# The Alethe Proof Format

**A Speculative Specification and Reference**
**(Work in Progress)**

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

# 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. It is designed around a few core concepts, such as a natural-deduction style structure and resolution. Section 2 informally introduces those concepts and also shows the notations used for them along the way. In the subsequent Section 3 we will see the formal semantic of the concepts introduced before.

The concrete syntax and semantic is based on the SMT-LIB [3] format, which is widely used as input and output format of SMT solvers. Hence, the base logic is the many-sorted first order logic of SMT-LIB. This document assumes the reader is familiar with the SMT-LIB standard. As discussed in Section 4 the concrete syntax of Alethe is based on the SMT-LIB syntax.

The format is currently used by the CDCL($T$)-based 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 [9,11]. Furthermore, in the future the SMT solver CVC4 [2] will support Alethe as one of 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 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, renaming of variables, and other manipulations of bound variables [1].

At the end of a document Section 5 contains a full list of all proof rules used by SMT solvers supporting Alethe.

# 2 Core Concepts of Alethe

This section provides an overview of the core concepts of the proof format and also introduces some notation used throughout this paper. While the next section provides a formal semantic for the format, this overview of the core concept should be very 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 and can also be extended 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 $\epsilon$. 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.

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 $\varphi, \psi$ denote formulas, i.e., terms of sort **Bool**. We use $\sigma$ 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 $\varphi$ and $\bar{\psi}$ are syntactically equal and the number of leading negations is even for $\varphi$ and odd for $\bar{\psi}$, or vice versa. To simplify the notation we will omit the sort of terms when possible.

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

$$c_1, \ldots, c_k \ \rhd \ i. \quad \varphi_1, \ldots, \varphi_l \quad (\texttt{rule};\ p_1, \ldots, p_m;\ a_1, \ldots, a_n)$$

Each step has an 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. No two steps have the same index. Furthermore, a step has a clause $\varphi_1, \ldots, \varphi_l$ 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 $\texttt{rule}$ is taken from a set of possible proof rules. Furthermore, each step has a possibly empty set of premises $\{p_1, \ldots, p_n\}$ with $p_i \in \mathbb{N}$, and a rule-dependent and possibly empty list of arguments $[a_1, \ldots, a_m]$. 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 $c_1, \ldots, c_k$ 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. Section 5 provides an overview of all proof rules used by veriT.

**Assumptions.** The $\texttt{assume}$ rule introduces a term as an assumption. The proof starts with a number of $\texttt{assume}$ 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 $\texttt{subproof}$ rule. From an assumption $\varphi$ and a formula $\psi$ proved by intermediate steps from $\varphi$, the $\texttt{subproof}$ step deduces $\neg\varphi \lor \psi$ and discharges $\varphi$.

**Tautologous Rules and Simple Deduction.** Most rules introduce tautologies. One example is the $\texttt{and\_pos}$ rule: $\neg(\varphi_1 \land \varphi_2 \land \cdots \land \varphi_n) \lor \varphi_i$. Other rules operate on only one premise. Those rules are primarily used to simplify Boolean connectives during preprocessing. For example, the $\texttt{implies}$ rule removes an implication: From $\varphi_1 \Rightarrow \varphi_2$ it deduces $\neg\varphi_1 \lor \varphi_2$.

4

**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, usual disjunction is also used clauses simplifies checking the rules. 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 to 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.

**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 $\varphi$ 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 is the list $(x_1, t_1), \ldots, (x_n, t_n)$. While this information can be recovered from the term, providing this information explicitly aids reconstruction because the implicit reordering of equalities (see below) obscure 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 $\varphi_i$ 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 1.** The following example is the proof for the unsatisfiability of $(x+y < 1) \vee (3 < x)$, $x \simeq 2$, and $0 \simeq y$.

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

**Subproofs and Lemmas.** The proof format uses subproof to prove lemmas and to manipulate the context. To prove lemmas, a subproof can introduce local assumptions. The subproof *rule* discharges the local assumptions by concluding with an implication (written as a clause) which has the local assumptions as its antecedents. A step can only use steps from the same subproof as its premise. It is not possible to have premises from either a subproof at a deeper level or from an outer level.

As the example below shows, our notation for subproofs is a frame around the rules within the subproof. Subproofs are also used to manipulate the context.

**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 veriT proof format is the step context. The context is a possible empty list $[c_1, \ldots, c_l]$, where $c_i$ is either a variable or a variable-term tuple denoted $x_i \mapsto t_i$. Throughout this document $\Gamma$ denotes an arbitrary context. The context denotes a substitution.

