The Alethe Proof Format

A Speculative Specification and Reference

Haniel Barbosa[1] Mathias Fleury[2] Pascal Fontaine[3]
Hans-Jörg Schurr[4]

[1] Universidade Federal de Minas Gerais
[2] Johannes Kepler University Linz
[3] Université de Liège
[4] University of Lorraine, CNRS, Inria, and LORIA, Nancy

Contents

Version 0.1-17-g31b0d31 with comments. Typeset on November 30, 2021.

Foreword

This document is a speculative specification and reference of a proof format for SMT solvers. The format consists of a language to express proofs and a set of proof rules. On the one side, the language is inspired by natural-deduction and is based on the widely used SMT-LIB format. The language also includes a flexible mechanism to reason about bound variables which allows fine-grained preprocessing proofs. On the other side, the rules are structured around resolution and the introduction of theory lemmas, in the same way as CDCL($\mathcal{T}$)-based SMT solvers.

The specification is speculative in the sense that it is not yet cast in stone, but it will evolve over time. It emerged from a list of proof rules used by the SMT solver veriT collected in a document called "Proofonomicon". Following the fate presupposed by its name, it informally circulated among researchers interested in the proofs produced by veriT after a few months. We now polished this document and gave it a respectable name.

Instead of aiming for theoretical purity, our approach is pragmatic: the specification describes the format as it is in use right now. It will develop in parallel with practical support for the format within SMT solvers, proof checkers, and other tools. We believe it is not a perfect specification that fosters the adaption of a format, but great tooling. This document will be a guide to develop such tools.

Nevertheless, it not only serves as a norm to ensure compatibility between tools, it also allows us to uncover the unsatisfactory aspects that would otherwise be hidden deep within the nooks and crannies of solver and checker implementations. Every uncovered problem presents an opportunity to improve the format. The authors of this document overlap with the authors of those tools and we are committed to improve the tools, the format, and ultimately the specification together. This document is also an invitation to other researchers to join these efforts. To read the reference and provide feedback, or to even implement support for Alethe into their own tools. Please get in touch!

The authors in the summer of 2021.

1 Introduction

This document is a reference of the Alethe format. The format is designed to be a flexible format to represent unsatisfiability proofs generated by SMT solvers. The overall design follows a few core concepts, such as a natural-deduction style structure and rules generating and operating on ground first-order clauses. There are two parts in this document: the proof language and a proof calculus. Section 2 introduces the language. First informally, then formally. Section 3 discusses the core concepts behind the Alethe proof calculus. At the end of the document Section 4 presents a list of all proof rules used by SMT solvers supporting Alethe.

The semantics (Section 2.1) and concrete syntax (Section 2.2) are based on the SMT-LIB [3] format, widely used as the input and output standard for SMT solvers. Hence, Alethe's base logic is the many-sorted first-order logic of SMT-LIB. This document assumes the reader is familiar with the SMT-LIB standard.

The format is currently used by the SMT solver veriT [6]. 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. The Alethe proofs can be reconstructed by the `smt` tactic of the proof assistant Isabelle/HOL [10, 12], as well as by SMTCoq in the proof assistant Coq [9]. The SMT solver cvc5 (the successor of CVC4 [2]) supports Alethe experimentally as one of its multiple proof output formats.

In addition to this reference, the proof format has been discussed in past publications which provide valuable background information. The core of the format goes back to 2011 when two publications at the PxTP workshop outlined the fundamental ideas behind the format [4] and proposed rules for quantifier instantiation [8]. More recently the format has gained support for reasoning typically used for processing, such as Skolemization, substitutions, and other manipulations of bound variables [1].

2 The Alethe Language

This section provides an overview of the core concepts of the Alethe language and also introduces some notation used throughout this document. While the next section provides a formal definition of the language, this overview of the core concepts should be helpful for practitioners.

Multi-Sorted First-Order Logic. Many SMT solvers use the SMT-LIB language [3] as both its input and output language and Alethe builds on this language. This includes its multi-sorted first-order logic. The available sorts depend on the selected SMT-LIB theory/logic as well as on those defined by the user, but a distinguished **Bool** sort is always available.

In addition to the multi-sorted first-order logic used by SMT-LIB, Alethe also uses Hilbert's choice operator ε. 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. Alethe requires that ε is functional with respect to logical equivalence: if for two formulas φ, ψ that contain the free variable x, it holds that $(\forall x.\, \varphi \leftrightarrow \psi)$, then $(\varepsilon x.\, \varphi) \simeq (\varepsilon x.\, \psi)$ must also hold.

As a matter of notation, 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 ψ, or vice versa. To simplify the notation we will omit the sort of terms when possible.

Comment by Haniel Barbosa

This notation is clashing with the notation of sequences of symbols, like in $\bar{x}$ for $x_1, \dots, x_n$.

Maybe we could use an alternative notation for "normalization under double negation elimination"? Like $\llbracket\varphi\rrbracket_{\neg\neg}$? I find the use of "complementary literal" to refer this a bit confusing anyway. There are very few uses in the rules (only tautology, and/or simps), so we could even not introduce a notation and just refer to "normalized under double negation elimination".

Example 1. The following example shows a simple Alethe proof. It uses quantifier instantiation and resolution to show a contradiction. The sections below step-by-step describe the concepts necessary to understand the proof intuitively.

$$\begin{array}{lcr}
\rhd 1. & \forall x.\, P(x) & (\texttt{assume}) \\
\rhd 2. & \neg P(a) & (\texttt{assume}) \\
\rhd 3. & \neg(\forall x.\, P(x)) \lor P(a) & (\texttt{forall\_inst};\,;(x, a)) \\
\rhd 4. & \neg(\forall x.\, P(x)), P(a) & (\texttt{or}; 3) \\
\rhd 5. & \bot & (\texttt{resolution}; 1, 2, 4)
\end{array}$$

Steps. A proof in the Alethe language is an indexed list of steps. To mimic the concrete syntax we write a step in the form

$$c_1, \dots, c_k \;\rhd\; i.\;\; \varphi_1, \dots, \varphi_l \quad (\texttt{rule};\, p_1, \dots, p_n;\, a_1, \dots, a_m)$$

Each step has a unique index $i \in \mathbb{I}$, where $\mathbb{I}$ is a countable infinite set of valid indices. In the concrete syntax all SMT-LIB symbols are valid indices, but for examples we will use natural numbers. Furthermore, a step has a clause $\varphi_1, \dots, \varphi_l$ as its conclusion. If a step has the empty clause as its conclusion (i.e., $l = 0$) we will write $\bot$. While this muddles the water a bit with regards to steps which have the unit clause with the unit literal $\bot$ as their conclusion, it simplifies the notation. We will remark on the difference if it is relevant. The rule name `rule` is taken from a set of possible proof rules. Furthermore, each step has a possibly empty set of premises $\{p_1, \dots, p_n\}$ with $p_i \in \mathbb{I}$, and a rule-dependent and possibly empty list of arguments $[a_1, \dots, a_m]$. The list of premises only references earlier steps, such that the proof forms a directed acyclic graph. 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 list $c_1, \dots, c_k$ is the *context* of the step. Contexts have their own section below. Every proof ends with a step that has the empty clause as the conclusion and an empty context.

The example above consists of five steps. Step 4 and 5 use premises. Since step 3 introduces a tautology, it uses no premises. However, it uses arguments to express the substitution $x \mapsto a$ used to instantiate the quantifier. Step 4 translates the disjunction into a clause. In the example above, the contexts are all empty.

Assumptions. The `assume` command introduces a term as an assumption. The proof starts with a number of `assume` commands. Each such command corresponds to an input assertion. Additional assumptions can be introduced too. In this case each assumption must be discharged with an appropriate step. The rule `subproof` can be used to do so.

The example above uses two assumptions which are introduced in the first two steps.

Subproofs and Lemmas. Alethe uses subproofs to create new contexts

prove lemmas and to manipulate the context. To prove lemmas, a subproof can introduce local assumptions. The `subproof` *rule* discharges the local assumptions. From an assumption φ and a formula ψ proved by intermediate steps from φ, the `subproof` rule deduces the clause $\neg\varphi, \psi$ that discharges the local assumption φ. A `subproof` step can not use premise from a subproof nested within the current subproof.

