Proofonomicon

A Reference of the veriT Proof Format

The veriT Team and Contributors

June 23, 2022

Contents

Version 2021.06-40-gc20206d-rmx

1 Introduction

This document is a reference manual for the format used by the SMT solver veriT to print proofs of the unsatisfiability of the input problem. It is part of the veriT repository and represents the format as generated by the corresponding source code. Currently the introduction gives a general overview of the proof format and lacks a complete formal definition of the calculus. Section 5 gives a complete lists of proof rules currently used by veriT. This section is probably the one of greatest utility to the reader.

1.1 Overview

veriT [4] 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.

Similar to the proofs generated by Z3, veriT's proofs are based on the SMT-LIB language, but are otherwise different. Proofs are not terms, but a list of indexed steps. 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, veriT uses term sharing. This 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.

In addition to this reference, the proof format used by veriT has been discussed in publications which provide valuable background information: the fundamental ideas behind the proof format have been published in [3]; proposed rules for quantifier instantiation can be found in [5]; and the proof rules to express reasoning typically used for processing, such as Skolemization, renaming of variables, and other manipulations of bound variables have been published in [1]. A complete reconstruction of the proofs generated by veriT in Isabelle/HOL has recently been reported in [6]. Parts of this manual have been taken from this publication.

2 Notation

veriT uses the SMT-LIB language [2] as both its input and output language. Hence, this document builds on the concept introduced there. This includes the concrete syntax and the multi-sorted first-order logic.

The notation used throughout this manual follows the notation of the SMT-LIB standard. To simplify the notation we will omit the sort of terms when possible. 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. 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 3. 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 section 5 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:

$$c_1, \ldots, c_l \rhd \mathfrak{i}.\ \ \varphi\ \ (\texttt{rule};\ p_1, \ldots, p_n;\ a_1, \ldots, a_m)$$

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 $\epsilon x.\varphi$ stands for a value v, such that $\varphi[v/x]$ is true if such a value exists. Any value is possible otherwise. Thorough this document, we use $\mathfrak{i}, \mathfrak{j}, \mathfrak{k}, \mathfrak{l}, \mathfrak{m}, \mathfrak{n}$ for step indices.

3 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. 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 $(\neg\forall x_1 \dots x_n.\varphi) \vee \varphi[t_1/x_1] \dots [t_n/x_n]$, 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 $x_1 \coloneqq t_1, \dots, x_n \coloneqq t_n$. While this information can be recovered from the term, providing this information explicitly aids reconstruction because the implicit reordering of equalities (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.

$$\begin{array}{rlll} \Gamma, x \mapsto (\epsilon x.\varphi) \rhd \mathfrak{n}. & \varphi \simeq \psi & & (\dots) \\ & \vdots & & \\ \Gamma \rhd \mathfrak{m}. & (\exists x.\varphi) \simeq \psi & (\texttt{sko\_ex}; \mathfrak{n}) & \end{array}$$

Example 1. To illustrate how such a rule is applied, consider the following example taken from [1]. Here the term $\neg p(\epsilon x.\neg p(x))$ 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 $\epsilon x.\neg\varphi$.

$$\begin{array}{lll} x \mapsto (\epsilon x.\neg p(x)) & \rhd 1. & x \simeq \epsilon x.\neg p(x) & (\texttt{refl}) \\ x \mapsto (\epsilon x.\neg p(x)) & \rhd 2. & p(x) \simeq p(\epsilon x.\neg p(x)) & (\texttt{cong};\ 1) \\ & \rhd 3. & (\forall x.p(x)) \simeq p(\epsilon x.\neg p(x)) & (\texttt{sko\_forall};\ 2) \\ & \rhd 4. & (\neg\forall x.p(x)) \simeq \neg p(\epsilon x.\neg p(x)) & (\texttt{cong};\ 3) \end{array}$$

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 $(\neg\varphi_1) \vee (\neg\varphi_2) \vee \cdots \vee (\neg\varphi_n)$ 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 $\varphi_1, \varphi_2, \ldots, \varphi_n$.

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 $0 \simeq y$.

$$\begin{array}{lll} \rhd 1. & (3 < x) \vee (x + y < 1) & (\texttt{assume}) \\ \rhd 2. & x \simeq 2 & (\texttt{assume}) \\ \rhd 3. & 0 \simeq y & (\texttt{assume}) \\ \rhd 4. & \neg(3 < x) \vee \neg(x \simeq 2) & (\texttt{la\_generic};\ ;1.0, 1.0) \\ \rhd 5. & \neg(3 < x) & (\texttt{resolution}; 2, 4) \\ \rhd 6. & x + y < 1 & (\texttt{resolution}; 1, 5) \\ \rhd 7. & \neg(x + y < 1) \vee \neg(x \simeq 2) \vee \neg(0 \simeq y) & (\texttt{la\_generic};\ ;1.0, 1.0, 1.0) \\ \rhd 8. & \bot & (\texttt{resolution};\ 7, 6, 2, 3) \end{array}$$

Implicit transformations reordering of equalities. In addition to the explicit steps, veriT might reorder equalities, i.e. apply symmetry of the equality predicate, without generating steps. When this happens is somewhat restricted. Equalities are only reordered when the term below the equality change during proof search. One such case is the instantiation of universally quantified variables. If the variable that appears below an equality, then the equality might have an arbitrary order after the variable is instantiated.

4 The 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. The format follows the SMT-LIB standard when possible.

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[1]. The nonterminals ⟨`symbol`⟩, ⟨`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.

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

```
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))
15           :rule or :premises (t15))
16 (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)

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.

```
⟨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

4.1 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.

4.2 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[2] 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 simplify reconstruction veriT can optionally[3] 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.

5 List of Proof Rules

The following lists all rules produced by veriT. When n-ary operators are in the form $t_1 \vee \cdots \vee t_n$ this corresponds to the SMT-LIB string `(or t1 ... tn)`. Hence, we explicitly bracket to clarify the application. To differentiate between `or` from `cl`, we use $\dot{\vee}$ for the second operator. Nevertheless, since proof steps always start with `cl`, we write the literal of unit clauses directly. Furthermore, premises can be ordered arbitrarily and need not follow the order given in the rule definition.

1. `assume`

$$\triangleright \mathsf{i}. \qquad \phi \qquad (\texttt{assume})$$

where φ is equivalent to a formula asserted in the input problem.

2. `true`

$$\triangleright \mathsf{i}. \qquad \top \qquad (\texttt{true})$$

3. `false`

$$\triangleright \mathsf{i}. \qquad \neg\bot \qquad (\texttt{false})$$

4. `not_not`

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

$\triangleright$ i. $\neg(\neg\neg\varphi)\dot{\vee}\varphi$ (not_not)

5. and_pos

$\triangleright$ i. $\neg(\varphi_1 \wedge \cdots \wedge \varphi_n)\dot{\vee}\varphi_i$ (and_pos)
with $1 \leq i \leq n$.

6. and_neg

$\triangleright$ i. $(\varphi_1 \wedge \cdots \wedge \varphi_n)\dot{\vee}(\neg\varphi_1)\dot{\vee}\ldots\dot{\vee}(\neg\varphi_n)$ (and_neg)

7. or_pos

$\triangleright$ i. $\neg(\varphi_1 \vee \cdots \vee \varphi_n)\dot{\vee}\varphi_1\dot{\vee}\ldots\dot{\vee}\varphi_n$ (or_pos)

8. or_neg

$\triangleright$ i. $(\varphi_1 \vee \cdots \vee \varphi_n)\dot{\vee}(\neg\varphi_i)$ (or_neg)
with $1 \leq i \leq n$.

