# Automating Interactive Theorem Provers and Certifying Automatic Theorem Provers

Arjun Viswanathan

**Abstract**

Interactive theorem provers (ITPs) and automatic theorem provers (ATPs) are two distinct categories of theorem provers on different ends of the spectrum of theorem proving. On one hand, ITPs are typically robust tools with a small, verified kernel, making them highly reliable. However, they require user intervention in the proving process, only offering a minimal amount of automation. ATPs, on the other hand, are push-button theorem provers that use complex heuristics to prove theorems; as a consequence, they have a large code-base that is hard to maintain and susceptible to bugs. A lot of recent research has focused on bridging the gap between these two poles of theorem proving. Hammers and certified checkers are tools that were born from this research, that have different approaches to solving this problem. This work aims to comprehensively study these different tools with a focus on using SMT solvers as the ATPs enhancing automation in ITPs.

## 1 Introduction

Interactive theorem provers (ITPs) or proof assistants are software tools that allow formalizing of mathematical proofs. They provide an expressive logic to state theorems in, and an interactive interface through which the user can attempt to prove these theorems using methods called tactics. Generally, this interface mimics a written mathematical proof with a context and a goal that changes on the fly as one steps through the parts of the proof. The data structures provided by the ITP are minimal and the user's mathematical structures are defined on top of these, keeping the verified kernel of the ITP small. These proofs provide a high level of reliability but are hard to come up with from the user's point of view.

Automatic theorem provers (ATPs) have grown rapidly over the past decades and refer to tools that allow automatic proving of logical formulas. Interaction between the user and the ATP is kept to a minimum; ideally, the user would provide a theorem to the ATP and the ATP either proves it or comes up with a counter-example that disproves it. Satisfiability modulo theories (SMT) solvers and superposition provers are two popular categories of ATPs. Although both of these have different approaches to proving, they both look at a formula in

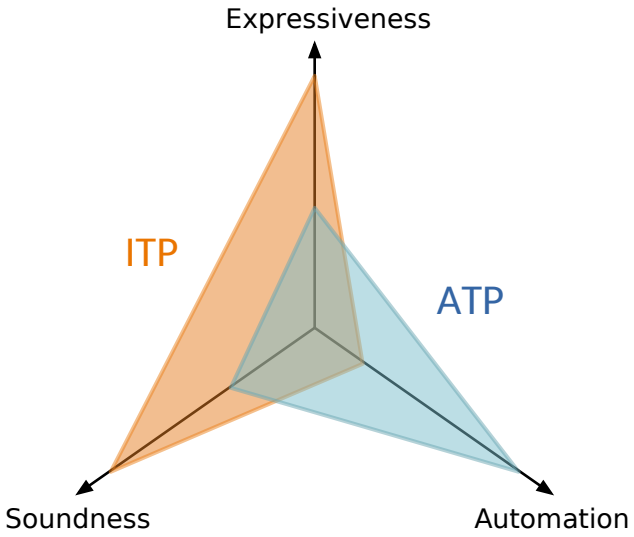


Figure 1: Comparing Interactive Theorem Provers (ITPs) and Automatic Theorem Provers. Credits: Chantal Keller (ATPs)

terms of its satisfiability. The satisfiability problem is the dual of the validity problem (proving a formula to be valid) — a formula can be proved to be valid by establishing that its negation is unsatisfiable.

ATPs and ITPs clearly have different modes of operations and offer distinct advantages: ITPs certify their results with a high degree of trustworthiness although reaching the proofs might take some time and effort from the user; in contrast, ATPs offer relatively quick and automated results, while not providing the same kind of guarantees that ITPs do. A natural research question is whether we can get the best of both worlds — reliable proofs with maximum automation. In this work, we study various tools that attempt to combine ATPs and ITPs.

After specifying the notation used in the rest of the paper, we describe SMT solvers and superposition provers in Section 3. Section 4 explores the state-of-the-art of ITPs. Section 5 and Section 6 explore hammers in general, and Sledgehammer's integration with SMT solvers in particular. In Section 7, we look at SMTCoq. Finally, we compare the 2 approaches in Section 10, then present some related work, and discuss our plans to take this research forward.

## 2 Technical Preliminaries

Most of the notation and technicalities are introduced where needed in this document. For a thorough introduction to first order logic and satisfiability modulo theories, see [10]. Briefly, literals are variables or negations of variables,

terms are built from literals, and formulas are built from terms and quantifiers. Most provers reason over arbitrary logical formulas which they convert into a normal form for easier processing. One such normal form is the conjunction nomal form (CNF). A formula in CNF is a conjunction of clauses (disjunctions of literals). Additionally, SMT deals with a sorted FOL - sort is a formal synonym for type.

We also look at various rule systems. In general a rule looks like this:

$$\frac{premise_1 \quad premise_2 \quad ... \quad premise_n}{conclusion}\ rule-name$$

The rule called *rule-name* has $n$ premises $premise_1, ..., premise_n$ and a conclusion *conclusion*. The rule can be understood as follows — if all $n$ premises are true, then it follows that the *conclusion* is true. Usually, the premises and the conclusion are formulas but the Z3 SMT solver uses a sequent calculus where these are sequents. These are introduced later before their usage.

We discuss various proof systems that give refutation proofs of validity. A refutation proof employs the duality between validity and satisfiability. To prove a formula $F$, it suffices to prove that $\neg F$ is unsatisfiable. If there is no way to satisfy $\neg F$, then by refutation, $F$ must be true. This form of proofs is a recurring theme in this work.

The following abbreviations are repeatedly used: SAT (SAT), satisfiability modulo theories (SMT), unsat (unsatisfiable), first-order logic (FOL), higher-order logic (HOL).

# 3 Automatic Theorem Provers

## 3.1 SMT Solvers

Boolean satisfiability, often called the SAT problem, is the problem of determining if a Boolean formula is satisfiable, that is, whether there is an assigment of the values of $True$ or $False$ to the variables of the formula so that the entire formula evaluates to $True$. For example,

$$(x \vee y) \wedge z$$

can be satisfied by the assignment $\{x = True,\ y = False,\ z = True\}$. On the other hand,

$$x \wedge \neg x$$

is unsatisfiable, no matter what value is assigned to $x$.

Satisfiability Modulo Theories [10] or SMT lifts SAT to a level that includes theories, by considering formulas whose atoms can be not just propositional variables but also atomic formulas in some logical theory of interest. For example,

$$(a = b) \wedge (b = c) \wedge \neg(a = c)$$

is a formula that is unsatisfiable in the theory of equality over uninterpreted functions, in which we have the axioms of equality and the only information about functions/constants we have is syntactical. This formula is "unsat" because, by transitivity of $a = b$ and $b = c$, we have $a = c$. SMT allows us to be more expressive with our formulas, but this comes at the cost of more complicated procedures.

SMT solvers have plenty of applications in formal methods and software verification. For instance, SMT solvers are used in the back-end of model checkers [7], which take as input mathematical models of a software system, and verify whether they satisfy a particular temporal property or not. Another area of application is symbolic execution [8], which is to analyze multiple execution paths of a program based on abstractions of the inputs. Other uses of SMT solvers include program synthesis [16], static analysis, and generation of logical interpolants [28].