**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.

$$\Gamma, x \mapsto (\epsilon x.\varphi) \; \triangleright \; n. \quad \varphi \simeq \psi \qquad (\ldots)$$
$$\Gamma \; \triangleright \; m. \quad (\exists x.\varphi) \simeq \psi \quad (\texttt{sko\_ex}; \; n)$$

**Example 3.** 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 $\epsilon x.\neg\varphi$.

| | | | |
|---|---|:---:|---:|
| $x \mapsto (\epsilon x.\neg p(x))$ | $\triangleright 1.$ | $x \simeq \epsilon x.\neg p(x)$ | (refl) |
| $x \mapsto (\epsilon x.\neg p(x))$ | $\triangleright 2.$ | $p(x) \simeq p(\epsilon x.\neg p(x))$ | (cong; 1) |
| | $\triangleright 3.$ | $(\forall x.p(x)) \simeq p(\epsilon x.\neg p(x))$ | (sko_forall; 2) |
| | $\triangleright 4.$ | $(\neg\forall x.p(x)) \simeq \neg p(\epsilon x.\neg p(x))$ | (cong; 3) |

**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 change 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.

## 3 The Concrete Semantics

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

### 3.1 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 $\Gamma$ 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 $\Gamma$ fixes a set of variables and specifies a substitution $subst(\Gamma)$. The substitution is the identity for $\varnothing$ and is defined as follows in the other cases:

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

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

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

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

Transformations of terms (and formulas) are justified by judgments of the form $\Gamma \; \triangleright \; t \simeq u$, where $\Gamma$ 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 $\Gamma$. Semantically, the judgment expresses the equality of the terms $\Gamma(t)$ and $u$, universally quantified on variables fixed by $\Gamma$. 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.

$$\models (\exists x.\ \varphi[x]) \implies \varphi[\varepsilon x.\ \varphi] \tag{$\varepsilon_1$}$$

$$\models (\forall x.\ \varphi \simeq \psi) \implies (\varepsilon x.\ \varphi) \simeq (\varepsilon x.\ \psi) \tag{$\varepsilon_2$}$$

$$\models (\text{let } \bar{x}_n \simeq \bar{s}_n \text{ in } t[\bar{x}_n]) \simeq t[\bar{s}_n] \tag{let}$$

The rules are:

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

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

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

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

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

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

- $\textsc{Taut}_{\mathcal{T}}$ 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 $\Gamma$'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 $\mathsf{f}$, including the logical connectives.

- $\textsc{Bind}$ 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 $\Gamma$). In particular, $y$ should not appear fixed or on either side of a substitution in the context.

- SKO$_\exists$ and SKO$_\forall$ exploit $(\varepsilon_1)$ to replace a quantified variable with a suitable witness, simulating skolemization. We can think of the $\varepsilon$ 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, 10], but we push the idea further and use it to simplify the inference system and make it more uniform.

**Example 4.** The following derivation tree justifies the expansion of a 'let' expression:

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

### 3.2 Correctness

**Theorem 4.1** (Soundness of Inferences [1, Theorem 11]). If the judgment $\Gamma\ \ \rhd\ t \simeq u$ 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 $\mathcal{T} = \mathcal{T}_1 \cup \cdots \cup \mathcal{T}_n \cup \simeq \cup\ \varepsilon \cup$ let.

