The Alethe Proof Format

An Evolving Specification and Reference

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

[1] Universidade Federal de Minas Gerais, Brazil
[2] Albert-Ludwig-Universität Freiburg, Germany
[3] Université de Liège, Belgium
[4] The University of Iowa, Iowa City, USA

Contents

Version caeee59-changed. Typeset on December 20, 2024.

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(T)-based SMT solvers.

The specification 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.

1 Introduction

This document is a reference of the Alethe[1] proof format. Alethe is designed to be a flexible format to represent unsatisfiability proofs generated by SMT solvers. Alethe proofs can be consumed by other systems, such as interactive theorem provers or proof checkers. The design is based on natural-deduction style structure and rules generating

[1] Alethe is a genus of small birds that occur in West Africa [9]. The name was chosen because it resembles the Greek word αλήθεια (alítheia) – truth.

and operating on first-order clauses. The Alethe proof format consists of two parts: the proof language based on SMT-LIB and a collection of proof rules. Section 2 introduces the language. First as an abstract language, then as a concrete syntax. Section 3 then discusses an abstract procedure to check Alethe proofs. This abstract checking procedure specifies the semantics of Alethe proofs. The Alethe proof rules are discussed in two sections. First, Section 4 discusses the core concepts behind the rules. Second, Section 5 presents a list of all proof rules currently used by veriT.

Alethe follows a few core design principles. First, proofs should be easy to understand by humans to ensure working with Alethe proofs is easy. Second, the language of the format should directly correspond to the language used by the solver. Since many solvers use the SMT-LIB language, Alethe also uses this language. Therefore, Alethe's base logic is the many-sorted first-order logic of SMT-LIB. Third, the format should be uniform for all theories used by SMT solvers. With the exception of clauses for propositional reasoning, there is no dedicated syntax for any theory.

The Alethe format was originally developed for the SMT solver veriT. 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 SMT solver cvc5 [1] (the successor of CVC4) supports Alethe experimentally as one of its multiple proof output formats. Alethe proofs can be reconstructed by the `smt` tactic of the proof assistant Isabelle/HOL [7, 8]. The SMTCoq tool can reconstruct an older version of the format in the proof assistant Coq [6]. An effort to update the tool to the latest version of Alethe is ongoing. Furthermore, Carcara is an experimental high-performance proof checker written in Rust.[2]

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 [5]. More recently, the format has gained support for reasoning typically used for processing, such as skolemization, substitutions, and other manipulations of bound variables [2].

1.1 Notations

The notation used in this document is similar to the notation used by the SMT-LIB standard. The Alethe proof format uses the SMT-LIB logic. Since the SMT-LIB language is based on S-expressions, SMT-LIB formulas are written using a λ-calculus style. That is, instead of $f(1, 2)$, we write $(f\,1\,2)$. However, connectives that are usually written using infix notation, also use infix notation here. That is, we write $t_1 \lor t_2$, not $(\lor\, t_1\, t_2)$.

We use x, y, z to indicate variables, f, g for functions, and P, Q for predicates (functions with co-domain sort **Bool**. To indicate terms we use t, u and to indicate formulas (terms of sort **Bool**) we use φ, ψ. To distinguish syntactic equality and the SMT-LIB equality predicate, we write $=$ for the former, and $\approx$ for the latter. We will write pre-defined

[2] Available at `https://github.com/ufmg-smite/carcara`.

SMT-LIB symbols, such as sorts and functions, in bold (e.g., **Bool**, **ite**).

We will use θ to denote a substitution. The notation $[x_1 \mapsto t_1, ..., x_n \mapsto t_n]$ denotes the substitution that maps x_i to t_i for $1 \leq i \leq n$ and corresponds to the identity function for all other variables. If θ and η are two substitutions, then $\theta\eta$ denotes the result of first applying θ and then η (i.e., $\eta(\theta(.))$). A substitution can naturally be extended to a function that maps terms to terms by replacing the occurrences of free variables. The application of a substitution θ to a term t (i.e., $\theta(t)$) is capture-avoiding; bound variables in t are renamed as necessary.

We write $t[u]$ for a term that contains the term u as a subterm. If u is subsequently replaced by a term v, we write $t[v]$ for the new term. We also use this notation with multiple terms. The notation $t[u_1, \dots, u_n]$ stands for a term may contain the pairwise distinct terms $u_1, \dots, u_n$. Then, $t[s_1, \dots, s_n]$ is the respective term where the variables $u_1, \dots, u_n$ are simultaneously replaced by $s_1, \dots, s_n$. Usually, $u_1, \dots, u_n$ will be variables.

Note that we will introduce the Alethe specific notation to write proof steps in the following sections.

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 chapter. The section first introduces an abstract notation to write Alethe proofs. Then, it introduces the concrete, SMT-LIB-based syntax. Finally, we show how a concrete Alethe proof can be checked.

Example 1. The following example shows a simple Alethe proof expressed in the abstract notation used in this document. It uses quantifier instantiation and resolution to show a contradiction. The paragraphs below describe the concepts necessary to understand the proof step by step.

1. $\rhd$	$\forall x. (Px)$	assume
2. $\rhd$	$\neg(Pa)$	assume
3. $\rhd$	$\neg(\forall x. (Px)) \vee (Pa)$	forall_inst $[(x, a)]$
4. $\rhd$	$\neg(\forall x. (Px)), (Pa)$	(or 3)
5. $\rhd$	$\bot$	(resolution $1, 2, 4$)

Many-Sorted First-Order Logic. Alethe builds on the SMT-LIB language. This includes its many-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 the distinguished **Bool** sort is always available. However, Alethe also extends this logic with Hilbert's choice operator ε. The term $\varepsilon x. \varphi[x]$ stands for a value v such that $\varphi[v]$ 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 $(\forall x. \varphi \approx \psi)$, then $(\varepsilon x. \varphi) \approx (\varepsilon x. \psi)$ must also hold. Note that choice terms can only appear in Alethe proofs, not in SMT-LIB problems.

Steps. A proof in the Alethe language is an indexed list of steps. To mimic the concrete syntax of Alethe proofs, proof steps in the abstract notation have the form

$$i.\ c_1, \dots, c_j \triangleright \qquad\qquad l_1, \dots, l_k \qquad\qquad (\mathsf{rule}\ p_1, \dots, p_n)\,[a_1, \dots, a_m]$$

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, $l_1, \dots, l_k$ is a clause with the literals l_i. It is the conclusion of the step. If a step has the empty clause as its conclusion (i.e., $k = 0$) we write $\bot$. While this muddles the water a bit with regard 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 (see Section 5). Furthermore, each step has a possibly empty set of premises $\{p_1, \dots, p_n\} \subseteq \mathbb{I}$, 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. If the list of premises is empty, we will drop the parentheses around the proof rule. 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 $c_1, \dots, c_j$ is the *context* of the step. Contexts are discussed below. Every proof ends with a step that has the empty clause as the conclusion and an empty context. The list of proof rules in Section 5 also uses this notation to define the proof rules.

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. An assume step introduces a term as an assumption. The proof starts with a number of assume steps. Each such step corresponds to an input assertion. Within a subproof, 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. In the concrete syntax, assume steps have a dedicated command `assume` to clearly distinguish them from normal steps that use the `step` command (see Section 2.1).

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

Subproofs and Lemmas. Alethe uses subproofs to prove lemmas and to create and 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 from φ, the subproof rule deduces the clause $\neg\varphi, \psi$ that discharges the local assumption φ. A subproof step cannot use a premise from a subproof nested within the current subproof.

Subproofs are also used to manipulate the context. As the example below shows, the abstract notation denotes subproofs by a frame around the steps in the subproof. In this case the subproof concludes with a step that does not use the subproof rule, but another rule, such as the bind rule.

Example 2. This example shows a refutation of the formula $(2+2) \approx 5$. The proof uses a subproof to prove the lemma $((2+2) \approx 5) \Rightarrow 4 \approx 5$.

1. $\triangleright$	$(2+2) \approx 5$	assume
2. $\triangleright$	$(2+2) \approx 5$	assume
3. $\triangleright$	$(2+2) \approx 4$	sum_simplify
4. $\triangleright$	$4 \approx 5$	(trans $2, 3$)
5. $\triangleright$	$\neg((2+2) \approx 5), 4 \approx 5$	subproof
6. $\triangleright$	$(4 \approx 5) \approx \bot$	eq_simplify
7. $\triangleright$	$\neg((4 \approx 5) \approx \bot), \neg(4 \approx 5), \bot$	equiv_pos2
8. $\triangleright$	$\bot$	(resolution $1, 5, 6, 7$)

Contexts. A specificity of the Alethe proofs is the step context. Alethe contexts are a general mechanism to write substitutions and to change them by attaching new elements. A context is a possibly empty list $c_1, \dots, c_l$, where each element 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* the variable. The second case is a mapping. Throughout this chapter, Γ denotes an arbitrary context.

Every context Γ induces a capture-avoiding substitution $\mathrm{subst}(\Gamma)$. If Γ is the empty list, $\mathrm{subst}(\Gamma)$ is the empty substitution, i.e, the identity function. The first case fixes x_n and allows the context to shadow a previously defined substitution for x_n:

$$\mathrm{subst}([c_1, \dots, c_{n-1}, x_n]) \text{ is } \mathrm{subst}([c_1, \dots, c_{n-1}]), \text{ but } x_n \text{ maps to } x_n.$$

When Γ ends in a mapping, the substitution is extended with this mapping:

$$\mathrm{subst}([c_1, \dots, c_{n-1}, x_n \mapsto t_n]) = \mathrm{subst}([c_1, \dots, c_{n-1}]) \circ \{x_n \mapsto t_n\}.$$

The following example illustrates this idea.

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

Contexts are used to express proofs about the preprocessing of terms. The conclusions of proof rules that use contexts always have the form

i. $\Gamma \triangleright$	$t \approx u$	(rule, ...)

where the term t is the original term, and u is the term after preprocessing. Tautologies with contexts correspond to judgments $\vDash_T \mathrm{subst}(\Gamma)(t) \approx u$. Note that, some proof rules require an empty context. In the list of proof rules in Section 5 this is indicated by omitting the Γ.

The substitution induced by Γ is capture-avoiding. Hence, some bound variables could be renamed in $\mathrm{subst}(\Gamma)(t)$ with respect to the original term t. A consequence of this is that steps that use a context must be checked under α-equivalence. The bind rule can be used to express renaming of bound variables explicitly. The refl rule, on the other hand, can be exploited to directly rename bound variables without an explicit proof.

Formally, the context can be translated to λ-abstractions and applications. This is discussed in Section 3.

Example 3. This example shows a proof that uses a subproof with a context to rename a bound variable.

1.		$\triangleright$	$\forall x. (P x)$	assume
2.		$\triangleright$	$\neg(\forall y. (P y))$	assume
3.	$y, x \mapsto y$	$\triangleright$	$x \approx y$	refl
4.	$y, x \mapsto y$	$\triangleright$	$(P x) \approx (P y)$	(cong 3)
5.		$\triangleright$	$\forall x. (P x) \approx \forall y. (P y)$	bind
6.		$\triangleright$	$\neg(\forall x. (P x) \approx \forall y. (P y)), \neg(\forall x. (P x)), (\forall y. (P y))$	equiv_pos2
7.		$\triangleright$	$\bot$	(resolution $1, 2, 5, 6$)

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 sole exception is the topmost equality in the conclusion of steps with non-empty context. The order of the arguments of this equality can never change. As described above, all rules that accept a non-empty context have a conclusion of the form $t \approx u$. Since the context represents a substitution applied to the left-hand side, this equality symbol is not symmetric.

The SMT solver veriT currently applies this freedom in a restricted form: equalities are reordered only 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 everywhere.

2.1 The Syntax

The concrete text representation of the Alethe proofs is based on the SMT-LIB standard. Figure 1 shows an example proof as printed by veriT with light edits for readability. The format broadly follows the SMT-LIB standard. Input problems in the SMT-LIB format are scripts. An SMT-LIB script is a list of commands that manipulate the SMT solver. For example, `assert` introduces an assertion, `check-sat` starts solving, and `get-proof` instructs the SMT solver to print the proof. Alethe mirrors this structure. Therefore, beside the SMT-LIB logic and term language, it also uses commands to structure the proof. An Alethe proof is a list of commands.

Every Alethe proof is associated with an SMT-LIB problem that is proved by the Alethe proof. This can either be a concrete problem file or, if the incremental solving commands of SMT-LIB are used, the implicit problem constructed at the invocation of a `get-proof` command. In this document, we will call this SMT-LIB problem the

```
(assume h1 (not (p a)))
(assume h2 (forall ((z1 U)) (forall ((z2 U)) (p z2))))
...
(anchor :step t9 :args ((vr4 U) (:= (z2 U) vr4)))
(step t9.t1 (cl (= z2 vr4)) :rule refl)
(step t9.t2 (cl (= (p z2) (p vr4)))
:rule cong :premises (t9.t1))
(step t9 (cl (= (forall ((z2 U)) (p z2))
(forall ((vr4 U)) (p vr4))))
:rule bind)
...
(step t14 (cl (forall ((vr5 U)) (p vr5)))
:rule th_resolution :premises (t11 t12 t13))
(step t15 (cl (or (not (forall ((vr5 U)) (p vr5)))
(p a)))
:rule forall_inst :args ((:= vr5 a)))
(step t16 (cl (not (forall ((vr5 U)) (p vr5))) (p a))
:rule or :premises (t15))
(step t17 (cl) :rule resolution :premises (t16 h1 t14))
```

Figure 1: Example proof output. Assumptions are introduced; a subproof renames bound variables; the proof finishes with instantiation and resolution steps.

input problem. An Alethe proof inherits the namespace of its SMT-LIB problem. All symbols declared or defined in the input problem remain declared or defined, respectively. Furthermore, the symbolic names introduced by the `:named` annotation also stay valid and can be used in the script. For the purpose of the proof rules, terms are treated as if proxy names introduced by `:named` annotations have been expanded and annotations have been removed. For example, the term `(or (! (P a) :named baz) (! baz :foo))` and `(or (P a) (P a))` are considered to be syntactically equal. Here `:foo` is an arbitrary SMT-LIB annotation.

Figure 2 shows the grammar of the proof text. It is based on the SMT-LIB grammar, as defined in the SMT-LIB standard [3, Appendix B]. The non-terminals ⟨`attribute`⟩, ⟨`function_def`⟩, ⟨`sorted_var`⟩, and ⟨`term`⟩ are as defined in the standard. A special restriction applies to the ⟨`symbol`⟩ non-terminal. Alethe has an extended set of number literals. Since these can start with a negation sign, they overlap SMT-LIB's ⟨`symbol`⟩ non-terminal. For example, `-1` is a valid ⟨`symbol`⟩ in SMT-LIB. These sequences cannot be used as symbols when using Alethe. Note that symbols are also used to name user defined constants and functions in the input problem. Hence, Alethe cannot express proofs about problems that use such symbols.