Due to the recent emphasis on verifying results from SMT solvers [19, 41, 4], many SMT solvers are proof producing. When an SMT solver finds that a (quantifier-free) formula is satisfiable, an easily checkable *certificate* of this is a satisfying model of the formula. However, when a solver concludes that a formula is unsatisfiable, it is more difficult to produce an acceptable certificate since this certificate must contain, in effect, a proof of unsatisfiability. Most SMT solvers follow some version of the DPLL(T) algorithm, which tries to satisfy the formula by propagating assignments, making choices on assignments when necessary, and concluding "unsat" when all choices have been tried unsuccessfully. Since the resolution rule is central to this algorithm, a universally accepted proof calculus is one based on resolution. Specifically, a proof of unsatisfiability of a formula in conjunction normal form (CNF), is a tree with the input clauses and theory lemmas at the leaves, and the empty clause at the root. Each node of the tree is an application of rules that simplifies previous nodes. The primary rule used in these trees is resolution.

The resolution rule takes two clauses, a pivot element that occurs with opposite polarities in each of the clauses, and gives one clause that is a consequent of the premises. The idea is that, beginning with the input clauses and the theory lemmas (which might have their own sub-proof trees coming from separate internal solvers called theory solvers), the tree derives that the empty clause holds, which is the most basic notion of unsatisfiability since the there is no way to satisfy the empty clause.

$$\frac{\varphi_1 \vee ... \vee \varphi_n \vee \chi \quad \neg\chi \vee \psi_1 \vee ... \vee \psi_m}{\varphi_1 \vee ... \vee \varphi_n \vee \psi_1 \vee ... \vee \psi_m} \; \textit{resolution}$$

For instance, the following is a valid application of the resolution rule.

$$\frac{a \vee \neg b \quad b \vee c}{a \vee c}$$

The proof calculus for an SMT solver consists of proof rules such as resolution and a proof of unsatisfiability is a proof tree with the assertions asserted in the problem as leaves, as illustrated in Figure 2. Additionally, clauses asserted to be

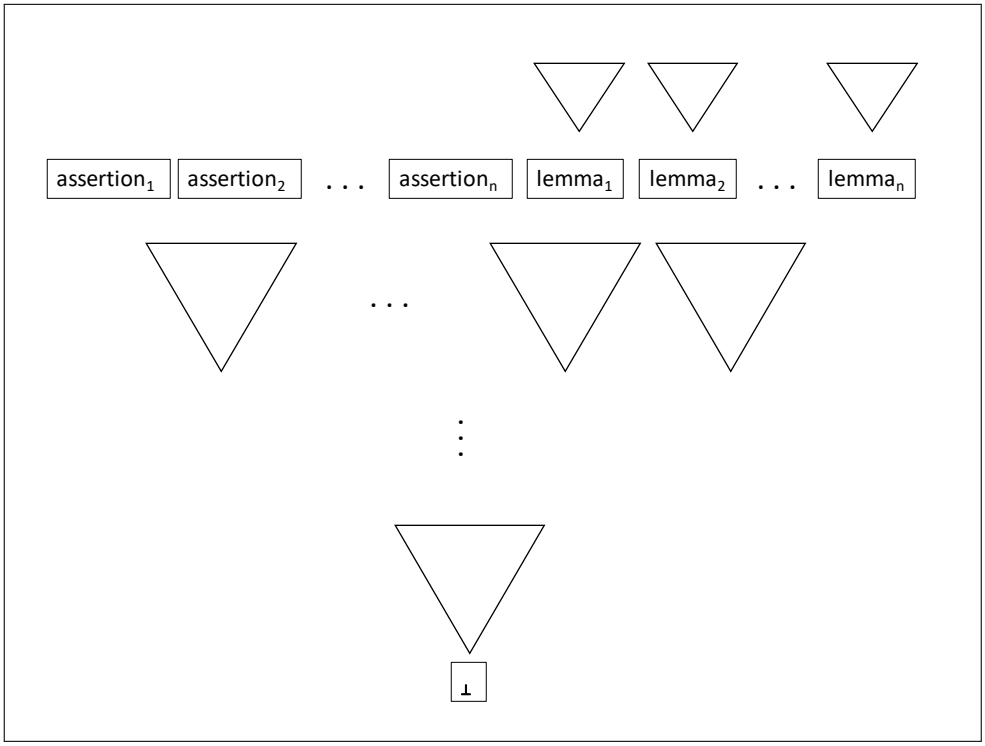


Figure 2: A proof tree of unsatisfiability.

true by a theory solver — called theory lemmas - can also be leaves. These are usually justified by the theory solver, possibly by a tree rooted at the lemma. Each node in the tree is an application of a proof rule, and the root concludes the empty clause $\bot$.

Even though this sort of calculus is common among proof-producing SMT solvers, there is no standard one and each solver uses its own variation. For example, CVC4 [11] uses a general proof framework called LFSC or Logical Framework with Side Conditions that mixes declarative rules with computable programs; veriT [17] uses a rule-based calculus defined in a SMT-LIB-like syntax; Z3 [23] uses a rule-based calculus with a relatively lower level of granularity.

## 3.2 Superposition Provers

Superposition provers or resolution provers differ from SMT solvers in that they focus on proving conjectures rather than finding a satisfying model for a set of formulas. While these problems are duals of each other, picking one over the other does make a difference in the kinds of problems that become solvable due 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 assumptions and a conjecture to prove. Additionally, while theories are built-in to SMT solvers, they need to be externally axiomatized for 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. Because theories aren't built-in to superposition provers, they distinguish axioms and assumptions in their inputs. Superposition provers, like SMT-solvers, produce resolution proofs.

The goal is to find the unsatisfiability of the negation of the conjecture along with the assumptions and the axioms, and this is done by refutation. The prover converts the input formulas into a set of clauses and uses a small set of inference rules to add implied clauses to the input set of clauses — ultimately it tries to derive the empty clause or figure out that this is not possible. These provers are resolution-based — the most important rule in the calculus is the *resolution* rule. This has evolved into the *paramodulation* rule to accommodate the notion of equality, and ultimately into the *superposition* rules that take into consideration, along with equality, an ordering on the terms in the clauses to reduce the number of rule applications.

Conceptually, a proof trace is a tree with the empty clause at the root, and the axioms/assumptions at the leaves. Each node is obtained using one or more inference rules applied to the previous nodes/leaves. In practice, the tree is implemented as a directed acyclic graph (DAG) since nodes are shared. The proof calculus is sound and refutationally complete — if the input is unsatisfiable, the empty clause will be derived. Termination, however, depends on the availability of resources (time and memory) when the input is satisfiable, and thus isn't guaranteed.

Popular resolution provers such as Vampire [35], E [54], and SPASS [60] adhere to the TPTP (Thousands of Problems for Theorem Provers) [56] input/output standard.

A superposition prover applies the inference rules to the input set of clauses until saturation — that is until the resultant set of clauses is closed under the inference rules. To keep this system efficient, provers use an ordering on the terms. The saturation algorithm used by the prover guides the process of choosing the next inference rule to apply based on this term ordering.

Given $S$, the set of input clauses, the possible outcomes of a superposition prover are:

1. The empty clause is derived from $S$, and $S$ is unsatisfiable.
2. The solver terminates without generating the empty clause, and $S$ is satisfiable.
3. The saturation algorithm does not terminate before the prover runs out of resources, and the empty clause is not generated. In this case, the result is unknown.

Since there may be redundancy in the application of inference rules and the generated clauses, many provers work on minimizing this redundancy in the interest of efficiency.

Resolution provers deal with theories by adding a user-provided axiomatization of the theory to the leaves of the proof DAG before running the saturation algorithm.