*Proof.* See [1, Theorem 11] $\qquad\square$

### 3.3 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 und is specialized.

```
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)

## 4  The Concrete 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 gram-
mar, 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:
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

---

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

$$
\begin{array}{rcl}
\langle proof \rangle & ::= & \langle proof\_command \rangle^* \\
\langle proof\_command \rangle & ::= & (\texttt{assume } \langle symbol \rangle \ \langle proof\_term \rangle \ ) \\
 & | & (\texttt{step } \langle symbol \rangle \ \langle clause \rangle \ \texttt{:rule } \langle symbol \rangle \\
 & & \quad \langle step\_annotation \rangle \ ) \\
 & | & (\texttt{anchor :step } \langle symbol \rangle \ ) \\
 & | & (\texttt{anchor :step } \langle symbol \rangle \ \texttt{:args } \langle proof\_args \rangle \ ) \\
 & | & (\texttt{define-fun } \langle function\_def \rangle \ ) \\
\langle clause \rangle & ::= & (\texttt{cl } \langle proof\_term \rangle^* \ ) \\
\langle step\_annotation \rangle & ::= & \texttt{:premises (} \ \langle symbol \rangle^+ \ ) \\
 & | & \texttt{:args } \langle proof\_args \rangle \\
 & | & \texttt{:premises (} \ \langle symbol \rangle^+ \ ) \ \texttt{:args } \langle proof\_args \rangle \\
\langle proof\_args \rangle & ::= & ( \ \langle proof\_arg \rangle^+ \ ) \\
\langle proof\_arg \rangle & ::= & \langle symbol \rangle \ \ | \ \ ( \ \langle symbol \rangle \ \langle proof\_term \rangle \ ) \\
\langle proof\_term \rangle & ::= & \langle term \rangle \text{ extended with} \\
 & & (\texttt{choice (} \ \langle sorted\_var \rangle^+ \ ) \ \langle proof\_term \rangle \ )
\end{array}
$$

Figure 2: The proof grammar

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.

## 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 $\Gamma$, 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. 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.

### 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.

## 5 List of Proof Rules

The following lists all rules of Alethe. They are categorized by type:

**Main Rules (Sect. 5.1)** contains the rules that justify the existence of the proof format, namely the `bind` and `subproof` rules.

**Skolemization Rules (Sect. 5.2)** that uses Hilbert choice.

**Basic Proof Rules (Sect. 5.3).**

**Arithmetic Rules (Sect. 5.4).**

**Quantifier Rules (Sect. 5.5)** that shows how to instantiate a formula.

**SAT handling (Sect. 5.6)** where resolution and removal of duplicated literals are presented.

**Clausification of** Boolean operators (Sect. 5.9) and if-then-else (Sect. 5.10) where structure is moved from terms to the clause structure.

---

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

**Simplifications (Sect. 5.11)** are a very important part of any SMT solver. veriT provides very detailed rules. To simplify checking, we encourage users to write one step per theory instead of combining all simplifications in a single step.

**Term Reshaping (Sect. 5.12)** that transform a term by adding or removing variables.

## 5.1 Main Rules

The following rules are the raison-d'être of the proof format: providing detailed reasoning under quantifiers.

### Rule 1: bind

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

$$\frac{\Gamma, y_1, \ldots, y_n, x_1 \mapsto y_1, \ldots, x_n \mapsto y_n, \ \rhd\, j. \qquad\qquad \varphi \leftrightarrow \varphi' \qquad\qquad (\ldots)}{\Gamma \ \rhd\, k. \qquad\qquad \forall x_1, \ldots, x_n.\varphi \leftrightarrow \forall y_1, \ldots, y_n.\varphi' \qquad\qquad (\texttt{bind})}$$

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

### Rule 2: 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.

$$
\begin{array}{lll}
\rhd\, i_1. & \psi_1 & (\texttt{input}) \\[2pt]
 & \vdots & \\[2pt]
\rhd\, i_n. & \psi_n & (\texttt{input}) \\[2pt]
 & \vdots & \\[2pt]
\rhd\, j. & \varphi & (\ldots) \\
\hline
\rhd\, k. & \neg\psi_1, \ldots, \neg\psi_n, \varphi & (\texttt{subproof})
\end{array}
$$

## 5.2 Skolemization Rules

### Rule 3: sko_ex

The `sko_ex` rule skolemizes existential quantifiers.

$$\frac{\Gamma, x_1 \mapsto (\epsilon x_1.\varphi), \ldots, x_n \mapsto (\epsilon x_n.\varphi), \ \rhd\, j. \qquad\qquad \varphi \leftrightarrow \psi \qquad\qquad (\ldots)}{\Gamma \ \rhd\, k. \qquad\qquad \exists x_1, \ldots, x_n.\varphi \leftrightarrow \psi \qquad\qquad (\texttt{sko\_ex})}$$

### Rule 4: sko_forall

The `sko_forall` rule skolemizes universal quantifiers.

$$\frac{\Gamma, x_1 \mapsto (\epsilon x_1.\neg\varphi), \ldots, x_n \mapsto (\epsilon x_n.\neg\varphi), \ \rhd\, j. \qquad\qquad \varphi \leftrightarrow \psi \qquad\qquad (\ldots)}{\Gamma \ \rhd\, k. \qquad\qquad \forall x_1, \ldots, x_n.\varphi \leftrightarrow \psi \qquad\qquad (\texttt{sko\_forall})}$$

