Reconstructing veriT Proofs in Isabelle/HOL

Mathias Fleury
Max-Planck-Institut für Informatik,
Saarland Informatics Campus, Saarbrücken, Germany
mathias.fleury@mpi-inf.mpg.de
Graduate School of Computer Science,
Saarland Informatics Campus, Saarbrücken, Germany
s8mafleu@stud.uni-saarland.de

Hans-Jörg Schurr
University of Lorraine, CNRS, Inria, and
LORIA, Nancy, France
hans-jorg.schurr@inria.fr

Automated theorem provers are now commonly used within interactive theorem provers to discharge an increasingly large number of proof obligations. To maintain the trustworthiness of a proof, the automatically found proof must be verified inside the proof assistant. We present here a reconstruction procedure in the proof assistant Isabelle/HOL for proofs generated by the satisfiability modulo theories solver veriT which is part of the `smt` tactic. We describe in detail the architecture of our improved reconstruction method and the challenges we faced in designing it. Our experiments show that the veriT-powered `smt` tactic is regularly suggested by Sledgehammer as the fastest method to automatically solve proof goals.

1 Introduction

Proof assistants are used in verification, formal mathematics, and other areas to provide trustworthy and machine-checkable formal proofs of theorems. Proof automation allows the user to focus on the core of their argument by reducing the burden of manual proof. A successful approach to automation is to invoke an external automatic theorem prover (ATP), such as a satisfiability modulo theories (SMT) solver [5] and to reconstruct any generated proofs using the proof assistant's inference kernel. The usefulness of this approach depends on the encoding of the proof goal into the language of the ATP, the capabilities of the ATP, the quality of the generated proof output, and the reconstruction routine itself.

In the proof assistant Isabelle/HOL this approach is implemented in the `smt` tactic [8].[1] This tactic encodes the proof goal into the SMT-LIB language [4] and calls the SMT solver Z3 [14]. If Z3 is successful in finding a proof of the input problem, the generated proof is reconstructed inside Isabelle. The proof format, and hence the reconstruction process, is specific to Z3. If the reconstruction is successful, the initial proof goal holds in the Isabelle/HOL logic. The reconstruction, however, might fail due to errors (either due to a weakness in the reconstruction or due to errors to the solver) or timing out.

We have previously developed [2, 3] a prototype to reconstruct proofs generated by the SMT solver veriT [9]. In this paper we have extended these works with proper support of term sharing, tested it on a much larger scale, and present the reconstruction method in more detail. Furthermore, we reworked the syntax of the proof output to adhere stronger to the SMT-LIB standard. Given the variety of capabilities between ATPs, a greater diversity in supported systems increases the number of proof goals which can be solved by automated tools. Moreover, the fine-grained proofs produced by veriT might allow for a higher success rate in reconstruction. Lastly, the reconstruction efforts provide valuable insights for the design of proof formats.

[1] Technically, `smt` is a proof method, but the difference (whether it requires an Isabelle context) does not matter here.

Submitted to:
PxTP 2019

Similar to the proofs generated by Z3, veriT's proofs are based on the SMT-LIB language, but are otherwise different. Proofs are a list of indexed steps which can reference steps appearing before them in the list. Steps without references are tautologies and assumptions. The last step is always the deduction of the empty clause. Furthermore, steps can be marked as subproofs, which are used for local assumptions and to reason about bound variables. To shorten the proof length, we use term sharing, which is implemented using the standard SMT-LIB name annotation mechanism. Major differences to the proof format used by Z3 are the fine-grained steps for Skolemization and the presence of steps for the manipulations of bound variables [2].

Our reconstruction routine inside Isabelle is structured as a pipeline. Once the proof is parsed into a datatype and the term sharing is unfolded, the SMT-LIB terms are translated into Isabelle terms. At this point the proof can be replayed step-by-step. Special care has to be taken to handle Skolem terms, subproofs, and the unfolding of the encoding into the first-order logic of the SMT solver.

We validate our reconstruction approach in two ways. First, we replace the calls of the `smt` tactic that are currently using Z3 by veriT. Second, we use Sledgehammer to validate the utility of veriT as a backend solver for the `smt` tactic. Sledgehammer uses external ATPs to find a collection of theorems from the background theory which are sufficient to prove the goal. It then tests a collection of automated tactics on this set of theorems and suggests the fastest successful tactic to the user. On theories from the Archive of Formal Proofs, Sledgehammer suggests the usage of the veriT-powered `smt` tactic on a significant number of proof steps. This suggests that the overall checking speed can be improved by switching to the veriT-powered `smt` tactic at these points.

2 The Proofs Generated by veriT

veriT is a CDCL(T)-based satisfiability modulo theories solver. It uses the SMT-LIB language as input and output language and also utilizes the many-sorted classical first-order logic defined by this language. If requested by the user, veriT outputs a proof if it can deduce that the input problem is unsatisfiable. In proof production mode, veriT supports the theory of uninterpreted functions, the theory of linear integer and real arithmetic, and quantifiers.

We assume the reader is familiar with many-sorted classical first-order logic. To simplify the notation we will omit the sort of terms, except when absolutely needed. The available sorts depend on the selected SMT-LIB theory and can also be extended by the user, but a distinguished **Bool** sort is always available. We use the symbols x, y, z for variables, f, g, h for functions, and P, Q for predicates, i.e., functions with result sort **Bool**. The symbols r, s, t, u stand for terms. The symbols φ, ψ denote formulas, i.e., terms of sort **Bool**. We use σ to denote substitutions and $t\sigma$ to denote the application of the substitution on the term t. To denote the substitution which maps x to t we write $[t/x]$. We use $=$ to denote syntactic equality and $\simeq$ to denote the sorted equality predicate. Since veriT implicitly removes double negations, we also use the notion of complementary literals very liberally: $\varphi = \bar{\psi}$ holds if the terms obtained after removing all leading negations from φ and $\bar{\psi}$ are syntactically equal and the number of leading negations is even for φ and odd for $\bar{\psi}$, or vice versa.

A proof generated by veriT is a list of steps. A step consists of an index $i \in \mathbb{N}$, a formula φ, a rule name R taken from a set of possible rules, a possibly empty set of premises $\{p_1, \ldots, p_n\}$ with $p_i \in \mathbb{N}$, a rule-dependent and possibly empty list of arguments $[a_1, \ldots, a_m]$, and a context Γ. The arguments a_i are either terms or tuples (x_i, t_i) where x_i is a variable and t_i is a term. The interpretation of the arguments is rule specific. The context is a possible empty list $[c_1, \ldots, c_l]$, where c_i stands for either a variable or a variable-term tuple (x_i, t_i). A context denotes a substitution as described in section 2.1. Every proof ends

with a step with the empty clause as the step term and empty context. The list of premises only references earlier steps, such that the proof forms a directed acyclic graph. In Appendix A we provide an overview of all proof rules used by veriT.