# 4 Proof Assistants

A proof assistant or interactive theorem prover (ITP) is a software tool used to formalize proofs by human-machine collaboration. Proof assistants originated with the Automath [22] and the Logic for Computable Functions (LCF) theorem prover [46], which is designed in the logic with the same name [38]. Both emphasized the principle of having a small trustable proof checker. In LCF-style theorem provers, theorems are implemented as an abstract datatype, and new theorems can only be constructed through a fixed set of functions, that correspond to the underlying logic's axiom schemata and inference rules, provided by this datatype. This keeps the trusted codebase to a minimum, and provides strong soundness guarantees. LCF-style theorem provers include the HOL family of ITPs: HOL Light [29], HOL4 [55] and Isabelle [61]. Automath is based on type theory and the Curry-Howard isomorphism, where propositions or theorem statements are types, and their proofs are programs inhabiting them. ITPs taking the Automath approach to theorem-proving include Coq [58], Agda [18], NuPRL [30], Matita [5], and Lean [24].

In the following sections, we focus on particular ATP integrations with Isabelle/HOL and Coq. Isabelle is a generic proof assistant that can be instantiated with a particular logic and calculus. Isabelle/HOL is an instance of Isabelle that provides an expressive HOL to prove theorems in its proof language Isar, and also consists of a large library of formally verified mathematics. The Coq proof assistant provides a functional programming language interface in Gallina, and implements a type theory called the calculus of inductive constructions (CIC). Similar to Automath, theorems are types in Coq, and proofs are programs inhabiting them. While these proofs are written manually by the user, Coq provides some automation in the form of *tactics*. Using tactics, the user must build a term whose type is the theorem being proved, and a proof ends with `Qed` which calls the Coq type checker to certify the proof.

# 5 Hammers

Hammers are automated tools that use ATPs to help automate the proving of goals within ITPs that depend on certain other lemmas. A hammer provides a method for automation within the ITP as follows. When a user calls the hammer's tactic on a goal, it sends the goal along with a set of premises (previously proven lemmas) to the ATP, and if the ATP is able to prove the goal,

it processes this proof in some way within the ITP to give the user a closed proof of the goal. A hammer is composed of a premise selector, which is a method to identify relevant premises that help prove a goal; a translator that bridges the gap between the logic of the ITP and that of the ATP(s); and a proof reconstructor that reconstructs in the ITP the external proof provided by the ATPs.

## 5.1 Premise Selection

A typical ITP goal $L$ has a set of hypotheses $H_1, H_2, ..., H_n$ and a conjecture $G$ to prove:

$$L : H_1 \to H_2 \to ... \to H_n \to G.$$

Proving $L$ is equivalent to proving the unsatisfiability of $\neg L$.

$$\begin{aligned}
\neg L &= \neg(H_1 \to H_2 \to ... \to H_n \to G) \\
&= \neg(\neg H_1 \vee \neg H_2 \vee ... \vee \neg H_n \vee G) && \text{by unfolding } \to \\
&= \neg\neg H_1 \wedge \neg\neg H_2 \wedge ... \wedge \neg\neg H_n \wedge \neg G && \text{by De Morgan's law} \\
&= H_1 \wedge H_2 \wedge ... \wedge H_n \wedge \neg G && \text{by double negation elimination}
\end{aligned}$$

So the problem can be equivalently stated in the context of ATPs as checking the unsatisfiability of a set of hypotheses along with the negation of the conjecture. For instance, in an SMT-LIB file, this would like this:

```
assert H_1
assert H_2
...
assert H_n
assert (not G)
check-sat
```

However, goals in ITPs aren't necessarily self-contained as in $L$. The hypotheses needed to prove $G$ might have been proven earlier. In fact, ITPs have large libraries of proven lemmas, any of which might be useful in proving a goal $G$. As such, picking the right hypotheses to send along with the negation of the goal to an ATP is challenging. This process is called premise selection. The most basic form of premise selection simply leaves it to the user to find relevant facts to prove a conjecture. But given that these libraries have hundreds of lemmas, using automatic methods to select premises is a more scalable and generalizable approach.

Modern hammers have plenty of ways to automate premise selection. Some of them use machine learning to learn axiom selection from previously successful proofs using a variety of methods based on Bayesian statistics [2], nearest-neighbor ranking functions [33], and non-linear kernel methods [2]. Others use simpler algorithms; the relevance filter by Meng and Paulson [45] for the Isabelle/HOL ITP selects relevant facts by giving priority to those with a larger

number of symbols in common with the goal. The Divvy system [52] also uses a syntactic relevance filtering technique; it also uses an ordering technique based on latent semantics called APRILS (Automated Prophesier of Relevance Incorporating Latent Semantics). Others use semantics instead of syntax to guide the selection process. For instance, the SRASS (Semantic Relevance Axiom Selection System) [57] finds countermodels of the conjecture and selects axioms that exclude the countermodels. Other work combines these various types of methods to increase efficiency [31, 36].

## 5.2 Translation

ITPs use expressive higher-order logics sometimes with set-theoretic notation, whereas ATPs are mostly restricted to first-order logic (FOL). Thus, not all ITP problems are transferable to an ATP. However, there is a large enough subset of problems provable by an ATP that can help an ITP user. When a goal is provable by an ATP, it needs to be soundly translated from the ITP's logic to that of the ATP. The translation can involve various tricks to eliminate higher-order constructs [44] such as higher-order quantification, partial function applications, and anonymous functions from the goal; and an encoding of complex types into simpler types, where applicable. Furthermore, resolution-based provers usually do not have built-in theories, so a theory must be externally axiomatized within the ATP. With SMT solvers, there maybe a stronger correspondence between some ITP types and sorts in the SMT solver, so the respective theory might serve as an axiomatization.

## 5.3 Proof Reconstruction

Once the goal is translated to a problem understandable to the ATP, the result from the ATP needs to be soundly translated back to a valid ITP output. As previously mentioned, the validity of a goal in the ITP corresponds to the unsatisfiability of its negation in the ATP. If it is satisfiable, this translates to a counterexample of the fact that the goal holds. Hammers sometimes use this feature of the ATP to avoid wasting effort trying to prove unprovable subgoals. More interestingly, if the goal is found to be unsatisfiable in the ATP, the refutation proof of its unsatisfiability needs to be processed by the ITP. In its simplest form, this pipeline consists of using the ATP as an oracle, that is, to consider the goal to be proven in the ITP if the ATP concludes that its negation is unsatisfiable. This compromises the ITP's trustworthiness and adds to the the trust-base, the entire ATP.

An improvement is to independently reconstruct the proof in the proof assistant once the ATP finds the goal to be unsatisfiable. In this case, the only information the ITP gets from the ATP is that the goal is provable given the premises. It does not care about how the ATP proved this conjecture. Essentially, the ATP acts as a relevance filter for the prover inside the ITP. This is done, for instance, with Isabelle and the Metis prover [49]. A more sophisticated but also more complex integration uses the proof steps used by the ATP

in addition to the premises to guide the construction of a proof in the ITP, as done in PRocH [32] which reconstructs TPTP proofs in HOL Light.