9. xor_pos1

$\triangleright$ i. $\neg(\varphi_1 \operatorname{xor} \varphi_2)\dot{\vee}\varphi_1\dot{\vee}\varphi_2$ (xor_pos1)

10. xor_pos2

$\triangleright$ i. $\neg(\varphi_1 \operatorname{xor} \varphi_2)\dot{\vee}(\neg\varphi_1)\dot{\vee}(\neg\varphi_2)$ (xor_pos2)

11. xor_neg1

$\triangleright$ i. $(\varphi_1 \operatorname{xor} \varphi_2)\dot{\vee}\varphi_1\dot{\vee}(\neg\varphi_2)$ (xor_neg1)

12. xor_neg2

$\triangleright$ i. $(\varphi_1 \operatorname{xor} \varphi_2)\dot{\vee}(\neg\varphi_1)\dot{\vee}\varphi_2$ (xor_neg2)

13. implies_pos

$\triangleright$ i. $\neg(\varphi_1 \to \varphi_2)\dot{\vee}(\neg\varphi_1)\dot{\vee}\varphi_2$ (implies_pos)

14. `implies_neg1`

▷ i. $(\varphi_1 \to \varphi_2)\dot{\vee}\varphi_1$ (`implies_neg1`)

15. `implies_neg2`

▷ i. $(\varphi_1 \to \varphi_2)\dot{\vee}(\neg\varphi_2)$ (`implies_neg2`)

16. `equiv_pos1`

▷ i. $\neg(\varphi_1 \leftrightarrow \varphi_2)\dot{\vee}\varphi_1\dot{\vee}(\neg\varphi_2)$ (`equiv_pos1`)

17. `equiv_pos2`

▷ i. $\neg(\varphi_1 \leftrightarrow \varphi_2)\dot{\vee}(\neg\varphi_1)\dot{\vee}\varphi_2$ (`equiv_pos2`)

18. `equiv_neg1`

▷ i. $\varphi_1 \leftrightarrow \varphi_2\dot{\vee}(\neg\varphi_1)\dot{\vee}(\neg\varphi_2)$ (`equiv_neg1`)

19. `equiv_neg2`

▷ i. $\varphi_1 \leftrightarrow \varphi_2\dot{\vee}\varphi_1\dot{\vee}\varphi_2$ (`equiv_neg2`)

20. `ite_pos1`

▷ i. $\neg(\mathrm{ite}\,\varphi_1\ \varphi_2\ \varphi_3)\dot{\vee}\varphi_1\dot{\vee}\varphi_3$ (`ite_pos1`)

21. `ite_pos2`

▷ i. $\neg(\mathrm{ite}\,\varphi_1\ \varphi_2\ \varphi_3)\dot{\vee}(\neg\varphi_1)\dot{\vee}\varphi_2$ (`ite_pos2`)

22. `ite_neg1`

▷ i. $\mathrm{ite}\,\varphi_1\ \varphi_2\ \varphi_3\dot{\vee}\varphi_1\dot{\vee}(\neg\varphi_3)$ (`ite_neg1`)

23. `ite_neg2`

▷ i. $\mathrm{ite}\,\varphi_1\ \varphi_2\ \varphi_3\dot{\vee}(\neg\varphi_1)\dot{\vee}(\neg\varphi_2)$ (`ite_neg2`)

24. `eq_reflexive`

$\triangleright$ i. $t \simeq t$ (`eq_reflexive`)

25. `eq_transitive`

$\triangleright$ i. $\neg(t_1 \simeq t_2)\dot{\vee} \dots \dot{\vee}\neg(t_{n-1} \simeq t_n)\dot{\vee} t_1 \simeq t_n$ (`eq_transitive`)

26. `eq_congruent`

$\triangleright$ i. $\neg(t_1 \simeq u_1)\dot{\vee} \dots \dot{\vee}\neg(t_n \simeq u_n)\dot{\vee} f(t_1, \dots, t_n) \simeq f(u_1, \dots, u_n)$ (`eq_congruent`)

27. `eq_congruent_pred`

$\triangleright$ i. $\neg(t_1 \simeq u_1)\dot{\vee} \dots \dot{\vee}\neg(t_n \simeq u_n)\dot{\vee} P(t_1, \dots, t_n) \simeq P(u_1, \dots, u_n)$ (`eq_congruent_pred`)

28. `distinct_elim`

This rule eliminates the $\mathrm{distinct}$ predicate. If called with one argument this predicate always holds:

$\triangleright$ i. $(\mathrm{distinct}\, t) \leftrightarrow \top$ (`distinct_elim`)

If applied to terms of type $\mathbf{Bool}$ more than two terms can never be distinct, hence only two cases are possible:

$\triangleright$ i. $(\mathrm{distinct}\, \varphi\ \psi) \leftrightarrow \neg(\varphi \leftrightarrow \psi)$ (`distinct_elim`)

and

$\triangleright$ i. $(\mathrm{distinct}\, \varphi_1\ \varphi_2\ \varphi_3 \dots) \leftrightarrow \bot$ (`distinct_elim`)

The general case is:

$\triangleright$ i. $(\mathrm{distinct}\, t_1 \dots t_n) \leftrightarrow \bigwedge_{i=1}^{n} \bigwedge_{j=i+1}^{n} t_i \not\simeq t_j$ (`distinct_elim`)

29. `la_rw_eq`

$\triangleright$ i. $(t \simeq u) \simeq (t \leq u \wedge u \leq t)$ (`la_rw_eq`)

Remark. While the connective could be an $\leftrightarrow$, currently an equality is used.

30. `la_generic`

A step of the `la_generic` rule represents a tautological clause of linear disequalities. It can be checked by showing that the conjunction of the negated disequalities is unsatisfiable. After

the application of some strengthening rules, the resulting conjunction is unsatisfiable, even if integer variables are assumed to be real variables.

A linear inequality is of term of the form $\sum_{i=0}^{n} c_i \times t_i + d_1 \bowtie \sum_{i=n+1}^{m} c_i \times t_i + d_2$ where $\bowtie \in \{=, <, >, \leq, \geq\}$, where $m \geq n$, c_i, d_1, d_2 are either integer or real constants, and for each i c_i and t_i have the same sort. We will write $s_1 \bowtie s_2$.

Let $l_1, \ldots, l_n$ be linear inequalities and $a_1, \ldots, a_n$ rational numbers, then a `la_generic` step has the form:

$$\rhd \mathsf{i}. \quad \varphi_1 \dot{\vee} \ldots \dot{\vee} \varphi_o \quad (\texttt{la\_generic}; ; a_1, \ldots, a_o)$$

where φ_i is either $\neg l_i$ or l_i, but never $s_1 \simeq s_2$.

If the current theory does not have rational numbers, then the a_i are printed using integer division. They should, nevertheless, be interpreted as rational numbers. If d_1 or d_2 are 0, they might not be printed.

To check the unsatisfiability of the negation of $\varphi_1 \vee \cdots \vee \varphi_o$ one performs the following steps for each literal. For each i, let $\varphi := \varphi_i$ and $a := a_i$.

1. If $\varphi = s_1 > s_2$, then let $\varphi := s_1 \leq s_2$. If $\varphi = s_1 \geq s_2$, then let $\varphi := s_1 < s_2$. If $\varphi = s_1 < s_2$, then let $\varphi := s_1 \geq s_2$. If $\varphi = s_1 \leq s_2$, then let $\varphi := s_1 > s_2$.
2. If $\varphi = \neg(s_1 \bowtie s_2)$, then let $\varphi := s_1 \bowtie s_2$.
3. Replace φ by $\sum_{i=0}^{n} c_i \times t_i - \sum_{i=n+1}^{m} c_i \times t_i \bowtie d$ where $d := d_2 - d_1$.
4. Now φ has the form $s_1 \bowtie d$. If all variables in s_1 are integer sorted: replace $\bowtie d$ according to table 1.
5. If $\bowtie$ is $\simeq$ replace l by $\sum_{i=0}^{m} a \times c_i \times t_i \simeq a \times d$, otherwise replace it by $\sum_{i=0}^{m} |a| \times c_i \times t_i \simeq |a| \times d$.

$\bowtie$	If d is an integer	Otherwise
$>$	$\geq d + 1$	$\geq \lfloor d \rfloor + 1$
$\geq$	$\geq d$	$\geq \lfloor d \rfloor + 1$

Table 1: Strengthening rules for `la_generic`.