## 5.3 Basic Proof Rules

### Rule 5: assume

$$\rhd\, i. \hspace{10em} \phi \hspace{10em} (\texttt{assume})$$

where $\varphi$ is equivalent to a formula asserted in the input problem.

### Rule 6: true

$$\rhd\, i. \hspace{10em} \top \hspace{10em} (\texttt{true})$$

### Rule 7: false

$$\rhd\, i. \hspace{10em} \neg\bot \hspace{10em} (\texttt{false})$$

### Rule 8: not_not

$$\rhd\, i. \hspace{8em} \neg(\neg\neg\varphi), \varphi \hspace{8em} (\texttt{not\_not})$$

**Remark.** This rule is useful to remove double negations from a clause. Take the clause with the double negation and resolve on $\varphi$.

## 5.4 Arithmetic Rules

### Rule 9: 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

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

where $\varphi_i$ 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 $\varphi := s_1 > s_2$.

14

2. If $\varphi = \neg(s_1 \bowtie s_2)$, then let $\varphi := s_1 \bowtie s_2$.

3. Replace $\varphi$ 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 $\varphi$ 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 $\geq 0$ if $\bowtie$ *is* $>$).

**Example 9.1.** 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 9.2.** 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 10: lia_generic**

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

$\triangleright i.$                             $\varphi_1, \ldots, \varphi_n$                           (`lia_generic`)

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

**Rule 11: la_disequality**

$\triangleright i.$             $t_1 \simeq t_2 \vee \neg(t_1 \leq t_2) \vee \neg(t_2 \leq t_1)$          (`la_disequality`)

**Rule 12: la_totality**

$\triangleright i.$                    $t_1 \leq t_2 \vee t_2 \leq t_1$                   (`la_totality`)

**Rule 13: la_tautology**

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

$\triangleright i.$                                $\varphi$                           (`la_tautology`)

where $\varphi$ 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`.

## 5.5 Quantifier Rules

**Rule 14: forall_inst**

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

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`.

## 5.6 Resolution Rules and Clause Simplifications

### Rule 15: th_resolution

This rule is the resolution of two or more clauses.

$$\triangleright i_1. \qquad\qquad \varphi_1^1, \ldots, \varphi_{k^1}^1 \qquad\qquad\qquad (\ldots)$$
$$\vdots$$
$$\triangleright i_n. \qquad\qquad \varphi_1^n, \ldots, \varphi_{k^n}^n \qquad\qquad\qquad (\ldots)$$
$$\triangleright\ j. \qquad\qquad \varphi_{s_1}^{r_1}, \ldots, \varphi_{s_m}^{r_m} \qquad (\texttt{th\_resolution}; i_1, \ldots, i_n)$$

where $\varphi_{s_1}^{r_1}, \ldots, \varphi_{s_m}^{r_m}$ are from $\varphi_j^i$ 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 16: 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 17: tautology

$$\triangleright i. \qquad\qquad \varphi_1, \ldots, \varphi_i, \ldots, \varphi_j, \ldots, \varphi_n \qquad\qquad (\ldots)$$
$$\triangleright j. \qquad\qquad \top \qquad\qquad (\texttt{tautology}; i)$$

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

### Rule 18: contraction

$$\triangleright i. \qquad\qquad \varphi_1, \ldots, \varphi_n \qquad\qquad\qquad (\ldots)$$
$$\vdots$$
$$\triangleright j. \qquad\qquad \varphi_{k_1}, \ldots, \varphi_{k_m} \qquad\qquad (\texttt{contraction}; i)$$

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

## 5.7 Equality and Congruence Reasonning

### Rule 19: refl

Either

$$\Gamma \, \triangleright j. \qquad\qquad\qquad\qquad t_1 \simeq t_2 \qquad\qquad\qquad\qquad\qquad \text{(refl)}$$

or

$$\Gamma \, \triangleright j. \qquad\qquad\qquad\qquad \varphi_1 \leftrightarrow \varphi_2 \qquad\qquad\qquad\qquad\qquad \text{(refl)}$$

where $\varphi_1$ and $\varphi_2$ ($P_1$ and $P_2$) are equal after applying the substitution induced by $\Gamma$.

**Rule 20: trans**

Either

$$\begin{array}{llll}
\Gamma \, \triangleright i. & \qquad\qquad t_1 \simeq t_2 & \qquad\qquad (\dots) \\
\Gamma \, \triangleright j. & \qquad\qquad t_2 \simeq t_3 & \qquad\qquad (\dots) \\
\Gamma \, \triangleright k. & \qquad\qquad t_1 \simeq t_3 & \qquad\qquad (\texttt{trans}; i, j)
\end{array}$$

or

$$\begin{array}{llll}
\Gamma \, \triangleright i. & \qquad\qquad \varphi_1 \leftrightarrow \varphi_2 & \qquad\qquad (\dots) \\
\Gamma \, \triangleright j. & \qquad\qquad \varphi_2 \leftrightarrow \varphi_3 & \qquad\qquad (\dots) \\
\Gamma \, \triangleright k. & \qquad\qquad \varphi_1 \leftrightarrow \varphi_3 & \qquad\qquad (\texttt{trans}; i, j)
\end{array}$$