Comment by Mathias Fleury

There are two ways to export one element from the subproof:

- rely on the order and take the last one
- add an explicit :premises in the conclusion stop to give the exported step

Subproofs are also used to manipulate the context. As the example below shows, within this document we denote subproofs by a frame around the rules within the subproof.

Example 2. This example show a contradiction proof for the formula $(2+2) \simeq 5$. The proof uses a subproof to prove the lemma $((2+2) \simeq 5) \Rightarrow 4 \simeq 5$.

$\triangleright$ 1.	$(2+2) \simeq 5$	(`assume`)
$\triangleright$ 2.	$(2+2) \simeq 5$	(`assume`)
$\triangleright$ 3.	$(2+2) \simeq 4$	(`plus_simplify`)
$\triangleright$ 4.	$4 \simeq 5$	(`trans`; $1, 2$)
$\triangleright$ 5.	$\neg((2+2) \simeq 5), 4 \simeq 5$	(`subproof`)
$\triangleright$ 6.	$(4 \simeq 5) \leftrightarrow \bot$	(`eq_simplify`)
$\triangleright$ 7.	$\neg((4 \simeq 5) \leftrightarrow \bot), \neg(4 \simeq 5), \bot$	(`equiv_pos2`)
$\triangleright$ 8.	$\bot$	(`resolution`; $1, 5, 7, 8$)

Contexts. A specialty of the Alethe proofs is the step context. The context is a possible empty list $[c_1, \dots, c_l]$, where c_i is either a variable or a variable-term tuple denoted $x_i \mapsto t_i$. In the first case, we say that c_i *fixes* its variable. Throughout this document Γ denotes an arbitrary context. Alethe contexts are a general mechanism to write substitutions and to change them by attaching new elements to the list.

Hence, every context Γ induces a substitution $\text{subst}(\Gamma)$. If Γ is the empty list, $\text{subst}(\Gamma)$ is the empty substitution, i.e, the identity function. When Γ ends in a mapping, the substitution is extended with this mapping: $\text{subst}([c_1, \ldots, c_{n-1}, x_n \mapsto t_n]) = \text{subst}([c_1, \ldots, c_{n-1}]) \circ [t_n/x_n]$. Finally, $\text{subst}([c_1, \ldots, c_{n-1}, x_n])$ is $\text{subst}([c_1, \ldots, c_{n-1}])$, but x_n maps to x_n. The last case fixes x_n and allows the context to shadow a previously defined substitution for x_n. The following example illustrates this idea:

$$\text{subst}([x \mapsto 7, x \mapsto g(x)]) = [g(7)/x]$$
$$\text{subst}([x \mapsto 7, x, x \mapsto g(x)]) = [g(x)/x]$$

Implicit Reordering of Equalities. In addition to the explicit steps, solvers might reorder equalities, i.e., apply symmetry of the equality predicate, without generating steps. The SMT solver veriT currently applies this liberty in a restricted form: equalities are only reordered when the term below the equality changes during proof search. One such case is the instantiation of universally quantified variables. If an instantiated variable appears below an equality, then the equality might have an arbitrary order after instantiation. Nevertheless, consumers of Alethe must consider the possible implicit reordering of equalities.

2.1 The Semantics of the Alethe Language

Most of the content is taken from the presentation and the correctness proof of the format [1].

2.1.1 Abstract Inference System

The inference rules used by our framework depend on a notion of *context* defined by the grammar

$$\Gamma \ ::= \ \varnothing \ \mid \ \Gamma,\, x \ \mid \ \Gamma,\, \bar{x}_n \mapsto \bar{s}_n$$

The empty context $\varnothing$ is also denoted by a blank. Each context entry either *fixes* a variable x or defines a *substitution* $\{\bar{x}_n \mapsto \bar{s}_n\}$. Any variables arising in the terms $\bar{s}_n$ will typically have been introduced in the context Γ on the left, but this is not required. If a context introduces the same variable several times, the rightmost entry shadows the others.

Abstractly, a context Γ fixes a set of variables and specifies a substitution $\mathit{subst}(\Gamma)$. The substitution is the identity for $\varnothing$ and is defined as follows in the other cases:

$$\mathit{subst}(\Gamma,\, x) = \mathit{subst}(\Gamma)[x \mapsto x] \qquad \mathit{subst}(\Gamma,\, \bar{x}_n \mapsto \bar{t}_n) = \mathit{subst}(\Gamma) \circ \{\bar{x}_n \mapsto \bar{t}_n\}$$

In the first equation, the $[x \mapsto x]$ update shadows any replacement of x induced by Γ. The examples below illustrate this subtlety:

$$\mathit{subst}(x \mapsto 7,\, x \mapsto \mathsf{g}(x)) = \{x \mapsto \mathsf{g}(7)\}$$
$$\mathit{subst}(x \mapsto 7,\, x,\, x \mapsto \mathsf{g}(x)) = \{x \mapsto \mathsf{g}(x)\}$$

We write $\Gamma(t)$ to abbreviate the capture-avoiding substitution $\mathit{subst}(\Gamma)(t)$.

Transformations of terms (and formulas) are justified by judgments of the form $\Gamma \ \rhd\ t \simeq u$, where Γ is a context, t is an unprocessed term, and u is the corresponding processed term. The free variables in t and u must appear in the context Γ. Semantically, the judgment expresses the equality of the terms $\Gamma(t)$ and u, universally quantified on variables fixed by Γ. Crucially, the substitution applies only on the left-hand side of the equality.

The inference rules for the transformations rely on equations that are presented here, then the rules are presented below, followed by explanations.

$$\models (\exists x.\, \varphi[x]) \implies \varphi[\varepsilon x.\, \varphi] \tag{ε_1}$$
$$\models (\forall x.\, \varphi \simeq \psi) \implies (\varepsilon x.\, \varphi) \simeq (\varepsilon x.\, \psi) \tag{ε_2}$$
$$\models (\text{let } \bar{x}_n \simeq \bar{s}_n \text{ in } t[\bar{x}_n]) \simeq t[\bar{s}_n] \tag{let}$$

The rules are:

$$\frac{}{\Gamma \ \rhd\ t \simeq u}\ \textsc{Taut}_{\mathcal{T}} \quad \text{if } \models_{\mathcal{T}} \Gamma(t) \simeq u$$

$$\frac{\Gamma \ \rhd\ s \simeq t \quad \Gamma \ \rhd\ t \simeq u}{\Gamma \ \rhd\ s \simeq u}\ \textsc{Trans} \quad \text{if } \Gamma(t) = t$$

$$\frac{\left(\Gamma \ \rhd\ t_i \simeq u_i\right)_{i=1}^{n}}{\Gamma \ \rhd\ \mathsf{f}(\bar{t}_n) \simeq \mathsf{f}(\bar{u}_n)}\ \textsc{Cong}$$

$$\frac{\Gamma, y, x \mapsto y \ \rhd\ \varphi \simeq \psi}{\Gamma \ \rhd\ (Qx.\, \varphi) \simeq (Qy.\, \psi)}\ \textsc{Bind} \quad \text{if } y \notin \mathit{FV}(Qx.\, \varphi) \cup V(\Gamma)$$

$$\frac{\Gamma, x \mapsto (\varepsilon x.\, \varphi) \ \rhd\ \varphi \simeq \psi}{\Gamma \ \rhd\ (\exists x.\, \varphi) \simeq \psi}\ \textsc{Sko}_{\exists} \qquad \frac{\Gamma, x \mapsto (\varepsilon x.\, \neg\, \varphi) \ \rhd\ \varphi \simeq \psi}{\Gamma \ \rhd\ (\forall x.\, \varphi) \simeq \psi}\ \textsc{Sko}_{\forall}$$

$$\frac{\left(\Gamma \ \rhd\ r_i \simeq s_i\right)_{i=1}^{n} \quad \Gamma, \bar{x}_n \mapsto \bar{s}_n \ \rhd\ t \simeq u}{\Gamma \ \rhd\ (\text{let } \bar{x}_n \simeq \bar{r}_n \text{ in } t) \simeq u}\ \textsc{Let} \quad \text{if } \Gamma(s_i) = s_i \text{ for all } i \in [n]$$

- $\textsc{Taut}_{\mathcal{T}}$ relies on an oracle $\models_{\mathcal{T}}$ to derive arbitrary lemmas in a theory $\mathcal{T}$. In practice, the oracle will produce some kind of certificate to justify the inference. An important special case, for which we use the name $\textsc{Refl}$, is syntactic equality up to renaming of bound variables; the side condition is then $\Gamma(t) =_{\alpha} u$. (We use $=_{\alpha}$ instead of $=$ because applying a substitution can rename bound variables.)