Finally, the sum of the resulting literals is trivially contradictory. The sum

$$\sum_{k=1}^{o} \sum_{i=1}^{m^o} c_i^o * t_i^o \bowtie \sum_{k=1}^{o} d^k$$

where c_i^k is the constant c_i of literal l_k, t_i^k is the term t_i of l_k, and d^k is the constant d of l_k. The operator $\bowtie$ is $\simeq$ if all operators are $\simeq$, $>$ if all are either $\simeq$ or $>$, and $\geq$ otherwise. The a_i must be sucht that the sum on the left-hand side is 0 and the right-hand side is > 0 (or ≥ 0 if $\bowtie is >$).

Example 30.1. A simple `la_generic` step in the logic `LRA` might look like this:

```
(step t10 (cl (not (> (f a) (f b))) (not (= (f a) (f b))))
    :rule la_generic :args (1.0 (- 1.0)))
```

To verify this we have to check the insatisfiability of $f(a) > f(b) \wedge f(a) = f(b)$ (Step 2). After step 3 we get $f(a) - f(b) > 0 \wedge f(a) - f(b) = 0$. Since we don't have an integer sort in this logic step 4 does not apply. Finally, after step 5 the conjunction is $f(a) - f(b) > 0 \wedge -f(a) + f(b) = 0$. This sums to $0 > 0$, which is a contradiction.

Example 30.2. The following `la_generic` step is from a `QF_UFLIA` problem:

```
(step t11 (cl (not (<= f3 0)) (<= (+ 1 (* 4 f3)) 1))
    :rule la_generic :args (1 (div 1 4)))
```

After normalization we get $-f_3 \geq 0 \wedge 4 \times f_3 > 0$. This time step 4 applies and we can strengthen this to $-f_3 \geq 0 \wedge 4 \times f_3 \geq 1$ and after multiplication we get $-f_3 \geq 0 \wedge f_3 \geq \frac{1}{4}$. Which sums to the contradiction $\frac{1}{4} \geq 0$.

31. `lia_generic`

This rule is a placeholder rule for integer arithmetic solving. It takes the same form as `la_generic`, without the additional arguments.

$$\triangleright \text{ i.} \qquad \varphi_1 \dot{\vee} \ldots \dot{\vee} \varphi_n \qquad (\texttt{lia\_generic})$$

with φ_i being linear inequalities. The disjunction $\varphi_1 \vee \cdots \vee \varphi_n$ is a tautology in the theory of linear integer arithmetic.

32. `la_disequality`

$$\triangleright \text{ i.} \qquad t_1 \simeq t_2 \vee \neg(t_1 \leq t_2) \vee \neg(t_2 \leq t_1) \qquad (\texttt{la\_disequality})$$

33. `la_totality`

$$\triangleright \text{ i.} \qquad t_1 \leq t_2 \vee t_2 \leq t_1 \qquad (\texttt{la\_totality})$$

34. `la_tautology`

This rule is a linear arithmetic tautology which can be checked without sophisticated reasoning. It has either the form:

$$\triangleright \text{ i.} \qquad \varphi \qquad (\texttt{la\_tautology})$$

where φ is either a linear inequality $s_1 \bowtie s_2$ or $\neg(s_1 \bowtie s_2)$. After performing step 1 to 3 of the process for checking the `la_generic` the result is trivially unsatisfiable.

The second form handles bounds on linear combinations. It is binary clause:

$\triangleright$ i. $\varphi_1 \vee \varphi_2$ (`la_tautology`)

It can be checked by using the procedure for `la_generic` with while setting the arguments to 1. Informally, the rule follows one of several cases:

- $\neg(s_1 \leq d_1) \vee s_1 \leq d_2$ where $d_1 \leq d_2$
- $s_1 \leq d_1 \vee \neg(s_1 \leq d_2)$ where $d_1 = d_2$
- $\neg(s_1 \geq d_1) \vee s_1 \geq d_2$ where $d_1 \geq d_2$
- $s_1 \geq d_1 \vee \neg(s_1 \geq d_2)$ where $d_1 = d_2$
- $\neg(s_1 \leq d_1) \vee \neg(s_1 \geq d_2)$ where $d_1 < d_2$

The inequalities $s_1 \bowtie d$ are are the result of applying normalization as for the rule `la_generic`.

35. `forall_inst`

$\triangleright$ i. $\neg(\forall x_1, \ldots, x_n.P) \vee P[t_1/x_1]\ldots[t_n/x_n]$ (`forall_inst`; ; $x_{k_1} := t_{k_1}, \ldots, x_{k_n} := t_{k_n}$)

where $k_1, \ldots, k_n$ is a permutation of $1, \ldots, n$ and x_i and k_i have the same sort. The arguments $x_{k_i} := t_{k_i}$ are printed as `(:= xki tki)`.

Remark. A rule simmilar to the `let` rule would be more appropriate. The resulting proof would be more fine grained and this would also be an opportunity to provide a proof for the clausification as currently done by `qnt_cnf`.

36. `qnt_join`

$\Gamma \triangleright$ i. $Qx_1, \ldots, x_n.Qx_{n+1}, \ldots, x_m.\varphi \leftrightarrow Qx_{k_1}, \ldots, x_{k_o}.\varphi$ (`qnt_join`)

where $m > n$, $Q \in \{\forall, \exists\}$, $k_1, \ldots, k_o$ is monotonic map to $1, \ldots, m$ such that $x_{k_1}, \ldots, x_{k_o}$ are pairwise distinct, and $\{x_1, \ldots, x_m\} = \{x_{k_1}, \ldots, x_{k_o}\}$.

37. `qnt_rm_unused`

$\Gamma \triangleright$ i. $Qx_1, \ldots, x_n.\varphi \leftrightarrow Qx_{k_1}, \ldots, x_{k_m}.\varphi$ (`qnt_rm_unused`)

where $m \leq n$, $Q \in \{\forall, \exists\}$, $k_1, \ldots, k_m$ is monotonic map to $1, \ldots, n$ and if $x \in \{x_j \mid j \in \{1, \ldots, n\} \wedge j \notin \{k_1, \ldots, k_m\}\}$ then x is not free in P.

38. `th_resolution`

This rule is the resolution of two or more clauses.

$$\begin{array}{lcl}
\rhd \mathfrak{i}^1. & \varphi_1^1 \dot{\vee} \ldots \dot{\vee} \varphi_{k^1}^1 & (\ldots) \\
& \vdots & \\
\rhd \mathfrak{i}^n. & \varphi_1^n \dot{\vee} \ldots \dot{\vee} \varphi_{k^n}^n & (\ldots) \\
& \vdots & \\
\rhd \ \mathfrak{j}. & \varphi_{s_1}^{r_1} \dot{\vee} \ldots \dot{\vee} \varphi_{s_m}^{r_m} & (\texttt{th_resolution}; \mathfrak{i}^1, \ldots, \mathfrak{i}^n)
\end{array}$$

where $\varphi_{s_1}^{r_1} \dot{\vee} \ldots \dot{\vee} \varphi_{s_m}^{r_m}$ are from φ_j^i and are the result of a chain of predicate resolution steps on the clauses $\mathfrak{i}^1$ to $\mathfrak{i}^n$. It is possible that $m = 0$, i.e. that the result is the empty clause.

This rule is only used when the resolution step is not emitted by the SAT solver. See the equivalent `resolution` rule for the rule emitted by the SAT solver.

Remark. While checking this rule is NP-complete, the `th_resolution`-steps produced by veriT are simple. Experience with reconstructing the step in Isabelle/HOL shows that checking can done by naive decision procedures. The vast majority of `th_resolution`-steps are binary resolution steps.

39. `resolution`

This rule is equivalent to the `the_resolution` rule, but it is emitted by the SAT solver instead of theory reasoners. The differentiation serves only informational purpose.

40. `refl`

Either

$$\Gamma \ \rhd \mathfrak{j}. \qquad t_1 \simeq t_2 \qquad (\texttt{refl})$$

or

$$\Gamma \ \rhd \mathfrak{j}. \qquad \varphi_1 \leftrightarrow \varphi_2 \qquad (\texttt{refl})$$

where φ_1 and φ_2 (P_1 and P_2) are equal after applying the substitution induced by Γ.