To mimic the actual proof text generated by veriT we will use the following notation to denote a step:

$$c_1, \ldots, c_l \rhd i. \quad \varphi \quad (\text{RULE}; p_1, \ldots, p_n; a_1, \ldots, a_m)$$

If an element of the context c_i is of the form (x_i, t_i), we will write $x_i \mapsto t_i$. If an element of the arguments a_i is of this form we will write $x_i := t_i$. Furthermore, the proofs can utilize Hilbert's choice operator ε. Choice acts like a binder. The term $\varepsilon x.\varphi$ stands for a value v, such that $\varphi[v/x]$ is true if such a value exists. Any value is possible otherwise.

The proof format used by veriT has been discussed in prior publications: the fundamental ideas behind the proof format have been discussed in [6]; proposed rules for quantifier instantiation can be found in [11]; and more recently, veriT gained proof rules to express reasoning typically used for processing, such as Skolemization, renaming of variables, and other manipulations of bound variables [2]. As veriT develops, so does the format of the proofs generated by it. There also have been efforts to improve the proof generation process. We now give an overview of the core ideas of the proofs generated by veriT before describing the concrete syntax of the proof output.

2.1 Core Concepts of the Proof Format

Assumptions. The ASSUME rule introduces a term as an assumption. The proof starts with a number of ASSUME steps. Each step corresponds to an assertion after some implicit transformations have been applied as described below. Additional assumptions can be introduced too. In this case each assumption must be discharged with an appropriate step. The only rule to do so is the SUBPROOF rule. From an assumption φ and a formula ψ proved by intermediate steps from φ, the SUBPROOF step deduces $\neg\varphi \vee \psi$ and discharges φ.

Tautologous rules and simple deduction. Most rules emitted by veriT introduce tautologies. One example is the AND_POS rule: $\neg(\varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_n) \vee \varphi_i$. Other rules operate on only one premise. Those rules are primarily used to simplify Boolean connectives during preprocessing. For example, the IMPLIES rule removes an implication: From $\varphi_1 \implies \varphi_2$ it deduces $\neg\varphi_1 \vee \varphi_2$.

Resolution. The proofs produced by veriT use a generalized propositional resolution rule with the rule name RESOLUTION or TH_RESOLUTION. Both names denote the same rule. The difference only serves to distinguish if the rule was introduced by the SAT solver or by a theory solver. The resolution step is purely propositional; there is currently no notion of a unifier.

The premises of a resolution step are clauses and the conclusion is a clause that has been derived from the premises by some binary resolution steps.

Quantifier Instantiation. To express quantifier instantiation, the rule FORALL_INST is used. It produces a formula of the form $(\neg\forall x_1 \ldots x_n.\varphi) \vee \varphi[t_1/x_1]\ldots[t_n/x_n]$, where φ is a term containing the free variables $(x_i)_{1\leq i\leq n}$, and t_i are new variable free terms with the same sort as x_i.

The arguments of a FORALL_INST step are the list $x_1 := t_1, \ldots, x_n := t_n$. While this information can be recovered from the term, providing this information explicitly aids reconstruction because the implicit transformations applied to terms (see below) obscure which terms have been used as instances. Existential quantifiers are handled by Skolemization.

Skolemization and other preprocessing steps. veriT uses the notion of a *context* to reason about bound variables. As defined above, a context is a (possibly empty) list of variables or variable term pairs. The context is modified like a stack: rules can either append elements to the right of the current context or remove elements from the right. A context Γ corresponds to a substitution σ_Γ. This substitution is recursively defined. If Γ is the empty list, then σ_Γ is the empty substitution, i.e., the identity function. If Γ is of the form Γ', x then $\sigma_\Gamma(v) = \sigma_{\Gamma'}(v)$ if $v \neq x$, otherwise $\sigma_\Gamma(v) = x$. Finally, if $\Gamma = \Gamma', x \mapsto \varphi$ then $\sigma_{\Gamma', x\mapsto\varphi} = \sigma_{\Gamma'} \circ [\varphi/x]$. Hence, the context allows one to build a substitution with the additional possibility to overwrite prior substitutions for a variable.

Contexts are processed step by step: If one step extends the context this new context is used in all subsequent steps in the step list until the context is modified again. Only a limited number of rules can be applied when the context is non-empty. All of those rules have equalities as premises and conclusion. A step with term $\varphi_1 \simeq \varphi_2$ and context Γ expresses the judgment that $\varphi_1\sigma_\Gamma = \varphi_2$.

One typical example for a rule with context is the SKO_EX rule, which is used to express Skolemization of an existentially quantified variable. It is a applied to a premise n with a context that maps a variable x to the appropriate Skolem term and produces a step m ($m > n$) where the veriable is quantified.

$$\frac{\Gamma, x \mapsto (\varepsilon x.\varphi) \ \rhd n.}{\Gamma \ \rhd m.} \quad \frac{\varphi \simeq \psi}{(\exists x.\varphi) \simeq \psi} \quad \frac{(\dots)}{(\text{SKO\_EX}; n)}$$

Example 1. To illustrate how such a rule is applied, consider the following example taken from [2]. Here the term $\neg p(\varepsilon x.\neg p(x))$ is Skolemized. The REFL rule expresses a simple tautology on the equality (reflexivity in this case), CONG is functional congruence, and SKO_FORALL works like SKO_EX, except that the choice term is $\varepsilon x.\neg\varphi$.

$$\begin{array}{lrr}
x \mapsto (\varepsilon x.\neg p(x)) \ \rhd 1. & x \simeq \varepsilon x.\neg p(x) & (\text{REFL}) \\
x \mapsto (\varepsilon x.\neg p(x)) \ \rhd 2. & p(x) \simeq p(\varepsilon x.\neg p(x)) & (\text{CONG}; 1) \\
\rhd 3. & (\forall x.p(x)) \simeq p(\varepsilon x.\neg p(x)) & (\text{SKO_FORALL}; 2) \\
\rhd 4. & (\neg\forall x.p(x)) \simeq \neg p(\varepsilon x.\neg p(x)) & (\text{CONG}; 3)
\end{array}$$

Linear arithmetic. Proofs for linear arithmetic use a number of straightforward rules, such as LA_TOTALITY: $t_1 \leq t_2 \vee t_2 \leq t_1$ and the main rule LA_GENERIC. The conclusion of an LA_GENERIC step is a tautology of the form $(\neg\varphi_1) \vee (\neg\varphi_2) \vee \dots \vee (\neg\varphi_n)$ where the φ_i are linear (in)equalities. Checking the validity of this formula amounts to checking the unsatisfiability of the system of linear equations $\varphi_1, \varphi_2, \dots, \varphi_n$. While Isabelle provides tactics to decide the validity of a set of linear equations, the non-trivial complexity of this task was a challenge for the proof reconstruction (see Section 3.2.4).