- $\textsc{Trans}$ needs the side condition because the term t appears both on the left-hand side of $\simeq$ (where it is subject to Γ's substitution) and on the right-hand side (where it is not). Without it, the two occurrences of t in the antecedent could denote different terms.

- $\textsc{Cong}$ can be used for any function symbol f, including the logical connectives.

- Bind is a congruence rule for quantifiers. The rule also justifies the renaming of the bound variable (from x to y). In the antecedent, the renaming is expressed by a substitution in the context. If $x = y$, the context is $\Gamma,\ x,\ x \mapsto x$, which has the same meaning as $\Gamma,\ x$. The side condition prevents an unwarranted variable capture: The new variable should not be a free variable in the formula where the renaming occurs ($y \notin FV(Qx.\, \varphi)$), and should be fresh in the context ($y \notin V(\Gamma)$, where $V(\Gamma)$ denotes the set of all variables occurring in Γ). In particular, y should not appear fixed or on either side of a substitution in the context.

- Sko$_\exists$ and Sko$_\forall$ exploit (ε_1) to replace a quantified variable with a suitable witness, simulating skolemization. We can think of the ε expression in each rule abstractly as a fresh function symbol that takes any fixed variables it depends on as arguments. In the antecedents, the replacement is performed by the context.

- Let exploits (let) to expand a 'let' expression. Again, a substitution is used. The terms $\bar{r}_n$ assigned to the variables $\bar{x}_n$ can be transformed into terms $\bar{s}_n$.

The antecedents of all the rules inspect subterms structurally, without modifying them. Modifications to the term on the left-hand side are delayed; the substitution is applied only in Taut. This is crucial to obtain compact proofs that can be checked efficiently. Some of the side conditions may look computationally expensive, but there are techniques to compute them fairly efficiently. Furthermore, by systematically renaming variables in Bind, we can satisfy most of the side conditions trivially.

The set of rules can be extended to cater for arbitrary transformations that can be expressed as equalities, using Hilbert choice to represent fresh symbols if necessary. The usefulness of Hilbert choice for proof reconstruction is well known [5, 7, 11], but we push the idea further and use it to simplify the inference system and make it more uniform.

Example 3. The following derivation tree justifies the expansion of a 'let' expression:

$$\dfrac{\dfrac{}{\rhd\ \mathsf{a} \simeq \mathsf{a}}\,\textsc{Cong} \qquad \dfrac{\dfrac{}{x \mapsto \mathsf{a}\ \ \rhd\ x \simeq \mathsf{a}}\,\textsc{Refl} \qquad \dfrac{}{x \mapsto \mathsf{a}\ \ \rhd\ x \simeq \mathsf{a}}\,\textsc{Refl}}{x \mapsto \mathsf{a}\ \ \rhd\ \mathsf{p}(x,x) \simeq \mathsf{p}(\mathsf{a},\mathsf{a})}\,\textsc{Cong}}{\rhd\ (\text{let } x \simeq \mathsf{a} \text{ in } \mathsf{p}(x,x)) \simeq \mathsf{p}(\mathsf{a},\mathsf{a})}\,\textsc{Let}$$

2.1.2 Correctness

Theorem 3.1 (Soundness of Inferences [1, Theorem 11]). If the judgment $\Gamma \quad \rhd\ t \simeq u$ is derivable using the original inference system with the theories $\mathcal{T}_1, \ldots, \mathcal{T}_n$, then $\models_{\mathcal{T}} \Gamma(t) \simeq u$ with $\mathcal{T} = \mathcal{T}_1 \cup \cdots \cup \mathcal{T}_n \cup \simeq \cup\ \varepsilon \cup \text{let}$.