41. `trans`

Either

$$\begin{array}{lcl}
\Gamma \ \rhd \mathfrak{i}. & t_1 \simeq t_2 & (\ldots) \\
& \vdots & \\
\Gamma \ \rhd \mathfrak{j}. & t_2 \simeq t_3 & (\ldots) \\
& \vdots & \\
\Gamma \ \rhd \mathfrak{k}. & t_1 \simeq t_3 & (\texttt{trans}; \mathfrak{i}, \mathfrak{j})
\end{array}$$

or

$\Gamma \rhd \mathsf{i}.$ $\quad \varphi_1 \leftrightarrow \varphi_2$ $\quad (\dots)$

$\vdots$

$\Gamma \rhd \mathsf{j}.$ $\quad \varphi_2 \leftrightarrow \varphi_3$ $\quad (\dots)$

$\vdots$

$\Gamma \rhd \mathsf{k}.$ $\quad \varphi_1 \leftrightarrow \varphi_3$ $\quad (\mathtt{trans}; \mathsf{i}, \mathsf{j})$

42. `cong`

Either

$\Gamma \rhd \mathsf{i}_1.$ $\quad t_1 \simeq u_1$ $\quad (\dots)$

$\vdots$

$\Gamma \rhd \mathsf{i}_n.$ $\quad t_n \simeq u_n$ $\quad (\dots)$

$\vdots$

$\Gamma \rhd \mathsf{j}.$ $\quad \mathrm{f}(t_1, \dots, t_n) \simeq \mathrm{f}(u_1, \dots, u_n)$ $\quad (\mathtt{cong}; \mathsf{i}_1, \dots, \mathsf{i}_n)$

where f is an n-ary function symbol, or

$\Gamma \rhd \mathsf{i}_1.$ $\quad \varphi_1 \simeq \psi_1$ $\quad (\dots)$

$\vdots$

$\Gamma \rhd \mathsf{i}_n.$ $\quad \varphi_n \simeq \psi_n$ $\quad (\dots)$

$\vdots$

$\Gamma \rhd \mathsf{j}.$ $\quad \mathrm{P}(\varphi_1, \dots, \varphi_n) \leftrightarrow \mathrm{P}(\psi_1, \dots, \psi_n)$ $\quad (\mathtt{cong}; \mathsf{i}_1, \dots, \mathsf{i}_n)$

where P is an n-ary predicate symbol.

43. `and`

$\rhd \mathsf{i}.$ $\quad \varphi_1 \wedge \dots \wedge \varphi_n$ $\quad (\dots)$

$\vdots$

$\rhd \mathsf{j}.$ $\quad \varphi_i$ $\quad (\mathtt{and}; \mathsf{i})$

44. `tautology`

$\rhd \mathsf{i}.$ $\quad \varphi_1 \dot{\vee} \dots \dot{\vee} \varphi_i \dot{\vee} \dots \dot{\vee} \varphi_j \dot{\vee} \dots \dot{\vee} \varphi_n$ $\quad (\dots)$

$\vdots$

$\rhd \mathsf{j}.$ $\quad \top$ $\quad (\mathtt{tautology}; \mathsf{i})$

and $\varphi_i = \bar{\varphi}_j$.

45. `not_or`

$\triangleright$ i. $\quad \neg(\varphi_1 \vee \cdots \vee \varphi_n)$ $\quad (\dots)$

$\vdots$

$\triangleright$ j. $\quad \neg\varphi_i$ $\quad (\texttt{not\_or}; \text{i})$

46. `or`

$\triangleright$ i. $\quad \varphi_1 \vee \cdots \vee \varphi_n$ $\quad (\dots)$

$\vdots$

$\triangleright$ j. $\quad \varphi_1 \dot{\vee} \dots \dot{\vee} \varphi_n$ $\quad (\texttt{or}; \text{i})$

Remark. This rule deconstructs the `or` operator into a `cl`.

Example 46.1. An application of the `or` rule.

```
(step t15 (cl (or (= a b) (not (<= a b)) (not (<= b a))))
    :rule la_disequality)
(step t16 (cl     (= a b) (not (<= a b)) (not (<= b a)))
    :rule or :premises (t15))
```

47. `not_and`

$\triangleright$ i. $\quad \neg(\varphi_1 \wedge \cdots \wedge \varphi_n)$ $\quad (\dots)$

$\vdots$

$\triangleright$ j. $\quad \neg\varphi_1 \dot{\vee} \dots \dot{\vee} \neg\varphi_n$ $\quad (\texttt{not\_and}; \text{i})$

48. `xor1`

$\triangleright$ i. $\quad \operatorname{xor} \varphi_1\ \varphi_2$ $\quad (\dots)$

$\vdots$

$\triangleright$ j. $\quad \varphi_1 \dot{\vee} \varphi_2$ $\quad (\texttt{xor1}; \text{i})$

49. `xor2`

$\triangleright$ i. $\quad \operatorname{xor} \varphi_1\ \varphi_2$ $\quad (\dots)$

$\vdots$

$\triangleright$ j. $\quad \neg\varphi_1 \dot{\vee} \neg\varphi_2$ $\quad (\texttt{xor2}; \text{i})$

50. `not_xor1`

▷ i.	$\neg(\mathrm{xor}\,\varphi_1\ \varphi_2)$	$(\dots)$
	$\vdots$	
▷ j.	$\varphi_1 \dot{\vee} \neg\varphi_2$	(not_xor1; i)

51. not_xor2

▷ i.	$\neg(\mathrm{xor}\,\varphi_1\ \varphi_2)$	$(\dots)$
	$\vdots$	
▷ j.	$\neg\varphi_1 \dot{\vee} \varphi_2$	(not_xor2; i)

52. implies

▷ i.	$\varphi_1 \to \varphi_2$	$(\dots)$
	$\vdots$	
▷ j.	$\neg\varphi_1 \dot{\vee} \varphi_2$	(implies; i)

53. not_implies1

▷ i.	$\neg(\varphi_1 \to \varphi_2)$	$(\dots)$
	$\vdots$	
▷ j.	φ_1	(not_implies1; i)

54. not_implies2

▷ i.	$\neg(\varphi_1 \to \varphi_2)$	$(\dots)$
	$\vdots$	
▷ j.	$\neg\varphi_2$	(not_implies2; i)

55. equiv1

▷ i.	$\varphi_1 \leftrightarrow \varphi_2$	$(\dots)$
	$\vdots$	
▷ j.	$\neg\varphi_1 \dot{\vee} \varphi_2$	(equiv1; i)

56. equiv2

▷ i.	$\varphi_1 \leftrightarrow \varphi_2$	$(\dots)$
	$\vdots$	
▷ j.	$\varphi_1 \dot{\vee} \neg\varphi_2$	(equiv2; i)

57. `not_equiv1`

$\triangleright$ i. $\quad \neg(\varphi_1 \leftrightarrow \varphi_2)$ $\quad (\dots)$

$\vdots$

$\triangleright$ j. $\quad \varphi_1 \dot{\vee} \varphi_2$ $\quad (\texttt{not\_equiv1}; \text{i})$

58. `not_equiv2`

$\triangleright$ i. $\quad \neg(\varphi_1 \leftrightarrow \varphi_2)$ $\quad (\dots)$

$\vdots$

$\triangleright$ j. $\quad \neg\varphi_1 \dot{\vee} \neg\varphi_2$ $\quad (\texttt{not\_equiv2}; \text{i})$

59. `ite1`

$\triangleright$ i. $\quad \mathrm{ite}\,\varphi_1\ \varphi_2\ \varphi_3$ $\quad (\dots)$

$\vdots$

$\triangleright$ j. $\quad \varphi_1 \dot{\vee} \varphi_3$ $\quad (\texttt{ite1}; \text{i})$

60. `ite2`

$\triangleright$ i. $\quad \mathrm{ite}\,\varphi_1\ \varphi_2\ \varphi_3$ $\quad (\dots)$