**Rule 21: cong**

Either

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

where f is an $n$-ary function symbol, or

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

where P is an $n$-ary predicate symbol.

**Rule 22: eq_reflexive**

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

**Rule 23: eq_transitive**

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

**Rule 24: eq_congruent**

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

**Rule 25: eq_congruent_pred**

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

18

## 5.8 Clausification of Quantifiers

**Rule 26: qnt_cnf**

$$\rhd i. \qquad\qquad \neg(\forall x_1, \ldots, x_n.\varphi) \vee \forall x_{k_1}, \ldots, x_{k_m}.\varphi' \qquad\qquad (\texttt{qnt\_cnf})$$

This is a placeholder rule for clausification of a term under a universal quantifier. This is used by conflicting instantiation. $\varphi'$ is one of the clause of the clause normal form of $\varphi$. The variables $x_{k_1}, \ldots, x_{k_m}$ are a permutation of $x_1, \ldots, x_n$ plus additional variables added by prenexing $\varphi$. 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:

$$\Phi(\neg\varphi, 1) := \Phi(\varphi, 0)$$
$$\Phi(\neg\varphi, 0) := \Phi(\varphi, 1)$$
$$\Phi(\varphi_1 \vee \cdots \vee \varphi_n, 1) := \Phi(\varphi_1, 1) \vee \cdots \vee \Phi(\varphi_n, 1)$$
$$\Phi(\varphi_1 \wedge \cdots \wedge \varphi_n, 1) := \Phi(\varphi_1, 1) \wedge \cdots \wedge \Phi(\varphi_n, 1)$$
$$\Phi(\varphi_1 \vee \cdots \vee \varphi_n, 0) := \Phi(\varphi_1, 0) \wedge \cdots \wedge \Phi(\varphi_n, 0)$$
$$\Phi(\varphi_1 \wedge \cdots \wedge \varphi_n, 0) := \Phi(\varphi_1, 0) \vee \cdots \vee \Phi(\varphi_n, 0)$$
$$\Phi(\varphi_1 \rightarrow \varphi_2, 1) := (\Phi(\varphi_1, 0) \vee \Phi(\varphi_2, 1)) \wedge (\Phi(\varphi_2, 0) \vee \Phi(\varphi_1, 1))$$
$$\Phi(\varphi_1 \rightarrow \varphi_2, 0) := (\Phi(\varphi_1, 1) \wedge \Phi(\varphi_2, 0)) \vee (\Phi(\varphi_2, 1) \wedge \Phi(\varphi_1, 0))$$
$$\Phi(\text{ite } \varphi_1\ \varphi_2\ \varphi_3, 1) := (\Phi(\varphi_1, 0) \vee \Phi(\varphi_2, 1)) \wedge (\Phi(\varphi_1, 1) \vee \Phi(\varphi_3, 1))$$
$$\Phi(\text{ite } \varphi_1\ \varphi_2\ \varphi_3, 0) := (\Phi(\varphi_1, 1) \wedge \Phi(\varphi_2, 0)) \vee (\Phi(\varphi_1, 0) \wedge \Phi(\varphi_3, 0))$$
$$\Phi(\forall x_1, \ldots, x_n.\varphi, 1) := \forall x_1, \ldots, x_n.\Phi(\varphi, 1)$$
$$\Phi(\exists x_1, \ldots, x_n.\varphi, 1) := \exists x_1, \ldots, x_n.\Phi(\varphi, 1)$$
$$\Phi(\forall x_1, \ldots, x_n.\varphi, 0) := \exists x_1, \ldots, x_n.\Phi(\varphi, 0)$$
$$\Phi(\exists x_1, \ldots, x_n.\varphi, 0) := \forall x_1, \ldots, x_n.\Phi(\varphi, 0)$$
$$\Phi(\varphi, 1) := \varphi$$
$$\Phi(\varphi, 0) := \neg\varphi$$

## 5.9 Clausification of Boolean Operators

**Rule 27: and**

$$\rhd i. \qquad\qquad \varphi_1 \wedge \cdots \wedge \varphi_n \qquad\qquad (\ldots)$$
$$\rhd j. \qquad\qquad \varphi_i \qquad\qquad (\texttt{and}; i)$$

**Rule 28: not_or**

$$\rhd i. \qquad\qquad \neg(\varphi_1 \vee \cdots \vee \varphi_n) \qquad\qquad (\ldots)$$
$$\rhd j. \qquad\qquad \neg\varphi_i \qquad\qquad (\texttt{not\_or}; i)$$

**Rule 29: or**

$$\triangleright i. \qquad\qquad \varphi_1 \vee \cdots \vee \varphi_n \qquad\qquad (\dots)$$
$$\triangleright j. \qquad\qquad \varphi_1, \dots, \varphi_n \qquad\qquad (\texttt{or}; i)$$

**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**

$$\triangleright i. \qquad\qquad \neg(\varphi_1 \wedge \cdots \wedge \varphi_n) \qquad\qquad (\dots)$$
$$\triangleright j. \qquad\qquad \neg\varphi_1, \dots, \neg\varphi_n \qquad\qquad (\texttt{not\_and}; i)$$

**Rule 31: xor1**

$$\triangleright i. \qquad\qquad \mathrm{xor}\ \varphi_1\ \varphi_2 \qquad\qquad (\dots)$$
$$\triangleright j. \qquad\qquad \varphi_1, \varphi_2 \qquad\qquad (\texttt{xor1}; i)$$

**Rule 32: xor2**

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

**Rule 33: not_xor1**

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

**Rule 34: not_xor2**

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

**Rule 35: implies**

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

**Rule 36: not_implies1**

$$\triangleright i. \qquad\qquad \neg(\varphi_1 \rightarrow \varphi_2) \qquad\qquad (\dots)$$
$$\triangleright j. \qquad\qquad \varphi_1 \qquad\qquad (\texttt{not\_implies1}; i)$$

**Rule 37: not_implies2**

$$\triangleright\, i. \qquad\qquad \neg(\varphi_1 \to \varphi_2) \qquad\qquad (\dots)$$
$$\triangleright\, j. \qquad\qquad \neg\varphi_2 \qquad\qquad (\texttt{not\_implies2}; i)$$

**Rule 38: equiv1**

$$\triangleright\, i. \qquad\qquad \varphi_1 \leftrightarrow \varphi_2 \qquad\qquad (\dots)$$
$$\triangleright\, j. \qquad\qquad \neg\varphi_1, \varphi_2 \qquad\qquad (\texttt{equiv1}; i)$$