Proof. See [1, Theorem 11] □

```
 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 instantiation and resolution steps (line 10–15)

2.1.3 More Concrete Rules

The high-level rules presented in the previous paragraph are useful for presentation purpose, but they are hard to check. We specialize them by reducing applicability. In particular the Taut rule is very hard to check and is specialized.

2.2 The Syntax

The concrete text representation of the Alethe proofs is based on the SMT-LIB standard. Figure 1 shows an exemplary proof as printed by veriT with light edits for readability. The format follows the SMT-LIB standard when possible.

Figure 2 shows the grammar of the proof text. 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.

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:

[1]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_annotation⟩? ⟨args_annotation⟩
⟨premises_annotation⟩ ::= :premises ( ⟨symbol⟩+)
 ⟨args_annotation⟩   ::=  :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

assumptions are neither tautological nor derived from premises. The `step` command, on the other hand, introduces a derived or tautological formula. 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. This operator, however, 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.

Table 1: Special Rules

Rule	Description
`assume` (1)	Repetition of an input assumption.
`subproof` (9)	Concludes a subproof and discharges local assumptions.

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 an 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.

Comment by Haniel Barbosa

Bruno pointed out what looks to me like an issue with how we are currently printing variable arguments for subproofs whose anchors introduce bound variables or substitutions for variables: without giving the types of the variables, it's not possible to know their types without some deep forward looking into next proof steps. I think then that the types should be printed for bound variables and variables occurring in substitutions.

FWIW, it's very simple to change veriT to do this. I did so in a branch for Bruno so that the checker he's implementing does not need to implement more complicated solutions to figure out the type of the variables.

I'm curious though: how is this handled in the Isabelle/HOL reconstruction?

Comment by Hans-Jörg Schurr

Indeed the substitution induced by an anchor should always be fully sorted. I think, the sorts should also be checked during proof checking.

What syntax did you choose to print the sorts? I would like to understand better when it's necessary to print a sort. Certainly, one can often directly deduce the sort of a variable from the substitute term. On the other hand, it is certainly necessary to print the sorts of fixed variables. Furthermore, constructions such as 'x1:=x2, x2:=x1' need sort annotations. In theory we could probably eliminate the last case. Is there a case where the substitute is a complex term, but it is still not clear what sort it is?

For Isabelle/HOL it's not a problem since its reconstruction looks ahead to the conclusion of the 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.

Comment by Hans-Jörg Schurr

There is an open question with regard to the best way to print subproofs:

There is an implicit relation between the last step of the subproof and the step concluding the subproof. → what if we would, for some reason, have some crap after the last step of the subproof? We cannot accommodate this yet.

There are multiple solutions to solve this implicit dependency.

• One is to give the final step of the subproof as a premise to the step concluding the subproof. → downside: normally it's forbidden to use steps from within a deeper proof

• We could have the conclusion of the subproof already in the anchor. or we could turn the current structure upside down → downside: a solver that just dumps out steps can not no always know the conclusion when opening a subproof. For example, when simplifications are applied.

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 any 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 expanded.

3 The Alethe proof calculus

Together with the language, the Alethe format also comes with a set of proof rule. Section 4 gives a full list of all proof rules. Currently, the proof rules correspond to the

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

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

rules used by the veriT solver. For the rest of this section, we will discuss some general concepts related to the rules.

Tautologous Rules and Simple Deduction. Most rules 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 $\varphi_1 \Rightarrow \varphi_2$ it deduces $\neg\varphi_1 \vee \varphi_2$.

Resolution. CDCL(T)-based SMT solvers, and especially their SAT solvers, are fundamentally based on clauses. Hence, Alethe also uses clauses. Nevertheless, since SMT solvers do not enforce a strict clausal normal-form, ordinary disjunction is also used. Keeping clauses and disjunctions distinct, simplifies rule checking. For example, in the case of resolution there is a clear distinction between unit clauses where the sole literal starts with a disjunction, and non-unit clauses. The syntax for clauses uses the `cl` operator, while disjunctions use the standard SMT-LIB `or` operator. The `or` *rule* is responsible for converting disjunctions into clauses.

The Alethe proofs 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 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.

Comment by Hans-Jörg Schurr

We have to clarify that resolution counts the number of leading negations to determine polarity. This is important for double negation elimination.

Furthermore, as Pascal noted, we should also fold the two rules into one and use an attribute to distinguish the two cases.

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 $x_1, \ldots x_n$, and t_i is a new variable free term with the same sort as x_i for each i.

The arguments of a `forall_inst` step is the list $(x_1, t_1), \ldots, (x_n, t_n)$. While this information can be recovered from the term, providing it explicitly helps reconstruction because the implicit reordering of equalities (see below) obscures which terms have been used as instances. Existential quantifiers are handled by Skolemization.

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 $\neg\varphi_1, \neg\varphi_2, \ldots, \neg\varphi_n$ where the φ_i are linear (in)equalities. Checking the validity of these formulas amounts to checking the unsatisfiability of the system of linear equations $\varphi_1, \varphi_2, \ldots, \varphi_n$. The annotation of an `la_generic` step contains a coefficient for each (in)equality. The result of forming the linear combination of the literals with the coefficients is a trivial inequality between constants.

Example 4. The following example is the proof for the unsatisfiability of $(x+y<1) \vee (3<x)$, $x \simeq 2$, and $0 \simeq y$.

$$\begin{array}{lcr}
\rhd 1. & (3 < x) \vee (x+y<1) & (\texttt{assume}) \\
\rhd 2. & x \simeq 2 & (\texttt{assume}) \\
\rhd 3. & 0 \simeq y & (\texttt{assume}) \\
\rhd 4. & (3<x), (x+y<1) & (\texttt{or}; 1) \\
\rhd 5. & \neg(3<x), \neg(x \simeq 2) & (\texttt{la\_generic};\ ; 1.0, 1.0) \\
\rhd 6. & \neg(3<x) & (\texttt{resolution}; 2, 5) \\
\rhd 7. & x+y<1 & (\texttt{resolution}; 4, 6) \\
\rhd 8. & \neg(x+y<1), \neg(x \simeq 2) \vee \neg(0 \simeq y) & (\texttt{la\_generic};\ ; 1.0, -1.0, 1.0) \\
\rhd 9. & \bot & (\texttt{resolution};\ 8, 7, 2, 3)
\end{array}$$

Skolemization and Other Preprocessing Steps. 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 variable is quantified.

$$\begin{array}{rlcr}
\Gamma, x \mapsto (\varepsilon x.\varphi) \ \rhd & n. & \varphi \simeq \psi & (\ldots) \\
\Gamma \ \rhd & m. & (\exists x.\varphi) \simeq \psi & (\texttt{sko\_ex};\ n)
\end{array}$$

Example 5. To illustrate how such a rule is applied, consider the following example taken from [1]. 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 $\varepsilon x.\neg\varphi$.

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

3.1 Classifications of the Rules

Section 4 provides a list of all proof rules supported by Alethe. To make this long list more accessible, this section first lists multiple classes of proof rules. The classes are not mutually exclusive: for example, the `la_generic` rule is both a linear arithmetic rule and introduces a tautology. Table 2.2 lists rules that serve a special purpose. Table 3 lists all rules that introduce tautologies. That is, regular rules that do not use premises.

Within the tables, the number in brackets corresponds to the number of the rule in Section 4.

Table 2: Resolution and Related Rules

Rule	Description
`resolution` (6)	Chain resolution of two or more clauses.
`th_resolution` (5)	As `resoltuion`, but used by theory solvers.
`tautology` (7)	Simplification of tautological clauses to $\top$.
`contraction` (8)	Removal of duplicated literals.

Table 3: Rules Introducing Tautologies

Rule	Conclusion
`true` (2)	$\top$
`false` (3)	$\neg\bot$
`not_not` (4)	$\neg(\neg\neg\varphi), \varphi$
`la_generic` (10)	Tautologous disjunction of linear inequalities.
`lia_generic` (11)	Tautologous disjunction of linear integer inequalities.
`la_disequality` (12)	$t_1 \simeq t_2 \vee \neg(t_1 \leq t_2) \vee \neg(t_2 \leq t_1)$
`la_totality` (13)	$t_1 \leq t_2 \vee t_2 \leq t_1$
`la_tautology` (14)	A trivial linear tautology.
`forall_inst` (18)	Quantifier instantiation.
`refl` (19)	Reflexivity after applying the context.
`eq_reflexive` (22)	$t \simeq t$ without context.
`eq_transitive` (23)	$\neg(t_1 \simeq t_2), \dots, \neg(t_{n-1} \simeq t_n), t_1 \simeq t_n$
`eq_congruent` (24)	$\neg(t_1 \simeq u_1), \dots, \neg(t_n \simeq u_n), f(t_1, \dots, t_n) \simeq f(u_1, \dots, u_n)$
`eq_congruent_pred` (25)	$\neg(t_1 \simeq u_1), \dots, \neg(t_n \simeq u_n), P(t_1, \dots, t_n) \simeq P(u_1, \dots, u_n)$
`qnt_cnf` (26)	Clausification of a quantified formula.
`and_pos` (42)	$\neg(\varphi_1 \wedge \cdots \wedge \varphi_n), \varphi_k$
`and_neg` (43)	$(\varphi_1 \wedge \cdots \wedge \varphi_n), \neg\varphi_1, \dots, \neg\varphi_n$
`or_pos` (44)	$\neg(\varphi_1 \vee \cdots \vee \varphi_n), \varphi_1, \dots, \varphi_n$
`or_neg` (45)	$(\varphi_1 \vee \cdots \vee \varphi_n), \neg\varphi_k$
`xor_pos1` (46)	$\neg(\varphi_1 \operatorname{xor} \varphi_2), \varphi_1, \varphi_2$
`xor_pos2` (47)	$\neg(\varphi_1 \operatorname{xor} \varphi_2), \neg\varphi_1, \neg\varphi_2$
`xor_neg1` (48)	$\varphi_1 \operatorname{xor} \varphi_2, \varphi_1, \neg\varphi_2$
`xor_neg2` (49)	$\varphi_1 \operatorname{xor} \varphi_2, \neg\varphi_1, \varphi_2$
`implies_pos` (50)	$\neg(\varphi_1 \rightarrow \varphi_2), \neg\varphi_1, \varphi_2$
`implies_neg1` (51)	$\varphi_1 \rightarrow \varphi_2, \varphi_1$
`implies_neg2` (52)	$\varphi_1 \rightarrow \varphi_2, \neg\varphi_2$
`equiv_pos1` (53)	$\neg(\varphi_1 \leftrightarrow \varphi_2), \varphi_1, \neg\varphi_2$
`equiv_pos2` (54)	$\neg(\varphi_1 \leftrightarrow \varphi_2), \neg\varphi_1, \varphi_2$
`equiv_neg1` (55)	$\varphi_1 \leftrightarrow \varphi_2, \neg\varphi_1, \neg\varphi_2$
`equiv_neg2` (56)	$\varphi_1 \leftrightarrow \varphi_2, \varphi_1, \varphi_2$
`ite_pos1` (59)	$\neg(\operatorname{ite} \varphi_1\ \varphi_2\ \varphi_3), \varphi_1, \varphi_3$
`ite_pos2` (60)	$\neg(\operatorname{ite} \varphi_1\ \varphi_2\ \varphi_3), \neg\varphi_1, \varphi_2$
`ite_neg1` (61)	$\operatorname{ite} \varphi_1\ \varphi_2\ \varphi_3, \varphi_1, \neg\varphi_3$