Alethe proofs are 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 semantics of an assumption justifies the presence of a dedicated

```
              A ⟨symbol⟩ is an SMT-LIB ⟨symbol⟩ that is not a
                                 −⟨numeral⟩/⟨positive_numeral⟩,
                                 −⟨numeral⟩, or − ⟨decimal⟩.
                   ⟨proof⟩  :=   ⟨proof_command⟩*
           ⟨proof_command⟩  :=   (assume ⟨symbol⟩ ⟨proof_term⟩ ⟨attribute⟩* )
                             |   (step ⟨symbol⟩ ⟨clause⟩ :rule ⟨symbol⟩
                                   ⟨premises_annotation⟩?
                                   ⟨args_annotation⟩? ⟨attribute⟩* )
                             |   (anchor :step ⟨symbol⟩
                                   ⟨context_annotation⟩? ⟨attribute⟩* )
                             |   (define-fun ⟨function_def⟩ )
                  ⟨clause⟩  :=   (cl ⟨proof_term⟩* )
              ⟨proof_term⟩  :=   ⟨term⟩ extended with
                                 (choice ( ⟨sorted_var⟩ ) ⟨proof_term⟩ )
                             |   ⟨rational⟩
                             |   ⟨nonpositive_numeral⟩
                             |   ⟨nonpositive_decimal⟩
     ⟨premises_annotation⟩  :=   :premises ( ⟨symbol⟩+)
         ⟨args_annotation⟩  :=   :args ( ⟨step_arg⟩+ )
                ⟨step_arg⟩  :=   ⟨symbol⟩ | ( ⟨symbol⟩ ⟨proof_term⟩ )
      ⟨context_annotation⟩  :=   :args ( ⟨context_assignment⟩+ )
      ⟨context_assignment⟩  :=   ⟨sorted_var⟩
                             |   (:= ⟨sorted_var⟩ ⟨proof_term⟩ )
        ⟨positive_numeral⟩  :=   any ⟨numeral⟩ except 0
                ⟨rational⟩  :=   −?⟨numeral⟩/⟨positive_numeral⟩
     ⟨nonpositive_numeral⟩  :=   −⟨numeral⟩
     ⟨nonpositive_decimal⟩  :=   −⟨decimal⟩
```

Figure 2: The proof grammar.

command: assumptions are neither tautological nor derived from premises. The **step** command, on the other hand, introduces a derived or tautological clause. 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. It must be unique for each **assume** and **step** command. A special restriction applies to the **assume** commands not within a subproof, which reference assertions in the input SMT-LIB problem. To simplify proof checking, the **assume** command must use the name assigned to the asserted formula if there is any. For example, if the input problem contains `(assert (! (P c) :named foo))`, then the proof must refer to this assertion (if it is needed in the proof) as `(assume foo (P c))`. Note that an SMT-LIB problem can assign a name to a term at any point, not only at its first occurrence. If a term has more than one name, any can be picked.

The second argument of **step** and **assume** is the conclusion of 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 to the standard `or` function extended to one argument, where it is equal to the identity, and zero arguments, where it is equal to `false`. Every step must use the `cl` operator, even if its conclusion is a unit clause. 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.

Furthermore, the syntax uses annotations as used by SMT-LIB. The original SMT-LIB syntax uses the non-terminal ⟨`attribute`⟩. The Alethe syntax uses some predefined annotation. To simplify parsing, the order in which those must be printed is strict. The **:premises** annotation denotes the premises and is skipped if the rule does not require premises. If the rule carries arguments, the **:args** annotation is used to denote them. Anchors have two annotations: **:step** provides the name of the step that concludes the subproof and **:args** provides the context as sorted variables and assignments. Note that in this annotation, the ⟨`symbol`⟩ non-terminal is intended to be a variable. After those pre-defined annotations, the solver can use additional annotations. This can be used for debugging, or other purposes. A consumer of Alethe proofs *must* be able to parse proofs that contain such annotations.

Terms The non-terminal ⟨`proof_term`⟩ is an extended version of the SMT-LIB non-terminal ⟨`term`⟩. First, it has an additional production for the `choice` binder. Second, it has productions to express rationals and negative integers concisely. A difficulty when parsing SMT-LIB terms is that numerical constants are not easy to distinguish from general terms. For example, $-\frac{1}{2}$ is written as `(/ (- 1) 2)`. The ⟨`rational`⟩ non-terminal makes it possible to write this constant as a single literal: `-1/2`. Furthermore, the non-terminals ⟨`nonpositive_numeral`⟩ and ⟨`nonpositive_decimal`⟩ achieve the same for unary negation.

The sorting rules for ⟨`proof_term`⟩ are as for SMT-LIB terms with one key difference. The sort of terms produced by ⟨`rational`⟩, ⟨`decimal`⟩, and ⟨`nonpositive_decimal`⟩ is always `Real`. The sort of terms produced by ⟨`integer`⟩ and ⟨`nonpositive_numeral`⟩ is always `Int`. For example, in standard SMT-LIB, the term `(+ 5 3)` has the sort `Int` in the logic `QF_LIA`, but has the sort `Real` in `QF_LRA`. In Alethe, this term always has the sort `Int`.

Subproofs The abstract notation denotes subproofs by marking them with a vertical line. To map this notation to a list of commands, Alethe uses the **anchor** command. This command indicates the start of a subproof. A subproof is concluded by a matching **step** command. This step must use a *concluding rule* (such as subproof, bind, and so forth).

After the **anchor** command, the proof uses **assume** commands to list the assumptions of the subproof. Subsequently, the subproof is a list of **step** commands that can use prior steps in the subproofs as premises. It is not allowed to issue **assume** commands after the first **step** command of a subproof has been issued.

In the abstract notation, every step is associated with a context. The concrete syntax uses anchors to optimize this. The context is manipulated in a nested way: if a step pops $c_1, \dots, c_n$ from a context Γ, there is an earlier step which pushes precisely $c_1, \dots, c_n$ onto the context. Since contexts can only be manipulated by push and pop, context manipulations are nested. The **anchor** commands push onto the context and the corresponding **step** commands pop from the context. To indicate these changes to the context, every anchor is associated with a list of fixed variables and mappings. The list is provided by the `:args` annotation. If the list is empty, the `:args` annotation is omitted[3]. Note that, when an **anchor** command extends a context Γ with some mappings $x_1 \mapsto t_1, \dots, x_n \mapsto t_n$, then the terms t_i are normalized by applying the substitution $\mathrm{subst}(\Gamma)$ to t_i. This is because the definition on page 6 extends the context by composing the substitutions.

The `:step` annotation of the anchor command is used to indicate the **step** command that will end the subproof. The clause of this **step** command is the conclusion of the subproof. While it is possible to infer the step that concludes a subproof from the structure of the proof, the explicit annotation simplifies proof checking and makes the proof easier to read. If the subproof uses a context, the `:args` annotation of the **anchor** command indicates the arguments added to the context for this subproof. The annotation provides the sort of fixed variables.

In the example proof (Figure 1) a subproof starts at the **anchor** command. It ends with the bind steps that finishes 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 the subproof. Hence, a proof checking tool can remove the steps of the subproof from memory after checking it.

Example 4. This example shows the proof from Example 3 expressed as a concrete proof.

```
(assume h1 (forall ((x S)) (P x)))
(assume h2 (not (forall ((y S)) (P y))))
(anchor :step t5 :args ((y S) (:= (x S) y)))
(step t3 (cl (= x y)) :rule refl)
(step t4 (cl (= (P x) (P y))) :rule cong :premises (t3))
(step t5 (cl (= (forall ((x S)) (P x)) (forall ((y S)) (P y))))
:rule bind)
(step t6 (cl (not (= (forall ((x S)) (P x)) (forall ((y S)) (P y))))
(not (forall ((x S)) (P x)))
(forall ((y S)) (P y))) :rule equiv_pos2)
(step t7 (cl) :rule resolution :premises (h1 h2 t5 t6))
```

[3] The only rule that allows an empty list is the subproof rule. Since this rule corresponds to implication introduction, it does not interact with binders.

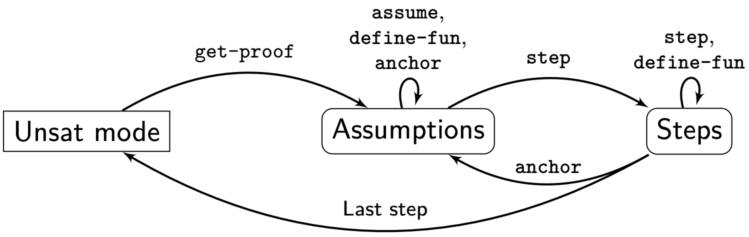

Figure 3: Abstract view of the transitions in an Alethe proof.

Alethe Proof Printing States Figure 2.1 shows the states of an Alethe proof abstractly. To generate a proof, the SMT solver must be in the *Unsat mode*, i.e., the solver determined that the problem is unsatisfiable. The SMT-LIB problem script then requests the proof by invoking the **`get-proof`** command. It is possible that this command fails. For example, if proof production was not activated up front. If there is no error, the proof is printed and printing starts with the assumptions. The solver prints the proof as a list of commands according to the state. The states ensure one constraint is maintained: assumptions can only appear at either the beginning of a proof or right after a subproof is started by the **`anchor`** command. They cannot be mixed with ordinary proof steps. This simplifies reconstruction. Each assumption printed at the beginning of the proof corresponds to assertions in the input problem, up to symmetry of equality. Proof printing concludes after the last step is printed and the solver returns to the Unsat mode and the user can issue further commands. Usually the last step is an outermost step (i.e., not within a subproof) that concludes the proof by deriving the empty clause, but this is not necessary. The solver is allowed to print some additional, useless, steps after the proof is concluded.

Sharing and Skolem Terms Usually, SMT solvers store terms internally in an efficient manner. A term data structure with perfect sharing ensures that every term is stored in memory precisely once. When printing the proof, this compact storage is unfolded. This leads to a blowup of the proof size.

Alethe can optionally use sharing[4] to print common subterms only once. This is realized using the standard naming mechanism of SMT-LIB. A term t is annotated 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. Alethe continues to use the names already used in the input problem. Hence, terms that already have a name in the input problem can be replaced by that name and new names introduced in the proof must not use names already used in the input problem.

To limit the number of names, an SMT solver can use the following simple approach used by veriT. Before printing the proof, it iterates over all terms of the proof and recursively descend into the terms. It marks every unmarked subterm it visits. If a

[4] For veriT this can be activated by the command-line option `--proof-with-sharing`.

visited term is already marked, the solver assigns a new name to this term. If a term already has a name, it does not descend further into the term. By doing so, it ensures that only terms that appear as child of two different parent terms get a name. Since a named term is replaced with its name after it first appearance, a term that only appears as a child of one single term does not need a distinct name. Thanks to the perfect sharing representation, testing if a term is marked takes constant time and the overall traversal takes linear time in the proof size.

To simplify reconstruction, Alethe can optionally[5] define Skolem constants as functions. In this case, the proof contains a list of **`define-fun`** commands that define shorthand 0-ary functions for the `(choice…)` terms needed. Without this option, no **`define-fun`** commands are issued, and the constants are expanded.

Implicit Transformations Overall, the following aspects are treated implicitly by Alethe.

- Symmetry of equalities that are not top-most equalities in steps with non-empty context.
- The order of literals in the clauses.
- The unfolding of names introduced by `(!` t `:named` s `)` in the original SMT-LIB problem or in the proof.
- The removal of other SMT-LIB annotations of the form `(!` t … `)`.
- The unfolding of function symbols introduced by **`define-fun`**.[6]
- If the input problem is in a logic without integers, then constants from ⟨`numeral`⟩ in the input problem will be printed using ⟨`decimal`⟩ or ⟨`rational`⟩ in the proof.

Alethe proofs contain steps for other aspects that are commonly left implicit, such as renaming of bound variables, and the application of substitutions.

3 Checking Alethe Proofs

In this section we present an abstract procedure to check if an Alethe proof is well-formed and valid. An Alethe proof is well-formed only if its anchors and steps are balanced. To check that this is the case, we replace innermost subproofs by holes until there is no subproof left. If the resulting reduced proof is free of anchors and steps that use concluding rules, then the overall proof is well-formed. To check if a proof is valid we have to check if all steps of a subproof adhere to the conditions of their rules before replacing the subproof by a hole. If all subproofs are valid and all steps in the reduced proof adhere to the conditions of their rule, then the entire proof is valid.

[5] For veriT by using the command-line option `--proof-define-skolems`.

[6] For veriT this is only used when the user introduces veriT to print Skolem terms as defined functions. User defined functions in the input problem are not supported by veriT in proof production mode.

Formally, an Alethe proof P is a list $[C_1, \dots, C_n]$ of steps and anchors. Since every step uses an unique index, we assume that each step C_i in P uses i as its index. The context only changes at anchors and subproof-concluding steps. Therefore, the elements of $C_1, \dots, C_n$ that are steps are not associated with a context. Instead, the context can be computed from the prior anchors. The anchors only ever extend the context.

To check an Alethe proof we can iteratively eliminate the first-innermost subproof, i.e., the innermost subproof that does not come after a complete subproof. The restriction to the first subproofs simplifies the calculation of the context of the steps in the subproof.

Definition 4.1 (First-Innermost Subproof)**.** Let P be the proof $[C_1, \dots, C_n]$ and $1 \leq \mathit{start} < \mathit{end} \leq n$ be two indices such that

- C_{start} is an anchor,
- C_{end} is a step that uses a concluding rule,
- no C_k with $k < \mathit{start}$ uses a concluding rule,
- no C_k with $\mathit{start} < k < \mathit{end}$ is an anchor or a step that uses a concluding rule.

Then $[C_{\mathit{start}}, \dots, C_{\mathit{end}}]$ is the first-innermost subproof of P.

Example 5. The proof in Example 4 has only one subproof and this subproof is also a first-innermost subproof. It is the subproof

```
(anchor :step t5 :args ((y S) (:= (x S) y)))
(step t3 (cl (= x y)) :rule refl)
(step t4 (cl (= (P x) (P y))) :rule cong :premises (t3))
(step t5 (cl (= (forall ((x S)) (P x)) (forall ((y S)) (P y))))
:rule bind)
```

It is easy to calculate the context of the first-innermost subproof.

Definition 5.1 (Calculated Context)**.** Let $[C_{\mathit{start}}, \dots, C_{\mathit{end}}]$ be the first-innermost subproof of P. Let $A_1, \dots, A_m$ be the anchors among $C_1, \dots, C_{\mathit{start}-1}$.

The calculated context of C_i is the context

$$c_{1,1}, \dots, c_{1,n_1}, \dots, c_{m,1}, \dots, c_{m,n_m}$$

where $c_{k,1}, \dots, c_{k,n_k}$ is the list of fixed variables and mappings associated with A_k.

Note that if C_i is an anchor, its calculated context does not contain the elements associated with C_i. Therefore, the context of C_{start} is the context of the steps before the subproof. Furthermore, the step C_{end} is the concluding step of the subproof and must have the same context as the steps surrounding the subproof. Hence, the context of C_{end} is the calculated context of C_{start}.

Example 6. The calculated context of the steps `t3` and `t5` in Example 4 is the context $x \mapsto y$. The calculated context of the concluding step `t5` and the anchor is empty.