**Rule 39: equiv2**

$$\triangleright\, i. \qquad\qquad \varphi_1 \leftrightarrow \varphi_2 \qquad\qquad (\dots)$$
$$\triangleright\, j. \qquad\qquad \varphi_1, \neg\varphi_2 \qquad\qquad (\texttt{equiv2}; i)$$

**Rule 40: not_equiv1**

$$\triangleright\, i. \qquad\qquad \neg(\varphi_1 \leftrightarrow \varphi_2) \qquad\qquad (\dots)$$
$$\triangleright\, j. \qquad\qquad \varphi_1, \varphi_2 \qquad\qquad (\texttt{not\_equiv1}; i)$$

**Rule 41: not_equiv2**

$$\triangleright\, i. \qquad\qquad \neg(\varphi_1 \leftrightarrow \varphi_2) \qquad\qquad (\dots)$$
$$\triangleright\, j. \qquad\qquad \neg\varphi_1, \neg\varphi_2 \qquad\qquad (\texttt{not\_equiv2}; i)$$

**Rule 42: and_pos**

$$\triangleright\, i. \qquad\qquad \neg(\varphi_1 \wedge \cdots \wedge \varphi_n), \varphi_k \qquad\qquad (\texttt{and\_pos})$$

with $1 \le k \le n$.

**Rule 43: and_neg**

$$\triangleright\, i. \qquad\qquad (\varphi_1 \wedge \cdots \wedge \varphi_n), \neg\varphi_1, \dots, \neg\varphi_n \qquad\qquad (\texttt{and\_neg})$$

**Rule 44: or_pos**

$$\triangleright\, i. \qquad\qquad \neg(\varphi_1 \vee \cdots \vee \varphi_n), \varphi_1, \dots, \varphi_n \qquad\qquad (\texttt{or\_pos})$$

**Rule 45: or_neg**

$$\triangleright\, i. \qquad\qquad (\varphi_1 \vee \cdots \vee \varphi_n), \neg\varphi_k \qquad\qquad (\texttt{or\_neg})$$

with $1 \le k \le n$.

**Rule 46: xor_pos1**

$$\triangleright\, i. \qquad\qquad \neg(\varphi_1 \,\mathrm{xor}\, \varphi_2), \varphi_1, \varphi_2 \qquad\qquad (\texttt{xor\_pos1})$$

**Rule 47: xor_pos2**

$$\triangleright\, i. \qquad\qquad \neg(\varphi_1 \,\mathrm{xor}\, \varphi_2), \neg\varphi_1, \neg\varphi_2 \qquad\qquad (\texttt{xor\_pos2})$$

**Rule 48: xor_neg1**

$\triangleright i.$ $\qquad\qquad\qquad\qquad \varphi_1 \operatorname{xor} \varphi_2, \varphi_1, \neg\varphi_2$ $\qquad\qquad\qquad\qquad$ $(\texttt{xor\_neg1})$

**Rule 49: xor_neg2**

$\triangleright i.$ $\qquad\qquad\qquad\qquad \varphi_1 \operatorname{xor} \varphi_2, \neg\varphi_1, \varphi_2$ $\qquad\qquad\qquad\qquad$ $(\texttt{xor\_neg2})$

**Rule 50: implies_pos**

$\triangleright i.$ $\qquad\qquad\qquad\qquad \neg(\varphi_1 \rightarrow \varphi_2), \neg\varphi_1, \varphi_2$ $\qquad\qquad\qquad\qquad$ $(\texttt{implies\_pos})$

**Rule 51: implies_neg1**

$\triangleright i.$ $\qquad\qquad\qquad\qquad \varphi_1 \rightarrow \varphi_2, \varphi_1$ $\qquad\qquad\qquad\qquad$ $(\texttt{implies\_neg1})$

**Rule 52: implies_neg2**

$\triangleright i.$ $\qquad\qquad\qquad\qquad \varphi_1 \rightarrow \varphi_2, \neg\varphi_2$ $\qquad\qquad\qquad\qquad$ $(\texttt{implies\_neg2})$

**Rule 53: equiv_pos1**

$\triangleright i.$ $\qquad\qquad\qquad\qquad \neg(\varphi_1 \leftrightarrow \varphi_2), \varphi_1, \neg\varphi_2$ $\qquad\qquad\qquad\qquad$ $(\texttt{equiv\_pos1})$

**Rule 54: equiv_pos2**

$\triangleright i.$ $\qquad\qquad\qquad\qquad \neg(\varphi_1 \leftrightarrow \varphi_2), \neg\varphi_1, \varphi_2$ $\qquad\qquad\qquad\qquad$ $(\texttt{equiv\_pos2})$

**Rule 55: equiv_neg1**

$\triangleright i.$ $\qquad\qquad\qquad\qquad \varphi_1 \leftrightarrow \varphi_2, \neg\varphi_1, \neg\varphi_2$ $\qquad\qquad\qquad\qquad$ $(\texttt{equiv\_neg1})$

**Rule 56: equiv_neg2**

$\triangleright i.$ $\qquad\qquad\qquad\qquad \varphi_1 \leftrightarrow \varphi_2, \varphi_1, \varphi_2$ $\qquad\qquad\qquad\qquad$ $(\texttt{equiv\_neg2})$

## 5.10 Clausification of If-Then-Else

**Rule 57: ite1**

$\triangleright i.$ $\qquad\qquad\qquad\qquad \operatorname{ite} \varphi_1\ \varphi_2\ \varphi_3$ $\qquad\qquad\qquad\qquad$ $(\dots)$
$\triangleright j.$ $\qquad\qquad\qquad\qquad \varphi_1, \varphi_3$ $\qquad\qquad\qquad\qquad$ $(\texttt{ite1}; i)$

**Rule 58: ite2**

$\triangleright i.$ $\qquad\qquad\qquad\qquad \operatorname{ite} \varphi_1\ \varphi_2\ \varphi_3$ $\qquad\qquad\qquad\qquad$ $(\dots)$
$\triangleright j.$ $\qquad\qquad\qquad\qquad \neg\varphi_1, \varphi_2$ $\qquad\qquad\qquad\qquad$ $(\texttt{ite2}; i)$

22

**Rule 59: ite_pos1**

$\rhd\, i.$ $\qquad\qquad\qquad\qquad \neg(\text{ite}\,\varphi_1\,\varphi_2\,\varphi_3), \varphi_1, \varphi_3$ $\qquad\qquad\qquad$ $(\texttt{ite\_pos1})$

**Rule 60: ite_pos2**

$\rhd\, i.$ $\qquad\qquad\qquad\qquad \neg(\text{ite}\,\varphi_1\,\varphi_2\,\varphi_3), \neg\varphi_1, \varphi_2$ $\qquad\qquad\qquad$ $(\texttt{ite\_pos2})$

**Rule 61: ite_neg1**

$\rhd\, i.$ $\qquad\qquad\qquad\qquad \text{ite}\,\varphi_1\,\varphi_2\,\varphi_3, \varphi_1, \neg\varphi_3$ $\qquad\qquad\qquad$ $(\texttt{ite\_neg1})$

**Rule 62: ite_neg2**

$\rhd\, i.$ $\qquad\qquad\qquad\qquad \text{ite}\,\varphi_1\,\varphi_2\,\varphi_3, \neg\varphi_1, \neg\varphi_2$ $\qquad\qquad\qquad$ $(\texttt{ite\_neg2})$

**Rule 63: not_ite1**

$\rhd\, i.$ $\qquad\qquad\qquad\qquad \neg(\text{ite}\,\varphi_1\,\varphi_2\,\varphi_3)$ $\qquad\qquad\qquad\qquad$ $(\ldots)$