Example 2. The following example is the proof generated by veriT for the unsatisfiability of $(x + y < 1) \vee (3 < x)$, $x \simeq 2$, and $0 \simeq y$.

$$\begin{array}{lll}
\rhd 1. & (3 < x) \vee (x + y < 1) & (\text{ASSUME}) \\
\rhd 2. & x \simeq 2 & (\text{ASSUME}) \\
\rhd 3. & 0 \simeq y & (\text{ASSUME}) \\
\rhd 4. & \neg(3 < x) \vee \neg(x \simeq 2) & (\text{LA_GENERIC}) \\
\rhd 5. & \neg(3 < x) & (\text{RESOLUTION}; 2, 4) \\
\rhd 6. & x + y < 1 & (\text{RESOLUTION}; 1, 5) \\
\rhd 7. & \neg(x + y < 1) \vee \neg(x \simeq 2) \vee \neg(0 \simeq y) & (\text{LA_GENERIC}) \\
\rhd 8. & \bot & (\text{RESOLUTION}; 7, 6, 2, 3)
\end{array}$$

```
1  (assume h1 (not (p a)))
2  (assume h2 (forall ((z1 U)) (forall ((z2 U)) (p z2))))
3  ...
4  (anchor :step t9 :args ((:= z2 vr4)))
5  (step t9.t1 (cl (= z2 vr4)) :rule refl)
6  (step t9.t2 (cl (= (p z2) (p vr4))) :rule cong :premises (t9.t1))
7  (step t9 (cl (= (forall ((z2 U)) (p z2)) (forall ((vr4 U)) (p vr4))))
8            :rule bind)
9  ...
10 (step t14 (cl (forall ((vr5 U)) (p vr5)))
11           :rule th_resolution :premises (t11 t12 t13))
12 (step t15 (cl (or (not (forall ((vr5 U)) (p vr5))) (p a)))
13           :rule forall_inst :args ((:= vr5 a)))
14 (step t16 (cl (not (forall ((vr5 U)) (p vr5))) (p a)) :rule or :premises (t15))
15 (step t17 (cl) :rule resolution :premises (t16 h1 t14))
```

Figure 1: Example proof output. Assumptions are introduced (line 1–2); a subproof renames bound variables (line 4–8); the proof finishes with instantiaton and resolution steps (line 10–15)

Implicit transformations. In addition to the explicit steps, veriT performs some transformations on proof terms implicitly without creating steps. To ensure compatibility with future versions of veriT, proof reconstruction must assume that those transformations are applied between any two steps. Furthermore, veriT can not introduce additional types of implicit transformations.

- Removal of double negation: Formulas of the form $\neg(\neg\varphi)$ are silently simplified to φ.
- Removal of repeated literals: If the step formula is of the form $\varphi_1 \vee \varphi_2 \vee \cdots \vee \varphi_n$ with $\varphi_i = \varphi_j$ for some $i \neq j$, then φ_j is removed. This is repeated until no more terms can be removed.
- Simplification of tautological formulas: If the step formula is of the form $\varphi_1 \vee \varphi_2 \vee \cdots \vee \varphi_n$ with $\varphi_i = \bar{\varphi}_j$ for some $i \neq j$, then the formula is replaced by $\top$.
- Reorienting equalities: veriT applies the symmetry of equality implicitly.

2.2 Concrete Syntax

The concrete text representation of the proofs generated by veriT is based on the SMT-LIB standard. Figure 1 shows an exemplary proof as printed by veriT lightly edited for readability.

We also reworked the proof syntax. Our goal is to follow the SMT-LIB standard when possible. While those modifications do not aid reconstruction inside Isabelle/HOL, they will simplify further development of the proof output. Previously, veriT produced nested steps. This was changed to a flat list of commands. The arguments of the commands are now given as annotations instead of a flat list. Since the changes are syntactical, the old format is still supported by veriT and can be selected using a command-line switch[2].

Figure 2 shows the grammar of the proof text generated by veriT. It is based on the SMT-LIB grammar, as defined in the SMT-LIB standard version 2.6 Appendix B[3]. The nonterminals ⟨symbol⟩,

[2]The option `--proof-version=N`, where `N` is either `1`, `2`, or `3`.

[3]Available online at: `http://smtlib.cs.uiowa.edu/language.shtml`

```
⟨proof⟩ ::= ⟨proof_command⟩*
⟨proof_command⟩ ::= (assume ⟨symbol⟩ ⟨proof_term⟩ )
  | (step ⟨symbol⟩ ⟨clause⟩ :rule ⟨symbol⟩ ⟨step_annotation⟩ )
  | (anchor :step ⟨symbol⟩ )
  | (anchor :step ⟨symbol⟩ :args ⟨proof_args⟩ )
  | (define-fun ⟨function_def⟩ )
⟨clause⟩ ::= (cl ⟨proof_term⟩* )
⟨step_annotation⟩ ::= :premises ( ⟨symbol⟩+ )
  | :args ⟨proof_args⟩
  | :premises ( ⟨symbol⟩+ ) :args ⟨proof_args⟩
⟨proof_args⟩ ::= ( ⟨proof_arg⟩+ )
⟨proof_arg⟩ ::= ⟨symbol⟩ | ( ⟨symbol⟩ ⟨proof_term⟩ )
⟨proof_term⟩ ::= ⟨term⟩ extended with (choice ( ⟨sorted_var⟩+ ) ⟨proof_term⟩ )
```

Figure 2: The proof grammar

⟨function_def⟩, ⟨sorted_var⟩, and ⟨term⟩ are as defined in the standard. The ⟨proof_term⟩ is the recursive ⟨term⟩ nonterminal redefined with the additional production for the choice binder.

Input problems in the SMT-LIB standard contain a list of *commands* that modify the internal state of the solver. In agreement with this approach veriT's proofs are also formed by a list of commands. The assume command introduces a new assumption. While this command could also be expressed using the step command with a special rule, the special semantic of an assumption justifies the presence of a dedicated command: assumptions are neither tautological nor derived from premises. The step command, on the other hand, introduces a derived or tautological term. Both commands assume and step require an index as the first argument to later refer back to it. This index is an arbitrary SMT-LIB symbol. The only restriction is that it must be unique for each assume and step command. The second argument is the term introduced by the command. For a step, this term is always a clause. To express disjunctions in SMT-LIB the or operator is used. Unfortunately, this operator needs at least two arguments and cannot represent unary or empty clauses. To circumvent this we introduce a new cl operator. It corresponds the standard or function extended to one argument, where it is equal to the identity, and zero arguments, where it is equal to false. The :premises annotation denotes the premises and is skipped if they are none. If the rule carries arguments, the :args annotation is used to denote them.

