for Programming and Automated Reasoning*, pages 85–95, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[42] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A Theorem Prover for Program Checking. *J. ACM*, 52(3):365–473, may 2005.

[43] I. Dillig and T. Dillig. Explain: A Tool for Performing Abductive Inference. In N. Sharygina and H. Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 684–689. Springer, 2013.

[44] I. Dillig, T. Dillig, K. L. McMillan, and A. Aiken. Minimum Satisfying Assignments for SMT. In P. Madhusudan and S. A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 394–409. Springer, 2012.

[45] C. Documentation. Conversion Rules in Coq - Coq Documentation. https://coq.github.io/doc/master/refman/language/core/conversion.html.

[46] J. Duprat. Library coq.bool.bvector.

[47] M. Echenim and N. Peltier. A Superposition Calculus for Abductive Reasoning. *J. Autom. Reason.*, 57(2):97–134, 2016.

[48] M. Echenim, N. Peltier, and Y. Sellami. A Generic Framework for Implicate Generation Modulo Theories. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, volume 10900 of *Lecture Notes in Computer Science*, pages 279–294. Springer, 2018.

[49] M. Echenim, N. Peltier, and S. Tourret. Prime Implicate Generation in Equational Logic. *J. Artif. Intell. Res.*, 60:827–880, 2017.

[50] B. Ekici, A. Mebsout, C. Tinelli, C. Keller, G. Katz, A. Reynolds, and C. Barrett. SMTCoq: A Plug-In for Integrating SMT Solvers into Coq. In *Proceedings of 29th International Conference on Computer Aided Verification (CAV 2017)*, volume 10427 of *Lecture Notes in Computer Science*, pages 126–133. Springer, 2017.

[51] B. Ekici, A. Viswanathan, Y. Zohar, C. Barrett, and C. Tinelli. Verifying Bit-vector Invertibility Conditions in Coq (Extended Abstract). *Electronic Proceedings in Theoretical Computer Science*, 301:18–26, aug 2019.

[52] B. Ekici, A. Viswanathan, Y. Zohar, C. Tinelli, and C. Barrett. Formal Verification of Bit-Vector Invertibility Conditions in Coq. In U. Sattler and M. Suda, editors, *Frontiers of Combining Systems*, pages 41–59, Cham, 2023. Springer Nature Switzerland.

[53] M. Fleury and C. Weidenbach. A Verified SAT Solver Framework including Optimization and Partial Valuations. In E. Albert and L. Kovacs, editors, *LPAR23. LPAR-23: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 73 of *EPiC Series in Computing*, pages 212–229. EasyChair, 2020.

[54] A. Fuchs, A. Goel, J. Grundy, S. Krstic, and C. Tinelli. Ground interpolation for the theory of equality. *Logical Methods in Computer Science*, 8(1), 2012.

[55] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In R. Alur and D. A. Peled, editors, *Computer Aided Verification*, pages 175–188, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[56] Y. Ge, C. Barrett, and C. Tinelli. Solving Quantified Verification Conditions Using Satisfiability Modulo Theories. In F. Pfenning, editor, *Automated Deduction – CADE-21*, pages 167–182, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[57] Y. Ge and L. M. de Moura. Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 -*

July 2, 2009. Proceedings, volume 5643 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2009.

[58] G. Gonthier and A. Mahboubi. An introduction to small scale reflection in Coq. *J. Formaliz. Reason.*, 3(2):95–152, 2010.

[59] M. Gordon. *From LCF to HOL: A Short History*, page 169–185. MIT Press, Cambridge, MA, USA, 2000.

[60] M. Gordon, R. Milner, C. P. Wadsworth, and P. T. Christopher. Edinburgh LCF: a mechanized logic of computation. 1978.

[61] A. Gupta and A. L. Fisher. Representation and Symbolic Manipulation of Linearly Inductive Boolean Functions. In *Proceedings of the 1993 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '93, pages 192–199, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[62] F. Haifani, P. Koopmann, S. Tourret, and C. Weidenbach. Connection-minimal abduction in *EL* via translation to FOL. In J. Blanchette, L. Kovács, and D. Pattinson, editors, *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, volume 13385 of *Lecture Notes in Computer Science*, pages 188–207. Springer, 2022.

[63] J. Harrison. HOL Light: An Overview. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 60–66, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[64] G. Huet. The Gallina specification language: A case study. In R. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, pages 229–240, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[65] J. Hurd. First-Order Proof Tactics in Higher-Order Logic Theorem Provers. pages 56–68.

[66] C. Inria and contributors. Library Coq.Zarith.Zorder. https://coq.github.io/doc/v8.13/stdlib/Coq.ZArith.Zorder.html.

[67] M. Jonáš. *Satisfiability of Quantified Bit-Vector Formulas Theory and Practice*. PhD thesis, 2019.

[68] S. Lescuyer. *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq. (Formalisation et developpement d'une tactique reflexive pour la demonstration automatique en coq)*. PhD thesis, University of Paris-Sud, Orsay, France, 2011.

[69] D. Loveland, A. Sabharwal, and B. Selman. *DPLL: The Core of Modern Satisfiability Solvers*, pages 315–335. Springer International Publishing, Cham, 2016.

[70] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. 1990.