# 6 Sledgehammer

Sledgehammer is a component of Isabelle/HOL that uses external ATPs to guide Isabelle/HOL's proof methods. Sledgehammer uses both resolution provers and SMT solvers for proof automation. When a user wishes to find a proof for a conjecture using Sledgehammer, the tool picks a few hundred relevant lemmas from Isabelle's libraries and sends them along with the conjecture to the ATP. The conjecture is translated from Isabelle's polymorphic higher-order logic (HOL) to the ATP's first order logic (FOL). Sledgehammer sends this query in parallel to all available ATPs. If an ATP is able to prove the conjecture, Isabelle's own FOL subprover - Metis - sets out to reprove the goal with the same premises. Essentially, the role of the ATP in this integration is to filter out relevant facts for Metis to efficiently work with and also to eliminate disprovable sub-goals from Metis's search space. This is useful in Isabelle because Metis is considerably slower than the external ATPs.

In the rest of this section, we expand on the work done to integrate SMT solvers with Isabelle/HOL via Sledgehammer. Sledgehammer provides the *smt* tactic to utilize this feature.

This integration, like other hammer integrations, consists of a premise selection phase, a translation from Isabelle's HOL to the SMT solvers' FOL, and the reconstruction of proofs found by the SMT solver with inference rules of Isabelle/HOL. Reconstruction involves either passing the minimal amount of facts needed to prove the goal by the external solver to Metis, Sledgehammer's internal prover or, Sledgehammer can reconstruct the proof of the external prover, inference by inference. The SMT solvers integrated in Sledgehammer are Z3, Yices [25], and CVC3 [9]. The integration with CVC3 was later extended to one with CVC4. Yices and CVC3 are trusted as oracles. Trusting SMT-solvers is not a dependable technique for previously mentioned reasons. The integration with Z3 involves proof production by Z3 and reconstruction of these proofs in Isabelle.

## 6.1 Translation

The integration of Sledgehammer with SMT solvers consists of a translation from HOL to many-sorted FOL. The target of the translation is the SMTLIB standard for SMT-solvers. The translation is sound - validity of the translated problem implies validity of the original problem - but not complete - validity of the original problem does not necessarily imply validity of the translated problem. HOL is more expressive than FOL. As such, all first-order terms are representable in HOL. Translation of this first-order subset of HOL is pretty straightforward. More interestingly, the translation has to deal with certain higher-order features that have no obvious first-order counterpart. Some of the

incompleteness of the translation does come from the inability to translate some of these features.

The type system of HOL consists of type variables and compound types. A type variable is a schematic type that can be instantiated to any particular type. For instance, $\alpha \to \beta$ is the type of all functions from some type $\alpha$ to some type $\beta$. This represents a polymorphic type which is handled using monomorphization — the process of generating all instances of a set of schematic terms based on a set of monomorphic terms until a fixed point is reached. This process can be nonterminating. The translation performs a fixed number of monomorphization steps since the terms that actually contribute to proofs are typically those that are generated by the first few steps. Compound types are types constructed from other types — $\kappa\ \tau_1 ... \tau_n$ is a type composed of the types $\tau_1, ..., \tau_n$. After recursively monomorphizing the types $\tau_1, ..., \tau_n$, the translation represents the compound type $\kappa\ \tau_1 ... \tau_n$ as a first-order type $\kappa^n$.

An anonymous function from HOL is translated to a named function and a quantified constraint is added to specify the function's behavior. This is called $\lambda-$lifting. Specifically, a term $t$ that contains a $\lambda-$abstraction (anonymous function) $\lambda x.u$ is translated to a term $t$ with all occurrences of this anonymous function substituted with $c$ and the constraint $\forall x.c\ x = u$ is added as an axiom.

HOL also allows for partial applications of functions, which are handled in the translation by using a constant `app` that represents explicit applications. If $c$ has arity $m+1$ and is represented in the problem with at least $m$ arguments, then $c\ t_1 ... t_m$ is represented as is, and this term can be applied to another argument $t_{m+1}$ as $\texttt{app}\ (c\ t_1 ... t_m)\ t_{m+1}$ along with an axiomatization of what it means to construct terms with `app`:

$$\forall t_1, t_2.\ \texttt{app}\ t_1\ t_2 = t_1\ t_2.$$

The translation is less general than the translation used by Sledgehammer for superposition provers, in that, since (monomorphic) types are embedded in SMT solvers, some of the types from Isabelle such as integers and reals are mapped to their corresponding SMT types. In addition, this work was extended with support for bit-vectors or machine integers [15]. The Isabelle/HOL counterpart to the SMT-LIB theory of bit-vectors was developed by Dawson [21]. All the SMT-LIB bit-vector operations have corresponding definitions in the bit-vector libraries of HOL4 and Isabelle/HOL.

An evaluation on this translation showed that SMT solvers improved and complemented the proof automation achieved with resolution provers alone on many trivial and non-trivial problems.

## 6.2 The Z3 Proof System

Z3's proof system is a sequent calculus that uses 38 proof rules. Some of these rules are presented in Figure 3 using sequents $\Gamma \vdash \varphi$ where $\Gamma$ is a set of formulas called the hypotheses and $\varphi$ is a formula called the proposition. A rule consists of one or more sequents as premises and a sequent as the conclusion. Similar to resolution provers, a proof is represented as a tree in theory, but implemented

$$\frac{}{\vdash \top}\ \texttt{true} \qquad \frac{\varphi \in \Pi}{\{\varphi\} \vdash \varphi}\ \texttt{asserted} \qquad \frac{\Gamma_1 \vdash \varphi_1 \quad \Gamma_2 \vdash \varphi_1 \to \varphi_2}{\Gamma_1 \cup \Gamma_2 \vdash \varphi_2}\ \texttt{mp}$$

$$\frac{}{\{\varphi\} \vdash \varphi}\ \texttt{hypothesis} \qquad \frac{\Gamma \setminus \{l_1, ..., l_n\} \vdash \neg l_1 \vee ... \vee \neg l_n}{\Gamma \cup \{l_1, ..., l_n\} \vdash \bot}\ \texttt{lemma}$$

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \leftrightarrow \top}\ \texttt{iff}_{\top} \qquad \frac{\Gamma \vdash \neg\varphi}{\Gamma \vdash \varphi \leftrightarrow \bot}\ \texttt{iff}_{\bot} \qquad \frac{\Gamma \vdash \varphi_1 \leftrightarrow \varphi_2}{\Gamma \vdash \varphi_1 \sim \varphi_2}\ \texttt{iff}_{\sim} \qquad \frac{}{\vdash t \simeq t}\ \texttt{refl}_{\simeq}$$

$$\frac{\Gamma \vdash l_1 \wedge ... \wedge l_n}{\Gamma \vdash l_i}\ \texttt{elim}_{\wedge} \qquad \frac{\Gamma \vdash \neg(l_1 \vee ... \vee l_n)}{\Gamma \vdash \neg l_i}\ \texttt{elim}_{\neg\vee} \qquad \frac{\Gamma \vdash t_1 \simeq t_2}{\Gamma \vdash t_2 \simeq t_1}\ \texttt{symm}_{\simeq}$$

$$\frac{\Gamma_1 \vdash t_1 \simeq t_2 \quad \Gamma_2 \vdash t_2 \simeq t_3}{\Gamma_1 \cup \Gamma_2 \vdash t_1 \simeq t_3}\ \texttt{trans}_{\simeq} \qquad \frac{\Gamma \vdash (t_1 \approx t_2) \simeq (t_1' \approx t_2')}{\Gamma \vdash (t_2 \approx t_1) \simeq (t_2' \approx t_1')}\ \texttt{comm}_{\simeq,\approx}$$

Figure 3: Simple Z3 proof rules.

as a directed acyclic graph (DAG) by Z3 to due to node sharing. A proof tree consists of axioms from the proof system as leaves, and application of rules as nodes, and since proof trees document unsatisfiability of the input formulas, the root of the tree concludes the empty clause $\bot$, which is represented by the sequent $\Pi' \vdash \bot$ where $\Pi'$ is the *unsat core* — a subset of the assertions $\Pi$ initially given to Z3 to be shown unsatisfiable.