`ite_neg2` (62)	ite φ_1 φ_2 $\varphi_3, \neg\varphi_1, \neg\varphi_2$
`connective_def` (65)	Definition of the boolean connectives.
`and_simplify` (66)	Simplification of a conjunction.
`or_simplify` (67)	Simplification of a disjunction.
`not_simplify` (68)	Simplification of a boolean negation.
`implies_simplify` (69)	Simplification of an implication.
`equiv_simplify` (70)	Simplification of an equivalence.
`bool_simplify` (71)	Simplification of combined boolean connectives.
`ac_simp` (72)	Flattening of nested $\vee$ or $\wedge$.
`ite_simplify` (73)	Simplification of if-then-else.
`qnt_simplify` (74)	Simplification of constant quantified formulas.
`qnt_join` (76)	Joining of consecutive quantifiers.
`qnt_rm_unused` (77)	Removal of unused quantified variables.
`eq_simplify` (78)	Simplification of equality.
`div_simplify` (79)	Simplification of division.
`prod_simplify` (80)	Simplification of products.
`unary_minus_simplify` (81)	Simplification of the unary minus.
`minus_simplify` (82)	Simplification of subtractions.
`sum_simplify` (83)	Simplification of sums.
`comp_simplify` (84)	Simplification of arithmetic comparisons.
`distinct_elim` (86)	Elimination of the distinction predicate.
`la_rw_eq` (87)	$(t \simeq u) \simeq (t \leq u \wedge u \leq t)$
`nary_elim` (88)	Replace n-ary operators with binary application.

Table 4: Linear Arithmetic Rules

Rule	Description
`la_generic` (10)	Tautologous disjunction of linear inequalities.
`lia_generic` (11)	Tautologous disjunction of linear integer inequalities.
`la_disequality` (12)	$t_1 \simeq t_2 \vee \neg(t_1 \leq t_2) \vee \neg(t_2 \leq t_1)$
`la_totality` (13)	$t_1 \leq t_2 \vee t_2 \leq t_1$
`la_tautology` (14)	A trivial linear tautology.
`la_rw_eq` (87)	$(t \simeq u) \simeq (t \leq u \wedge u \leq t)$
`div_simplify` (79)	Simplification of division.
`prod_simplify` (80)	Simplification of products.
`unary_minus_simplify` (81)	Simplification of the unary minus.
`minus_simplify` (82)	Simplification of subtractions.
`sum_simplify` (83)	Simplification of sums.
`comp_simplify` (84)	Simplification of arithmetic comparisons.

,,,,,,,,

,,,,,,,,,,,,,,,

4 List of Proof Rules

The following lists all rules of Alethe.

Rule 1: assume

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

where φ corresponds to a formula asserted in the input problem up to the orientation of equalities.

Remark. This rule can not be used by the `(step ...)` command. Instead it corresponds to the dedicated `(assume ...)` command.

Rule 2: true

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

Rule 3: false

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

Rule 4: not_not

$$\triangleright\, i. \qquad \neg(\neg\neg\varphi), \varphi \qquad (\texttt{not\_not})$$

Remark. This rule is useful to remove double negations from a clause by resolving a clause with the double negation on φ.

Rule 5: th_resolution

This rule is the resolution of two or more clauses.

$$\begin{array}{lll}
\triangleright\, i_1. & \varphi^1_1, \dots, \varphi^1_{k^1} & (\dots) \\
 & \vdots & \\
\triangleright\, i_n. & \varphi^n_1, \dots, \varphi^n_{k^n} & (\dots) \\
\triangleright\ \ j. & \varphi^{r_1}_{s_1}, \dots, \varphi^{r_m}_{s_m} & (\texttt{th\_resolution}; i_1, \dots, i_n)
\end{array}$$

where $\varphi^{r_1}_{s_1}, \dots, \varphi^{r_m}_{s_m}$ are from φ^i_j and are the result of a chain of predicate resolution steps on the clauses of i_1 to 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.

Rule 6: 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.

Rule 7: tautology

$$\begin{array}{lcr} \rhd\, i. & \varphi_1,\dots,\varphi_i,\dots,\varphi_j,\dots,\varphi_n & (\dots) \\ \rhd\, j. & \top & (\texttt{tautology}; i) \end{array}$$

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

Rule 8: contraction

$$\begin{array}{lcr} \rhd\, i. & \varphi_1,\dots,\varphi_n & (\dots) \\ & \vdots & \\ \rhd\, j. & \varphi_{k_1},\dots,\varphi_{k_m} & (\texttt{contraction}; i) \end{array}$$

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 removes duplicated literals.

Rule 9: 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.

$$\begin{array}{|lcr} \rhd\, i_1. & \psi_1 & (\texttt{input}) \\ & \vdots & \\ \rhd\, i_n. & \psi_n & (\texttt{input}) \\ & \vdots & \\ \rhd\ \ j. & \varphi & (\dots) \\ \hline \rhd\, k. & \neg\psi_1,\dots,\neg\psi_n,\varphi & (\texttt{subproof}) \end{array}$$

Rule 10: 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 $l_1,\dots,l_n$ be linear inequalities and $a_1,\dots,a_n$ rational numbers, then a `la_generic` step has the form

$$\rhd\, i.\quad \varphi_1,\dots,\varphi_o \quad (\texttt{la\_generic};; a_1,\dots,a_o)$$

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 $\varphi := s_1 > s_2$. This negates the literal.

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 9.

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 9: Strengthening rules for `la_generic`.

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

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

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 such that the sum on the left-hand side is 0 and the right-hand side is > 0 (or ≥ 0 if $\bowtie$ *is* $>$).

Example 10.1. 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 10.2. 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 $\frac{1}{4} \geq 0$.

Rule 11: lia_generic

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

$$\triangleright\, i. \qquad \varphi_1, \dots, \varphi_n \qquad (\texttt{lia\_generic})$$

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.

Remark. Since this rule can introduce a disjunction of arbitrary linear integer inequalities without any additional hints, proof checking can be NP-hard. Hence, this rule should be avoided when possible.

Rule 12: la_disequality

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

Rule 13: la_totality

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

Rule 14: la_tautology

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

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

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:

$$\triangleright\, i. \qquad \varphi_1 \vee \varphi_2 \qquad (\texttt{la\_tautology})$$

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`.

Rule 15: bind

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

$$\begin{array}{|lcr}
 & \vdots & \\
\Gamma, y_1, \dots, y_n, x_1 \mapsto y_1, \dots, x_n \mapsto y_n, \;\triangleright j. & \varphi \leftrightarrow \varphi' & (\dots) \\
\hline
\Gamma \;\triangleright k. & \forall x_1, \dots, x_n . \varphi \leftrightarrow \forall y_1, \dots, y_n . \varphi' & (\texttt{bind})
\end{array}$$

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

Rule 16: sko_ex

The `sko_ex` rule skolemizes existential quantifiers.

$$\begin{array}{|lcr}
 & \vdots & \\
\Gamma, x_1 \mapsto \varepsilon_1, \dots, x_n \mapsto \varepsilon_n \;\triangleright j. & \varphi \leftrightarrow \psi & (\dots) \\
\hline
\Gamma \;\triangleright k. & \exists x_1, \dots, x_n . \varphi \leftrightarrow \psi & (\texttt{sko\_ex})
\end{array}$$

where ε_i stands for $\varepsilon x_i . (\exists x_{i+1}, \dots, x_n . \varphi[x_1 \mapsto \varepsilon_1, \dots, x_{i-1} \mapsto \varepsilon_{i-1}])$.

Rule 17: sko_forall

The `sko_forall` rule skolemizes universal quantifiers.

$$\begin{array}{|lcr}
 & \vdots & \\
\Gamma, x_1 \mapsto (\varepsilon x_1 . \neg\varphi), \dots, x_n \mapsto (\varepsilon x_n . \neg\varphi) \;\triangleright j. & \varphi \leftrightarrow \psi & (\dots) \\
\hline
\Gamma \;\triangleright k. & \forall x_1, \dots, x_n . \varphi \leftrightarrow \psi & (\texttt{sko\_forall})
\end{array}$$

Rule 18: forall_inst

$$\begin{array}{lll}
\triangleright i. & \neg(\forall x_1, \dots, x_n . P) \vee & (\texttt{forall\_inst}; ; (x_{k_1}, t_{k_1}), \dots, (x_{k_n}, t_{k_n})) \\
 & P[t_1/x_1] \dots [t_n/x_n] &
\end{array}$$

where $k_1, \dots, k_n$ is a permutation of $1, \dots, n$ 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`.

Rule 19: refl

Either

$$\Gamma \;\triangleright j. \qquad t_1 \simeq t_2 \qquad (\texttt{refl})$$

or

$$\Gamma \;\triangleright j. \qquad \varphi_1 \leftrightarrow \varphi_2 \qquad (\texttt{refl})$$

where, if $\sigma = \text{subst}(\Gamma)$, the terms $\varphi_1\sigma$ and φ_2 (the formulas $P_1\sigma$ and P_2) are syntactically equal up to the orientation of equalities.

Remark. This is the only rule that requires the application of the context.

Rule 20: trans
Either

$$\begin{array}{lllr}
\Gamma & \rhd i_1. & t_1 \simeq t_2 & (\dots) \\
\Gamma & \rhd i_2. & t_2 \simeq t_3 & (\dots) \\
\Gamma & \rhd i_n. & t_n \simeq t_{n+1} & (\dots) \\
\Gamma & \rhd\ j. & t_1 \simeq t_{n+1} & (\texttt{trans}; i_1, \dots, i_n, j)
\end{array}$$

or

$$\begin{array}{lllr}
\Gamma & \rhd i_1. & \varphi_1 \leftrightarrow \varphi_2 & (\dots) \\
\Gamma & \rhd i_2. & \varphi_2 \leftrightarrow \varphi_3 & (\dots) \\
\Gamma & \rhd i_n. & \varphi_n \leftrightarrow \varphi_{n+1} & (\dots) \\
\Gamma & \rhd\ j. & \varphi_1 \leftrightarrow \varphi_{n+1} & (\texttt{trans}; i_1, \dots, i_n, j)
\end{array}$$

Comment by Mathias Fleury

The `trans` rules comes in three flavors that can be distinguished by the attribute:

ordered and oriented the equalities are given in the correct order and are oriented in the right direction;

ordered and unoriented the equalities are given in the correct order, but the equalitis like "$t_1 \simeq t_2$" can be used as "$t_1 \simeq t_2$" or "$t_2 \simeq t_1$";

unordered and unoriented the equalities are not ordered either.

Rule 21: cong
Either

$$\begin{array}{lllr}
\Gamma & \rhd i_1. & t_1 \simeq u_1 & (\dots) \\
\Gamma & \rhd i_n. & t_n \simeq u_n & (\dots) \\
\Gamma & \rhd\ j. & \mathrm{f}(t_1, \dots, t_n) \simeq \mathrm{f}(u_1, \dots, u_n) & (\texttt{cong}; i_1, \dots, i_n)
\end{array}$$

where f is an n-ary function symbol, or

$$\begin{array}{lllr}
\Gamma & \rhd i_1. & \varphi_1 \simeq \psi_1 & (\dots) \\
\Gamma & \rhd i_n. & \varphi_n \simeq \psi_n & (\dots) \\
\Gamma & \rhd\ j. & \mathrm{P}(\varphi_1, \dots, \varphi_n) \leftrightarrow \mathrm{P}(\psi_1, \dots, \psi_n) & (\texttt{cong}; i_1, \dots, i_n)
\end{array}$$

where P is an n-ary predicate symbol.

Rule 22: eq_reflexive

$$\rhd i. \qquad t \simeq t \qquad (\texttt{eq\_reflexive})$$

Rule 23: eq_transitive

$$\rhd i. \qquad \neg(t_1 \simeq t_2), \dots, \neg(t_{n-1} \simeq t_n), t_1 \simeq t_n \qquad (\texttt{eq\_transitive})$$

Rule 24: eq_congruent

$$\rhd i. \qquad \neg(t_1 \simeq u_1), \dots, \neg(t_n \simeq u_n), f(t_1, \dots, t_n) \simeq f(u_1, \dots, u_n) \qquad (\texttt{eq\_congruent})$$

Rule 25: eq_congruent_pred

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

Rule 26: qnt_cnf

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

This rule expresses 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 $x_{k_1}, \dots, x_{k_m}$ are a permutation of $x_1, \dots, x_n$ 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:

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

Remark. This is a placeholder rule that combines the many steps done during clausification.

Rule 27: and

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

$\triangleright\, j.$ φ_i (`and`; i)

Rule 28: not_or

▷ $i.$	$\neg(\varphi_1 \lor \cdots \lor \varphi_n)$	$(\dots)$
▷ $j.$	$\neg\varphi_i$	$(\texttt{not\_or}; i)$

Rule 29: or

▷ $i.$	$\varphi_1 \lor \cdots \lor \varphi_n$	$(\dots)$
▷ $j.$	$\varphi_1, \dots, \varphi_n$	$(\texttt{or}; i)$

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

Example 29.1. 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))
```