A first-innermost subproof is valid if all its steps adhere to the conditions of their rule and only use premises that occur before them in the subproof. The conditions of each rule are listed in Section 5.

Definition 6.1 (Valid First-Innermost Subproof)**.** Let $[C_{\mathit{start}}, \dots, C_{\mathit{end}}]$ be the first-innermost subproof of P. The subproof is *valid* if

- all steps C_i with $\mathit{start} < i < \mathit{end}$ only use premises C_j with $\mathit{start} < j < i$,
- all C_i that are steps adhere to the conditions of their rule under the calculated context of C_i,
- the step C_{end} adheres to the conditions of its rule under the calculated context of C_{start}.

The only rule that can discharge assumptions in a subproof is the subproof rule. Therefore, an admissible subproof can only contain assume step if C_{end} is the subproof rule.

To eliminate a subproof we can replace the subproof with a hole that has at its conclusion the conclusion of the subproof. This is safe as long as the subproof that is eliminated is valid (see Section 3.2).

Definition 6.2. The function E eliminates the first-innermost subproof from a proof if there is one. Let P be a proof $[C_1, \dots C_n]$. Then $E(P) = P$ if P has no first-innermost subproof. Otherwise, P has the first-innermost subproof $[C_{\mathit{start}}, \dots, C_{\mathit{end}}]$, and $E(P) = [C_1, \dots, C_{\mathit{start}-1}, C', C_{\mathit{end}+1}, \dots, C_n]$ where C' is a new step that uses the hole rule and has the index, conclusion, and premises of C_{end}.

It is important to add the premises of C_{end} to C'. The let rule can use additional premises and omitting those premises results in an unsound step. We can apply E iteratively to a proof P until we reach the least fixed point. Since P is finite we will always reach a fixed point in finitely many steps. The result is a list $[P_0, P_1, P_2, \dots, P_{\mathit{last}}]$ where $P_0 = P$, $P_1 = E(P)$, $P_2 = E(E(P))$ and $P_{\mathit{last}} = E(P_{\mathit{last}})$.

Example 7. Applying E to the proof in Example 4 gives us the proof

```
(assume h1 (forall ((x S)) (P x)))
(assume h2 (not (forall ((y S)) (P y))))
(step t5 (cl (= (forall ((x S)) (P x)) (forall ((y S)) (P y))))
:rule hole)
(step t6 (cl (= (forall ((x S)) (P x)) (forall ((y S)) (P y)))
(not (forall ((x S)) (P x)))
(forall ((y S)) (P y)))) :rule equiv_pos2)
(step t7 (cl) :rule resolution :premises (h1 h2 t5 t6))
```

Since this proof contains no subproof, it is also P_{last}.

Definition 7.1 (Well-Formed Proof)**.** The Alethe proof P is well-formed if every step uses a unique index and P_{last} contains no anchor or step that uses a concluding rule.

Definition 7.2 (Valid Alethe Proof)**.** The proof P is a *valid Alethe proof* if

- P is well-formed,
- P does not contain any step that uses the hole rule,
- P_{last} contains a step that has the empty clause as its conclusion,
- the first-innermost subproof of every P_i, $i < last$ is valid,
- all steps C_i in P_{last} only use premises C_j in P_{last} with $1 \leq j < i$,
- all steps C_i in P_{last} adhere to the conditions of their rule under the empty context.

The condition that P contains no hole ensures that the original proof is complete and holes are only introduced by eliminating valid subproofs.

Example 8. The proof in Example 4 is valid. The only subproof is valid, the proof contains no hole, and P_{last} contains the step t7 that concludes with the empty clause.

It is sometimes useful to speak about the steps that are not within a subproof. We call such a step an *outermost step*. In a well-formed proof those are the steps of P_{last}.

3.1 Contexts and Metaterms

We now direct our attention to subproofs with contexts. It is useful to give precise semantics to contexts to have the tools to check that rules that use contexts are sound. Contexts are local in the sense that they affect only the proof step they are applied to. For the full details on contexts see [2]. The presentation here is adapted from this publication, but omits some details.

To handle subproofs with contexts, we translate the contexts into λ-terms. This allows us to leverage the λ-calculus as an existing well-understood theory of binders. These λ-terms are called *metaterms*.

Definition 8.1 (Metaterm)**.** Metaterms are expressions constructed by the grammar

$$M ::= \boxed{\mathrm{t}} \mid \lambda x.\, M \mid (\lambda \bar{x}_n.\, M)\, \bar{t}_n$$

where t is an ordinary term and t_i and x_i have matching sorts for all $0 \leq i \leq 1$.

According to this definition, a metaterm is either a boxed term, a λ-abstraction, or an application to an uncurried λ-term. The annotation $\boxed{\mathrm{t}}$ delimits terms from the context, a simple λ-abstraction is used to express fixed variables, and the application expresses simulations substitution of n terms.[7]

[7] The box annotation used here is unrelated to the boxes within the SMT solver discussed in the introduction.

We use $=_{\alpha\beta}$ to denote syntactic equivalence modulo α-equivalence and β-reduction.

Proof steps with contexts can be encoded into proof steps with empty contexts, but with metaterms. A proof step

$$\text{i. } \Gamma \rhd \qquad t \approx u \qquad (\mathsf{rule}\ \bar{p}_n)\ [\bar{a}_m]$$

is encoded into

$$\text{i. } \rhd \qquad L(\Gamma)[t] \approx R(\Gamma)[u] \qquad (\mathsf{rule}\ \bar{p}_n)\ [\bar{a}_m]$$

where

$$\begin{aligned} L(\emptyset)[t] &= \boxed{t} & R(\emptyset)[u] &= \boxed{u} \\ L(x, \bar{c}_m)[t] &= \lambda x.\, L(\bar{c}_m)[t] & R(x, \bar{c}_m)[u] &= \lambda x.\, R(\bar{c}_m)[u] \\ L(\bar{x}_n \mapsto \bar{s}_n, \bar{c}_m)[t] &= (\lambda \bar{x}_n.\, L(\bar{c}_m)[t])\bar{s}_n & R(\bar{x}_n \mapsto \bar{s}_n, \bar{c}_m)[u] &= R(\bar{c}_m)[u] \end{aligned}$$

To achieve the same effect as using the subst() function described above, we can translate the terms into metaterms, perform β-reduction, and rename bound variables if necessary [2, Lemma 11].

Example 9. The example on page 6 becomes

$$\begin{aligned} &L(x \mapsto 7, x \mapsto g(x))[x] = (\lambda x.\, (\lambda x.\, \boxed{x})\, (g(x)))\, 7 =_{\alpha\beta} \boxed{g(7)} \\ &L(x \mapsto 7, x, x \mapsto g(x))[x] = (\lambda x.\, \lambda x.\, (\lambda x.\, \boxed{x})\, (g(x)))\, 7 =_{\alpha\beta} \lambda x.\, \boxed{g(x)} \end{aligned}$$

Most proof rules that operate with contexts can easily be translated into proof rules using metaterms. The exception are the tautologous rules, such as refl and the $\cdots$_simplify rules.

Steps that use such rules always encode a judgment $\models \Gamma \rhd t \approx u$. With the encoding described above we get $L(\Gamma)[t] \approx R(\Gamma)[u] =_{\alpha\beta} \lambda \bar{x}_n.\, \boxed{t'} \approx \lambda \bar{x}_n.\, \boxed{u'}$ with some terms t', u'. To handle those terms, we use the reify() function. This function is defined as

$$\mathrm{reify}(\lambda \bar{x}_n.\, \boxed{t} \approx \lambda \bar{x}_n.\, \boxed{u}) = \forall \bar{x}_n.\, (t \approx u).$$

Therefore, all tautological rules with contexts represent a judgment
$\models \mathrm{reify}(T \approx U)$ where $T =_{\alpha\beta} L(\Gamma)[t]$ and $U =_{\alpha\beta} R(\Gamma)[u]$.

Example 10. Consider the step

$$\text{i. } y, x \mapsto y \rhd \qquad x + 0 \approx y \qquad \mathsf{sum\_simplify}$$

Translating the context into metaterms leads to

i. $\triangleright$ $(\lambda y.\, (\lambda x.\, \boxed{\mathrm{x} + 0})\, y) \approx (\lambda y.\, \boxed{\mathrm{y}})$ sum_simplify

Applying β-reduction leads to

i. $\triangleright$ $(\lambda y.\, \boxed{\mathrm{y} + 0}) \approx (\lambda y.\, \boxed{\mathrm{y}})$ sum_simplify

Finally, using reify() leads to

i. $\triangleright$ $\forall y.\, (y + 0 \approx y)$ sum_simplify

This obviously holds in the theory of linear arithmetic.

3.2 Soundness

Any proof calculus should be sound. In the case of Alethe, most proof rules are standard rules, or simple tautologies. The rules that use context are unusual, but those proof rules were previously shown to be sound [2]. Alethe does not use any rules that are merely satisfiability preserving. The skolemization rules replace the bound variables with choice terms instead of fresh symbols.[8] All Alethe rules express semantic implications. Overall, we assume in this document that the proof rules and proofs written in the abstract notation are sound.

Nevertheless, a modest gap remains. The concrete, command-based syntax does not precisely correspond to the abstract notation. In this section we will address the soundness of concrete Alethe proofs.

Theorem 10.1 (Soundness of Concrete Alethe Proofs)**.** If there is a valid Alethe proof $P = [C_1, \dots, C_n]$ that has the formulas $\varphi_1, \dots, \varphi_m$ as the conclusions of the outermost assume steps, then

$$\varphi_1, \dots, \varphi_m \vDash \bot.$$

Here, $\vDash$ represents semantic consequence in the many-sorted first order logic of SMT-LIB with the theories of uninterpreted functions and linear arithmetic extended with the choice operator and clauses.

To show the soundness of a valid Alethe proof $P = [C_1, \dots, C_n]$, we can use the same approach as for the definition of validity: consider first-innermost subproof first and then replace them by holes. Since valid proofs do not contain holes, we have to generalize the induction to allow holes that were introduced by the elimination of subproofs. We start with simple subproofs with empty contexts and without nested subproofs.

Lemma 10.1. Let P be a proof that contains a valid first-innermost subproof where C_{end} is a subproof step. Let ψ be the conclusion of C_{end}. Then $\vDash \psi$ holds.

[8] The `define-fun` function can introduce fresh symbols, but we will assume here that those commands have been eliminated by unfolding the definition.

Proof. First, we use induction on the number of steps n after C_{start}. Let ψ_n be the conclusion of $C_{start+n}$ and V_n the conclusions of the assume steps in $[C_{start}, \dots, C_{start+n}]$. Assumptions always introduce unit clauses. We will identify unit clauses with their single literal. We will show $V_n \vDash \psi_n$ if $start + n < end$.

If $n = 1$, then $C_{start+n} = C_{start+1}$ must either be a tautology, or an assume step. In the first case, $\vDash \psi_{start+1}$ holds, and in the second case $\psi_{start+1} \vDash \psi_{start+1}$ holds.

For subsequent n, $C_{start+n}$ is either an ordinary step, or an assume step. In the second case, $\psi_{start+n} \vDash \psi_{start+n}$ which can be weakened to $V_n \vDash \psi_{start+n}$. In the first case, the step $C_{start+n}$ has a set of premises S. For each step $C_{start+i} \in S$ we have $i < n$ and $V_i \vDash \psi_{start+i}$ due to the induction hypothesis. Since $i < n$, the premises V_i are a subset of V_n and we can weaken $V_i \vDash \psi_{start+i}$ to $V_n \vDash \psi_{start+i}$. Since all premises of $C_{start+n}$ are the consequence of V_n we get $V_n \vDash \psi_n$.

The step C_{end-1} is the last step of the subproof that does not use a concluding rule. At this step we have $V_{end-1} \vDash \psi_{end-1}$. Since C_{end} is not an assume step, the set $V_{end-1} = \{\varphi_1, \dots, \varphi_m\}$ contains all assumptions in the subproof. By the deduction theorem we get

$$\vDash \varphi_1 \wedge \cdots \wedge \varphi_m \rightarrow \psi_{end-1}.$$

This can be transformed into the clause

$$\vDash \neg\varphi_1, \cdots, \neg\varphi_m, l_1, \dots, l_o.$$

where $l_1, \dots, l_o$ are the literals of ψ_{end-1}. This clause is exactly the conclusion of the step C_{end} according to the definition of the subproof rule. □

We can do the same reasoning as for Lemma 10.1 for subproofs with contexts. This is slightly complicated by the let rule that can use extra premises.

Lemma 10.2. Let P be a proof that contains a valid first-innermost subproof where C_{end} is a step using one of: bind, sko_ex, sko_forall, onepoint, let.

Then $V \vDash \Gamma \rhd \psi$ where Γ is the calculated context of C_{start} and ψ is the conclusion of C_{end}. The set V is empty if C_{end} does not use the let rule. Otherwise, it contains all conclusions of the assume steps among $[C_\delta, \dots, C_{start}]$ where δ is either the largest index $\delta < start$ such that s_δ is an anchor, or 1 if no such index exist. Hence, there is no anchor between C_δ and C_{start}.

Proof. The step C_{start} is an anchor due to the definition of innermost-first subproof. Let $c_1, \dots, c_n$ be the context introduced by the anchor C_{start}, and let Γ be the calculated context of C_{start}. $\Gamma' = \Gamma, c_1, \dots, c_n$. is the calculated context of the steps in the subproof after C_{start}.

The step C_{end} is a step

$$\begin{array}{r|llr} & & \cdots & \\ end-1. & \Gamma' \rhd & \psi' & (\dots) \\ \hline end. & \Gamma \rhd & \psi & (\mathsf{rule}\ i_1, \dots, i_n) \end{array}$$

Since we assume the step C_{end} is correctly employed, $\vDash \Gamma \rhd \psi$ holds, as long as $\vDash \Gamma' \rhd \psi'$ holds.

We perform the same induction as for Lemma 10.1 over the steps in $[C_{start}, \dots, C_{end}]$. Since C_{end} does not use the subproof rule, the subproof does not contain any assumptions and V_i stays empty. Again, we are interested in the step C_{end-1}. At this step we get $\vDash \Gamma' \rhd \psi'$.

Only the let rule uses additional premises $C_{i_1}, \dots, C_{i_n}$. Hence, for all other rules, the conclusion cannot depend on any step outside the subproof and V is empty. Due to the definition of first-innermost subproof, all steps $C_{i_1}, \dots, C_{i_n}$ are in the same subproof that starts at C_δ.

The steps $C_{i_1}, \dots, C_{i_n}$ might depend on some assume steps that appear before them in their subproof. This is the case if the steps are outermost steps, or if the subproof that starts at C_δ concludes with a subproof step. In this case we can, as we saw in the proof of Lemma 10.1, weaken their judgments to include all assumptions in $[C_\delta, \dots, C_{start}]$.

If the subproof that starts at C_δ concludes with any other rule, then there cannot be any assumptions and V is empty. □

By using Lemma 10.1 and Lemma 10.2 we can now show that a valid, concrete Alethe proof is sound. That is, we can show Theorem 10.1.