A representative subset of the proof rules are documented in [14] and some of them are repeated here to illustrate the workings of the Z3 proof system. In addition to the simple rules from Figure 3, resolution is a rule that's central to the proof calculus. Resolution was introduced in Section 3.1 and a special case of it called unit resolution is represented as follows in the sequent calculus of Z3, where $I = \{1, ..., n\}$ and $J$ is a non-empty subset of $I$ :

$$\frac{\Gamma \vdash \bigvee_{i \in I} l_i \quad \langle \Gamma_j \vdash \neg l_j \rangle_{j \in J}}{\Gamma \cup \bigcup_{j \in J} \Gamma_j \vdash \bigvee_{i \in I \setminus J} l_i}\ \texttt{unit-resolution}$$

Additionally, Z3 has a set of rules for converting input formulas into equisatisfiable formulas in conjunctive normal form (CNF) with polynomial complexity, based on Tseitin's CNF conversion method [59]. Tseitin's method introduces auxiliary variables and constraints asserting their equivalence with sub-formulas that aren't in CNF, until the entire formula is in CNF. Z3 runs this algorithm lazily and optimizes it to avoid introducing too many new variables. A formula is in negation normal form (NNF) if negations are applied only to variables and the only Boolean operators in it are conjunction and disjunction. NNF is an intermediate step in the CNF conversion process when quantifiers are involved and Z3 has a set of rules for translation of formulas to NNF in linear time.

The main rule for reasoning about equality is the *congruence* rule:

$$\frac{\langle \Gamma_j \vdash t_j \simeq t'_j \rangle_{j \in J} \quad \langle t_i \equiv t'_i \rangle_{i \in I \setminus J}}{\bigcup_{j \in J} \Gamma_j \vdash f(t_1, ..., t_n) \simeq f(t'_1, ..., t'_n)} \ \texttt{cong}$$

Equality ($=$), equivalence ($\iff$), and equisatisfiability ($\sim$) are congruence relations, and $\simeq$ represents any one of these relations. $\equiv$ represents syntactic equivalence of terms. $I$ and $J$ are defined as above for resolution.

Z3 is also proof producing for quantified formulas. $Q$ represents either of the two quantifiers — the universal quantifier $\forall$ or the existential quantifier $\exists$ — and $\overline{x}$ represents a list of variables $x_1, ..., x_n$. The following proof rule allows introduction of quantifiers:

$$\frac{\Gamma \vdash \varphi_1[\overline{x}] \sim \varphi_2[\overline{x}]}{\Gamma \vdash (Q\overline{x}.\varphi_1[\overline{x}]) \sim (Q\overline{x}.\varphi_2[\overline{x}])} \ \texttt{intro}_Q$$

The universal quantifier can be eliminated by instantiation and an existential quantifiers can be eliminated using skolemization.

$$\frac{}{\neg(\forall \overline{x}.\varphi[\overline{x}]) \vee \varphi[\overline{x} \mapsto \overline{t}]} \ \texttt{inst}_\forall$$

Note the representation of implication $a \to b$ in its CNF form $\neg a \vee b$.

Skolemization [47] is the process of removing existential quantifiers from formulas using constants called Skolem constants as witnesses.

$$\frac{}{(\exists \overline{x}.\varphi[\overline{y}, \overline{x}]) \sim \varphi[\overline{y}, \overline{f}(\overline{y})]} \ \texttt{sk}_\exists \qquad \frac{}{(\neg\forall \overline{x}.\varphi[\overline{y}, \overline{x}]) \sim (\neg\varphi[\overline{y}, \overline{f}(\overline{y})])} \ \texttt{sk}_\forall$$

Besides these, Z3 provides theory rules, that conclude disjunctions of theory literals called theory lemmas, for theory specific reasoning and rewriting rules. These are called `th-lemma` rules. The theory of bit-vectors has special `th-lemma-bv` rules owing to the fact that bit-vector proofs involve reasoning about the corresponding bits of a bit-vector — a process called bit-blasting.

Rewriting is an important part of SMT solvers — this process takes care of various simplifications and transformations to canonical forms. Z3 provides a set of metatheorems that are rewriting patterns that may repeatedly occur such as symmetry of equality, common propositional tautologies, etc. Sledgehammer allows users to add rules to the set of metatheorems. If one of Z3's rewrite rules fails to be reconstructed in Sledgehammer, the user can add a metatheorem that matches the pattern to fix the failure. For the bit-vector extension [15], schematic theorems that play the same roles as metatheorems have proved to be crucially time-saving. If a schematic theorem's conclusion matches a term, the entire theorem is instantiated by the scheme. Besides commutativity and associativity rules specific to bit-vector operations, examples include simplification rules such as the neutrality of the bit-vector representing 0 for bit-wise disjunction.

### 6.3 Proof Reconstruction

Sledgehammer *smt* tactic trusts the Yices and CVC3 solvers, so only proofs produced by Z3 are reconstructed in Isabelle. The Z3 proofs are reconstructed step-by-step, and currently supported theories are equality, linear arithmetic, and bit-vector arithmetic. Since Z3's FOL is a subset of Isabelle's HOL, formulas in Z3 proofs are easily representable as terms in Isabelle/HOL. The Z3 proof tree is scanned in depth-first post-order, starting from the root node, and reconstructed in Isabelle, step-by-step. Each proof node, contains information such as rule name, references to premises and propositions, and is represented as an algebraic datatype with a unique ID assigned by Z3. Sledgehammer stores these in a balanced tree with logarithmic lookup using the IDs as keys. Reconstruction of a proof node represents a proven Isabelle/HOL theorem of the node's conclusion. Every node's premises are discharged as theorems, which are then used to derive the node's conclusion. Once reconstructed, a proof node can be reused by Sledgehammer. If $\Pi$ is the set of asserted formulas, $\Pi'$ represents the unsat core returned by Z3 — the minimal subset of $\Pi$ required to prove the problem unsat. After reconstructing the root node, Sledgehammer checks the rest of the tree to see that only the formulas from $\Pi'$ remain as leaves.

Proof reconstruction is sound by construction and does not extend the trusted code base beyond Isabelle's LCF kernel, but it is incomplete. In fact, this work led to the discoveries of soundness bugs in Z3 that were subsequently fixed by the developers.

For the theory of bit-vectors, the *smt* tactic involves some additional optimizations to offset the complexity added by bit-blasting - a process that can cause an exponential blow-up. One of these - schematic theorem instantiation - is described in the previous sub-section. Memoization of theorems, that is, caching of certain computationally expensive bit-vector specific theorems in a structure called a term net also proved useful. These techniques, and many useful bit-vector rewrite rules — are used with HOL4 and Isabelle/HOL's existing automation for bit-vector reasoning.

Since Z3's proofs are relatively coarse-grained, proof reconstruction in Sledgehammer has to perform proof search when the proof step is not reflected in Z3's proof. According to [14], enhancing Z3 with more descriptive rules would help improve the *smt* tactic and strengthen the integration.