Rule 30: not_and

▷ $i.$	$\neg(\varphi_1 \land \cdots \land \varphi_n)$	$(\dots)$
▷ $j.$	$\neg\varphi_1, \dots, \neg\varphi_n$	$(\texttt{not\_and}; i)$

Rule 31: xor1

▷ $i.$	$\mathrm{xor}\, \varphi_1\ \varphi_2$	$(\dots)$
▷ $j.$	φ_1, φ_2	$(\texttt{xor1}; i)$

Rule 32: xor2

▷ $i.$	$\mathrm{xor}\, \varphi_1\ \varphi_2$	$(\dots)$
▷ $j.$	$\neg\varphi_1, \neg\varphi_2$	$(\texttt{xor2}; i)$

Rule 33: not_xor1

▷ $i.$	$\neg(\mathrm{xor}\, \varphi_1\ \varphi_2)$	$(\dots)$
▷ $j.$	$\varphi_1, \neg\varphi_2$	$(\texttt{not\_xor1}; i)$

Rule 34: not_xor2

▷ $i.$	$\neg(\mathrm{xor}\, \varphi_1\ \varphi_2)$	$(\dots)$
▷ $j.$	$\neg\varphi_1, \varphi_2$	$(\texttt{not\_xor2}; i)$

Rule 35: implies

▷ $i.$	$\varphi_1 \rightarrow \varphi_2$	$(\dots)$
▷ $j.$	$\neg\varphi_1, \varphi_2$	$(\texttt{implies}; i)$

Rule 36: not_implies1

$\triangleright\, i.$	$\neg(\varphi_1 \rightarrow \varphi_2)$	$(\ldots)$
$\triangleright\, j.$	φ_1	(`not_implies1`; i)

Rule 37: not_implies2

$\triangleright\, i.$	$\neg(\varphi_1 \rightarrow \varphi_2)$	$(\ldots)$
$\triangleright\, j.$	$\neg\varphi_2$	(`not_implies2`; i)

Rule 38: equiv1

$\triangleright\, i.$	$\varphi_1 \leftrightarrow \varphi_2$	$(\ldots)$
$\triangleright\, j.$	$\neg\varphi_1, \varphi_2$	(`equiv1`; i)

Rule 39: equiv2

$\triangleright\, i.$	$\varphi_1 \leftrightarrow \varphi_2$	$(\ldots)$
$\triangleright\, j.$	$\varphi_1, \neg\varphi_2$	(`equiv2`; i)

Rule 40: not_equiv1

$\triangleright\, i.$	$\neg(\varphi_1 \leftrightarrow \varphi_2)$	$(\ldots)$
$\triangleright\, j.$	φ_1, φ_2	(`not_equiv1`; i)

Rule 41: not_equiv2

$\triangleright\, i.$	$\neg(\varphi_1 \leftrightarrow \varphi_2)$	$(\ldots)$
$\triangleright\, j.$	$\neg\varphi_1, \neg\varphi_2$	(`not_equiv2`; i)

Rule 42: and_pos

$\triangleright\, i.$	$\neg(\varphi_1 \land \cdots \land \varphi_n), \varphi_k$	(`and_pos`)

with $1 \leq k \leq n$.

Rule 43: and_neg

$\triangleright\, i.$	$(\varphi_1 \land \cdots \land \varphi_n), \neg\varphi_1, \ldots, \neg\varphi_n$	(`and_neg`)

Rule 44: or_pos

$\triangleright\, i.$	$\neg(\varphi_1 \lor \cdots \lor \varphi_n), \varphi_1, \ldots, \varphi_n$	(`or_pos`)

Rule 45: or_neg

$\triangleright\, i.$	$(\varphi_1 \lor \cdots \lor \varphi_n), \neg\varphi_k$	(`or_neg`)

with $1 \leq k \leq n$.