The anchor and define-fun commands are used for subproofs and sharing, respectively. The define-fun command corresponds exactly to the define-fun command of the SMT-LIB language.

2.3 Subproofs

As the name suggests, the SUBPROOF rule expresses subproofs. This is possible because its application is restricted: the assumption used as premise for the SUBPROOF step must be the assumption introduced last. Hence, the ASSUME, SUBPROOF pairs are nested. The context is manipulated in the same way: if a step pops $c_1, \ldots, c_n$ from a context Γ, there is a earlier step which pushes precisely $c_1, \ldots, c_n$ onto the context. Since contexts can only be manipulated by push and pop, context manipulations are also nested.

Because of this nesting, veriT uses the concept of subproofs. A subproof is started right before an ASSUME command or a command which pushes onto the context. We call this point the *anchor*. The subproof ends with the matching SUBPROOF command or command which pops from the context, respectively. The `:step` annotation of the anchor command is used to indicate the `step` command which will end the subproof. The term of this `step` command is the conclusion of the subproof. If the subproof uses a context, the `:args` annotation of the `anchor` command indicates the arguments added to the context for this subproof. In the example proof (Figure 1) a subproof starts on line four. It ends on line seven with the BIND steps which finished the proof for the renaming of the bound variable `z2` to `vr4`.

A further restriction applies: only the conclusion of a subproof can be used as a premise outside of the subproof. Hence, a proof checking tool can remove the steps of the subproof from memory after checking it.

2.4 Sharing and Skolem Terms

The proof output generated by veriT is generally large. One reason for this is that veriT can store terms internally much more efficiently. By utilizing a perfect sharing data structure, every term is stored in memory precisely once. When printing the proof this compact storage is unfolded.

The user of veriT can optionally activate sharing[4] to print common subterms only once. This is realized using the standard naming mechanism of SMT-LIB. In the language of SMT-LIB it is possible to annotate every term t with a name n by writing `(! t :named n )` where n is a symbol. After a term is annotated with a name, the name can be used in place of the term. This is a purely syntactical replacement.

To limit the number of names introduced we use a simple approach: before printing the proof we iterate over all terms of the proof and recursively descend into the terms. We mark every unmarked subterm we visit. If we visit a marked term, this term gets a name. If a term already has a name, we do not descend further into this term. By doing so, we ensure that only terms that appear as child of two different parent terms get a name. Thanks to the perfect sharing representation testing if a term is marked takes constant time and the overall traversal takes linear time in the proof size.

To simplify reconstruction veriT can optionally[5] define Skolem constants as functions. If activated, this option adds a list of `define-fun` command to define shorthand 0-ary functions for the `(choice ...)` terms needed. Without this option, no `define-fun` commands are issued and the constants are inlined.

3 Proof Reconstruction in Isabelle/HOL

Proof reconstruction is done in Isabelle in two steps presented in Figure 3: first, the proof is parsed and the terms are transformed into Isabelle terms (Section 3.1). Then we can reconstruct the proof itself by reconstructing the steps one-by-one. Some of the steps require some care (Section 3.2).

3.1 Parsing and Preprocessing

Parsing the proof is simple thanks to the infrastructure developed to reconstruct Z3 proofs for the `smt` tactic. This infrastructure is able to parse a generalized version of the SMT-LIB syntax, including the proofs generated by veriT. It produces a raw version of the proof. We only have to extract the structure

[4] By using the command-line option `--proof-with-sharing`.
[5] By using the command-line option `--proof-define-skolems`.

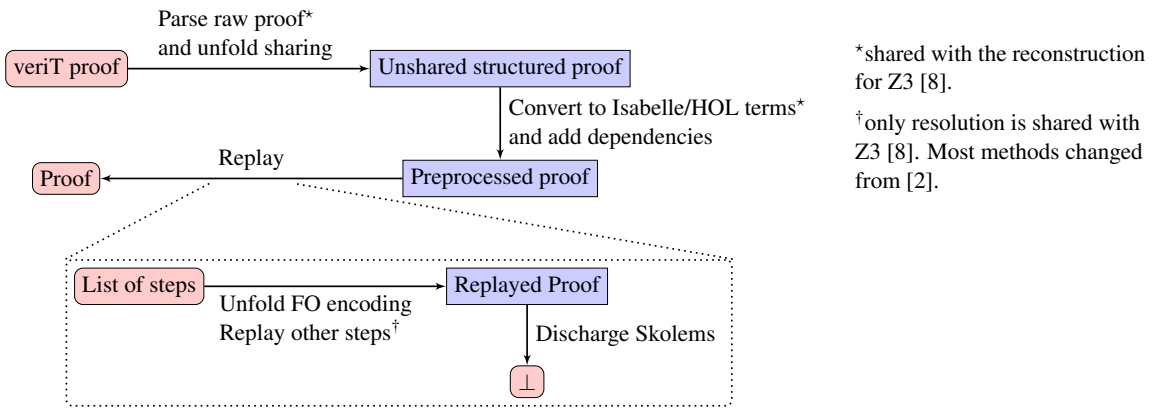

Figure 3: The reconstruction pipeline

(indices, steps, ...) from the raw proof. During parsing of the raw proof we also unfold the sharing, because Isabelle does not offer any sharing functionality.

The first transformation is a change of the disjunction representation. In the proof output, veriT represents the outermost disjunction as a multiset by using the `cl` operator. In Isabelle, we replace this multiset by a disjunction. veriT explicitly applies the rule OR to convert disjunctions to multisets. In Isabelle, these steps are simply the identity.

After that, we convert the SMT-LIB terms to Isabelle terms. This reuses again some of the infrastructure developed for Z3. An important difference with Z3 is the declaration of variables. When converting to Isabelle terms, types are inferred. However, some expressions like $x \mapsto y$ of the context cannot be typed without extracting the types from the conclusion.

Finally, we preprocess the proofs to ease the reconstruction further:

- We add the implicit dependency between the last step of each subproof and its conclusion. In the example of Figure 1, it is the dependency from `t9` to `t9.t2`. Spelling it out explicitly makes the reconstruction more regular.
- We add missing dependencies to the definitions of Skolem terms: veriT applies definitions implicitly, but we have to unfold the definitions explicitly to reconstruct Skolemization steps in Isabelle.