Proof. Since $P = [C_1, \dots, C_n]$ is valid, all steps that do not use the hole rule adhere to their rule. Since we assume that the abstract notation and the rules are sound, we only have to worry about the steps using the hole rule. Those should be sound, i.e., for a hole step with the conclusion ψ, premises V, and context Γ the judgment $V \vDash \Gamma \rhd \psi$ must hold.

Since P is a valid proof there is a sequence $[P_0, \dots, P_{last}]$ as discussed in Section 3. For $i < last$, $E(P_i) = P_{i+1}$ replaces the first-innermost subproof in P_i by a hole with the conclusion ψ. Furthermore, the context of the introduced hole corresponds to the context Γ of the start of the subproof. Since P is a valid proof, the first-innermost subproof eliminated by E is always valid. Therefore, we can apply Lemma 10.1 or Lemma 10.2 to conclude that the hole introduced by E is sound.

Since P_0 does not contain any holes, the holes in each proof P_i are all introduced by innermost-first subproof elimination. Therefore, they are sound. In consequence, all holes in P_{last} are sound and we can perform the same argument as in the proof of Lemma 10.1 to the proof P_{last}.

Let j be the index of the step in P_{last} that concludes with the empty clause. Let $start = 1$ and $end = j$ in the argument of Lemma 10.1. This shows that $V \vDash \bot$, where V is the conclusion of the assume steps in the sublist $[C_1, \dots, C_j]$ of P_{last}. We can weaken this by adding the conclusions of the assume steps in $[C_j, \dots, C_n]$ of P_{last} to get $\varphi_1, \dots, \varphi_m \vDash \bot$. □

4 Core Concepts of the Alethe Rules

Together with the language, the Alethe format also includes a set of proof rule. Section 5 gives a full list of all proof rules. Currently, the proof rules correspond to the rules that the solver veriT can emit. 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: $\neg(\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n), \varphi_i$. Other rules derive their conclusion from a single premise. Those rules are primarily used to simplify Boolean connectives during preprocessing. For example, the implies rule eliminates an implication: From $\varphi_1 \rightarrow \varphi_2$, it deduces $\neg\varphi_1, \varphi_2$.

Resolution. CDCL(T)-based SMT solvers, and especially their SAT solvers, are fundamentally based on resolution of clauses. Hence, Alethe also has dedicated clauses and a resolution proof rule. However, 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 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 resolution rules also implicitly simplifies repeated negations at the head of literals.

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 $(\neg\forall \bar{x}_n . \varphi) \vee \varphi[\bar{t}_n]$, where φ is a term containing the free variables $\bar{x}_n$, and for each i the ground term t_i is a new term with the same sort as x_i.[9]

The arguments of a forall_inst step are the list $(x_1, t_1), \ldots, (x_n, t_n)$. While this information can be recovered from the term, providing it explicitly helps reconstruction because the implicit reordering of equalities 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 $(t_1 \leq t_2 \vee t_2 \leq t_1)$[10] and the main rule la_generic. The conclusion of

[9] For historical reasons, forall_inst has a unit clause with a disjunction as its conclusion and not the clause $(\neg\forall \bar{x}_n . \varphi), \varphi[\bar{t}_n]$.

[10] This rule also has a unit clause with a disjunction as its conclusion.

an la_generic step is a tautology $\neg\varphi_1, \neg\varphi_2, \dots, \neg\varphi_n$ where the φ_i are linear (in)equalities. Checking the validity of this clause amounts to checking the unsatisfiability of the system of linear equations $\varphi_1, \varphi_2, \dots, \varphi_n$. 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 11. The following example is the proof for the unsatisfiability of $(x + y < 1) \vee (3 < x)$, $x \approx 2$, and $0 \approx y$.

1.	$\rhd$	$(3 < x) \vee (x + y < 1)$	assume
2.	$\rhd$	$x \approx 2$	assume
3.	$\rhd$	$0 \approx y$	assume
4.	$\rhd$	$(3 < x), (x + y < 1)$	(or 1)
5.	$\rhd$	$\neg(3 < x), \neg(x \approx 2)$	la_generic $[1.0, 1.0]$
6.	$\rhd$	$\neg(3 < x)$	(resolution $2, 5$)
7.	$\rhd$	$x + y < 1$	(resolution $4, 6$)
8.	$\rhd$	$\neg(x + y < 1), \neg(x \approx 2) \vee \neg(0 \approx y)$	la_generic $[1.0, -1.0, 1.0]$
9.	$\rhd$	$\bot$	(resolution $8, 7, 2, 3$)

Skolemization and Other Preprocessing Steps One typical example for a rule with context is the sko_ex rule that is used to express skolemization of an existentially quantified variable. The conclusion of a step that uses this rules is an equality. The left-hand side is a formula starting with an existential quantifier over some variable x. In the formula on the right-hand side, the variable is replaced by the appropriate Skolem term. To provide a proof for the replacement, the sko_ex step uses one premise. The premise has a context that maps the existentially quantified variable to the appropriate Skolem term.

i.	$\Gamma, x \mapsto (\varepsilon x.\, \varphi)$	$\rhd$	$\varphi \approx \psi$	$(\dots)$
j.	Γ	$\rhd$	$(\exists x.\, \varphi) \approx \psi$	(sko_ex)

Example 12. To illustrate how such a rule is applied, consider the following example taken from [2]. Here the term $\neg p(\varepsilon x.\neg p(x))$ is skolemized. The refl rule expresses a simple tautology on the equality (reflexivity in this case), cong is functional congruence, and sko_forall works like sko_ex, except that the choice term is $\varepsilon x.\neg\varphi$.

1.	$x \mapsto (\varepsilon x.\, \neg(p\, x))$	$\rhd$	$x \approx \varepsilon x.\, \neg(p\, x)$	refl
2.	$x \mapsto (\varepsilon x.\, \neg(p\, x))$	$\rhd$	$(p\, x) \approx p(\varepsilon x.\, \neg(p\, x))$	(cong 1)
3.		$\rhd$	$(\forall x.\, (p\, x)) \approx (p\, (\varepsilon x.\, \neg(p\, x)))$	(sko_forall 2)
4.		$\rhd$	$(\neg\forall x.\, (p\, x)) \approx \neg(p\, (\varepsilon x.\, \neg(p\, x)))$	(cong 3)

4.1 Bitvector Reasoning with Bitblasting

A standard approach to handle bitvector reasoning in SMT solvers is bitblasting. Bitblasting works by translating bitvector functions to propositional formulas that model the logical circuit of the bitvector function.

To express bitblasting in Alethe proof rules, the the Alethe calculus uses multiple families of helper functions: **bbT**, $\mathbf{bitOf}_m$, **bvsize**, and $\mathbf{bv}_n^i$. Functions in the families are distinguished either by overloading (**bbT** and **bvsize**) or by explicit indexing ($\mathbf{bitOf}_m$ and $\mathbf{bv}_n^i$). To avoid name clashes with user defined functions, **bbT** is written as `@bbT`, **bitOf** as `@bitOf`, **bvsize** as `@bvsize`, and **bv** as `@bv`. The SMT-LIB standard specifies that simple symbols starting with "`@`" are reserved for solver generated functions.

The family **bbT** consists of one function for each bitvector sort (**BitVec** n).

$$\mathbf{bbT} : \underbrace{\mathbf{Bool} \dots \mathbf{Bool}}_{n} \, (\mathbf{BitVec}\ n).$$

Intuitively, the function **bbT** takes a list of boolean arguments and packs them into a bitvector. Let $\langle u_1, \dots, u_n \rangle$ denote a bitvector of sort (**BitVec** n) where $u_i = \top$ if the bit at position i is 1, and $u_i = \bot$ otherwise. The bit u_n is the least significant bit. Then

$$\mathbf{bbT}\ v_1 \dots v_n = \langle v_1, \dots, v_n \rangle.$$

The **bbT** functions could be defined in terms of standard SMT-LIB functions.

$$\begin{aligned}
\mathbf{bbT}\ v_1 \dots v_n := {} & \mathbf{concat}\,(\mathbf{concat}\,(\dots \\
& (\mathbf{concat}\,(\mathbf{ite}\, v_1\ \langle\top\rangle\ \langle\bot\rangle)\ (\mathbf{ite}\, v_2\ \langle\top\rangle\ \langle\bot\rangle)) \\
& \dots \\
& (\mathbf{ite}\, v_{n-1}\ \langle\top\rangle\ \langle\bot\rangle)) \\
& (\mathbf{ite}\, v_n\ \langle\top\rangle\ \langle\bot\rangle))
\end{aligned}$$

The functions $\mathbf{bitOf}_m$ are the inverse of **bbT**. They extract a bit of a bitvector as a boolean. Just as the built in **extract** symbol, $\mathbf{bitOf}_m$ is used as an indexed symbol. Hence, for $m \leq n$, we write `(_ @bitOf` m `)`, to denote functions

$$\mathbf{bitOf}_m : (\mathbf{BitVec}\ n) \rightarrow \mathbf{Bool}.$$

These functions are defined as

$$\mathbf{bitOf}_m \langle u_1, \dots, u_n \rangle := u_m.$$

The functions **bvsize** return the size of a bitvector. Formally, there is one **bvsize** for for each bitvector sort (**BitVec** n). Each **bvsize** is a constant function that returns n. Using notation:

$$\begin{aligned}
&\mathbf{bvsize} : (\mathbf{BitVec}\ n) \rightarrow \mathbb{N} \\
&\mathbf{bvsize}\ b := n
\end{aligned}$$

Finally, $\mathbf{bv}_n^i$ is a family of constants indexed by two parameters: a bitvector length n, and a natural number i. We write `(_ @bv`n i `)` for $\mathbf{bv}_n^i$. The space before n is omitted for historical reasons. Each $\mathbf{bv}_n^i$ is the bitvector constant that represents the bitvector of length n that encodes the integer i. Formally, it corresponds to `nat2bv[n](i)`, where `nat2bv` is defined as in the SMT-LIB standard.[11]

5 The Alethe Rules

This section provides a list of all proof rules supported by Alethe. To make this long list more accessible, the 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. The number in brackets is the position of the rule in the overall list of proof rules. Table 1 lists rules that serve a special purpose. Table 3 lists all rules that introduce tautologies. That is, regular rules that do not use premises.

The subsequent section, starting at 5.2, defines all rules and provides example proofs for complicated rules. The index of proof rules on page 55 can be used to quickly find the definition of rules.

5.1 Classifications of the Rules

Table 1: Special rules.

Rule	Description
assume (18)	Introduction of an assumption.
hole (19)	Placeholder for rules not defined here.
subproof (27)	Concludes a subproof and discharges local assumptions.

Table 2: Resolution and related rules.

Rule	Description
resolution (24)	Chain resolution of two or more clauses.
th_resolution (23)	As resolution, but used by theory solvers.
tautology (25)	Simplification of tautological clauses to $\top$.
contraction (26)	Removal of duplicated literals.

Table 3: Rules introducing tautologies.

Rule	Description
true (20)	$\top$
false (21)	$\neg\bot$
not_not (22)	$\neg(\neg\neg\varphi), \varphi$
la_generic (28)	Tautologous disjunction of linear inequalities.
lia_generic (29)	Tautologous disjunction of linear integer inequalities.

[11] See `https://smt-lib.github.io/theories-FixedSizeBitVectors.shtml`.

la_disequality (30)	$t_1 \approx t_2 \vee \neg(t_1 \leq t_2) \vee \neg(t_2 \leq t_1)$
la_totality (31)	$t_1 \leq t_2 \vee t_2 \leq t_1$
la_tautology (32)	A trivial linear tautology.
la_mult_pos (33)	Multiplication with a positive factor.
la_mult_neg (34)	Multiplication with a negative factor.
forall_inst (38)	Quantifier instantiation.
refl (39)	Reflexivity after applying the context.
eq_reflexive (42)	$t \approx t$ without context.
eq_transitive (43)	$\neg(t_1 \approx t_2), \ldots, \neg(t_{n-1} \approx t_n), t_1 \approx t_n$
eq_congruent (44)	$\neg(t_1 \approx u_1), \ldots, \neg(t_n \approx u_n), f(t_1, \ldots, t_n) \approx f(u_1, \ldots, u_n)$
eq_congruent_pred (45)	$\neg(t_1 \approx u_1), \ldots, \neg(t_n \approx u_n), P(t_1, \ldots, t_n) \approx P(u_1, \ldots, u_n)$
qnt_cnf (46)	Clausification of a quantified formula.
and_pos (64)	$\neg(\varphi_1 \wedge \ldots \wedge \varphi_n), \varphi_k$
and_neg (65)	$(\varphi_1 \wedge \ldots \wedge \varphi_n), \neg\varphi_1, \ldots, \neg\varphi_n$
or_pos (66)	$\neg(\varphi_1 \vee \ldots \vee \varphi_n), \varphi_1, \ldots, \varphi_n$
or_neg (67)	$(\varphi_1 \vee \ldots \vee \varphi_n), \neg\varphi_k$; with $1 \leq k \leq n$
xor_pos1 (68)	$\neg(\varphi_1 \,\mathbf{xor}\, \varphi_2), \varphi_1, \varphi_2$
xor_pos2 (69)	$\neg(\varphi_1 \,\mathbf{xor}\, \varphi_2), \neg\varphi_1, \neg\varphi_2$
xor_neg1 (70)	$\varphi_1 \,\mathbf{xor}\, \varphi_2, \varphi_1, \neg\varphi_2$
xor_neg2 (71)	$\varphi_1 \,\mathbf{xor}\, \varphi_2, \neg\varphi_1, \varphi_2$
implies_pos (72)	$\neg(\varphi_1 \rightarrow \varphi_2), \neg\varphi_1, \varphi_2$
implies_neg1 (73)	$\varphi_1 \rightarrow \varphi_2, \varphi_1$
implies_neg2 (74)	$\varphi_1 \rightarrow \varphi_2, \neg\varphi_2$
equiv_pos1 (75)	$\neg(\varphi_1 \approx \varphi_2), \varphi_1, \neg\varphi_2$
equiv_pos2 (76)	$\neg(\varphi_1 \approx \varphi_2), \neg\varphi_1, \varphi_2$
equiv_neg1 (77)	$\varphi_1 \approx \varphi_2, \neg\varphi_1, \neg\varphi_2$
equiv_neg2 (78)	$\varphi_1 \approx \varphi_2, \varphi_1, \varphi_2$
ite_pos1 (81)	$\neg(\mathbf{ite}\ \varphi_1\ \varphi_2\ \varphi_3), \varphi_1, \varphi_3$
ite_pos2 (82)	$\neg(\mathbf{ite}\ \varphi_1\ \varphi_2\ \varphi_3), \neg\varphi_1, \varphi_2$
ite_neg1 (83)	$(\mathbf{ite}\ \varphi_1\ \varphi_2\ \varphi_3), \varphi_1, \neg\varphi_3$
ite_neg2 (84)	$(\mathbf{ite}\ \varphi_1\ \varphi_2\ \varphi_3), \neg\varphi_1, \neg\varphi_2$
connective_def (87)	Definition of some connectives.
and_simplify (88)	Simplification of a conjunction.
or_simplify (89)	Simplification of a disjunction.
not_simplify (90)	Simplification of a Boolean negation.
implies_simplify (91)	Simplification of an implication.
equiv_simplify (92)	Simplification of an equivalence.
bool_simplify (93)	Simplification of combined Boolean connectives.
ac_simp (94)	Flattening and removal of duplicates for $\vee$ or $\wedge$.
ite_simplify (95)	Simplification of if-then-else.
qnt_simplify (96)	Simplification of constant quantified formulas.
qnt_join (98)	Joining of consecutive quantifiers.
qnt_rm_unused (99)	Removal of unused quantified variables.
eq_simplify (100)	Simplification of equality.

div_simplify (101)	Simplification of division.
prod_simplify (102)	Simplification of products.
unary_minus_simplify (103)	Simplification of the unary minus.
minus_simplify (104)	Simplification of subtractions.
sum_simplify (105)	Simplification of sums.
comp_simplify (106)	Simplification of arithmetic comparisons.
distinct_elim (109)	Elimination of the **distinct** operator.
la_rw_eq (110)	$(t \approx u) \approx (t \leq u \wedge u \leq t)$
nary_elim (111)	Eliminate n-ary application of operators via binary applications.
eq_symmetric (119)	Symmetry of equality as equivalence.