Rule 46: xor_pos1

$\triangleright\, i.$	$\neg(\varphi_1 \operatorname{xor} \varphi_2), \varphi_1, \varphi_2$	(`xor_pos1`)

Rule 47: xor_pos2

$\triangleright\, i.$ $\neg(\varphi_1 \operatorname{xor} \varphi_2), \neg\varphi_1, \neg\varphi_2$ (`xor_pos2`)

Rule 48: xor_neg1

$\triangleright\, i.$ $\varphi_1 \operatorname{xor} \varphi_2, \varphi_1, \neg\varphi_2$ (`xor_neg1`)

Rule 49: xor_neg2

$\triangleright\, i.$ $\varphi_1 \operatorname{xor} \varphi_2, \neg\varphi_1, \varphi_2$ (`xor_neg2`)

Rule 50: implies_pos

$\triangleright\, i.$ $\neg(\varphi_1 \rightarrow \varphi_2), \neg\varphi_1, \varphi_2$ (`implies_pos`)

Rule 51: implies_neg1

$\triangleright\, i.$ $\varphi_1 \rightarrow \varphi_2, \varphi_1$ (`implies_neg1`)

Rule 52: implies_neg2

$\triangleright\, i.$ $\varphi_1 \rightarrow \varphi_2, \neg\varphi_2$ (`implies_neg2`)

Rule 53: equiv_pos1

$\triangleright\, i.$ $\neg(\varphi_1 \leftrightarrow \varphi_2), \varphi_1, \neg\varphi_2$ (`equiv_pos1`)

Rule 54: equiv_pos2

$\triangleright\, i.$ $\neg(\varphi_1 \leftrightarrow \varphi_2), \neg\varphi_1, \varphi_2$ (`equiv_pos2`)

Rule 55: equiv_neg1

$\triangleright\, i.$ $\varphi_1 \leftrightarrow \varphi_2, \neg\varphi_1, \neg\varphi_2$ (`equiv_neg1`)

Rule 56: equiv_neg2

$\triangleright\, i.$ $\varphi_1 \leftrightarrow \varphi_2, \varphi_1, \varphi_2$ (`equiv_neg2`)

Rule 57: ite1

$\triangleright\, i.$ $\operatorname{ite} \varphi_1\ \varphi_2\ \varphi_3$ $(\ldots)$
$\triangleright\, j.$ φ_1, φ_3 (`ite1`; i)

Rule 58: ite2

$$\begin{array}{lcr} \rhd i. & \text{ite}\,\varphi_1\ \varphi_2\ \varphi_3 & (\dots) \\ \rhd j. & \neg\varphi_1, \varphi_2 & (\texttt{ite2}; i) \end{array}$$

Rule 59: ite_pos1

$$\begin{array}{lcr} \rhd i. & \neg(\text{ite}\,\varphi_1\ \varphi_2\ \varphi_3), \varphi_1, \varphi_3 & (\texttt{ite\_pos1}) \end{array}$$

Rule 60: ite_pos2

$$\begin{array}{lcr} \rhd i. & \neg(\text{ite}\,\varphi_1\ \varphi_2\ \varphi_3), \neg\varphi_1, \varphi_2 & (\texttt{ite\_pos2}) \end{array}$$

Rule 61: ite_neg1

$$\begin{array}{lcr} \rhd i. & \text{ite}\,\varphi_1\ \varphi_2\ \varphi_3, \varphi_1, \neg\varphi_3 & (\texttt{ite\_neg1}) \end{array}$$

Rule 62: ite_neg2

$$\begin{array}{lcr} \rhd i. & \text{ite}\,\varphi_1\ \varphi_2\ \varphi_3, \neg\varphi_1, \neg\varphi_2 & (\texttt{ite\_neg2}) \end{array}$$

Rule 63: not_ite1

$$\begin{array}{lcr} \rhd i. & \neg(\text{ite}\,\varphi_1\ \varphi_2\ \varphi_3) & (\dots) \\ \rhd j. & \varphi_1, \neg\varphi_3 & (\texttt{not\_ite1}; i) \end{array}$$

Rule 64: not_ite2

$$\begin{array}{lcr} \rhd i. & \neg(\text{ite}\,\varphi_1\ \varphi_2\ \varphi_3) & (\dots) \\ \rhd j. & \neg\varphi_1, \neg\varphi_2 & (\texttt{not\_ite2}; i) \end{array}$$

Rule 65: connective_def

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

$$\begin{array}{llcr} \Gamma & \rhd i. & \varphi_1\,\text{xor}\,\varphi_2 \leftrightarrow (\neg\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \neg\varphi_2) & (\texttt{connective\_def}) \\ \Gamma & \rhd i. & \varphi_1 \leftrightarrow \varphi_2 \leftrightarrow (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) & (\texttt{connective\_def}) \\ \Gamma & \rhd i. & \text{ite}\,\varphi_1\ \varphi_2\ \varphi_3 \leftrightarrow (\varphi_1 \rightarrow \varphi_2) \wedge (\neg\varphi_1 \rightarrow \varphi_3) & (\texttt{connective\_def}) \\ \Gamma & \rhd i. & \forall \mathrm{x}_1, \dots, \mathrm{x}_\mathrm{n}.\, \varphi \leftrightarrow \neg\exists \mathrm{x}_1, \dots, \mathrm{x}_\mathrm{n}.\, \neg\varphi & (\texttt{connective\_def}) \end{array}$$

Rule 66: and_simplify

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

$$\begin{array}{llcr} \Gamma & \rhd i. & \varphi_1 \wedge \dots \wedge \varphi_n \leftrightarrow \psi & (\texttt{and\_simplify}) \end{array}$$

where ψ is 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 $\varphi_i = \bar{\varphi}_j$

Rule 67: or_simplify
This rule simplifies an $\vee$ term by applying equivalent transformations as long as possible. Hence, the general form is

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

where ψ is 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 $\varphi_i = \bar{\varphi}_j$

Rule 68: not_simplify
This rule simplifies an $\neg$ term by applying equivalent transformations as long as possible. Hence, the general form is

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

where ψ is the transformed term.
The possible transformations are:

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

Rule 69: implies_simplify
This rule simplifies an $\rightarrow$ term by applying equivalent transformations as long as possible. Hence, the general form is

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

where ψ is the transformed term.
The possible transformations are:

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

Rule 70: equiv_simplify
This rule simplifies an $\leftrightarrow$ term by applying equivalent transformations as long as possible. Hence, the general form is

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

where ψ is the transformed term.
The possible transformations are:

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

Rule 71: bool_simplify
This rule simplifies a boolean term by applying equivalent transformations as long as possible. Hence, the general form is

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

where ψ is the transformed term.
The possible transformations are:

- $\neg(\varphi_1 \rightarrow \varphi_2) \leftrightarrow (\varphi_1 \wedge \neg\varphi_2)$

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

Rule 72: ac_simp
This rule simplifies nested occurences of $\vee$ or $\wedge$:

$$\Gamma \;\triangleright i. \qquad \psi \leftrightarrow \varphi_1 \circ \cdots \circ \varphi_n \qquad (\texttt{ac\_simp})$$

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.

If-Then-Else Operators

Rule 73: ite_simplify
This rule simplifies an if-then-else term by applying equivalent transformations until fix point[4] Depending on the sort of the ite-term the rule can have one of two forms. If the sort is **Bool** it has the form

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

where ψ is the transformed term.
Otherwise, it has the form

$$\Gamma \;\triangleright i. \qquad \mathrm{ite}\,\varphi\; t_1\; t_2 \simeq u \qquad (\texttt{ite\_simplify})$$

where u is the transformed term.
The possible transformations are:

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

[4]Note however that the order of the application is important, since the set of rules is not confluent. For example, the term $(\mathrm{ite}\,\top\; t_1\; t_2 \leftrightarrow t_1)$ can be simplified into both p and $(not(notp))$ depending on the order of applications.

- $\mathrm{ite}\,\psi\;\bot\;\top \leftrightarrow \neg\psi$
- $\mathrm{ite}\,\psi\;\top\;\varphi \leftrightarrow \psi \vee \varphi$
- $\mathrm{ite}\,\psi\;\varphi\;\bot \leftrightarrow \psi \wedge \varphi$
- $\mathrm{ite}\,\psi\;\bot\;\varphi \leftrightarrow \neg\psi \wedge \varphi$
- $\mathrm{ite}\,\psi\;\varphi\;\top \leftrightarrow \neg\psi \vee \varphi$

Rule 74: qnt_simplify

This rule simplifies a $\forall$-formula with a constant predicate.

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

where φ is either $\top$ or $\bot$.