## 7 SMTCoq

SMTCoq implements a skeptical cooperation between Coq and proof-producing SMT and SMT solvers. It can be used as a proof checker for proofs produced by SAT and SMT solvers. More relevantly, it is a Coq plugin that uses SAT and SMT solvers to prove goals in Coq. It is written and proved correct in Coq, with some OCaml support for translation and preprocessing. SMTCoq allows the user to query a number of SAT/SMT solvers and if the solver is able to validate the query, that is, it determines that the negation of the conjecture is

unsatisfiable, it produces a proof of unsatisfiability, which is checked in Coq.

SMTCoq's uses *computational reflection* [3] to invoke Coq's computation within proofs. Proofs can be shortened by increasing their computational part, due to the Curry-Howard isomorphism that Coq is based on. Reflection requires the SMT solver's first-order terms to have two representations. FOL terms are represented in Coq using Coq's first-order terms in a *shallow embedding* and also using datatypes defined in Coq — the *deep embedding*. Thus, FOL statements are naturally represented in Coq in its shallow embedding. Reflection allows to prove these statements using computations on their deep embedding. SMTCoq provides the tools necessary to switch between the two embeddings — an *interpretion* function compiles deep terms to their Coq counterparts; and *reification* - written in OCaml without compromising soundness — transforms shallow terms to the corresponding deep terms.

SMTCoq represents the quantifier free formulas from the SMT solver in Coq as Boolean terms. Thus, a Boolean decision procedure in SMTCoq, checks proof certificates from solvers. However, `Prop` is the type of propositions in Coq, and thus is the type of proofs. The Ssreflect plugin allows reflection from the `bool` type in Coq to `Prop`. This is crucial since Booleans are easier to manipulate via computations and propositions are harder to prove.

SMTCoq is sound — proven to be correct in Coq - but not complete. This means that when an SMT solver proves a conjecture, we can we be confident of its response since we get a `Qed` proof in Coq. However, if SMTCoq is not able to prove a conjecture, we cannot be sure that its unprovable. Since certain theories can make SMT solving undecidable, completeness is not a realistic goal.

Since different SMT solvers produce resolution proofs of unsatisfiability in different formats, SMTCoq has its own input certificate format. The proofs from the different SMT solvers are translated to an SMTCoq certificate which can then be checked within Coq. Given an input query, the SMT solver first converts it into CNF (conjunction normal form) which is better suited for resolution proofs. The final resolution proof might also consist of smaller proofs of theory lemmas from the theory solver.

SMTCoq takes a modular approach of checking an SMT proof. The checker is divided into a main checker that delegates parts of the proof to small checkers. The proof is divided into steps that small checkers can check. There is a small checker for CNF conversion, one for resolution, and one for each theory. SMT solvers perform a preprocessing step to rewrite certain input formulas to a simplified version, and SMTCoq has a step and a checker corresponding to this rewriting phase. The main checker divides the proof into steps, gives the step to the relevant small checker, and checks that in the end, the empty clause is deduced from the initial query. Since this initial query is the negation of our conjecture, deriving the empty clause from it signifies its unsatisfiability. Each small checker operates independently and maintains an invariant that allows the checking process to be split this way. Specifically, the correctness of the main checker depends on the correctness of each small checker. Since each small checker transforms the initial query, generated from the initial goal, by replacing clauses by new ones, its correctness guarantees that it will not make

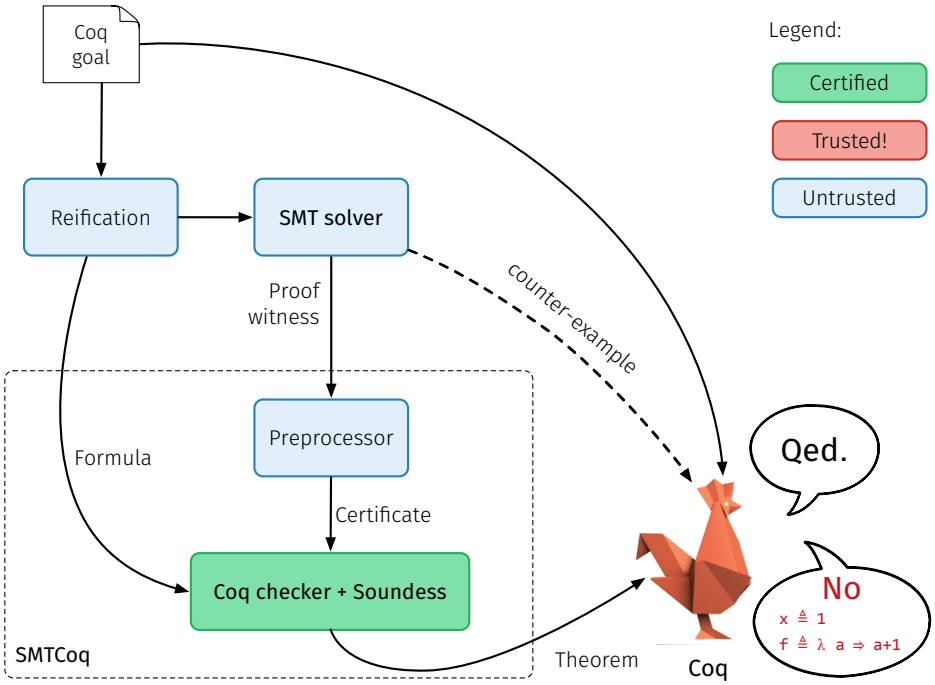


Figure 4: SMTCoq's integration with SMT solvers. Credits: Alain Mebsout

unsound transformations. In other words, the small checker guarantees that its transformations preserve the (un)satisfiability of the current query.

SMTCoq uses Coq's native arrays to store the clauses, a set of which represent the goal. The main checker handles this initial array of clauses representing the negation of the goal in CNF, and each small checker computes a clause that is implied from a subset of the clauses. For efficiency, SMTCoq replaces the unnecessary clauses with new ones when it knows that they will not be useful anymore. After all the steps are handled by the small checkers, the main checker checks that the final implied clause is the empty clause. Sub-terms of deep terms are hash-consed (i.e., cached for maximal term sharing. Additionally, variables and literals of FOL are represented in Coq using machine integers on which fast computations can be performed to manipulate them.

Currently SMTCoq supports the SAT-solvers zChaff [39] and Glucose [6], and the SMT-solvers CVC4 and veriT.

# 8 Comparison

Sledgehammer has three possible integrations with SMT-solvers: it either (*i*) trusts the SMT-solver as an oracle, (*ii*) uses the SMT-solver as a relevance filter on premises so it can prove goals efficiently using Metis, or (*iii*) reconstructs

proofs produced by the SMT solver to produce goals without compromising the soundness guarantees of the small LCF-kernel. The third of these is most comparable to how SMTCoq operates — it sends Coq conjectures to the SAT or SMT solver, which returns a proof of the conjecture if it finds it to be provable; SMTCoq then checks this proof within Coq by reflection. SMTCoq can also be used as a checker for proofs produced by SMT solvers. In fact, while there exists an independent LFSC checker for LFSC proofs provided by CVC4, veriT produces proofs but does not have a dedicated checker. It relies on integrations with proof assistants to check its output. In this section, we compare the usage of SMTCoq and Sledgehammer according to their ability to use SMT-solvers to automate proofs within their respective ITPs.