Table 4: Linear arithmetic rules.

Rule	Description
la_generic (28)	Tautologous disjunction of linear inequalities.
lia_generic (29)	Tautologous disjunction of linear integer inequalities.
la_disequality (30)	$t_1 \approx t_2 \vee \neg(t_1 \leq t_2) \vee \neg(t_2 \leq t_1)$
la_totality (31)	$t_1 \leq t_2 \vee t_2 \leq t_1$
la_tautology (32)	A trivial linear tautology.
la_mult_pos (33)	Multiplication with a positive factor.
la_mult_neg (34)	Multiplication with a negative factor.
la_rw_eq (110)	$(t \approx u) \approx (t \leq u \wedge u \leq t)$
div_simplify (101)	Simplification of division.
prod_simplify (102)	Simplification of products.
unary_minus_simplify (103)	Simplification of the unary minus.
minus_simplify (104)	Simplification of subtractions.
sum_simplify (105)	Simplification of sums.
comp_simplify (106)	Simplification of arithmetic comparisons.

Table 5: Quantifier handling.

Rule	Description
forall_inst (38)	Instantiation of a universal quantifier.
bind (35)	Renaming of bound variables.
sko_ex (36)	Skolemization of an existential quantifier.
sko_forall (37)	Skolemization of an universal quantifier.
qnt_cnf (46)	Clausification of quantified formulas.
qnt_simplify (96)	Simplification of constant quantified formulas.
onepoint (97)	The one-point rule.
qnt_join (98)	Joining of consecutive quantifiers.
qnt_rm_unused (99)	Removal of unused quantified variables.

Table 6: Skolemization rules.

Rule	Description
sko_ex (36)	Skolemization of existential variables.
sko_forall (37)	Skolemization of universal variables.

Table 7: Clausification rules. These rules can be used to perform propositional clausification.

Rule	Description
and (47)	And elimination.
not_or (48)	Elimination of a negated disjunction.
or (49)	Disjunction to clause.
not_and (52)	Distribution of negation over a conjunction.
xor1 (53)	From $(\mathbf{xor}\,\varphi_1\,\varphi_2)$ to φ_1, φ_2.
xor2 (54)	From $(\mathbf{xor}\,\varphi_1\,\varphi_2)$ to $\neg\varphi_1, \neg\varphi_2$.
not_xor1 (55)	From $\neg(\,\mathbf{xor}\,\varphi_1\,\varphi_2)$ to $\varphi_1, \neg\varphi_2$.
not_xor2 (56)	From $\neg(\,\mathbf{xor}\,\varphi_1\,\varphi_2)$ to $\neg\varphi_1, \varphi_2$.
implies (57)	From $\varphi_1 \rightarrow \varphi_2$ to $\neg\varphi_1, \varphi_2$.
not_implies1 (58)	From $\neg(\varphi_1 \rightarrow \varphi_2)$ to φ_1.
not_implies2 (59)	From $\neg(\varphi_1 \rightarrow \varphi_2)$ to $\neg\varphi_2$.
equiv1 (60)	From $\varphi_1 \approx \varphi_2$ to $\neg\varphi_1, \varphi_2$.
equiv2 (61)	From $\varphi_1 \approx \varphi_2$ to $\varphi_1, \neg\varphi_2$.
not_equiv1 (62)	From $\neg(\varphi_1 \approx \varphi_2)$ to φ_1, φ_2.
not_equiv2 (63)	From $\neg(\varphi_1 \approx \varphi_2)$ to $\neg\varphi_1, \neg\varphi_2$.
and_pos (64)	$\neg(\varphi_1 \land \ldots \land \varphi_n), \varphi_k$
and_neg (65)	$(\varphi_1 \land \ldots \land \varphi_n), \neg\varphi_1, \ldots, \neg\varphi_n$
or_pos (66)	$\neg(\varphi_1 \lor \ldots \lor \varphi_n), \varphi_1, \ldots, \varphi_n$
or_neg (67)	$(\varphi_1 \lor \ldots \lor \varphi_n), \neg\varphi_k$
xor_pos1 (68)	$\neg(\varphi_1\,\mathbf{xor}\,\varphi_2), \varphi_1, \varphi_2$
xor_pos2 (69)	$\neg(\varphi_1\,\mathbf{xor}\,\varphi_2), \neg\varphi_1, \neg\varphi_2$
xor_neg1 (70)	$\varphi_1\,\mathbf{xor}\,\varphi_2, \varphi_1, \neg\varphi_2$
xor_neg2 (71)	$\varphi_1\,\mathbf{xor}\,\varphi_2, \neg\varphi_1, \varphi_2$
implies_pos (72)	$\neg(\varphi_1 \rightarrow \varphi_2), \neg\varphi_1, \varphi_2$
implies_neg1 (73)	$\varphi_1 \rightarrow \varphi_2, \varphi_1$
implies_neg2 (74)	$\varphi_1 \rightarrow \varphi_2, \neg\varphi_2$
equiv_pos1 (75)	$\neg(\varphi_1 \approx \varphi_2), \varphi_1, \neg\varphi_2$
equiv_pos2 (76)	$\neg(\varphi_1 \approx \varphi_2), \neg\varphi_1, \varphi_2$
equiv_neg1 (77)	$\varphi_1 \approx \varphi_2, \neg\varphi_1, \neg\varphi_2$
equiv_neg2 (78)	$\varphi_1 \approx \varphi_2, \varphi_1, \varphi_2$
let (107)	Elimination of the **let** operator.
distinct_elim (109)	Elimination of the **distinct** operator.
nary_elim (111)	Elimination of n-ary application of operators.

Table 8: Simplification rules. These rules represent typical operator-level simplifications.

Rule	Description
connective_def (87)	Definition of the Boolean connectives.
and_simplify (88)	Simplification of a conjunction.
or_simplify (89)	Simplification of a disjunction.
not_simplify (90)	Simplification of a Boolean negation.
implies_simplify (91)	Simplification of an implication.
equiv_simplify (92)	Simplification of an equivalence.
bool_simplify (93)	Simplification of combined Boolean connectives.
ac_simp (94)	Flattening and removal of duplicates for $\vee$ or $\wedge$.
ite_simplify (95)	Simplification of if-then-else.
qnt_simplify (96)	Simplification of constant quantified formulas.
onepoint (97)	The one-point rule.
qnt_join (98)	Joining of consecutive quantifiers.
qnt_rm_unused (99)	Removal of unused quantified variables.
eq_simplify (100)	Simplification of equalities.
div_simplify (101)	Simplification of division.
prod_simplify (102)	Simplification of products.
unary_minus_simplify (103)	Simplification of the unary minus.
minus_simplify (104)	Simplification of subtractions.
sum_simplify (105)	Simplification of sums.
comp_simplify (106)	Simplification of arithmetic comparisons.
qnt_simplify (96)	Simplification of constant quantified formulas.

Table 9: Bitvector rules.

Rule	Description
bitblast_extract (114)	Bitblasting of **extract**.
bitblast_ult (115)	Bitblasting of **ult**.
bitblast_add (116)	Bitblasting of **add**.

Table 10: Rules used by cvc5, but not by veriT.

Rule	Description
bitblast_extract (114)	Bitblasting of **extract**.
bitblast_ult (115)	Bitblasting of **ult**.
bitblast_add (116)	Bitblasting of **add**.
la_mult_pos (33)	Multiplication with a positive factor.
la_mult_neg (34)	Multiplication with a negative factor.
symm (117)	Symmetry of equality.
not_symm (118)	Symmetry of not-equal.
reordering (51)	Reording of the literals in a clause.

Table 11: Rules used by the Carcara elaborator.

Rule	Description
weakening (50)	Weakening of a clause.
eq_symmetric (119)	Symmetry of equality as equivalence.

5.2 Rule List

Rule 1: pbblast_bveq

Consider bitvectors $\mathbf{x}$ and $\mathbf{y}$ of length n. The pseudo-boolean bitblasting of their equality is expressed by:

$i.\ \rhd \quad (= x\ y) \approx A \quad$ (pbblast_bveq)

The term "A" is the conjunction of PseudoBoolean constraints:

$$\bigwedge_{i=0}^{n-1} (x_i - y_i = 0)$$

Or similarly, using a single constraint:

$$\sum_{i=0}^{n-1} 2^i x_{n-i-1} - \sum_{i=0}^{n-1} 2^i y_{n-i-1} = 0$$

Rule 2: pbblast_bvult

The 'unsigned-less-than' operation over BitVectors with n bits is expressed using PseudoBoolean inequalities by:

$i.\ \rhd \quad (\textbf{bvult}\ x\ y) \approx A \quad$ (pbblast_bvult)

The term "A" is 'true' iff:

$$\sum_{i=0}^{n-1} 2^i \mathbf{y}_{n-i-1} - \sum_{i=0}^{n-1} 2^i \mathbf{x}_{n-i-1} \geq 1.$$

Rule 3: pbblast_bvugt

The 'unsigned-greater-than' operation over BitVectors with n bits is expressed using PseudoBoolean inequalities by:

$i.\ \rhd \quad (\textbf{bvugt}\ x\ y) \approx A \quad$ (pbblast_bvugt)

The term "A" is 'true' iff:

$$\sum_{i=0}^{n-1} 2^i \mathbf{x}_{n-i-1} - \sum_{i=0}^{n-1} 2^i \mathbf{y}_{n-i-1} \geq 1.$$

Or in terms of pbblast_bvult:

$i.\ \rhd \quad (\textbf{bvugt}\ x\ y) \approx (\textbf{bvult}\ y\ x) \quad$ (pbblast_bvugt)

Rule 4: pbblast_bvuge

The 'unsigned-greater-or-equal' operation over BitVectors with n bits is expressed using PseudoBoolean inequalities by:

$i.\ \rhd \quad (\textbf{bvuge}\ x\ y) \approx A \quad$ (pbblast_bvuge)

The term "A" is 'true' iff:

$$\sum_{i=0}^{n-1} 2^i \mathbf{x}_{n-i-1} - \sum_{i=0}^{n-1} 2^i \mathbf{y}_{n-i-1} \geq 0.$$

Rule 5: pbblast_bvule

The 'unsigned-greater-or-equal' operation over BitVectors with n bits is expressed using PseudoBoolean inequalities by:

$i.$ $\rhd$ $(\mathbf{bvule}\ x\ y) \approx A$ (pbblast_bvule)

The term "A" is 'true' iff:

$$\sum_{i=0}^{n-1} 2^i \mathbf{y}_{n-i-1} - \sum_{i=0}^{n-1} 2^i \mathbf{x}_{n-i-1} \geq 0.$$

Or in terms of pbblast_bvuge:

$i.$ $\rhd$ $(\mathbf{bvule}\ x\ y) \approx (\mathbf{bvuge}\ y\ x)$ (pbblast_bvule)

Rule 6: pbblast_bvslt

The 'signed-less-than' operation over BitVectors with n bits is expressed using PseudoBoolean inequalities by:

$i.$ $\rhd$ $(\mathbf{bvslt}\ x\ y) \approx A$ (pbblast_bvslt)

The term "A" is 'true' iff:

$$-(2^{n-1})\mathbf{y}_0 + \sum_{i=0}^{n-2} 2^i \mathbf{y}_{n-i-1} + 2^{n-1}\mathbf{x}_0 - \sum_{i=0}^{n-2} 2^i \mathbf{x}_{n-i-1} \geq 1$$

Rule 7: pbblast_bvsgt

The 'signed-greater-than' operation over BitVectors with n bits is expressed using PseudoBoolean inequalities by:

$i.$ $\rhd$ $(\mathbf{bvsgt}\ x\ y) \approx A$ (pbblast_bvsgt)

The term "A" is 'true' iff:

$$-(2^{n-1})\mathbf{x}_0 + \sum_{i=0}^{n-2} 2^i \mathbf{x}_{n-i-1} + 2^{n-1}\mathbf{y}_0 - \sum_{i=0}^{n-2} 2^i \mathbf{y}_{n-i-1} \geq 1$$

Or in terms of pbblast_bvslt:

$i.$ $\rhd$ $(\mathbf{bvsgt}\ x\ y) \approx (\mathbf{bvslt}\ y\ x)$ (pbblast_bvsgt)

Rule 8: pbblast_bvsge

The 'signed-greater-or-equal' operation over BitVectors with n bits is expressed using PseudoBoolean inequalities by:

$i.$ $\rhd$ $(\mathbf{bvsge}\ x\ y) \approx A$ (pbblast_bvsge)

The term "A" is 'true' iff:

$$-(2^{n-1})\mathbf{x}_0 + \sum_{i=0}^{n-2} 2^i \mathbf{x}_{n-i-1} + 2^{n-1}\mathbf{y}_0 - \sum_{i=0}^{n-2} 2^i \mathbf{y}_{n-i-1} \geq 0$$

Rule 9: pbblast_bvsle

The 'signed-less-or-equal' operation over BitVectors with n bits is expressed using PseudoBoolean inequalities by:

i. ▷ $(\mathbf{bvsle}\ x\ y) \approx A$ (pbblast_bvsle)

The term "A" is 'true' iff:

$$-(2^{n-1})\mathbf{y}_0 + \sum_{i=0}^{n-2} 2^i \mathbf{y}_{n-i-1} + 2^{n-1}\mathbf{x}_0 - \sum_{i=0}^{n-2} 2^i \mathbf{x}_{n-i-1} \geq 0$$

Or in terms of pbblast_bvsge:

i. ▷ $(\mathbf{bvsle}\ x\ y) \approx (\mathbf{bvsge}\ y\ x)$ (pbblast_bvsle)

Rule 10: pbblast_pbbvar

Conversion from a BitVector of n bits to n PseudoBoolean variables passed to pbbT:

i. ▷ $x \approx \mathbf{pbbT}\ x_1 \ldots x_{n+1}$ (pbblast_pbbvar)