Rule 75: onepoint

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

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

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

The terms $t_{j_1}, \dots, t_{j_o}$ are the points of the variables $x_{j_1}, \dots, x_{j_o}$. 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 75.1. An application of the `onepoint` rule on the term $\forall x, y.\, x \simeq y \rightarrow f(x) \wedge f(y)$ 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)
```

Rule 76: qnt_join

$$\Gamma \;\rhd i. \qquad Qx_1, \dots, x_n.Qx_{n+1}, \dots, x_m.\varphi \leftrightarrow Qx_{k_1}, \dots, x_{k_o}.\varphi \qquad (\texttt{qnt\_join})$$

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

Rule 77: qnt_rm_unused

$$\Gamma \;\rhd i. \qquad Qx_1, \dots, x_n.\varphi \leftrightarrow Qx_{k_1}, \dots, x_{k_m}.\varphi \qquad (\texttt{qnt\_rm\_unused})$$

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

Rule 78: eq_simplify

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

$$\Gamma \;\rhd i. \qquad t_1 \simeq t_2 \leftrightarrow \varphi \qquad (\texttt{eq\_simplify})$$

where ψ is 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.

Rule 79: div_simplify

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

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

The possible transformations are:

- $t/t = 1$
- $t/1 = t$
- $t_1/t_2 = t_3$ 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.

Rule 80: prod_simplify

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

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

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.

Rule 81: unary_minus_simplify
This rule is either

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

or

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

where u is the negated numerical constant t.

Rule 82: minus_simplify
This rule simplifies a subtraction by applying equivalent transformations. The general form is

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

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.
- $t - 0 = t$
- $0 - t = -t$

Rule 83: sum_simplify
This rule simplifies a sum by applying equivalent transformations as long as possible. The general form is

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

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.

Rule 84: comp_simplify

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

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

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.
- $t \leq t \leftrightarrow \top$
- $t_1 \geq t_2 \leftrightarrow t_2 \leq t_1$
- $t_1 < t_2 \leftrightarrow \neg(t_2 \leq t_1)$
- $t_1 > t_2 \leftrightarrow \neg(t_1 \leq t_2)$

Rule 85: let

This rule eliminats let. It has the form

$$\begin{array}{lll}
\Gamma \ \rhd i_1. & t_1 \simeq s_1 & (\dots) \\
 & \vdots & \\
\Gamma \ \rhd i_n. & t_n \simeq s_n & (\dots) \\
 & \vdots & \\
\Gamma, x_1 \mapsto s_1, \dots, x_n \mapsto s_n, \ \rhd j. & u \simeq u' & (\dots) \\
\hline
\Gamma \ \rhd k. & (\text{let}\, x_1 := t_1, \dots, x_n := t_n . u) \simeq u' & (\texttt{let}; i_1 \dots i_n)
\end{array}$$

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.

Rule 86: distinct_elim

This rule eliminates the distinct predicate. If called with one argument this predicate always holds:

$$\rhd i. \qquad (\text{distinct}\, t) \leftrightarrow \top \qquad (\texttt{distinct\_elim})$$

If applied to terms of type **Bool** more than two terms can never be distinct, hence only two cases are possible:

$$\rhd i. \qquad (\text{distinct}\, \varphi\ \psi) \leftrightarrow \neg(\varphi \leftrightarrow \psi) \qquad (\texttt{distinct\_elim})$$

and

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

The general case is

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

Rule 87: la_rw_eq

$\triangleright\, i.$ $\qquad (t \simeq u) \simeq (t \leq u \land u \leq t)$ $\qquad$ (`la_rw_eq`)

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

Rule 88: nary_elim

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

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

$\Gamma\ \triangleright\, i.$ $\qquad \bigcirc_{i=1}^{n} t_i \leftrightarrow (\dots (t_1 \circ t_2) \circ t_3) \circ \dots t_n)$ $\qquad$ (`nary_elim`)

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

$\Gamma\ \triangleright\, i.$ $\qquad \bigcirc_{i=1}^{n} t_i \leftrightarrow (t_1 \circ \cdots \circ (t_{n-2} \circ (t_{n-1} \circ t_n) \dots)$ $\qquad$ (`nary_elim`)

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

$\Gamma\ \triangleright\, i.$ $\qquad \bigcirc_{i=1}^{n} t_i \leftrightarrow (t_1 \circ t_2) \land (t_2 \circ t_3) \land \cdots \land (t_{n-1} \circ t_n)$ $\qquad$ (`nary_elim`)

Rule 89: bfun_elim

$\triangleright\, i.$ $\qquad \psi$ $\qquad (\dots)$

$\triangleright\, j.$ $\qquad \varphi$ $\qquad$ (`bfun_elim`; i)

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 **Bool**. Hence, $\exists x.t$ becomes $t[\bot/x] \lor t[\top/x]$ and $\forall x.t$ becomes $t[\bot/x] \land t[\top/x]$. If n variables of sort **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 $\text{ite}\, P\ f(x, \top, y)\ f(x, \bot, y)$. If the argument is already the constant $\top$ or $\bot$ it is ignored.

Rule 90: ite_intro

Either

$$\triangleright\, i. \qquad t \simeq (t' \land u_1 \land \cdots \land u_n) \qquad (\texttt{ite\_intro})$$

or

$$\triangleright\, i. \qquad \varphi \leftrightarrow (\varphi' \land u_1 \land \cdots \land u_n) \qquad (\texttt{ite\_intro})$$

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

$$\text{ite}\, \psi_i\ (s_i \simeq r_i)\ (s_i \simeq r'_i)$$

or

$$\text{ite}\, \psi_i\ (s_i \leftrightarrow r_i)\ (s_i \leftrightarrow r'_i)$$

if s_i is of sort **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.

5 Changelog

Unreleased

Apply major changes to the structure of the document to clarify the difference between the *language* and the *rules*.

List of rules:

- Improve description of `sko_ex`.

Corrections:

- Grammar: the `choice` binder can only bind one variable.

Clarifications:

- Clarify functionality of choice in introduction.
- Add illustrating example to introduction.
- Normalize printing of (variable, term) arguments in the abstract notation.
- Fix linear arithmetic example in introduction.

0.1 — July 10, 2021

This is the first public release of this document. It coincides with the seventh PxTP Workshop.

References

[1] Haniel Barbosa, Jasmin C. Blanchette, Mathias Fleury, and Pascal Fontaine. Scalable fine-grained proofs for formula processing. *Journal of Automated Reasoning*, 2019.

[2] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV 2011*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.

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

[4] Frédéric Besson, Pascal Fontaine, and Laurent Théry. A flexible proof format for SMT: A proposal. In Pascal Fontaine and Aaron Stump, editors, *PxTP 2011*, pages 15–26, 2011.

[5] Sascha Böhme and Tjark Weber. Fast LCF-style proof reconstruction for Z3. In Matt Kaufmann and Lawrence Paulson, editors, *ITP 2010*, volume 6172 of *LNCS*, pages 179–194. Springer, 2010.

[6] Thomas Bouton, Diego C. B. de Oliveira, David Déharbe, and Pascal Fontaine. veriT: An open, trustable and efficient SMT-solver. In Renate A. Schmidt, editor, *CADE 2009*, volume 5663 of *LNCS*, pages 151–156. Springer, 2009.

[7] Hans de Nivelle. Translation of resolution proofs into short first-order proofs without choice axioms. *Inf. Comput.*, 199(1-2):24–54, 2005.

[8] David Déharbe, Pascal Fontaine, and Bruno Woltzenlogel Paleo. Quantifier inference rules for SMT proofs. In Pascal Fontaine and Aaron Stump, editors, *PxTP 2011*, pages 33–39, 2011.

[9] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. SMTCoq: A plug-in for integrating SMT solvers into Coq. In Rupak Majumdar and Viktor Kunčak, editors, *CAV 2017*, volume 10426 of *LNCS*, pages 126–133. Springer, 2017.

[10] Mathias Fleury and Hans-Jörg Schurr. Reconstructing veriT proofs in Isabelle/HOL. In Giselle Reis and Haniel Barbosa, editors, *Proceedings Sixth Workshop on Proof eXchange for Theorem Proving, Natal, Brazil, August 26, 2019*, volume 301 of *Electronic Proceedings in Theoretical Computer Science*, pages 36–50. Open Publishing Association, 2019.

[11] Lawrence C. Paulson and Kong Woei Susanto. Source-level proof reconstruction for interactive theorem proving. In Klaus Schneider and Jens Brandt, editors, *TPHOLs 2007*, volume 4732 of *LNCS*, pages 232–245. Springer, 2007.

[12] Hans-Jörg Schurr, Mathias Fleury, and Martin Desharnais. Reliable reconstruction of fine-grained proofs in a proof assistant. In André Platzer and Geoff Sutcliffe, editors, *CADE 28*, LNCS, pages 450–467, Cham, 2021. Springer International Publishing.