$\rhd\, j.$ $\qquad\qquad\qquad\qquad\quad \varphi_1, \neg\varphi_3$ $\qquad\qquad\qquad\qquad$ $(\texttt{not\_ite1}; i)$

**Rule 64: not_ite2**

$\rhd\, i.$ $\qquad\qquad\qquad\qquad \neg(\text{ite}\,\varphi_1\,\varphi_2\,\varphi_3)$ $\qquad\qquad\qquad\qquad$ $(\ldots)$
$\rhd\, j.$ $\qquad\qquad\qquad\qquad\quad \neg\varphi_2, \neg\varphi_2$ $\qquad\qquad\qquad\qquad$ $(\texttt{not\_ite2}; i)$

## 5.11 Simplifications

The following rules are pure simplification rules. We give a very detailed list. Combining several simplifications at once is possible, but we encourage proofs to do so for only one single theory at a time.

### 5.11.1 Boolean Operators

### Rule 65: connective_def

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

$\Gamma\ \rhd\, i.$ $\qquad\qquad \varphi_1\,\text{xor}\,\varphi_2 \leftrightarrow (\neg\varphi_1 \land \varphi_2) \lor (\varphi_1 \land \neg\varphi_2)$ $\qquad$ $(\texttt{connective\_def})$

$\Gamma\ \rhd\, i.$ $\qquad\qquad\quad \varphi_1 \leftrightarrow \varphi_2 \leftrightarrow (\varphi_1 \rightarrow \varphi_2) \land (\varphi_2 \rightarrow \varphi_1)$ $\qquad$ $(\texttt{connective\_def})$

$\Gamma\ \rhd\, i.$ $\qquad\qquad \text{ite}\,\varphi_1\,\varphi_2\,\varphi_3 \leftrightarrow (\varphi_1 \rightarrow \varphi_2) \land (\neg\varphi_1 \rightarrow \neg\varphi_3)$ $\qquad$ $(\texttt{connective\_def})$

### Rule 66: and_simplify

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

$$\Gamma \, \rhd \, i. \qquad\qquad\qquad \varphi_1 \wedge \cdots \wedge \varphi_n \leftrightarrow \psi \qquad\qquad\qquad \text{(and\_simplify)}$$

where $\psi$ 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

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

where $\psi$ 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

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

where $\psi$ 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

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

where $\psi$ 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

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

where $\psi$ 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

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

where $\psi$ 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 $\wedge$:

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

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

### 5.11.2 If-Then-Else Operators

**Rule 73: 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 **Bool** it has the form

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

where $\psi$ is the transformed term.

Otherwise, it has the form

$$\Gamma \ \rhd i. \qquad\qquad \text{ite } \varphi \ t_1 \ t_2 \simeq u \qquad\qquad (\texttt{ite\_simplify})$$

where $u$ is the transformed term.

The possible transformations are:

- ite $\top \ t_1 \ t_2 \leftrightarrow t_1$

- ite $\bot \ t_1 \ t_2 \leftrightarrow t_2$

- ite $\psi \ t \ t \leftrightarrow t$

- ite $\neg\varphi\, t_1\, t_2 \leftrightarrow$ ite $\varphi\, t_2\, t_1$

- ite $\top\, t_1\, t_2 \leftrightarrow t_1$

- ite $\bot\, t_1\, t_2 \leftrightarrow t_2$

- ite $\psi$ (ite $\psi\, t_1\, t_2$) $t_3 \leftrightarrow$ ite $\psi\, t_1\, t_3$

- ite $\psi\, t_1$ (ite $\psi\, t_2\, t_3$) $\leftrightarrow$ ite $\psi\, t_1\, t_3$

- ite $\psi\, \top\, \bot \leftrightarrow \psi$

- ite $\psi\, \bot\, \top \leftrightarrow \neg\psi$

- ite $\psi\, \top\, \varphi \leftrightarrow \psi \vee \varphi$

- ite $\psi\, \varphi\, \bot \leftrightarrow \psi \wedge \varphi$

- ite $\psi\, \bot\, \varphi \leftrightarrow \neg\psi \wedge \varphi$

- ite $\psi\, \varphi\, \top \leftrightarrow \neg\psi \vee \varphi$

### 5.11.3 Quantifiers

**Rule 74: qnt_simplify**
This rule simplifies a $\forall$ term with a constant predicate.

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

where $\varphi$ 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.

$$\frac{\Gamma, x_{k_1}, \ldots, x_{k_m}, x_{j_1} \mapsto t_{j_1}, \ldots, x_{j_o} \mapsto t_{j_o},\ \rhd j. \qquad\qquad \varphi \leftrightarrow \varphi' \qquad\qquad (\ldots)}{\Gamma\ \rhd k. \qquad\qquad Qx_1, \ldots, x_n.\varphi \leftrightarrow Qx_{k_1}, \ldots, x_{k_m}.\varphi'} \quad (\texttt{onepoint})$$

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 $\varphi$.

**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 $\forall x, y.\, x \simeq y \to f(x) \wedge f(y)$ 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**

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

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

**Rule 77: qnt_rm_unused**

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