The Isabelle semantics of the proof steps are slightly different than the semantics in veriT. First, the context is seen as a list of equalities instead of a list of mappings. Second, the conclusion uses the equality symbol instead of $\simeq$ and a substitution. If the proof step is $y \mapsto z, x \mapsto s \ \triangleright \ n.\ \varphi \simeq \psi$, Isabelle sees it as: $y = z, x = s \ \blacktriangleright \ n.\ \varphi = \psi$, where the context are assumptions. The advantage of this different semantics is that it requires fewer transformations on the input term, as it avoids adding lambda abstractions, and makes the Isabelle tactics easier to use for reconstructions.

The difference in the semantics is small and rarely important. They only differ for variables that can syntactically appear on the left and on the right-hand side with different semantics. For example, consider $y \mapsto z, x \mapsto y \ \triangleright \ n.\ P\,x\,y \simeq P\,y\,z$. This is a tautology, because $P\,x\,y$ is $P\,y\,z$ after substitution (remember that the substitution only applies on the left-hand side). However, the naive conversion to the Isabelle version yields $y = z, x = y \ \blacktriangleright \ n.\ P\,x\,y \simeq P\,y\,z$, which is a different term, namely $P\,z\,z = P\,z\,z$

To avoid the difference in semantics, we rename terms when they have already been bound: we rename the occurrences on the right-hand side of $\simeq$ of y by the new fresh name xy. The step $y \mapsto z, x \mapsto y \ \triangleright \ n.\ \varphi \simeq \psi$ becomes $y = z, x = xy \ \blacktriangleright \ n.\ \varphi = \psi[xy/x]$.

3.2 Reconstructing Parsed Proofs

After parsing, we reconstruct the proof steps in Isabelle. Overall, the proof reconstruction works by replaying each step and unifying the assumptions with the premises. At the end, we get a proof of $\bot$.

For most steps, the rule can be spelled out as an Isabelle theorem and the only issues are implicit steps (Section 3.2.1). Unlike the reconstruction of Z3 proofs, we reconstruct subproofs as they are printed by veriT (Section 3.2.2). Finally, some rules require special care or are tricky to reconstruct: Skolemization steps, if done naively, can produce terms that are too large to be handled efficiently (Section 3.2.3); Isabelle's arithmetic procedure is incomplete and not very efficient when reconstructing arithmetic steps (Section 3.2.4); for efficiency, some rules are reconstructed heuristically (Section 3.2.5).

3.2.1 Application of Theorems

Most rules that can be applied are either tautologies or applications of theorems that can be easily expressed: The rule TRUE is the tautology used to prove that the theorem $\top$ holds. Similarly, the transitivity rule EQ_TRANSITIVE transforms the assumptions $(t_j \simeq t_{j+1})_{j<n}$ into $t_0 \simeq t_n$.

In practice, there are two main difficulties: double negations can be simplified and equalities can be reoriented. The reorientation is implementation dependent, which prohibits us from relying on the order as given by the input problem. The reordering and simplification have consequences that either make reconstruction harder or require additional annotations in the proof output:

- Duplicate literals are implicitly removed. This is rarely an issue in practice, but we have seen this happening in some test cases like the ITE2 rule. This rule introduces the tautology $(\mathit{if}\ \neg\varphi\ \mathit{then}\ \psi_1\ \mathit{else}\ \psi_2) \vee \varphi \vee \psi_2$, but if $\varphi = \psi_2$ the it produces the simplified clause $(\mathit{if}\ \neg\varphi\ \mathit{then}\ \psi_1\ \mathit{else}\ \psi_2) \vee \varphi$.
- The rules can be applied up to additional negations. For example, the ITE2 rule can be applied to get $(\mathit{if}\ \varphi\ \mathit{then}\ \psi_1\ \mathit{else}\ \psi_2) \vee (\neg\varphi) \vee \psi_2$.

The individual steps are reconstructed by:

- taking into account the additional information provided in the proof output. This can require some preprocessing on the formula: in veriT instantiation (rule FORALL_INST) can be done to quantifiers that do not appear at the outermost level, but inside the formula. Preprocessing is used to transform $\forall x.(P \implies \neg(\exists y.Qy))$ into $\forall xy.(P \implies Qy)$. This is easier to reconstruct, because all quantifiers to instantiate are now at the outmost level and forall quantifiers.
- applying the theorem or finding the instantiations and then using `simp` to reorder the equalities and prove that the terms are equal. For the ITE2 on $(\mathit{if}\ \neg\varphi\ \mathit{then}\ \psi_1\ \mathit{else}\ \psi_2) \vee \varphi' \vee \psi_2'$, we identify the terms, φ, ψ_1, and ψ_2, and generate the tautology $(\mathit{if}\ \neg\varphi\ \mathit{then}\ \psi_1\ \mathit{else}\ \psi_2) \vee \varphi \vee \psi_2$, that can be used by `simp` to discharge the goal by showing $\varphi = \varphi'$ and $\psi_2 = \psi_2'$. The search space is very large and the search can be very time consuming during the reconstruction.
- providing various version of the lemmas to accommodate negations: for Isabelle, the theorem $(\mathit{if}\ \neg\varphi\ \mathit{then}\ \psi_1\ \mathit{else}\ \psi_2) \vee \varphi \vee \psi_2$ cannot be applied to prove $(\mathit{if}\ \varphi\ \mathit{then}\ \psi_1\ \mathit{else}\ \psi_2) \vee \neg\varphi \vee \psi_2$.

In practice the reordering happens mostly when producing new terms (during parsing or instantiation). However, we do not want to rely on this specific behavior which could change in a future version.

3.2.2 Subproofs

Unlike Z3, veriT has subproofs. Subproofs fall into two categories: proofs used to justify proof steps (e.g. for Skolemization) and lemmas with assumptions and fixed variables. In Isabelle, both are modeled by the notions of contexts that encapsulate all the assumptions and fixed variables present at a given point.

The first kind of subproofs are proofs of *lemmas* that come with additional assumptions. They are used for example for proofs like $P \Longrightarrow \bot$. P is an assumption of the proof (given by an ASSUME command) and $\bot$ is the conclusion. In Isabelle, we start by extracting all the assumptions when entering the proof. This creates a new context. Then, we replay the proof in the new context. The ASSUME commands are now entailed by the context are replayed as such. Finally, the conclusion is exported to the outer context.