Rule 11: pbblast_pbbconst

Constraints on each bit of the constant BitVector b:

i. ▷ $(b \approx \mathbf{pbbT}\ r) \wedge \bigwedge_{i=0}^{n-1} (r_i = \mathbf{PB\_ZERO\_OR\_ONE}(b_{n-i-1}))$ (pbblast_pbbconst)

In which we expand $\mathbf{PB\_ZERO\_OR\_ONE}(b_i)$ into:

- $(b_i = 0)$ if b_i is 0
- $(b_i = 1)$ if b_i is 1

Rule 12: pbblast_bvxor

The 'bvxor' operation over BitVectors with n bits is expressed using PseudoBoolean inequalities by:

i. ▷ $(\mathbf{bvxor}\ x\ y) \approx [r_0, \ldots, r_1] \wedge A$ (pbblast_bvxor)

The term "A" is the conjunction of these PseudoBoolean inequalities, for $0 \leq i < n$:

$$\begin{aligned}
-\mathbf{r}_i + \mathbf{x}_i + \mathbf{y}_i &\geq 0 \\
-\mathbf{r}_i - \mathbf{x}_i - \mathbf{y}_i &\geq -2 \\
\mathbf{r}_i + \mathbf{x}_i - \mathbf{y}_i &\geq 0 \\
\mathbf{r}_i - \mathbf{x}_i + \mathbf{y}_i &\geq 0
\end{aligned}$$

Rule 13: pbblast_step_bvand

The 'bvand' operation over BitVectors with n bits is expressed using PseudoBoolean inequalities by:

i. ▷ $(\mathbf{bvand}\ x\ y) \approx [r_0, \ldots, r_1] \wedge A$ (pbblast_step_bvand)

The term "A" is the conjunction of these PseudoBoolean inequalities, for $0 \leq i < n$:

$$\begin{aligned}
x_i - r_i &\geq 0 \\
y_i - r_i &\geq 0 \\
r_i - x_i - y_1 &\geq -1
\end{aligned}$$

Rule 14: cp_addition

A step of the cp_addition rule represents the addition of two pseudo-Boolean constraints using cutting planes reasoning. Pseudo-Boolean constraints are 0-1 integer linear inequalities of the form:

$$\sum_i a_i \cdot l_i \geq A$$

where A is called **constant**, a_i are **coefficients**, and l_i are **literals**, that are either:

- **plain** literal, a term `x`;
- **negated** literal, a term of the form `(- 1 x)`

where the `x` value is a pseudo-boolean variable, i.e. it will resolve to values `0` (false) or `1` (true). All these values are of sort `Int`.
To form a summation we use a list of added terms of form, `(+ <T1> <T2> ... 0)` and each term is `(* a_i <L1>)`, with a coefficient and a literal.

The cp_addition rule allows two pseudo-Boolean constraints to be added together, combining their coefficients and constants. Negated and plain literals over the same variable cancel each other.
Formally, given two constraints:

$$\sum_i a_i \cdot l_i \geq A \quad \text{and} \quad \sum_i b_i \cdot l_i \geq B,$$

the result of applying the cp_addition rule is:

$$\sum_i (a_i + b_i) \cdot l_i \geq (A + B).$$

A cp_addition step in the proof has the form:

$$\begin{array}{lcr} i_1. \ \triangleright & \sum_i a_i l_i \geq A & \\ i_2. \ \triangleright & \sum_i b_i l_i \geq B & \\ j. \ \ \triangleright & \sum_i (a_i + b_i) l_i \geq (A + B) & (\text{cp\_addition } i_1, i_2) \end{array}$$

To verify a cp_addition step, one must check that the two given pseudo-Boolean constraints are valid and that their combination satisfies the addition rule.

Example 14.1. A simple cp_addition step might look like this:

```
(assume c1 (>= (+ (* 1 x1) 0) 1))
(assume c2 (>= (+ (* 1 x2) 0) 1))
(step t1 (cl (>= (+ (* 1 x1) (* 1 x2) 0) 2))
:rule cp_addition :premises (c1 c2))
```

In this example, we are adding two constraints.

$$x_1 \geq 1 \quad \text{and} \quad x_2 \geq 1.$$

After applying the cp_addition rule, the combined constraint is:

$$x_1 + x_2 \geq 1$$

Example 14.2. This cp_addition example has negated literals that cancel each other:

```
(assume c1 (>= (+ (* 2 x1) (* 3 x2) 0) 2))
(assume c2 (>= (+ (* 1 (- 1 x1)) (* 3 (- 1 x2)) 0) 4))
(step t1 (cl (>= (+ (* 1 x1) 0) 2))
:rule cp_addition :premises (c1 c2))
```

In this example, we are adding two constraints.

$$2x_1 + 3x_2 \geq 2 \quad \text{and} \quad \overline{x_1} + 3\overline{x_2} \geq 4.$$

After applying the cp_addition rule, the combined constraint is:

$$x_1 \geq 2$$

After simplification, this results in a contradiction.

Rule 15: cp_multiplication
A constraint can be multiplied by any $c \in \mathbb{N}^+$:

$$\begin{array}{llr} i.\ \triangleright & \sum_i a_i l_i \geq A & \\ j.\ \triangleright & \sum_i c a_i l_i \geq cA & (\text{cp\_multiplication}\ \ i)\ [c] \end{array}$$

Rule 16: cp_divison
A constraint can be divided by any $c \in \mathbb{N}^+$, and the the ceiling of this division in applied:

$$\begin{array}{llr} i.\ \triangleright & \sum_i a_i l_i \geq A & \\ j.\ \triangleright & \sum_i \lceil \frac{a_i}{c} \rceil l_i \geq \lceil \frac{A}{c} \rceil & (\text{cp\_divison}\ \ i)\ [c] \end{array}$$

Remark. This rule needs constraints in **normalized form** i.e. no negative coefficients, no negative constant.

Rule 17: cp_saturation
A constraint can replace its coefficients by the minimum between them and the constant:

$$\begin{array}{llr} i.\ \triangleright & \sum_i a_i l_i \geq A & \\ j.\ \triangleright & \sum_i \min(a_i, A) \cdot l_i \geq A & (\text{cp\_saturation}\ \ i) \end{array}$$

Remark. This rule needs constraints in **normalized form** i.e. no negative coefficients, no negative constant.

Rule 18: assume

$i.\ \triangleright$ φ assume

where φ corresponds up to the orientation of equalities to a formula asserted in the input problem, or φ is a local assumption in a subproof.

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

Rule 19: hole

$i.\ \triangleright$ φ (hole $p_2, \dots, p_n$) $[a_1, \dots, a_m]$

where φ is any well-formed formula.

This rule can be used to express holes in the proof. It can be used by solvers as a placeholder for proof steps that are not yet expressed by the proof rules in this document. A proof checker *must not* accept a proof as valid that contains this rule even if the checker can somehow check this rule. However, it is possible for checkers to have a dedicated status for proofs that contain this rule and are otherwise valid. Any other tool can accept or reject proofs that contain this rule.

The premises and arguments are arbitrary, but must follow the syntax for premises and arguments.

Rule 20: true

$i.\ \triangleright$ $\top$ true

Rule 21: false

$i.\ \triangleright$ $\neg\bot$ false

Rule 22: not_not

$i.\ \triangleright$ $\neg(\neg\neg\varphi),\ \varphi$ 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 23: th_resolution

This rule is the resolution of two or more clauses.

$i_1.\ \triangleright$ $l^1_1,\ \dots,\ l^1_{k^1}$ (…)

$\vdots$

$i_n.\ \triangleright$ $l^n_1,\ \dots,\ l^n_{k^n}$ (…)

$j.\ \triangleright$ $l^{r_1}_{s_1},\ \dots,\ l^{r_m}_{s_m}$ (th_resolution $i_1, \dots, i_n$)

where $l^{r_1}_{s_1}, \dots, l^{r_m}_{s_m}$ are from l^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. When performing resolution steps, the rule implicitly merges repeated negations at the start of the formulas l^i_j. For example, the formulas $\neg\neg\neg P$ and $\neg\neg P$ can serve as pivots during resolution. The first formula is interpreted as $\neg P$ and the second as just P for the purpose of performing resolution steps.

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. The definition given here is very general. The motivation for this is to ensure the definition covers all possible resolution steps generated by the existing proof generation code in veriT. It will be restricted after a full review of the code. As a consequence of this checking this rule is theoretically NP-complete. In practice, however, 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 24: resolution

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

Rule 25: tautology

$$\begin{array}{llr} i. \rhd & l_1, \ldots, l_k, \ldots, l_m, \ldots, l_n & (\ldots) \\ j. \rhd & \top & (\textsf{tautology}\ i) \end{array}$$

and l_k, l_m are such that

$$l_k = \underbrace{\neg \ldots \neg}_{o} \varphi$$

$$l_m = \underbrace{\neg \ldots \neg}_{p} \varphi$$

and one of o, p is odd and the other even. Either can be 0.

Rule 26: contraction

$$\begin{array}{llr} i. \rhd & l_1, \ldots, l_n & (\ldots) \\ j. \rhd & l_{k_1}, \ldots, l_{k_m} & (\textsf{contraction}\ i) \end{array}$$

where $m \leq n$ and $k_1 \ldots k_m$ is a monotonic map to $1 \ldots n$ such that $l_{k_1} \ldots l_{k_m}$ are pairwise distinct and $\{l_1, \ldots, l_n\} = \{l_{k_1}, \ldots, l_{k_m}\}$. Hence, this rule removes duplicated literals.

Rule 27: subproof

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

$$\begin{array}{l|lcr} i_1. & \rhd & \varphi_1 & \textsf{assume} \\ & & \vdots & \\ i_n. & \rhd & \varphi_n & \textsf{assume} \\ & & \vdots & \\ j. & \rhd & \psi & (\ldots) \\ \hline \mathrm{k.} & \rhd & \neg\varphi_1, \ldots, \neg\varphi_n, \psi & \textsf{subproof} \end{array}$$

Rule 28: 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 $\bowtie \in \{\approx, <, >, \leq, \geq\}$, 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

$i.$ ▷ φ_1, ..., φ_o la_generic $[a_1, ..., a_o]$

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

The constants a_i are of sort `Real` and must be printed using one of the productions ⟨`rational`⟩ ⟨`decimal`⟩, ⟨`nonpositive_decimal`⟩.

To check the unsatisfiability of the negation of $\varphi_1 \vee \dots \vee \varphi_o$ 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 the table below.
5. If $\bowtie$ is $\approx$ replace l by $\sum_{i=0}^{m} a \times c_i \times t_i \approx a \times d$, otherwise replace it by $\sum_{i=0}^{m} |a| \times c_i \times t_i \approx |a| \times d$.

The replacements that can be performed by step 4 above are

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

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 $\approx$ if all operators are $\approx$, $>$ if all are either $\approx$ 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 28.1. 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 $(fa) > (fb) \wedge (fa) \approx (fb)$ (step 2). After step 3 we get $(fa) - (fb) > 0 \wedge (fa) - (fb) \approx 0$. Since we don't have an integer sort in this logic step 4 does not apply. Finally, after step 5 the conjunction is $(fa) - (fb) > 0 \wedge -(fa) + (fb) \approx 0$. This sums to $0 > 0$, which is a contradiction.

Example 28.2. 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 (1.0 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 29: lia_generic

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

$i.\ \triangleright$ $\quad \varphi_1,\, \ldots,\, \varphi_o \quad$ (lia_generic)

with φ_i being linear inequalities. The disjunction $\varphi_1 \vee \ldots \vee \varphi_n$ 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 30: la_disequality

$i.\ \triangleright$ $\quad t_1 \approx t_2 \vee \neg(t_1 \leq t_2) \vee \neg(t_2 \leq t_1) \quad$ (la_disequality)

Rule 31: la_totality

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

Rule 32: la_tautology

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

$i.\ \triangleright$ $\quad \varphi \quad$ (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:

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

It can be checked by using the procedure for la_generic 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 the result of applying normalization as for the rule la_generic.

Rule 33: la_mult_pos
Either of the form:

$$i.\ \rhd \qquad (t_1 > 0 \land t_2 \bowtie t_3) \rightarrow t_1 * t_2 \bowtie t_1 * t_3 \qquad (\mathsf{la\_mult\_pos})$$

with $\bowtie \in \{<, >, \leq, \geq, \approx\}$.

Or of the form:

$$i.\ \rhd \qquad (t_1 > 0 \land \neg(t_2 \approx t_3)) \rightarrow \neg(t_1 * t_2 \approx t_1 * t_3) \qquad (\mathsf{la\_mult\_pos})$$

Rule 34: la_mult_neg
Either of the form:

$$i.\ \rhd \qquad (t_1 < 0 \land t_2 \bowtie t_3) \rightarrow t_1 * t_2 \bowtie_{inv} t_1 * t_3 \qquad (\mathsf{la\_mult\_neg})$$

with $\bowtie \in \{<, >, \leq, \geq, \approx\}$ and $\bowtie_{inv}$ being defined according to the following table.

$\bowtie$	$\bowtie_{inv}$ is defined as:
$<$	$>$
$\leq$	$\geq$
$\approx$	$\approx$
$>$	$<$
$\geq$	$\leq$

Or of the form:

$$i.\ \rhd \qquad (t_1 < 0 \land \neg(t_2 \approx t_3)) \rightarrow \neg(t_1 * t_2 \approx t_1 * t_3) \qquad (\mathsf{la\_mult\_neg})$$

Rule 35: bind
The bind rule is used to rename bound variables.

$$\begin{array}{l|lcr}
 & & \vdots & \\
j. & \Gamma, y_1, \dots, y_n, x_1 \mapsto y_1, \dots, x_n \mapsto y_n \ \rhd & \varphi \approx \varphi' & (\dots) \\
\hline
k. & \Gamma \qquad\qquad\qquad\qquad\qquad \rhd & Qx_1, \dots, x_n.\varphi \approx Qy_1, \dots, y_n.\varphi' & \mathsf{bind}
\end{array}$$

where $Q \in \{\forall, \exists\}$, and the variables $y_1, \dots, y_n$ are neither free in $Qx_1, \dots, x_n.\varphi$ nor occur in Γ.

Rule 36: sko_ex
The sko_ex rule skolemizes existential quantifiers.

$$\begin{array}{l|lcr}
 & & \vdots & \\
j. & \Gamma, x_1 \mapsto \varepsilon_1, \dots, x_n \mapsto \varepsilon_n \ \rhd & \varphi \approx \psi & (\dots) \\
\hline
k. & \Gamma \qquad\qquad\qquad\quad \rhd & \exists x_1, \dots, x_n.\varphi \approx \psi & \mathsf{sko\_ex}
\end{array}$$

where ε_i stands for $\varepsilon x_i.(\exists x_{i+1}, \dots, x_n.\varphi)$.