$\vdots$

$\triangleright$ j. $\quad \neg\varphi_1 \dot{\vee} \varphi_2$ $\quad (\texttt{ite2}; \text{i})$

61. `not_ite1`

$\triangleright$ i. $\quad \neg(\mathrm{ite}\,\varphi_1\ \varphi_2\ \varphi_3)$ $\quad (\dots)$

$\vdots$

$\triangleright$ j. $\quad \varphi_1 \dot{\vee} \neg\varphi_3$ $\quad (\texttt{not\_ite1}; \text{i})$

62. `not_ite2`

$\triangleright$ i. $\quad \neg(\mathrm{ite}\,\varphi_1\ \varphi_2\ \varphi_3)$ $\quad (\dots)$

$\vdots$

$\triangleright$ j. $\quad \neg\varphi_2 \dot{\vee} \neg\varphi_2$ $\quad (\texttt{not\_ite2}; \text{i})$

63. `ite_intro`

Either

$\triangleright$ i. $\quad t \simeq (t' \wedge u_1 \wedge \cdots \wedge u_n)$ $\quad (\dots)$

or

$\rhd$ i. $\varphi \leftrightarrow (\varphi' \wedge u_1 \wedge \cdots \wedge u_n)$ $(\dots)$

The term t (the formula φ) contains the ite operator. Let $s_1, \dots, s_n$ be the terms starting with ite, i.e. $s_i := \mathrm{ite}\, \psi_i\, r_i\, r_i'$, then u_i has the form:

$$\mathrm{ite}\, \psi_i\, (s_i \simeq r_i)\, (s_i \simeq r_i')$$

or

$$\mathrm{ite}\, \psi_i\, (s_i \leftrightarrow r_i)\, (s_i \leftrightarrow r_i')$$

if s_i is of sort $\mathbf{Bool}$. The term t' (the formular φ') is equal to the term t (the formular φ') up to the reordering of equalities where one argument is an ite term.

Remark. This rule stems from the introduction of fresh constants for if-then-else terms inside veriT. Internally s_i is a new constant symbol and the φ on the right side of the equality is φ with the if-then-else terms replaced by the constants. Those constants are unfolded during proof printing. Hence, the slightly strange form and the reordering of equalities.

64. `contraction`

$\rhd$ i. $\varphi_1 \dot\vee \dots \dot\vee \varphi_n$ $(\dots)$

$\vdots$

$\rhd$ j. $\varphi_{k_1} \dot\vee \dots \dot\vee \varphi_{k_m}$ $(\texttt{contraction}; \mathrm{i})$

where $m \leq n$ and $k_1 \dots k_m$ is a monotonic map to $1 \dots n$ such that $\varphi_{k_1} \dots \varphi_{k_m}$ are pairwise distinct and $\{\varphi_1, \dots, \varphi_n\} = \{\varphi_{k_1} \dots \varphi_{k_m}\}$. Hence, this rule remove duplicated literals.

65. `connective_def`

This rule is used to replace connectives by their definition. It can be one of the following:

$\Gamma \rhd$ i. $\varphi_1 \,\mathrm{xor}\, \varphi_2 \leftrightarrow (\neg\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \neg\varphi_2)$ $(\texttt{connective\_def})$

$\Gamma \rhd$ i. $\varphi_1 \leftrightarrow \varphi_2 \leftrightarrow (\varphi_1 \to \varphi_2) \wedge (\varphi_2 \to \varphi_1)$ $(\texttt{connective\_def})$

$\Gamma \rhd$ i. $\mathrm{ite}\, \varphi_1\, \varphi_2\, \varphi_3 \leftrightarrow (\varphi_1 \to \varphi_2) \wedge (\neg\varphi_1 \to \neg\varphi_3)$ $(\texttt{connective\_def})$

66. `ite_simplify`

This rule simplifies an if-then-else term by applying equivalent transformations as long as possible. Depending on the sort of the ite-term the rule can have one of two forms. If the sort is $\mathbf{Bool}$ it has the form

$\Gamma \rhd$ i. $\mathrm{ite}\, \varphi\, t_1; t_2 \leftrightarrow \psi$ $(\texttt{ite\_simplify})$

with ψ being the transformed term.

Otherwise, it has the form

$\Gamma \rhd \mathfrak{i}.$ $\qquad \mathrm{ite}\,\varphi\ t_1\ t_2 \simeq u$ $\qquad$ (`ite_simplify`)

with u being the transformed term.

The possible transformations are:

- $\mathrm{ite}\,\top\ t_1\ t_2 \leftrightarrow t_1$
- $\mathrm{ite}\,\bot\ t_1\ t_2 \leftrightarrow t_2$
- $\mathrm{ite}\,\psi\ t\ t \leftrightarrow t$
- $\mathrm{ite}\,\neg\varphi\ t_1\ t_2 \leftrightarrow \mathrm{ite}\,\varphi\ t_2\ t_1$
- $\mathrm{ite}\,\top\ t_1\ t_2 \leftrightarrow t_1$
- $\mathrm{ite}\,\bot\ t_1\ t_2 \leftrightarrow t_2$
- $\mathrm{ite}\,\psi\ (\mathrm{ite}\,\psi\ t_1\ t_2)\ t_3 \leftrightarrow \mathrm{ite}\,\psi\ t_1\ t_3$
- $\mathrm{ite}\,\psi\ t_1\ (\mathrm{ite}\,\psi\ t_2\ t_3) \leftrightarrow \mathrm{ite}\,\psi\ t_1\ t_3$
- $\mathrm{ite}\,\psi\ \top\ \bot \leftrightarrow \psi$
- $\mathrm{ite}\,\psi\ \bot\ \top \leftrightarrow \neg\psi$
- $\mathrm{ite}\,\psi\ \top\ \varphi \leftrightarrow \psi \vee \varphi$
- $\mathrm{ite}\,\psi\ \varphi\ \bot \leftrightarrow \psi \wedge \varphi$
- $\mathrm{ite}\,\psi\ \bot\ \varphi \leftrightarrow \neg\psi \wedge \varphi$
- $\mathrm{ite}\,\psi\ \varphi\ \top \leftrightarrow \neg\psi \vee \varphi$

67. `eq_simplify`

This rule simplifies an $\simeq$ term by applying equivalent transformations as long as possible. Hence, the general form is:

$\Gamma \rhd \mathfrak{i}.$ $\qquad t_1 \simeq t_2 \leftrightarrow \varphi$ $\qquad$ (`eq_simplify`)

with ψ being the transformed term.

The possible transformations are:

- $t \simeq t \leftrightarrow \top$
- $t_1 \simeq t_2 \leftrightarrow \bot$ if t_1 and t_2 are different numeric constants.
- $\neg(t \simeq t) \leftrightarrow \bot$ if t is a numeric constant.

68. `and_simplify`

This rule simplifies an $\wedge$ term by applying equivalent transformations as long as possible. Hence, the general form is:

$$\Gamma \ \triangleright \mathrm{i}. \qquad \varphi_1 \wedge \cdots \wedge \varphi_n \leftrightarrow \psi \qquad (\texttt{and\_simplify})$$

with ψ being the transformed term.

The possible transformations are:

- $\top \wedge \cdots \wedge \top \leftrightarrow \top$
- $\varphi_1 \wedge \cdots \wedge \varphi_n \leftrightarrow \varphi_1 \wedge \cdots \wedge \varphi_{n'}$ where the right hand side has all $\top$ literals removed.
- $\varphi_1 \wedge \cdots \wedge \varphi_n \leftrightarrow \varphi_1 \wedge \cdots \wedge \varphi_{n'}$ where the right hand side has all repeated literals removed.
- $\varphi_1 \wedge \cdots \wedge \bot \wedge \cdots \wedge \varphi_n \leftrightarrow \bot$
- $\varphi_1 \wedge \cdots \wedge \varphi_i \wedge \cdots \wedge \varphi_j \wedge \cdots \wedge \varphi_n \leftrightarrow \bot$ if $\varphi_i = \bar{\varphi}_j$

69. `or_simplify`

This rule simplifies an $\vee$ term by applying equivalent transformations as long as possible. Hence, the general form is:

$$\Gamma \ \triangleright \mathrm{i}. \qquad (\varphi_1 \vee \cdots \vee \varphi_n) \leftrightarrow \psi \qquad (\texttt{or\_simplify})$$

with ψ being the transformed term.