Replaying *subproofs* is similar to replaying lemmas in the proof: we enter contexts with new assumptions and variables, depending on the rule. At the end of the subproof the last step is exported back to the outer context and is used to discharge the conclusion. For example, the subproof of a simple BIND step will be of the form $\forall x\, y.\, (x = y \Longrightarrow P\, x = Q\, y)$ to prove that $(\forall x.\, P\, x) = (\forall y.\, Q\, y)$

3.2.3 Skolemizations

Skolemization is an important but subtle point, which slightly differs between Isabelle and veriT. While defining the constants is easy, the definitions themselves do not exactly match the natural ones and reconstructing the proof can be difficult.

Technically, Skolem constants are not introduced with a definition, but as an assumption of the form $\overline{X} = (\varepsilon x.\, \ldots)$. At the end of the reconstruction, we get the theorem $\forall \overline{X}.\, (\overline{X} = (\varepsilon x.\, \neg P\, x) \Longrightarrow \bot)$, from which we can trivially derive the theorem $\bot$.

Internally, veriT directly Skolemizes formulas: The term $\forall x\, y.\, P\, x\, y$ becomes after Skolemization $P\, \overline{X}\, \overline{Y}$, i.e., $P\ (\varepsilon x.\, \neg(\forall y.\, P\, x\, y))\ (\varepsilon y.\, \neg P\ (\varepsilon x.\, \neg(\forall y.\, P\, x\, y)))$, where $\overline{X}$ and $\overline{Y}$ are defined to be the two Skolem constants. However, in the logged proof, $\forall x\, y.\, P\, x\, y$ becomes $\forall y.\, P\, \overline{X}\, y$, i.e., $\forall y.\, P\ (\varepsilon x.\, \neg(\forall y.\, P\, x\, y))\ y$, which in turns naturally becomes $P\, \overline{X}\ \left(\varepsilon y.\, \neg P\, \overline{X}\, y\right)$. Therefore, in Isabelle, we fold the definition of the Skolems inside each other to get $\overline{Y} = \left(\varepsilon y.\, \neg P\, \overline{X}\, y\right)$ and try to prove the goal. This might, however, fail due to the implicit steps. Hence, if required, we unfold all definitions and prove the result. This could explode for non-trivial terms, but we did not have issues with this during our experiments.

A major issue of the reconstruction is the size of generated terms. While developing the reconstruction, we found a case where four variables were Skolemized in a single step, and the generated term was so big that Isabelle was not able to replace the third variable by the equivalent choice: the application of the theorem $(\forall x.\, P\, x) \Longleftrightarrow P\, (\varepsilon x.\, \neg P\, x)$ was too slow. We now aggressively fold the Skolem constants inside the term.

3.2.4 Arithmetic

To replay arithmetic steps, we use Isabelle's procedure `linarith`. This tactic is a decision procedure for real numbers, but not for integers or natural numbers. Internally, it uses the Fourier–Motzkin elimination [17]: it derives a contradiction via a linear combination of the equations.

veriT with proof production only supports linear arithmetic. On linear problems, however, it is stronger than Isabelle's tactic: Isabelle does not simplify equations. If we have the equations $5 \times x + 10 \times y \simeq 15$, it will not be simplified to $x + 2 \times y \simeq 3$ in Isabelle. This happens neither as preprocessing, nor during the search for the linear combination. In one case over the Archive of Formal Proofs, this makes the following problem impossible to reconstruct:

$$\begin{array}{cccccccccccc} \neg & 0 & \leq & y & \wedge & \neg & 10 \times x & < & 4 + 14 \times z & \wedge \\ & 10 \times x & \leq & 15 + 25 \times y & \wedge & \neg & 10 \times x + 10 \times z & \leq & 30 + 25 \times y & \end{array}$$

This goal is produced as an arithmetic tautology by veriT, but `linarith` is not able to prove it. Before simplification, the inequality $16 \leq 10 \times x - 25 \times y$ is derived. After simplification, the equivalent (but

SMT calls	Number of occurrences
Successful reconstruction	447
Failed reconstruction	4
veriT timeouts	47
veriT unknown	4

Table 1: Result of using veriT instead of Z3 in existing `smt` calls

seemingly stronger) inequality $20 \leq 10 \times x - 25 \times y$ is derived because x and y are integers. The coefficients of the second inequality are different enough to allow `linarith` to find a contradiction, which it was unable to find otherwise.

We strengthened the reconstruction by implementing a simplification procedure that divides each equation by its greatest common divisor. It could be activated more globally, but currently conflicts with two other simplification procedures: one of them sorts terms, while the other does not.

3.2.5 Other Rules

The reconstruction of the rules is often guided by the efficiency of the reconstruction, how often a rule is used, and concrete examples. One of the most prominent rules is CONNECTIVE_EQUIV. It is a simplification step and can involve simplifications of the Boolean structure and arithmetic. At first, we reconstructed CONNECTIVE_EQUIV steps with `auto`, a tactic that simplifies the terms and performs some logical reasoning. However, this turned out to be too inefficient on large terms. Moreover, often only the Boolean structure is modified and not the terms or the order of equalities. Therefore, we now first abstract over the non-Boolean terms and check only the modifications on the Boolean structure by `fast`. Only if this fails is `auto` tried. If that also fails, `metis` is tried as a fallback tactic. We do not attempt to select the right tactic, but simply try them in this order.

4 Experimental Results

We experiment on the Isabelle reconstruction in two ways. The first one is to replace all the `smt` calls that are in the Isabelle distribution and are currently powered by Z3 by the version of `smt` with veriT. These `smt` calls have been selected by the developer of the library who provided the theory, because Z3 is able to find a proof and the reconstruction is fast. While this experiment provides insight into the performance of the veriT-powered `smt` tactic relative to the Z3-powered variant, it does not tell us if the veriT-powered one is a useful and supplementary addition to the family of automated tactics provided by Isabelle. Towards that end, we try to generate new veriT-powered `smt` calls by using Sledgehammer [7], an Isabelle tool able to find proofs.

4.1 Replacing the `smt` calls

There are already many `smt` calls in the theories included in the Isabelle distribution and the Archive of Formal Proofs. The latter did not allow `smt` calls until a few years ago. We replaced the Z3 as a backend for the `smt` calls, by veriT. The results are summarized in Table 1. Testing revealed:

- that veriT is not able to find all the proofs that Z3 is able to find. On the one hand, this does not corroborate the findings from the SMT competition where veriT performs better than Z3 on some

categories. On the other hand, the problems have been specifically selected to be solvable by Z3. We did not include the problems specifically relying on Z3 extensions (e.g. the division operator) or features not supported by veriT (bit vectors).