Rule 37: sko_forall
The sko_forall rule skolemizes universal quantifiers.

$$\begin{array}{l|lcr}
 & & \vdots & \\
j. & \Gamma, x_1 \mapsto (\varepsilon x_1.\neg\varphi), \dots, x_n \mapsto (\varepsilon x_n.\neg\varphi) \ \rhd & \varphi \approx \psi & (\dots) \\
\hline
k. & \Gamma \qquad\qquad\qquad\qquad\qquad\qquad \rhd & \forall x_1, \dots, x_n.\varphi \approx \psi & \mathsf{sko\_forall}
\end{array}$$

Rule 38: forall_inst

$i.\ \triangleright$	$\neg(\forall x_1, \dots, x_n . P) \lor P[x_1 \mapsto t_1] \dots [x_n \mapsto t_n]$	forall_inst $[t_1, \dots, t_n]$

where x_i and t_i have the same sort.

Example 38.1. An application of the forall_inst rule.

```
(step t16 (cl (or (not (forall ((x S) (y T)) (P y x    )))
(P b (f a))
:rule forall_inst :args ((f a) b)
```

Rule 39: refl

$j.\ \triangleright\ \Gamma$	$t_1 \approx t_2$	refl

where, if $\sigma = \text{subst}(\Gamma)$, the terms $t_1\sigma$ and t_2 are syntactically equal up to renaming of bound variables and the orientation of equalities.

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

Rule 40: trans

$i_1.\ \triangleright\ \Gamma$	$t_1 \approx t_2$	$(\dots)$
$i_2.\ \triangleright\ \Gamma$	$t_2 \approx t_3$	$(\dots)$
	$\vdots$	
$i_n.\ \triangleright\ \Gamma$	$t_n \approx t_{n+1}$	$(\dots)$
$j.\ \ \triangleright\ \Gamma$	$t_1 \approx t_{n+1}$	(trans $i_1, \dots, i_n$)

Rule 41: cong

$i_1.\ \triangleright\ \Gamma$	$t_1 \approx u_1$	$(\dots)$
$i_2.\ \triangleright\ \Gamma$	$t_2 \approx u_2$	$(\dots)$
	$\vdots$	
$i_n.\ \triangleright\ \Gamma$	$t_n \approx u_n$	$(\dots)$
$j.\ \ \triangleright\ \Gamma$	$(f t_1 \cdots t_n) \approx (f u_1 \cdots u_n)$	(cong $i_1, \dots, i_n$)

where f is any function symbol of appropriate sort.

Rule 42: eq_reflexive

$i.\ \triangleright$	$t \approx t$	eq_reflexive

Rule 43: eq_transitive

$i.\ \triangleright$	$\neg(t_1 \approx t_2), \dots\ , \neg(t_{n-1} \approx t_n), t_1 \approx t_n$	eq_transitive

Rule 44: eq_congruent

$i.\ \triangleright$	$\neg(t_1 \approx u_1), \dots\ , \neg(t_n \approx u_n), (f t_1 \cdots t_n) \approx (f u_1 \cdots u_n)$	eq_congruent

Rule 45: eq_congruent_pred

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

where P is a function symbol with co-domain sort **Bool**.

Rule 46: qnt_cnf

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

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

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

Rule 47: and

$i.\ \triangleright$	$\varphi_1 \land \cdots \land \varphi_n$	$(\dots)$
$j.\ \triangleright$	φ_k	(and i) k

and $1 \leq k \leq n$.

Rule 48: not_or

$i.\ \triangleright$	$\neg(\varphi_1 \lor \cdots \lor \varphi_n)$	$(\dots)$
$j.\ \triangleright$	$\neg\varphi_k$	(not_or i) k

and $1 \leq k \leq n$.

Rule 49: or

$i.\ \triangleright$	$\varphi_1 \lor \cdots \lor \varphi_n$	$(\dots)$
$j.\ \triangleright$	$\varphi_1, \cdots, \varphi_n$	(or i)

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

Example 49.1. 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 50: weakening

$i. \triangleright$	$\varphi_1, \cdots, \varphi_n$	$(\ldots)$
$j. \triangleright$	$\varphi_1, \cdots, \varphi_n, \psi_1, \ldots, \psi_m$	(weakening i)

where $m \geq 1$.

Rule 51: reordering

$i. \triangleright$	$\varphi_1, \cdots, \varphi_n$	$(\ldots)$
$j. \triangleright$	$\psi_1, \cdots, \psi_n$	(reordering i)

where the multisets $\{\varphi_1, \cdots, \varphi_n\}$ and $\{\psi_1, \cdots, \psi_n\}$ are the same. That is, the conclusion of the rule is a reordering of the literals in the premise.

Rule 52: not_and

$i. \triangleright$	$\neg(\varphi_1 \land \ldots \land \varphi_n)$	$(\ldots)$
$j. \triangleright$	$\neg\varphi_1, \ldots, \neg\varphi_n$	(not_and i)

Rule 53: xor1

$i. \triangleright$	$(\mathbf{xor}\; \varphi_1\; \varphi_2)$	$(\ldots)$
$j. \triangleright$	φ_1, φ_2	(xor1 i)

Rule 54: xor2

$i. \triangleright$	$(\mathbf{xor}\; \varphi_1\; \varphi_2)$	$(\ldots)$
$j. \triangleright$	$\neg\varphi_1, \neg\varphi_2$	(xor2 i)

Rule 55: not_xor1

$i. \triangleright$	$\neg(\mathbf{xor}\; \varphi_1\; \varphi_2)$	$(\ldots)$
$j. \triangleright$	$\varphi_1, \neg\varphi_2$	(not_xor1 i)

Rule 56: not_xor2

$i. \triangleright$	$\neg(\mathbf{xor}\; \varphi_1\; \varphi_2)$	$(\ldots)$
$j. \triangleright$	$\neg\varphi_1, \varphi_2$	(not_xor2 i)

Rule 57: implies

$i. \triangleright$	$\varphi_1 \rightarrow \varphi_2$	$(\ldots)$
$j. \triangleright$	$\neg\varphi_1, \varphi_2$	(implies i)

Rule 58: not_implies1

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

Rule 59: not_implies2

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

Rule 60: equiv1

$i. \triangleright$	$\varphi_1 \approx \varphi_2$	$(\ldots)$
$j. \triangleright$	$\neg\varphi_1, \varphi_2$	(equiv1 i)

Rule 61: equiv2

$i.\ \rhd$ $\varphi_1 \approx \varphi_2$ (...)
$j.\ \rhd$ $\varphi_1, \neg\varphi_2$ (equiv2 i)

Rule 62: not_equiv1

$i.\ \rhd$ $\neg(\varphi_1 \approx \varphi_2)$ (...)
$j.\ \rhd$ φ_1, φ_2 (not_equiv1 i)

Rule 63: not_equiv2

$i.\ \rhd$ $\neg(\varphi_1 \approx \varphi_2)$ (...)
$j.\ \rhd$ $\neg\varphi_1, \neg\varphi_2$ (not_equiv2 i)

Rule 64: and_pos

$i.\ \rhd$ $\neg(\varphi_1 \land \dots \land \varphi_n), \varphi_k$ and_pos k
with $1 \leq k \leq n$.

Rule 65: and_neg

$i.\ \rhd$ $(\varphi_1 \land \dots \land \varphi_n), \neg\varphi_1, \dots, \neg\varphi_n$ and_neg

Rule 66: or_pos

$i.\ \rhd$ $\neg(\varphi_1 \lor \dots \lor \varphi_n), \varphi_1, \dots, \varphi_n$ or_pos

Rule 67: or_neg

$i.\ \rhd$ $(\varphi_1 \lor \dots \lor \varphi_n), \neg\varphi_k$ or_neg k
with $1 \leq k \leq n$.

Rule 68: xor_pos1

$i.\ \rhd$ $\neg(\mathbf{xor}\, \varphi_1\, \varphi_2), \varphi_1, \varphi_2$ xor_pos1

Rule 69: xor_pos2

$i.\ \rhd$ $\neg(\mathbf{xor}\, \varphi_1\, \varphi_2), \neg\varphi_1, \neg\varphi_2$ xor_pos2

Rule 70: xor_neg1

$i.\ \rhd$ $(\mathbf{xor}\, \varphi_1\, \varphi_2), \varphi_1, \neg\varphi_2$ xor_neg1

Rule 71: xor_neg2

$i.\ \rhd$ $(\mathbf{xor}\, \varphi_1\, \varphi_2), \neg\varphi_1, \varphi_2$ xor_neg2

Rule 72: implies_pos

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

Rule 73: implies_neg1

$i.\ \rhd$ $\varphi_1 \rightarrow \varphi_2, \varphi_1$ implies_neg1

Rule 74: implies_neg2

$i.\ \rhd$ $\varphi_1 \rightarrow \varphi_2, \neg\varphi_2$ implies_neg2

Rule 75: equiv_pos1

$i.\ \rhd$ $\neg(\varphi_1 \approx \varphi_2), \varphi_1, \neg\varphi_2$ equiv_pos1

Rule 76: equiv_pos2

$i.\ \rhd$ $\neg(\varphi_1 \approx \varphi_2), \neg\varphi_1, \varphi_2$ equiv_pos2

Rule 77: equiv_neg1

$i.\ \triangleright$	$\varphi_1 \approx \varphi_2, \neg\varphi_1, \neg\varphi_2$	equiv_neg1

Rule 78: equiv_neg2

$i.\ \triangleright$	$\varphi_1 \approx \varphi_2, \varphi_1, \varphi_2$	equiv_neg2

Rule 79: ite1

$i.\ \triangleright$	$(\mathbf{ite}\,\varphi_1\,\varphi_2\,\varphi_3)$	(...)
$j.\ \triangleright$	φ_1, φ_3	(ite1 i)

Rule 80: ite2

$i.\ \triangleright$	$(\mathbf{ite}\,\varphi_1\,\varphi_2\,\varphi_3)$	(...)
$j.\ \triangleright$	$\neg\varphi_1, \varphi_2$	(ite2 i)

Rule 81: ite_pos1

$i.\ \triangleright$	$\neg(\mathbf{ite}\,\varphi_1\,\varphi_2\,\varphi_3), \varphi_1, \varphi_3$	(ite_pos1)

Rule 82: ite_pos2

$i.\ \triangleright$	$\neg(\mathbf{ite}\,\varphi_1\,\varphi_2\,\varphi_3), \neg\varphi_1, \varphi_2$	(ite_pos2)

Rule 83: ite_neg1

$i.\ \triangleright$	$(\mathbf{ite}\,\varphi_1\,\varphi_2\,\varphi_3, \varphi_1, \neg\varphi_3)$	(ite_neg1)

Rule 84: ite_neg2

$i.\ \triangleright$	$(\mathbf{ite}\,\varphi_1\,\varphi_2\,\varphi_3, \neg\varphi_1, \neg\varphi_2)$	(ite_neg2)

Rule 85: not_ite1

$i.\ \triangleright$	$\neg(\mathbf{ite}\,\varphi_1\,\varphi_2\,\varphi_3)$	(...)
$j.\ \triangleright$	$\varphi_1, \neg\varphi_3$	(not_ite1 i)

Rule 86: not_ite2

$i.\ \triangleright$	$\neg(\mathbf{ite}\,\varphi_1\,\varphi_2\,\varphi_3)$	(...)
$j.\ \triangleright$	$\neg\varphi_1, \neg\varphi_2$	(not_ite2 i)

Rule 87: connective_def

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

$i.\ \triangleright\ \Gamma$	$(\mathbf{xor}\,\varphi_1\,\varphi_2) \approx ((\neg\varphi_1 \land \varphi_2) \lor (\varphi_1 \land \neg\varphi_2))$	connective_def
$i.\ \triangleright\ \Gamma$	$(\varphi_1 \approx \varphi_2) \approx ((\varphi_1 \rightarrow \varphi_2) \land (\varphi_2 \rightarrow \varphi_1))$	connective_def
$i.\ \triangleright\ \Gamma$	$(\mathbf{ite}\,\varphi_1\,\varphi_2\,\varphi_3) \approx ((\varphi_1 \rightarrow \varphi_2) \land (\neg\varphi_1 \rightarrow \varphi_3))$	connective_def
$i.\ \triangleright\ \Gamma$	$(\forall x_1, \dots, x_n.\, \varphi) \approx \neg(\exists x_1, \dots, x_n.\, \neg\varphi)$	connective_def

Rule 88: and_simplify

This rule simplifies an $\land$ term by applying equivalence-preserving transformations until a fixed point is reached. Hence, the general form is

$i.\ \triangleright\ \Gamma$	$\varphi_1 \land \dots \land \varphi_n \approx \psi$	and_simplify

where ψ is the transformed term.

The possible transformations are:

- $\top \land \cdots \land \top \Rightarrow \top$
- $\varphi_1 \land \cdots \land \varphi_n \Rightarrow \varphi_1 \land \cdots \land \varphi_{n'}$ where the right-hand side has all $\top$ literals removed.
- $\varphi_1 \land \cdots \land \varphi_n \Rightarrow \varphi_1 \land \cdots \land \varphi_{n'}$ where the right-hand side has all repeated literals removed.
- $\varphi_1 \land \cdots \land \bot \land \cdots \land \varphi_n \Rightarrow \bot$
- $\varphi_1 \land \cdots \land \varphi_i \land \cdots \land \varphi_j \land \cdots \land \varphi_n \Rightarrow \bot$ and φ_i, φ_j are such that

  $$\varphi_i = \underbrace{\neg \ldots \neg}_{n} \psi$$
  $$\varphi_j = \underbrace{\neg \ldots \neg}_{m} \psi$$

  and one of n, m is odd and the other even. Either can be 0.

Rule 89: or_simplify

This rule simplifies an $\lor$ term by applying equivalence-preserving transformations until a fixed point is reached. Hence, the general form is

$i.\ \rhd \Gamma \qquad (\varphi_1 \lor \cdots \lor \varphi_n) \approx \psi \qquad$ or_simplify

where ψ is the transformed term.

The possible transformations are:

- $\bot \lor \cdots \lor \bot \Rightarrow \bot$
- $\varphi_1 \lor \cdots \lor \varphi_n \Rightarrow \varphi_1 \lor \cdots \lor \varphi_{n'}$ where the right-hand side has all $\bot$ literals removed.
- $\varphi_1 \lor \cdots \lor \varphi_n \Rightarrow \varphi_1 \lor \cdots \lor \varphi_{n'}$ where the right-hand side has all repeated literals removed.
- $\varphi_1 \lor \cdots \lor \top \lor \cdots \lor \varphi_n \Rightarrow \top$
- $\varphi_1 \lor \cdots \lor \varphi_i \lor \cdots \lor \varphi_j \lor \cdots \lor \varphi_n \Rightarrow \top$ and φ_i, φ_j are such that

  $$\varphi_i = \underbrace{\neg \ldots \neg}_{n} \psi$$
  $$\varphi_j = \underbrace{\neg \ldots \neg}_{m} \psi$$

  and one of n, m is odd and the other even. Either can be 0.

Rule 90: not_simplify

This rule simplifies an $\neg$ term by applying equivalence-preserving transformations until a fixed point is reached. Hence, the general form is

$i.\ \rhd \Gamma \qquad \neg\varphi \approx \psi \qquad$ not_simplify

where ψ is the transformed term.