Recall that ITPs have more expressive logics than ATPs, and that roughly, the logic of an ATP is a subset of that of an ITP. So they can be used to automate goals that are expressed within this subset. Sledgehammer goes a little further by describing a translation of certain HOL features into FOL features understood by SMT solvers. While this translation is incomplete, it captures enough HOL features to widen the set of problems that an SMT solver can help automate. Since SMTCoq does not deal with Coq's non-FOL constructs, SMTCoq's tactics cannot be called on Coq goals that contain, for instance, anonymous functions, partial function applications, or higher-order terms in general. SMTCoq does not deal with polymorphic types or compound types either. There is a correspondence between types in Coq and theories in SMT, and only Coq goals that have these particular types can use the SMT solver. The integer sort from SMT-LIB is mapped to Coq's $Z$ type, and Coq's $Bool$ type corresponds to propositional reasoning in SAT or SMT. However, SMTCoq defines custom types for machine integers and arrays. While these serve as rich, independent Coq libraries for these types, they are not standardized in Coq. In contrast, Sledgehammer only supports the theories of linear integer arithmetic and bit-vectors, besides equality over uninterpreted functions which is also supported by SMTCoq. Additionally, SMTCoq does not have support for quantified formulas. Although the goal in Coq is universally quantified, on the SMT side, it amounts to checking the unsatisfiability of a quantifier-free formula. This is not the case with formulas containing either alternating quantifiers, or those that occur outside the head of the formula — SMTCoq will fail on these. Sledgehammer uses Z3's quantifier proofs to support quantified fragments of its theories.

Premise selection is a crucial part of Sledgehammer. A lot of research has gone into optimizing premise selection to make proving more efficient. Consequently, the SMT integration also uses Sledgehammer's premise selection mechanisms to supply the SMT solver with the best possible hypotheses to prove a goal. This process is so imperative that using the SMT solver just as a relevance filter for premises has given useful results. In SMTCoq, each lemma is considered an independent entity with its set of hypotheses $H$ and goal $G$ that are translated to an SMT-LIB problem. If the proof of $G$ depends on conjectures stated outside of $H$, the SMT solver will likely not be able to prove them. This difference in ideology comes from the difference in axiomatizations of theories in

SMT solvers and resolution provers. Sledgehammer was originally coupled only with resolution provers that do not have a notion of theories. Any additional axioms of a theory must be appended to the input of a superposition prover. SMT solvers have theories encoded into them, which might have facts besides $H$ that might help prove $G$. However, this might not always be the case. In the next section, we propose an extension to SMTCoq using abduction to deal with this shortcoming.

The Sledgehammer integration with SMT solvers is specific to Z3 (although there are looser integrations with other SMT solvers). Although SMT solvers have a common input/output syntax in SMT-LIB, their proof formats are quite different. Extending Sledgehammer's *smt* tactic to other SMT solvers might not be straightforward due to the idiosyncrasies of Z3. On the other hand, SMTCoq claims to be generic and extensible. SMTCoq has a certificate format, that is different from the proof formats of each SMT solver. To extend SMTCoq's support with a new solver, one only needs to write a preprocessor that translates proofs from this solver's format to that of the SMTCoq certificate. The SMTCoq checker's soundness lemmas are also general enough to make it possible to add more theories simply by extending the type of terms in SMTCoq. These claims are corroborated by several successful extensions to SMTCoq. Initially, it had support only for the ZChaff SAT solver and the veriT SMT-solver, with the ability to reason in the theories of EUF and LIA. Since its inception, the CVC4 SMT solver and the theories of bit-vectors and arrays have been added to SMTCoq's capabilities [26].

# 9 Related Work

In this section, we present other integrations between ATPs and ITPs, and related research, that we didn't get into deeply in this work. [13] presents the framework for embedding the resolution calculus within type theoretic ITPs. Some instances of this embedding are: in [1], between the ${}_3TAP$ [12] ATP and the Karlsruhe Interactive Verifier (KIV) [50]; IVY [42], a verified prover in the ACL2 framework [34] that calls the Otter [43] theorem prover. Connections between ITPs and SMT solvers include ones between the PVS system [48] and the Yices SMT solver in [53], between the UCLID solver [37] and ACL2 in [40], and haRVey (now veriT) with Isabelle/HOL in [27].

CoqHammer [20] is a hammer for the Coq proof assistant. It's novelty comes from the fact that most current hammers are for LCF-style provers that have a HOL that is different from CIC. CoqHammer has a translation from Coq's CIC to untyped FOL which has a practical level of soundness and completeness.

# 10 Conclusion and Future Work

We have looked at various integrations between automatic and interactive theorem provers. Hammers that typically utilize resolution provers within ITPs have

complex premise selection methods to pick an optimal set of lemmas that will allow the ATP to prove the goal. We looked at an extension of Sledgehammer that applies this framework with an SMT solver. On the other hand, SMTCoq queries SAT and SMT solvers with independent lemmas in Coq and certifies the solver's result within Coq using computational reflection.

These illustrate the currently popular solutions to the premise selection problem - use smart methods to find a good set of premises that will enable the ATP to find a proof; alternatively, hope that the ATP's axiomatization is strong enough to have any extra information necessary, or fail otherwise. We suggest involving the user in the premise selection process with some help from the ATP. Our proposal involves using CVC4's abduction solver [51] in SMTCoq to do this. For a theory $T$, given a set of axioms $A$ and a goal $G$, abduction finds a formula $\phi$ (the abduct) — if it exists — such that

$$A \wedge \phi \models_T G$$

In other words, it finds a formula $\phi$ that is consistent with the axioms and when added to them, allows for the goal to be proven. When SMTCoq sends CVC4 a lemma $H \models G$ and CVC4 finds that $G$ does not follow from $H$, the idea would be to have it return a set of abducts for $G$ and $H$. The user would then prove the relevant abduct from previously proven lemmas and query CVC4 with that abduct added to $H$, thus enabling CVC4 to prove the goal. We hope to take SMTCoq's CVC4 integration forward this way and avoid the premise selection problem.

# References


[1] Ahrendt, Beckert, Hähnle, Menzel, Reif, Schellhorn, and Schmitt. *Integrating Automated and Interactive Theorem Proving*, pages 97–116. Springer Netherlands, Dordrecht, 1998.

[2] J. Alama, T. Heskes, D. Kühlwein, E. Tsivtsivadze, and J. Urban. Premise selection for mathematics by corpus analysis and kernel methods. *J. Autom. Reason.*, 52(2):191–213, 2014.

[3] S. F. Allen, R. L. Constable, D. J. Howe, and W. E. Aitken. The semantics of reflected proof. In *[1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 95–105, 1990.

[4] C. Artho, A. Biere, and M. Seidl. Model-based testing for verification backends. In M. Veanes and L. Viganò, editors, *Tests and Proofs*, pages 39–55, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[5] A. Asperti, W. Ricciotti, C. S. Coen, and E. Tassi. The matita interactive theorem prover. In *Proceedings of the 23rd International Conference on Automated Deduction*, CADE'11, page 64–69, Berlin, Heidelberg, 2011. Springer-Verlag.

[6] G. Audemard and L. Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, page 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[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] C. Barrett and C. Tinelli. Cvc3. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, page 298–302, Berlin, Heidelberg, 2007. Springer-Verlag.

[10] C. Barrett and C. Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking.*, pages 305–343. 2018.

[11] 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.