- a bug in the proof generation. veriT does not correctly print some substeps: a term is replaced by an equivalent term, but this replacement is not logged. We are currently fixing this bug in veriT. In Isabelle, this leads to an error in the reconstruction and we do not attempt to reconstruct the following steps.
- the problem in the reconstruction of arithmetic steps described in Section 3.2.4. Only one of those benchmarks could not be reconstructed without the simplification procedure.

The results are promising: we are able to reconstruct nearly all of the proofs that veriT is able to find. We cannot replace Z3 by veriT in the Isabelle distribution, but this was not the aim of this experiment.

4.2 Generating calls with Sledgehammer

Hammers, like Sledgehammer, select facts from the background theory, translate them to the input language of the provers, and then attempt to use the generated proof if a proof is found within a given timeout. The proof can be used in different ways. One approach is to gather the facts required to find a proof with one of the builtin tactics of the proof assistant. Another approach is to replay the proof within the core by an `smt`-like method or to translate it into the user-facing language of the assistant. Sledgehammer supports all of these approaches.

By default, Sledgehammer uses the following strategy: first, it tries several Isabelle tactics, including detailed proof reconstruction by the Z3-powered `smt` tactic, with a timeout of 1 s. If this is successful, it returns the tactic that was the fastest to prove the goal in Isabelle. This tactic can be inserted in the theory. If none of them is fast enough to find a proof, reconstruction of a proof in the user-facing language is attempted. We have changed Sledgehammer to additionally test the veriT-powered `smt` tactic. Sledgehammer tries to find a proof and to minimize the involved facts. Even when Z3 finds the proof, reconstruction with veriT can be faster.

We tested this approach on two formalizations: an ordered resolution prover [15, 16] and the SSA language [10, 19]. We tested all theories included in the formalization. We selected the first development because we knew that Sledgehammer was useful during the development. We selected the second formalization because it is a very different theme. Due to time constraints, we did not test more theories.

The results are given in Table 2. They show that veriT-powered `smt` calls happen in practice and can improve the speed of the overall proof processing. We do not know why veriT performs much worse on the formal SSA theory, but we believe that some rules that we do not reconstruct efficiently enough (possibly the QNT_SIMPLIFY rule that simplifies quantifiers) appear more often in this theory. The row 'Oracle' denotes calls to solvers that found a proof that could not be reconstructed. Many of these failed calls are proofs found by the SMT solver CVC4 that can either not be found or not be reconstructed by veriT and Z3-powered `smt`.

5 Related Work

Reconstruction of proofs generated by external theorem provers has been implemented in various systems including CVC in HOL Light [13], Z3 in HOL4 and Isabelle/HOL [8], and SMTCoq reconstructs veriT [1] and CVC4 [12] proofs in Coq. None of the other solvers produce detailed proofs or information on Skolemization. For veriT proofs, SMTCoq currently supports a different version of the proof output

Theory	Ordered Resolution Prover	Formal SSA
Found proofs	5019	5961
Z3-powered `smt` proofs	90	109
veriT-powered `smt` proofs	25	4
Oracle	9	63

Table 2: Proofs found by Sledgehammer on two Isabelle formalizations

(version 1) that has different rules and an older version of veriT (the version is from 2016), which does not record detailed information for Skolemization and has worse performance.

The reconstruction of Z3 proofs in HOL4 and Isabelle/HOL is one of the most advanced and well tested. It has been used to check proofs generated on problems from the SMT competition. Sadly, the code to read the SMT-LIB input problems was never included in the standard Isabelle distribution and is now lost. Proof reconstruction has been heavily tested and succeeds in more than 90% of the cases according to Sledgehammer benchmark [7, Section 9], and is very efficient.

The SMT solver CVC4 follows a different philosophy from veriT and Z3: it produces proofs in a logical framework with side conditions [18]. The output can contain programs to check certain rules. The CVC4 proof format is quite flexible but currently CVC4 does not produce proofs for quantifiers.

6 Conclusion and Future Work

We presented the syntax and semantics of the proofs generated by veriT and the reconstruction of those proofs in Isabelle. During the development, the format was extended to ease reconstruction by printing more information like the instantiations. We hope to integrate our code in the next Isabelle release.

Overall, having more details in the proofs helps to make the reconstruction more robust, because each step is simpler to check. For example, veriT detailed information on Skolemization, makes it easier to replay than the one from Z3: the reconstruction can call the ordered resolution prover `metis`. For now, the implicit simplifications prevents us from reconstructing proof more efficiently than Z3.

Another challenge is to translate the proof to the more readable Isar format. It is useful for two main reasons. First, it gives the Isabelle user more information on how the proof works and potentially what kind of lemmas would be interesting to create. Second, if the reconstruction fails, it allows the user to fix the failing part. Generating readable proofs can be done automatically by Sledgehammer for most solvers, but this does not work for veriT proofs. One reason is that, Sledgehammer does not support subproofs and inlining the assumptions each time is not very readable. Another reason is that the Skolems constants implicitly introduce a context where these constants are defined. This introduces an implicit dependency order between the definitions and every step where the defined constant appears. We could unfold the definitions to use the choice version instead, but that would harm the readability of the proof. Finally, the proofs generated by veriT often follow the scheme "φ holds; $\varphi \leftrightarrow \psi$ also holds; hence ψ holds", whereas "φ hence ψ" is easier to understand.

There are various useful pieces of information that are found by the solver but are not presented to the user. For example, in the case of linear arithmetic a contradiction is derived by finding a linear combination of the equations, but the coefficients are not printed. Therefore, Isabelle must find these same coefficients again. The reconstruction would be faster if they were in the proof output.

Acknowledgments. We thank Alex Brick, Daniel El Ouraoui, and Pascal Fontaine for suggesting many textual improvements. The work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka). Previous experiments were carried out using the Grid'5000 testbed (`https://www.grid5000.fr/`), supported by a scientific interest group hosted by Inria and including CNRS, RENATER, and several universities as well as other organizations.

References

[1] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry & Benjamin Werner (2011): *A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses*. In Jean-Pierre Jouannaud & Zhong Shao, editors: *CPP 2011*, *LNCS* 7086, Springer, pp. 135–150, doi:10.1007/978-3-642-25379-9_12.

[2] Haniel Barbosa, Jasmin C. Blanchette, Mathias Fleury & Pascal Fontaine (2019): *Scalable Fine-Grained Proofs for Formula Processing*. *Journal of Automated Reasoning*, doi:10.1007/s10817-018-09502-y.

[3] Haniel Barbosa, Jasmin C. Blanchette, Mathias Fleury, Pascal Fontaine & Hans-Jörg Schurr (2019): *Better SMT proofs for easier reconstruction*. In Thomas C. Hales, Cezary Kaliszyk, Ramana Kumar, Stephan Schulz & Josef Urban, editors: *AITP 2019*.