[71] G. Nelson and D. C. Oppen. Fast Decision Procedures Based on Congruence Closure. *J. ACM*, 27(2):356–364, apr 1980.

[72] A. Niemetz, M. Preiner, A. Reynolds, C. Barrett, and C. Tinelli. On Solving Quantified Bit-Vectors using Invertibility Conditions, 2018.

[73] A. Niemetz, M. Preiner, A. Reynolds, C. Barrett, and C. Tinelli. Solving Quantified Bit-Vectors Using Invertibility Conditions. In *Proceedings of 30th International Conference on Computer Aided Verification (CAV 2018)*, pages 236–255, 2018.

[74] A. Niemetz, M. Preiner, A. Reynolds, Y. Zohar, C. Barrett, and C. Tinelli. Towards Bit-Width-Independent Proofs in SMT Solvers. In P. Fontaine, editor, *Automated Deduction – CADE 27*, pages 366–384, Cham, 2019. Springer International Publishing.

[75] A. Niemetz, M. Preiner, A. Reynolds, Y. Zohar, C. W. Barrett, and C. Tinelli. Towards Satisfiability Modulo Parametric Bit-vectors. *J. Autom. Reason.*, 65(7):1001–1025, 2021.

[76] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[77] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[78] A. Nötzli, H. Barbosa, A. Niemetz, M. Preiner, A. Reynolds, C. W. Barrett, and C. Tinelli. Reconstructing Fine-Grained Proofs of Rewrites Using a Domain-Specific Language. In A. Griggio and N. Rungta, editors, *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, pages 65–74. IEEE, 2022.

[79] C. Paulin-Mohring. Inductive definitions in the system Coq rules and properties. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, pages 328–345, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[80] C. Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In B. W. Paleo and D. Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, Jan. 2015.

[81] L. C. Paulson and J. C. Blanchette. Three years of experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers. In G. Sutcliffe, S. Schulz, and E. Ternovska, editors, *The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011*, volume 2 of *EPiC Series in Computing*, pages 1–11. EasyChair, 2010.

[82] M. Preiner, A. Niemetz, and A. Biere. Counterexample-Guided Model Synthesis. In A. Legay and T. Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, volume 10205 of *Lecture Notes in Computer Science*, pages 264–280, 2017.

[83] A. Reynolds, H. Barbosa, and P. Fontaine. Revisiting Enumerative Instantiation. In D. Beyer and M. Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, volume 10806 of *Lecture Notes in Computer Science*, pages 112–131. Springer, 2018.

[84] A. Reynolds, H. Barbosa, D. Larraz, and C. Tinelli. Scalable Algorithms for Abduction via Enumerative Syntax-Guided Synthesis. In N. Peltier and V. Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, volume 12166 of *Lecture Notes in Computer Science*, pages 141–160. Springer, 2020.

[85] A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. W. Barrett. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In D. Kroening and C. S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 198–216. Springer, 2015.

[86] A. Reynolds, C. Tinelli, and L. M. de Moura. Finding conflicting instances of quantified formulas in SMT. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 195–202. IEEE, 2014.

[87] A. Reynolds, C. Tinelli, A. Goel, S. Krstic, M. Deters, and C. W. Barrett. Quantifier Instantiation Techniques for Finite Model Finding in SMT. In M. P. Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2013.

[88] H. Schurr, M. Fleury, H. Barbosa, and P. Fontaine. Alethe: Towards a Generic SMT Proof Format (extended abstract). In C. Keller and M. Fleury, editors, *Proceedings Seventh Workshop on Proof eXchange for Theorem Proving, PxTP 2021, Pittsburg, PA, USA, July 11, 2021*, volume 336 of *EPTCS*, pages 49–54, 2021.

[89] H. Schurr, M. Fleury, and M. Desharnais. Reliable Reconstruction of Fine-grained Proofs in a Proof Assistant. In A. Platzer and G. Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event,*

July 12-15, 2021, Proceedings, volume 12699 of *Lecture Notes in Computer Science*, pages 450–467. Springer, 2021.

[90] X. Shi, Y. Fu, J. Liu, M. Tsai, B. Wang, and B. Yang. CoqQFBV: A Scalable Certified SMT Quantifier-Free Bit-Vector Solver. In A. Silva and K. R. M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, volume 12760 of *Lecture Notes in Computer Science*, pages 149–171. Springer, 2021.

[91] A. Solar-Lezama. The Sketching Approach to Program Synthesis. In Z. Hu, editor, *Programming Languages and Systems*, pages 4–13, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[92] M. Sozeau. Equations: A Dependent Pattern-Matching Compiler. In *Proceedings of the 1st International Conference on Interactive Theorem Proving (ITP 2010)*, pages 419–434, 2010.

[93] S. Spies and Y. Forster. Undecidability of higher-order unification formalised in Coq. In J. Blanchette and C. Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 143–157. ACM, 2020.

[94] A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli. SMT proof checking using a logical framework. *Formal Methods in System Design*, 42(1):91–118, 2013.

[95] T. Weber. *SAT-based finite model generation for higher-order logic.* PhD thesis, Technical University Munich, Germany, 2008.

[96] K. Yang and J. Deng. Learning to Prove Theorems via Interacting with Proof Assistants. *CoRR*, abs/1905.09381, 2019.