The possible transformations are:

- $\bot \vee \cdots \vee \bot \leftrightarrow \bot$
- $\varphi_1 \vee \cdots \vee \varphi_n \leftrightarrow \varphi_1 \vee \cdots \vee \varphi_{n'}$ where the right hand side has all $\bot$ literals removed.
- $\varphi_1 \vee \cdots \vee \varphi_n \leftrightarrow \varphi_1 \vee \cdots \vee \varphi_{n'}$ where the right hand side has all repeated literals removed.
- $\varphi_1 \vee \cdots \vee \top \vee \cdots \vee \varphi_n \leftrightarrow \top$
- $\varphi_1 \vee \cdots \vee \varphi_i \vee \cdots \vee \varphi_j \vee \cdots \vee \varphi_n \leftrightarrow \top$ if $\varphi_i = \bar{\varphi}_j$

70. `not_simplify`

This rule simplifies an $\neg$ term by applying equivalent transformations as long as possible. Hence, the general form is:

$$\Gamma \ \triangleright \mathrm{i}. \qquad \neg\varphi \leftrightarrow \psi \qquad (\texttt{not\_simplify})$$

with ψ being the transformed term.

The possible transformations are:

- $\neg(\neg\varphi) \leftrightarrow \varphi$
- $\neg\bot \leftrightarrow \top$
- $\neg\top \leftrightarrow \bot$

71. `implies_simplify`

This rule simplifies an $\rightarrow$ term by applying equivalent transformations as long as possible. Hence, the general form is:

$$\Gamma \ \triangleright \mathrm{i}. \qquad \varphi_1 \rightarrow \varphi_2 \leftrightarrow \psi \qquad (\texttt{implies\_simplify})$$

with ψ being the transformed term.

The possible transformations are:

- $\neg\varphi_1 \rightarrow \neg\varphi_2 \leftrightarrow \varphi_2 \rightarrow \varphi_1$
- $\bot \rightarrow \varphi \leftrightarrow \top$
- $\varphi \rightarrow \top \leftrightarrow \top$
- $\top \rightarrow \varphi \leftrightarrow \varphi$
- $\varphi \rightarrow \bot \leftrightarrow \neg\varphi$
- $\varphi \rightarrow \varphi \leftrightarrow \top$
- $\neg\varphi \rightarrow \varphi \leftrightarrow \varphi$
- $\varphi \rightarrow \neg\varphi \leftrightarrow \neg\varphi$
- $(\varphi_1 \rightarrow \varphi_2) \rightarrow \varphi_2 \leftrightarrow \varphi_1 \vee \varphi_2$

72. `equiv_simplify`

This rule simplifies an $\leftrightarrow$ term by applying equivalent transformations as long as possible. Hence, the general form is:

$$\Gamma \ \triangleright \mathrm{i}. \qquad \varphi_1 \leftrightarrow \varphi_2 \leftrightarrow \psi \qquad (\texttt{equiv\_simplify})$$

with ψ being the transformed term.

The possible transformations are:

- $(\neg\varphi_1 \leftrightarrow \neg\varphi_2) \leftrightarrow (\varphi_1 \leftrightarrow \varphi_2)$
- $(\varphi \leftrightarrow \varphi) \leftrightarrow \top$

- $(\varphi \leftrightarrow \neg\varphi) \leftrightarrow \bot$
- $(\neg\varphi \leftrightarrow \varphi) \leftrightarrow \bot$
- $(\top \leftrightarrow \varphi) \leftrightarrow \varphi$
- $(\varphi \leftrightarrow \top) \leftrightarrow \varphi$
- $(\bot \leftrightarrow \varphi) \leftrightarrow \neg\varphi$
- $(\varphi \leftrightarrow \bot) \leftrightarrow \neg\varphi$

73. `bool_simplify`

This rule simplifies a boolean term by applying equivalent transformations as long as possible. Hence, the general form is:

$$\Gamma \ \rhd \mathrm{i}. \qquad \varphi \leftrightarrow \psi \qquad (\texttt{bool\_simplify})$$

with ψ being the transformed term.

The possible transformations are:

- $\neg(\varphi_1 \to \varphi_2) \leftrightarrow (\varphi_1 \wedge \neg\varphi_2)$
- $\neg(\varphi_1 \vee \varphi_2) \leftrightarrow (\neg\varphi_1 \wedge \neg\varphi_2)$
- $\neg(\varphi_1 \wedge \varphi_2) \leftrightarrow (\neg\varphi_1 \vee \neg\varphi_2)$
- $(\varphi_1 \to (\varphi_2 \to \varphi_3)) \leftrightarrow (\varphi_1 \wedge \varphi_2) \to \varphi_3$
- $((\varphi_1 \to \varphi_2) \to \varphi_2) \leftrightarrow (\varphi_1 \vee \varphi_2)$
- $(\varphi_1 \wedge (\varphi_1 \to \varphi_2)) \leftrightarrow (\varphi_1 \wedge \varphi_2)$
- $((\varphi_1 \to \varphi_2) \wedge \varphi_1) \leftrightarrow (\varphi_1 \wedge \varphi_2)$

74. `qnt_simplify`

This rule simplifies a $\forall$ term with a constant predicate.

$$\Gamma \ \rhd \mathrm{i}. \qquad \forall x_1, \ldots, x_n.\varphi \leftrightarrow \varphi \qquad (\texttt{qnt\_simplify})$$

where φ is either $\top$ or $\bot$.

75. `div_simplify`

This rule simplifies a division by applying equivalent transformations. The general form is:

$$\Gamma \ \rhd \mathrm{i}. \qquad \frac{t_1}{t_2} \simeq t_3 \qquad (\texttt{div\_simplify})$$

The possible transformations are:

- $\frac{t}{t} = 1$
- $\frac{t}{1} = t$
- $\frac{t_1}{t_2} = t_3$ if t_1 and t_2 are constants and t_3 is t_1 divided by t_2 according to the semantic of the current theory.

76. `prod_simplify`

This rule simplifies a product by applying equivalent transformations as long as possible. The general form is:

$$\Gamma \ \triangleright \mathfrak{i}. \qquad t_1 \times \cdots \times t_n \simeq u \qquad (\texttt{prod\_simplify})$$

where u is either a constant or a product.

The possible transformations are:

- $t_1 \times \cdots \times t_n = u$ where all t_i are constants and u is their product.
- $t_1 \times \cdots \times t_n = 0$ if any t_i is 0.
- $t_1 \times \cdots \times t_n = c \times t_{k_1} \times \cdots \times t_{k_n}$ where c ist the product of the constants of $t_1, \ldots, t_n$ and $t_{k_1}, \ldots, t_{k_n}$ is $t_1, \ldots, t_n$ with the constants removed.
- $t_1 \times \cdots \times t_n = t_{k_1} \times \cdots \times t_{k_n}$: same as above if c is 1.

77. `unary_minus_simplify`

This rule is either

$$\Gamma \ \triangleright \mathfrak{i}. \qquad -(-t) \simeq t \qquad (\texttt{unary\_minus\_simplify})$$

or

$$\Gamma \ \triangleright \mathfrak{i}. \qquad -t \simeq u \qquad (\texttt{unary\_minus\_simplify})$$

where u is the negated numerical constant t.

78. `minus_simplify`

This rule simplifies a subtraction by applying equivalent transformations. The general form is:

$$\Gamma \ \triangleright \mathfrak{i}. \qquad t_1 - t_2 \simeq u \qquad (\texttt{minus\_simplify})$$

The possible transformations are:

- $t - t = 0$

- $t_1 - t_2 = t_3$ where t_1 and t_2 are numerical constants and t_3 is t_2 subtracted from t_1.
- $t - 0 = t$
- $0 - t = -t$

79. `sum_simplify`

This rule simplifies a sum by applying equivalent transformations as long as possible. The general form is:

$$\Gamma \ \triangleright \mathrm{i}. \qquad t_1 + \cdots + t_n \simeq u \qquad (\texttt{sum\_simplify})$$

where u is either a constant or a product.