[4] Clark Barrett, Pascal Fontaine & Cesare Tinelli (2016): *The Satisfiability Modulo Theories Library (SMT-LIB)*. `www.SMT-LIB.org`.

[5] Clark Barrett, Roberto Sebastiani, Sanjit Seshia & Cesare Tinelli (2009): *Satisfiability Modulo Theories*. In Armin Biere, Marijn J. H. Heule, Hans van Maaren & Toby Walsh, editors: *Handbook of Satisfiability*, chapter 26, *Frontiers in Artificial Intelligence and Applications* 185, IOS Press, pp. 825–885.

[6] Frédéric Besson, Pascal Fontaine & Laurent Théry (2011): *A Flexible Proof Format for SMT: A Proposal*. In Pascal Fontaine & Aaron Stump, editors: *PxTP 2011*, pp. 15–26. Available at `https://hal.inria.fr/hal-00642544/`.

[7] Jasmin C. Blanchette, Sascha Böhme, Mathias Fleury, Steffen J. Smolka & Albert Steckermeier (2016): *Semi-intelligible Isar Proofs from Machine-Generated Proofs*. *Journal of Automated Reasoning* 56(2), pp. 155–200, doi:10.1007/s10817-015-9335-3.

[8] Sascha Böhme & Tjark Weber (2010): *Fast LCF-Style Proof Reconstruction for Z3*. In Matt Kaufmann & Lawrence C. Paulson, editors: *ITP 2010*, *LNCS* 6172, Springer, pp. 179–194, doi:10.1007/978-3-642-14052-5_14.

[9] Thomas Bouton, Diego C. B. de Oliveira, David Déharbe & Pascal Fontaine (2009): *veriT: An Open, Trustable and Efficient SMT-solver*. In Renate A. Schmidt, editor: *CADE 2009*, *LNCS* 5663, Springer, pp. 151–156, doi:10.1007/978-3-642-02959-2_12.

[10] Sebastian Buchwald, Denis Lohner & Sebastian Ullrich (2016): *Verified construction of static single assignment form*. In: *CC*, ACM, pp. 67–76, doi:10.1145/2892208.2892211.

[11] David Déharbe, Pascal Fontaine & Bruno Woltzenlogel Paleo (2011): *Quantifier Inference Rules for SMT Proofs*. In Pascal Fontaine & Aaron Stump, editors: *PxTP 2011*, pp. 33–39. Available at `https://hal.inria.fr/hal-00642535`.

[12] Burak Ekici, Guy Katz, Chantal Keller, Alain Mebsout, Andrew J. Reynolds & Cesare Tinelli (2016): *Extending SMTCoq, a Certified Checker for SMT (Extended Abstract)*. In Jasmin C. Blanchette & Cezary Kaliszyk, editors: *HaTT 2016*, *EPTCS* 210, pp. 21–29, doi:10.4204/EPTCS.210.5.

[13] Sean McLaughlin, Clark Barrett & Yeting Ge (2006): *Cooperating Theorem Provers: A Case Study Combining HOL-Light and CVC Lite*. *Electronic Notes in Theoretical Computer Science* 144(2), pp. 43–51, doi:10.1016/j.entcs.2005.12.005.

[14] Leonardo de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In C. R. Ramakrishnan & Jakob Rehof, editors: *TACAS 2008*, *LNCS* 4963, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[15] Anders Schlichtkrull, Jasmin C. Blanchette, Dmitriy Traytel & Uwe Waldmann (2018): *Formalization of Bachmair and Ganzinger's Ordered Resolution Prover*. Archive of Formal Proofs. `http://isa-afp.org/entries/Ordered_Resolution_Prover.html`, Formal proof development.

[16] Anders Schlichtkrull, Jasmin C. Blanchette, Dmitriy Traytel & Uwe Waldmann (2018): *Formalizing Bachmair and Ganzinger's Ordered Resolution Prover*. In: *IJCAR*, *LNCS* 10900, Springer, pp. 89–107, doi:10.1007/978-3-319-94205-6_7.

[17] Alexander Schrijver (1999): *Theory of Linear and Integer Programming*. Wiley - Interscience Series in Discrete Mathematics and Optimization, Wiley.

[18] Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean & Cesare Tinelli (2013): *SMT Proof Checking Using a Logical Framework*. Formal Methods in System Design 42(1), pp. 91–118, doi:10.1007/s10703-012-0163-3.

[19] Sebastian Ullrich & Denis Lohner (2016): *Verified Construction of Static Single Assignment Form*. Archive of Formal Proofs. `http://isa-afp.org/entries/Formal_SSA.html`, Formal proof development.

A List of Proof Rules

Rule	Description
TRUE, FALSE, AND_POS, AND_NEG, OR_POS, OR_NEG, IMPLIES_POS, IMPLIES_NEG1, IMPLIES_NEG2, EQUIV_POS1, EQUIV_POS2, EQUIV_NEG1, EQUIV_NEG2, ITE_POS1, ITE_POS2, ITE_NEG1, ITE_NEG2, EQ_REFLEXIVE, REFL,TRANS,CONG, NOT_OR, IMPLIES, NOT_IMPLIES1, NOT_IMPLIES2, EQUIV1,EQUIV2, NOT_EQUIV1, NOT_EQUIV2, ITE1, ITE2, NOT_ITE1, NOT_ITE2, ITE_INTRO	Simple rules without premises
OR, BIND, EQ_TRANSITIVE	Simple rules with premises
EQ_CONGRUENT, EQ_CONGRUENT_PRED	Congruence. Reconstruction can be problematic due to the FO encoding.
LA_RW_EQ, LA_GENERIC, LIA_GENERIC, LA_DISEQUALITY, LA_TOTALITY, LA_TAUTOLOGY	Linear arithmetics
FORALL_INST	Variable instantiation
TH_RESOLUTION, RESOLUTION	Resolution reconstructed with a simple SAT solver in Isabelle
CONNECTIVE_EQUIV	Arithmetics and logic simplification
TMP_AC_SIMP	Simplification modulo associativity and commutativity
SUBPROOF	Implication from assumption
SKO_EX, SKO_FORALL	Skolemization
QNT_SIMPLIFY, QNT_JOIN, QNT_RM_UNUSED, TMP_BFUN_ELIM	Quantifier simplification
LET, XOR1, XOR2, NOT_XOR1, NOT_XOR2, XOR_POS1, XOR_POS2, XOR_NEG1, XOR_NEG2, DISTINCT_ELIM	Unused: Isabelle does not generates XOR or lets
NLA_GENERIC, TMP_SKOLEMIZE	Unused: experimental features