where $m \leq n$ and $Q \in \{\forall, \exists\}$. Furthermore, $k_1, \ldots, k_m$ is a 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 $P$.

### 5.11.4 Equalities

**Rule 78: eq_simplify**
This rule simplifies an $\simeq$ term by applying equivalent transformations as long as possible. Hence, the general form is

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

where $\psi$ 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.

### 5.11.5 Arithmetic Operators

**Rule 79: div_simplify**

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

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

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

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

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

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

or

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

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

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

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

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

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

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

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 $\varphi$ is $\top$ if $t_1$ is strictly greater than $t_2$ and $\bot$ otherwise.

- $t < t \leftrightarrow \bot$

- $t_1 \le t_2 \leftrightarrow \varphi$ where $t_1$ and $t_2$ are numerical constants and $\varphi$ is $\top$ if $t_1$ is greater than $t_2$ or equal and $\bot$ otherwise.

- $t \le t \leftrightarrow \top$

- $t_1 \ge t_2 \leftrightarrow t_2 \le t_1$

- $t_1 < t_2 \leftrightarrow \neg(t_2 \le t_1)$

- $t_1 > t_2 \leftrightarrow \neg(t_1 \le t_2)$

### 5.11.6 Other Simplifications

**Rule 85: let**

This rule eliminats let. It has the form

$$\Gamma \ \rhd i_1. \qquad\qquad\qquad t_1 \simeq s_1 \qquad\qquad\qquad\qquad (\dots)$$

$$\vdots$$

$$\Gamma \ \rhd i_n. \qquad\qquad\qquad t_n \simeq s_n \qquad\qquad\qquad\qquad (\dots)$$

$$\vdots$$

$$\frac{\Gamma, x_1 \mapsto s_1, \dots, x_n \mapsto s_n, \ \rhd j. \qquad\qquad u \simeq u' \qquad\qquad (\dots)}{\Gamma \ \rhd k. \qquad\qquad (\text{let } x_1 := t_1, \dots, x_n := t_n.u) \simeq u' \qquad (\texttt{let}; i_1 \dots i_n)}$$

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:

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

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

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

and

$$\rhd i. \qquad\qquad\qquad (\text{distinct } \varphi_1 \ \varphi_2 \ \varphi_3 \dots) \leftrightarrow \bot \qquad\qquad (\texttt{distinct\_elim})$$

The general case is

$$\rhd i. \qquad\qquad (\text{distinct } t_1 \dots t_n) \leftrightarrow \bigwedge_{i=1}^{n} \bigwedge_{j=i+1}^{n} t_i \not\simeq t_j \qquad\qquad (\texttt{distinct\_elim})$$

**Rule 87: la_rw_eq**

$$\rhd i. \qquad\qquad\qquad (t \simeq u) \simeq (t \leq u \wedge u \leq t) \qquad\qquad\qquad (\texttt{la\_rw\_eq})$$

**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 $\wedge$ or $\vee$.

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

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

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

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

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

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

## 5.12 Term Reshaping

### Rule 89: bfun_elim

$$\triangleright i. \qquad\qquad\qquad \psi \qquad\qquad\qquad\qquad (\dots)$$
$$\triangleright j. \qquad\qquad\qquad \varphi \qquad\qquad\qquad\qquad (\texttt{bfun\_elim}; i)$$

The formula $\varphi$ is $\psi$ after boolean functions have been simplified. This happens in a two step process. Both steps recursively iterate over $\psi$. The first step expands quantified variable of type **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 **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.

### Rule 90: ite_intro
Either

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

or

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

The term $t$ (the formula $\varphi$) 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 **Bool**. The term $t'$ (the formular $\varphi'$) is equal to the term $t$ (the formular $\varphi'$) 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 $\varphi$ on the right side of the equality is $\varphi$ 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.

# 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] 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.

[10] 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.

[11] Hans-Jörg Schurr, Mathias Fleury, and Martin Desharnais. Reliable reconstruction of fine-grained proofs in a proof assistant. In *CADE 2021*, LNCS, 2021.