The possible transformations are:

- $t_1 + \cdots + t_n = c$ where all t_i are constants and c is their sum.
- $t_1 + \cdots + t_n = c + t_{k_1} + \cdots + t_{k_n}$ where c ist the sum of the constants of $t_1, \ldots, t_n$ and $t_{k_1}, \ldots, t_{k_n}$ is $t_1, \ldots, t_n$ with the constants removed.
- $t_1 + \cdots + t_n = t_{k_1} + \cdots + t_{k_n}$: same as above if c is 0.

80. `comp_simplify`

This rule simplifies a comparison by applying equivalent transformations as long as possible. The general form is:

$$\Gamma \ \triangleright \mathrm{i}. \qquad t_1 \bowtie t_n \leftrightarrow \psi \qquad (\texttt{comp\_simplify})$$

where $\bowtie$ is as for the proof rule `la_generic`, but never $\simeq$.

The possible transformations are:

- $t_1 < t_2 \leftrightarrow \varphi$ where t_1 and t_2 are numerical constants and φ is $\top$ if t_1 is strictly greater than t_2 and $\bot$ otherwise.
- $t < t \leftrightarrow \bot$
- $t_1 \leq t_2 \leftrightarrow \varphi$ where t_1 and t_2 are numerical constants and φ is $\top$ if t_1 is greater than t_2 or equal and $\bot$ otherwise.
- $t \leq t \leftrightarrow \top$
- $t_1 \geq t_2 \leftrightarrow t_2 \leq t_1$
- $t_1 < t_2 \leftrightarrow \neg(t_2 \leq t_1)$
- $t_1 > t_2 \leftrightarrow \neg(t_1 \leq t_2)$

81. `nary_elim`

This rule replaces n-ary operators with their equivalent application of the binary operator. It is never applied to $\wedge$ or $\vee$.

Thre cases are possible. If the operator $\circ$ is left associative, then the rule has the form

$$\Gamma \ \rhd \mathrm{i}. \qquad \bigcirc_{i=1}^{n} t_i \leftrightarrow (\dots (t_1 \circ t_2) \circ t_3) \circ \dots t_n) \qquad (\texttt{nary\_elim})$$

If the operator $\circ$ is right associative, then the rule has the form

$$\Gamma \ \rhd \mathrm{i}. \qquad \bigcirc_{i=1}^{n} t_i \leftrightarrow (t_1 \circ \dots \circ (t_{n-2} \circ (t_{n-1} \circ t_n) \dots) \qquad (\texttt{nary\_elim})$$

If the operator is *chainable*, then it has the form

$$\Gamma \ \rhd \mathrm{i}. \qquad \bigcirc_{i=1}^{n} t_i \leftrightarrow (t_1 \circ t_2) \wedge (t_2 \circ t_3) \wedge \dots \wedge (t_{n-1} \circ t_n) \qquad (\texttt{nary\_elim})$$

82. `ac_simp`

This rule simplifies nested occurences of $\vee$ or $\wedge$:

$$\Gamma \ \rhd \mathrm{i}. \qquad \psi \leftrightarrow \varphi_1 \circ \dots \circ \varphi_n \qquad (\texttt{ac\_simp})$$

where $\circ \in \{\vee, \wedge\}$ and ψ is a nested application of $\circ$. The literals φ_i are literals of the flattening of ψ with duplicates removed.

83. `bfun_elim`

$$\begin{array}{lcl} \rhd \mathrm{i}. & \psi & (\dots) \\ & \vdots & \\ \rhd \mathrm{j}. & \varphi & (\texttt{bfun\_elim}; \mathrm{i}) \end{array}$$

The formula φ is ψ after boolean functions have been simplified. This happens in a two step process. Both steps recursively iterate over ψ. The first step expands quantified variable of type $\mathbf{Bool}$. Hence, $\exists x.t$ becomes $t[\bot/x] \vee t[\top/x]$ and $\forall x.t$ becomes $t[\bot/x] \wedge t[\top/x]$. If n variables of sort $\mathbf{Bool}$ appear in a quantifier, the disjunction (conjunction) has 2^n terms. Each term replaces the variables in t according to the bits of a number which is increased by one for each subsequent term starting from zero. The left-most variable corresponds to the least significant bit.

The second step expands function argument of boolean types by introducing appropriate if-then-else terms. For example, consider $f(x, P, y)$ where P is some formula. Then we replace this term by $\mathrm{ite}\, P\ f(x, \top, y)\ f(x, \bot, y)$. If the argument is already the constant $\top$ or $\bot$ it is ignored.

84. `deep_skolemize`

This rule is only emitted when the option `--enable-deep-skolem` is given. This option is experimental and should not be used.

85. `qnt_cnf`

$\triangleright$ i. $\quad \neg(\forall x_1, \ldots, x_n.\varphi) \vee \forall x_{k_1}, \ldots, x_{k_m}.\varphi'$ (qnt_cnf)

This is a placeholder rule for clausification of a term under a universal quantifier. This is used by conflicting instantiation. φ' is one of the clause of the clause normal form of φ. The variables $x_{k_1}, \ldots, x_{k_m}$ are a permutation of $x_1, \ldots, x_n$ plus additional variables added by prenexing φ. Normalization is performed in two phases. First, the negative normal form is formed, then the result is prenexed. The result of the first step is $\Phi(\varphi, 1)$ where:

$$
\begin{aligned}
\Phi(\neg\varphi, 1) &:= \Phi(\varphi, 0)\\
\Phi(\neg\varphi, 0) &:= \Phi(\varphi, 1)\\
\Phi(\varphi_1 \vee \cdots \vee \varphi_n, 1) &:= \Phi(\varphi_1, 1) \vee \cdots \vee \Phi(\varphi_n, 1)\\
\Phi(\varphi_1 \wedge \cdots \wedge \varphi_n, 1) &:= \Phi(\varphi_1, 1) \wedge \cdots \wedge \Phi(\varphi_n, 1)\\
\Phi(\varphi_1 \vee \cdots \vee \varphi_n, 0) &:= \Phi(\varphi_1, 0) \wedge \cdots \wedge \Phi(\varphi_n, 0)\\
\Phi(\varphi_1 \wedge \cdots \wedge \varphi_n, 0) &:= \Phi(\varphi_1, 0) \vee \cdots \vee \Phi(\varphi_n, 0)\\
\Phi(\varphi_1 \to \varphi_2, 1) &:= (\Phi(\varphi_1, 0) \vee \Phi(\varphi_2, 1)) \wedge (\Phi(\varphi_2, 0) \vee \Phi(\varphi_1, 1))\\
\Phi(\varphi_1 \to \varphi_2, 0) &:= (\Phi(\varphi_1, 1) \wedge \Phi(\varphi_2, 0)) \vee (\Phi(\varphi_2, 1) \wedge \Phi(\varphi_1, 0))\\
\Phi(\mathrm{ite}\,\varphi_1\ \varphi_2\ \varphi_3, 1) &:= (\Phi(\varphi_1, 0) \vee \Phi(\varphi_2, 1)) \wedge (\Phi(\varphi_1, 1) \vee \Phi(\varphi_3, 1))\\
\Phi(\mathrm{ite}\,\varphi_1\ \varphi_2\ \varphi_3, 0) &:= (\Phi(\varphi_1, 1) \wedge \Phi(\varphi_2, 0)) \vee (\Phi(\varphi_1, 0) \wedge \Phi(\varphi_3, 0))\\
\Phi(\forall x_1, \ldots, x_n.\varphi, 1) &:= \forall x_1, \ldots, x_n.\Phi(\varphi, 1)\\
\Phi(\exists x_1, \ldots, x_n.\varphi, 1) &:= \exists x_1, \ldots, x_n.\Phi(\varphi, 1)\\
\Phi(\forall x_1, \ldots, x_n.\varphi, 0) &:= \exists x_1, \ldots, x_n.\Phi(\varphi, 0)\\
\Phi(\exists x_1, \ldots, x_n.\varphi, 0) &:= \forall x_1, \ldots, x_n.\Phi(\varphi, 0)\\
\Phi(\varphi, 1) &:= \varphi\\
\Phi(\varphi, 0) &:= \neg\varphi
\end{aligned}
$$

86. subproof

The subproof rule completes a subproof and discharges local assumptions. Every subproof starts with some input steps. The last step of the subproof is the conclusion.