[12] B. Beckert, R. Hähnle, P. Oel, and M. Sulzmann. The tableau-based theorem prover 3tap version 4.0. In *Proceedings of the 13th International Conference on Automated Deduction: Automated Deduction*, CADE-13, page 303–307, Berlin, Heidelberg, 1996. Springer-Verlag.

[13] M. Bezem, D. Hendriks, and H. de Nivelle. Automated proof construction in type theory using resolution. In D. McAllester, editor, *Automated Deduction - CADE-17*, pages 148–163, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[14] S. Böhme. *Proving Theorems of Higher-Order Logic with SMT Solvers*. PhD thesis, Technical University Munich, 2012.

[15] 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.

[16] J. Bornholt. Program synthesis explained. https://homes.cs.washington.edu/~bornholt/post/synthesis-explained.html, 2015. [Online; accessed 28-August-2018].

[17] 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.

[18] A. Bove, P. Dybjer, and U. Norell. A brief overview of agda – a functional language with dependent types. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 73–78, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[19] R. Brummayer and A. Biere. Fuzzing and delta-debugging smt solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, SMT '09, page 1–5, New York, NY, USA, 2009. Association for Computing Machinery.

[20] L. Czajka and C. Kaliszyk. Hammer for coq: Automation for dependent type theory. *Journal of Automated Reasoning*, 61, 06 2018.

[21] J. Dawson. Isabelle theories for machine words. *Electronic Notes in Theoretical Computer Science*, 250(1):55 – 70, 2009. Proceedings of the Seventh International Workshop on Automated Verification of Critical Systems (AVoCS 2007).

[22] N. G. de Bruijn. The mathematical language automath, its usage, and some of its extensions. In M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, editors, *Symposium on Automatic Demonstration*, pages 29–61, Berlin, Heidelberg, 1970. Springer Berlin Heidelberg.

[23] 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.

[24] 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.

[25] B. Dutertre. Yices 2.2. In A. Biere and R. Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[26] B. Ekici, G. Katz, C. Keller, A. Mebsout, A. J. Reynolds, and C. Tinelli. Extending smtcoq, a certified checker for SMT (extended abstract). In J. C. Blanchette and C. Kaliszyk, editors, *Proceedings First International Workshop on Hammers for Type Theories, HaTT@IJCAR 2016, Coimbra, Portugal, July 1, 2016*, volume 210 of *EPTCS*, pages 21–29, 2016.

[27] P. Fontaine, J.-Y. Marion, S. Merz, L. P. Nieto, and A. Tiu. Expressiveness + automation + soundness: Towards combining smt solvers and interactive proof assistants. In H. Hermanns and J. Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 167–181, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[28] 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.

[29] 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.

[30] P. B. Jackson. *The Nuprl Proof Development System, Version 4.2 Reference Manual and User's Guide*. The PRL Group at Cornell University, 1996.

[31] C. Kaliszyk and J. Urban. Mizar 40 for mizar 40. *CoRR*, abs/1310.2805, 2013.

[32] C. Kaliszyk and J. Urban. Proch: Proof reconstruction for hol light. In M. P. Bonacina, editor, *Automated Deduction – CADE-24*, pages 267–274, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[33] C. Kaliszyk and J. Urban. Stronger automation for flyspeck by feature weighting and strategy evolution. In J. C. Blanchette and J. Urban, editors, *Third International Workshop on Proof Exchange for Theorem Proving, PxTP 2013, Lake Placid, NY, USA, June 9-10, 2013*, volume 14 of *EPiC Series in Computing*, pages 87–95. EasyChair, 2013.

[34] M. Kaufmann and J. S. Moore. An acl2 tutorial. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 17–21, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[35] L. Kovács and A. Voronkov. First-order theorem proving and vampire. In N. Sharygina and H. Veith, editors, *Computer Aided Verification*, pages 1–35, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[36] D. Kühlwein, T. van Laarhoven, E. Tsivtsivadze, J. Urban, and T. Heskes. Overview and evaluation of premise selection techniques for large theory mathematics. In B. Gramlich, D. Miller, and U. Sattler, editors, *Automated Reasoning*, pages 378–392, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[37] S. K. Lahiri and S. A. Seshia. The uclid decision procedure. In R. Alur and D. A. Peled, editors, *Computer Aided Verification*, pages 475–478, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[38] J. Loeckx and K. Sieber. *LCF, A Logic for Computable Functions*, pages 199–214. Vieweg+Teubner Verlag, Wiesbaden, 1987.

[39] Y. S. Mahajan, Z. Fu, and S. Malik. Zchaff2004: An efficient sat solver. In H. H. Hoos and D. G. Mitchell, editors, *Theory and Applications of Satisfiability Testing*, pages 360–375, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[40] P. Manolios and S. K. Srinivasan. Verification of executable pipelined machines with bit-level interfaces. In *2005 International Conference on Computer-Aided Design, ICCAD 2005, San Jose, CA, USA, November 6-10, 2005*, pages 855–862. IEEE Computer Society, 2005.

[41] M. N. Mansur, M. Christakis, V. Wüstholz, and F. Zhang. Detecting critical bugs in smt solvers using blackbox mutational fuzzing, 2020.

[42] W. McCune and O. Shumsky. System description: Ivy. In D. McAllester, editor, *Automated Deduction - CADE-17*, pages 401–405, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[43] W. McCune and L. Wos. Otter - the cade-13 competition incarnations. *J. Autom. Reason.*, 18(2):211–220, Apr. 1997.

[44] J. Meng and L. C. Paulson. Translating higher-order clauses to first-order clauses. *J. Autom. Reason.*, 40(1):35–60, 2008.

[45] J. Meng and L. C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *J. Appl. Log.*, 7(1):41–57, 2009.

[46] R. Milner. Logic for computable functions: Description of a machine implementation. Technical report, Stanford, CA, USA, 1972.

[47] A. Nonnengart and C. Weidenbach. Chapter 6 - computing small clause normal forms. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, Handbook of Automated Reasoning, pages 335 – 367. North-Holland, Amsterdam, 2001.

[48] S. Owre, J. M. Rushby, and N. Shankar. Pvs: A prototype verification system. In D. Kapur, editor, *Automated Deduction—CADE-11*, pages 748–752, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[49] L. C. Paulson and K. W. Susanto. Source-level proof reconstruction for interactive theorem proving. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics*, pages 232–245, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[50] W. Reif. *The Kiv-approach to software verification*, pages 339–368. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.

[51] 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.

[52] A. Roederer, Y. Puzis, and G. Sutcliffe. Divvy: An atp meta-system based on axiom relevance ordering. In R. A. Schmidt, editor, *Automated Deduction – CADE-22*, pages 157–162, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[53] J. Rushby. Tutorial: Automated formal methods with pvs, sal, and yices. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, pages 262–262, 2006.

[54] S. Schulz. E - a brainiac theorem prover. *AI Commun.*, 15(2,3):111–126, Aug. 2002.

[55] K. Slind and M. Norrish. A brief overview of hol4. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 28–32, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[56] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[57] G. Sutcliffe and Y. Puzis. Srass - a semantic relevance axiom selection system. In F. Pfenning, editor, *Automated Deduction – CADE-21*, pages 295–310, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[58] L. Théry, P. Letouzey, and G. Gonthier. *Coq*, pages 28–35. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[59] G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.

[60] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. Spass version 3.5. In R. A. Schmidt, editor, *Automated Deduction – CADE-22*, pages 140–145, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[61] M. Wenzel, L. C. Paulson, and T. Nipkow. The isabelle framework. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 33–38, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.