$\triangleright$ $\mathfrak{i}_1$.	ψ_1	(input)
	$\vdots$	
$\triangleright$ $\mathfrak{i}_n$.	ψ_n	(input)
	$\vdots$	
$\triangleright$ j.	φ	$(\ldots)$
$\triangleright$ $\mathfrak{k}$.	$\neg\psi_1 \dot\vee \ldots \dot\vee \neg\psi_n \dot\vee \varphi$	(subproof)

87. bind

The `bind` rule is used to rename bound variables.

$$\dfrac{\begin{array}{lll} & \vdots & \\ \Gamma, y_1, \ldots, y_n, x_1 \mapsto y_1, \ldots, x_n \mapsto y_n, \rhd \mathrm{j}. & \varphi \leftrightarrow \varphi' & (\ldots) \end{array}}{\begin{array}{lll} \Gamma \rhd \mathfrak{k}. & \forall x_1, \ldots, x_n.\varphi \leftrightarrow \forall y_1, \ldots, y_n.\varphi' & (\texttt{bind}) \end{array}}$$

where the variables $y_1, \ldots, y_n$ is not free in $\forall x_1, \ldots, x_n.\varphi$.

88. `let`

This rule eliminats let. It has the form

$$\dfrac{\begin{array}{lll} \Gamma \rhd \mathrm{i}_1. & t_1 \simeq s_1 & (\ldots) \\ & \vdots & \\ \Gamma \rhd \mathrm{i}_n. & t_n \simeq s_n & (\ldots) \\ & \vdots & \\ & \vdots & \\ \Gamma, x_1 \mapsto s_1, \ldots, x_n \mapsto s_n, \rhd \mathrm{j}. & u \simeq u' & (\ldots) \end{array}}{\begin{array}{lll} \Gamma \rhd \mathfrak{k}. & (\mathrm{let}\, x_1 := t_1, \ldots, x_n := t_n.u) \simeq u' & (\texttt{let};\mathrm{i}_1 \ldots \mathrm{i}_n) \end{array}}$$

where $\simeq$ is replaced by $\leftrightarrow$ where necessary.

If for $t_i \simeq s_i$ the t_i and s_i are syntactically equal, the premise is skipped.

89. `onepoint`

The `onepoint` rule is the "one-point-rule". That is: it eliminates quantified variables that can only have one value.

$$\dfrac{\begin{array}{lll} & \vdots & \\ \Gamma, x_{k_1}, \ldots, x_{k_m}, x_{j_1} \mapsto t_{j_1}, \ldots, x_{j_o} \mapsto t_{j_o}, \rhd \mathrm{j}. & \varphi \leftrightarrow \varphi' & (\ldots) \end{array}}{\begin{array}{lll} \Gamma \rhd \mathfrak{k}. & Qx_1, \ldots, x_n.\varphi \leftrightarrow Qx_{k_1}, \ldots, x_{k_m}.\varphi' & (\texttt{onepoint}) \end{array}}$$

where $Q \in \{\forall, \exists\}$, $n = m + o$, $k_1, \ldots, k_m$ and $j_1, \ldots, j_o$ are monotone mappings to $1, \ldots, n$, and no x_{k_i} appears in $x_{j_1}, \ldots, x_{j_o}$.

The terms $t_{j_1}, \ldots, t_{j_o}$ are the points of the variables $x_{j_1}, \ldots, x_{j_o}$. Points are defined by equalities $x_i \simeq t_i$ with positive polarity in the term φ.

Remark. Since an eliminated variable x_i might appear free in a term t_j, it is necessary to replace x_i with t_i inside t_j. While this substitution is performed correctly, the proof for it is currently missing.

Example 89.1. An applichtion of the `onepoint` rule on the term $\forall x, y.\, x \simeq y \to \mathrm{f}(x) \wedge \mathrm{f}(y)$ look like this:

```
(anchor :step t3 :args ((:= y x)))
(step t3.t1 (cl (= x y)) :rule refl)
```

```
(step t3.t2 (cl (= (= x y) (= x x)))
    :rule cong :premises (t3.t1))
(step t3.t3 (cl (= x y)) :rule refl)
(step t3.t4 (cl (= (f y) (f x)))
    :rule cong :premises (t3.t3))
(step t3.t5 (cl (= (and (f x) (f y)) (and (f x) (f x))))
    :rule cong :premises (t3.t4))
(step t3.t6 (cl (=
        (=> (= x y) (and (f x) (f y)))
        (=> (= x x) (and (f x) (f x)))))
    :rule cong :premises (t3.t2 t3.t5))
(step t3 (cl (=
        (forall ((x S) (y S)) (=> (= x y) (and (f x) (f y))))
        (forall ((x S)) (=> (= x x) (and (f x) (f x))))))
    :rule qnt_simplify)
```

90. `sko_ex`

The `sko_ex` rule skolemizes existential quantifiers.

$$\frac{\begin{array}{l}\\ \Gamma, x_1 \mapsto (\epsilon x_1.\varphi), \ldots, x_n \mapsto (\epsilon x_n.\varphi),\ \triangleright \mathrm{j}.\end{array} \quad \begin{array}{c}\vdots \\ \varphi \leftrightarrow \psi\end{array} \quad \begin{array}{r}\\ (\ldots)\end{array}}{\Gamma\ \triangleright \mathrm{f}. \qquad \exists x_1, \ldots, x_n.\varphi \leftrightarrow \psi \qquad (\texttt{sko\_ex})}$$

91. `sko_forall`

The `sko_forall` rule skolemizes universal quantifiers.

$$\frac{\begin{array}{l}\\ \Gamma, x_1 \mapsto (\epsilon x_1.\neg\varphi), \ldots, x_n \mapsto (\epsilon x_n.\neg\varphi),\ \triangleright \mathrm{j}.\end{array} \quad \begin{array}{c}\vdots \\ \varphi \leftrightarrow \psi\end{array} \quad \begin{array}{r}\\ (\ldots)\end{array}}{\Gamma\ \triangleright \mathrm{f}. \qquad \forall x_1, \ldots, x_n.\varphi \leftrightarrow \psi \qquad (\texttt{sko\_forall})}$$

References

[1] Barbosa, H., Blanchette, J. C., Fleury, M., and Fontaine, P. Scalable fine-grained proofs for formula processing. *Journal of Automated Reasoning* (2019).

[2] Barrett, C., Fontaine, P., and Tinelli, C. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[3] Besson, F., Fontaine, P., and Théry, L. A flexible proof format for SMT: A proposal. In *PxTP 2011* (2011), P. Fontaine and A. Stump, Eds., pp. 15–26.

[4] Bouton, T., de Oliveira, D. C. B., Déharbe, D., and Fontaine, P. veriT: An open, trustable and efficient SMT-solver. In *CADE 2009* (2009), R. A. Schmidt, Ed., vol. 5663 of *LNCS*, Springer, pp. 151–156.

[5] Déharbe, D., Fontaine, P., and Woltzenlogel Paleo, B. Quantifier inference rules for SMT proofs. In *PxTP 2011* (2011), P. Fontaine and A. Stump, Eds., pp. 33–39.

[6] Fleury, M., and Schurr, H.-J. Reconstructing veriT proofs in Isabelle/HOL. In *Proceedings Sixth Workshop on Proof eXchange for Theorem Proving, Natal, Brazil, August 26, 2019* (2019), G. Reis and H. Barbosa, Eds., vol. 301 of *Electronic Proceedings in Theoretical Computer Science*, Open Publishing Association, pp. 36–50.