The possible transformations are:

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

Rule 91: implies_simplify
This rule simplifies an → term by applying equivalence-preserving transformations until a fixed point is reached. Hence, the general form is
$i.\ \rhd\ \Gamma \qquad \varphi_1 \rightarrow \varphi_2 \approx \psi \qquad$ implies_simplify
where ψ is the transformed term.

The possible transformations are:

- $\neg\varphi_1 \rightarrow \neg\varphi_2 \Rightarrow \varphi_2 \rightarrow \varphi_1$
- $\bot \rightarrow \varphi \Rightarrow \top$
- $\varphi \rightarrow \top \Rightarrow \top$
- $\top \rightarrow \varphi \Rightarrow \varphi$
- $\varphi \rightarrow \bot \Rightarrow \neg\varphi$
- $\varphi \rightarrow \varphi \Rightarrow \top$
- $\neg\varphi \rightarrow \varphi \Rightarrow \varphi$
- $\varphi \rightarrow \neg\varphi \Rightarrow \neg\varphi$

Rule 92: equiv_simplify
This rule simplifies a formula with the head symbol $\approx: \mathbf{Bool}\,\mathbf{Bool}\,\mathbf{Bool}$ by applying equivalence-preserving transformations until a fixed point is reached. Hence, the general form is
$i.\ \rhd\ \Gamma \qquad (\varphi_1 \approx \varphi_2) \approx \psi \qquad$ equiv_simplify
where ψ is the transformed term.

The possible transformations are:

- $(\neg\varphi_1 \approx \neg\varphi_2) \Rightarrow (\varphi_1 \approx \varphi_2)$
- $(\varphi \approx \varphi) \Rightarrow \top$
- $(\varphi \approx \neg\varphi) \Rightarrow \bot$
- $(\neg\varphi \approx \varphi) \Rightarrow \bot$
- $(\top \approx \varphi) \Rightarrow \varphi$
- $(\varphi \approx \top) \Rightarrow \varphi$
- $(\bot \approx \varphi) \Rightarrow \neg\varphi$

- $(\varphi \approx \bot) \Rightarrow \neg\varphi$

Rule 93: bool_simplify
This rule simplifies a boolean term by applying equivalence-preserving transformations until a fixed point is reached. Hence, the general form is
$i. \rhd \Gamma$ $\varphi \approx \psi$ bool_simplify
where ψ is the transformed term.

The possible transformations are:

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

Rule 94: ac_simp
This rule simplifies nested occurrences of $\vee$ or $\wedge$:
$i. \rhd \Gamma$ $\psi \approx \varphi_1 \circ \cdots \circ \varphi_n$ 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.

Rule 95: ite_simplify
This rule simplifies an if-then-else term by applying equivalence-preserving transformations until fixed point[12] It has the form
$i. \rhd \Gamma$ $(\mathbf{ite}\, \varphi\, t_1\, t_2) \approx u$ ite_simplify
where u is the transformed term.

The possible transformations are:

- $(\mathbf{ite}\, \top\, t_1\, t_2) \Rightarrow t_1$
- $(\mathbf{ite}\, \bot\, t_1\, t_2) \Rightarrow t_2$
- $(\mathbf{ite}\, \psi\, t\, t) \Rightarrow t$
- $(\mathbf{ite}\, \neg\varphi\, t_1\, t_2) \Rightarrow (\mathbf{ite}\, \varphi\, t_2\, t_1)$
- $(\mathbf{ite}\, \psi\, (\mathbf{ite}\, \psi\, t_1\, t_2)\, t_3) \Rightarrow (\mathbf{ite}\, \psi\, t_1\, t_3)$

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

- $(\textbf{ite}\,\psi\,t_1\,(\textbf{ite}\,\psi\,t_2\,t_3) \Rightarrow (\textbf{ite}\,\psi\,t_1\,t_3)$

- $(\textbf{ite}\,\psi\,\top\,\bot) \Rightarrow \psi$

- $(\textbf{ite}\,\psi\,\bot\,\top) \Rightarrow \neg\psi$

- $(\textbf{ite}\,\psi\,\top\,\varphi) \Rightarrow \psi \vee \varphi$

- $(\textbf{ite}\,\psi\,\varphi\,\bot) \Rightarrow \psi \wedge \varphi$

- $(\textbf{ite}\,\psi\,\bot\,\varphi) \Rightarrow \neg\psi \wedge \varphi$

- $(\textbf{ite}\,\psi\,\varphi\,\top) \Rightarrow \neg\psi \vee \varphi$

Rule 96: qnt_simplify

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

$$i.\ \triangleright\ \Gamma \qquad (\forall x_1, \dots, x_n . \varphi) \approx \varphi \qquad \textsf{qnt\_simplify}$$

where φ is either $\top$ or $\bot$.

Rule 97: onepoint

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

$$\begin{array}{lllr} & & \vdots & \\ j. & \Gamma, x_{k_1}, \dots, x_{k_m}, x_{j_1} \mapsto t_{j_1}, \dots, x_{j_o} \mapsto t_{j_o}\ \triangleright & \varphi \approx \varphi' & (\dots) \\ \hline k. & \Gamma & \triangleright\ Qx_1, \dots, x_n . \varphi \approx Qx_{k_1}, \dots, x_{k_m} . \varphi' & \textsf{onepoint} \end{array}$$

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 $x_i \approx t_i$ 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 97.1. An application of the onepoint rule on the term $(\forall x, y.\, x \approx y \rightarrow (f\,x) \wedge (f\,y))$ look like this:

```
(anchor :step t3 :args ((x S) (:= (y S) 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 onepoint)
```

Rule 98: qnt_join

$i.\ \rhd\ \Gamma \qquad Qx_1, \dots, x_n.\, (Qx_{n+1}, \dots, x_m.\, \varphi) \approx Qx_{k_1}, \dots, x_{k_o}.\, \varphi$ 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 99: qnt_rm_unused

$i.\ \rhd\ \Gamma \qquad Qx_1, \dots, x_n.\, \varphi \approx Qx_{k_1}, \dots, x_{k_m}.\, \varphi$ 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 $x \in \{x_j \mid j \in \{1, \dots, n\} \land j \in \not\{k_1, \dots, k_m\}\}$ then x is not free in P.

Rule 100: eq_simplify

This rule simplifies an $\approx$ term by applying equivalence-preserving transformations until a fixed point is reached. Hence, the general form is

$i.\ \rhd\ \Gamma \qquad (t_1 \approx t_2) \approx \varphi$ eq_simplify

where ψ is the transformed term.

The possible transformations are:

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

Rule 101: div_simplify

This rule simplifies a division by applying equivalence-preserving transformations. The general form is

$i.\ \rhd\ \Gamma \qquad (t_1 \,/\, t_2) \Rightarrow t_3$ div_simplify

The possible transformations are:

- $t \,/\, t \Rightarrow 1$
- $t \,/\, 1 \Rightarrow t$
- $t_1 \,/\, t_2 \Rightarrow t_3$ if t_1 and t_2 are constants and t_3 is t_1 divided by t_2 according to the semantics of the current theory.

Rule 102: prod_simplify

This rule simplifies a product by applying equivalence-preserving transformations until a fixed point is reached. The general form is

$i.\ \rhd\ \Gamma \qquad t_1 \times \cdots \times t_n \approx u$ prod_simplify

where u is either a constant or a product.

The possible transformations are:

- $t_1 \times \cdots \times t_n \Rightarrow u$ where all t_i are constants and u is their product.
- $t_1 \times \cdots \times t_n \Rightarrow 0$ if any t_i is 0.
- $t_1 \times \cdots \times t_n \Rightarrow c \times t_{k_1} \times \cdots \times t_{k_n}$ where c is 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.
- $t_1 \times \cdots \times t_n \Rightarrow t_{k_1} \times \cdots \times t_{k_n}$: same as above if c is 1.

Rule 103: unary_minus_simplify
This rule is either
$i. \rhd \Gamma \qquad -(-t) \approx t \qquad$ unary_minus_simplify
or
$i. \rhd \Gamma \qquad -t \approx u \qquad$ unary_minus_simplify
where u is the negated numerical constant t.

Rule 104: minus_simplify
This rule simplifies a subtraction by applying equivalence-preserving transformations. The general form is
$i. \rhd \Gamma \qquad t_1 - t_2 \approx u \qquad$ minus_simplify
The possible transformations are:

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

Rule 105: sum_simplify
This rule simplifies a sum by applying equivalence-preserving transformations until a fixed point is reached. The general form is
$i. \rhd \Gamma \qquad t_1 + \cdots + t_n \approx u \qquad$ sum_simplify
where u is either a constant or a product.

The possible transformations are:

- $t_1 + \cdots + t_n \Rightarrow c$ where all t_i are constants and c is their sum.
- $t_1 + \cdots + t_n \Rightarrow c + t_{k_1} + \cdots + t_{k_n}$ where c is 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.
- $t_1 + \cdots + t_n \Rightarrow t_{k_1} + \cdots + t_{k_n}$: same as above if c is 0.

Rule 106: comp_simplify
This rule simplifies a comparison by applying equivalence-preserving transformations until a fixed point is reached. The general form is
$i. \rhd \Gamma \qquad t_1 \bowtie t_2 \approx \psi \qquad$ comp_simplify
where $\bowtie \in \{<, >, \leq, \geq\}$.

The following transformation rules are used to simplify $t_1 \bowtie t_2$:

- $s_1 < s_2 \Rightarrow \top$ if s_1, s_2 are numerical constants and s_1 is strictly less than s_2
- $s_1 < s_2 \Rightarrow \bot$ if s_1, s_2 are numerical constants and s_1 is greater or equal than s_2
- $s_1 \leq s_2 \Rightarrow \top$ if s_1, s_2 are numerical constants and s_1 is less or equal than s_2
- $s_1 \leq s_2 \Rightarrow \bot$ if s_1, s_2 are numerical constants and s_1 is strictly greater than s_2
- $s < s \Rightarrow \bot$
- $s \leq s \Rightarrow \top$
- $s_1 \geq s_2 \Rightarrow s_2 \leq s_1$
- $s_1 < s_2 \Rightarrow \neg(s_2 \leq s_1)$
- $s_1 > s_2 \Rightarrow \neg(s_1 \leq s_2)$

Rule 107: let

This rule eliminates **let**. It has the form

$$\begin{array}{llccr}
i_1. & \Gamma & \rhd & t_1 \approx s_1 & (\dots) \\
 & & & \vdots & \\
i_n. & \Gamma & \rhd & t_n \approx s_n & (\dots) \\
 & & & \vdots & \\
j. & \vert\, \Gamma, x_1 \mapsto s_1, \dots, x_n \mapsto s_n & \rhd & u \approx u' & (\dots) \\
\hline
k. & \Gamma & \rhd & (\mathbf{let}\, x_1 = t_1,\ \dots,\ x_n = t_n\, \mathbf{in}\, u) \approx u' & (\mathsf{let}\ i_1,\ \dots,\ i_n)
\end{array}$$

The premise $i_1, \dots, i_n$ must be in the same subproof as the let step. If for $t_i \approx s_i$ the t_i and s_i are syntactically equal, the premise is omitted.

Rule 108: bind_let

This rule corresponds to the bind rule for `let`. It allows the renaming of the variables bound by the `let` step, the rewriting of the substituted terms, and the rewriting of the body of the `let`, resulting in a new `let` term. It has the form

$$\begin{array}{llccr}
i_1. & \Gamma & \rhd & t_1 \approx s_1 & (\dots) \\
 & & & \vdots & \\
i_n. & \Gamma & \rhd & t_n \approx s_n & (\dots) \\
 & & & \vdots & \\
j. & \vert\, \Gamma, y_1, \dots, y_n, x_1 \mapsto y_1, \dots, x_n \mapsto y_n & \rhd & u \approx u' & (\dots) \\
\hline
k. & \Gamma & \rhd & (\mathbf{let}\, x_1 = t_1,\ \dots,\ x_n = t_n\, \mathbf{in}\, u) \approx & (\mathsf{bind\_let}\ i_1,\ \dots,\ i_n) \\
 & & & (\mathbf{let}\, y_1 = s_1,\ \dots,\ y_n = s_n\, \mathbf{in}\, u') &
\end{array}$$

The variables $y_1, \dots, y_n$ are neither free in $(\mathbf{let}\, x_1 = t_1,\ \dots,\ x_n = t_n\ \mathbf{in}\, u)$ nor, for each y_i different from x_i, occur in Γ.

The premise $i_1, \dots, i_n$ must be in the same subproof as the bind_let step. If for $t_i \approx s_i$ the t_i and s_i are syntactically equal, the premise is omitted.

Example 108.1. The following example shows how this rule is used in a proof generated by Carcara's elaborator. It elaborates an implicit application of symmetry of equality.

```
(step t1 (cl (= (= 0 1) (= 1 0))) :rule eq_symmetric)
(anchor :step t2 :args ((p Bool) (:= (p Bool) p)))
(step t2.t1 (cl (= (= p false) (= false p))) :rule eq_symmetric)
(step t2 (cl (= (let ((p (= 0 1))) (= p false))
(let ((p (= 1 0))) (= false p))))
:rule bind_let :premises (t1))
```

Rule 109: distinct_elim

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

$i.\ \triangleright\ \Gamma \qquad (\textbf{distinct}\, t) \approx \top \qquad$ distinct_elim

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

$i.\ \triangleright\ \Gamma \qquad (\textbf{distinct}\, \varphi\, \psi) \approx \neg(\varphi \approx \psi) \qquad$ distinct_elim

and

$i.\ \triangleright\ \Gamma \qquad (\textbf{distinct}\, \varphi_1\, \varphi_2\, \varphi_3\, \dots) \approx \bot \qquad$ distinct_elim

The general case is

$i.\ \triangleright\ \Gamma \qquad (\textbf{distinct}\, t_1\, \dots\, t_n) \approx \bigwedge_{i=1}^{n} \bigwedge_{j=i+1}^{n} t_i \not\approx t_j \qquad$ distinct_elim

Rule 110: la_rw_eq

$i.\ \triangleright \qquad (t \approx u) \approx (t \leq u \land u \leq t) \qquad$ la_rw_eq

Rule 111: nary_elim

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

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

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

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

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

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

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

Rule 112: bfun_elim

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

$j.\ \triangleright \qquad \varphi \qquad$ (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 $t[x \mapsto \bot] \lor t[x \mapsto \top]$ and $(\forall x.\, t)$ becomes $t[x \mapsto \bot] \land t[x \mapsto \top]$. 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 $(\mathbf{ite}\,P(f\,x\top\ y)\,(f\,x\bot\,y))$. If the argument is already the constant $\top$ or $\bot$, it is ignored.

Rule 113: ite_intro

$i.\ \rhd \qquad t \approx (t' \wedge u_1 \wedge \ldots \wedge u_n) \qquad$ (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. $s_i := \mathbf{ite}\,\psi_i\,r_i\,r_i'$, then u_i has the form

$$\mathbf{ite}\,\psi_i\,(s_i \approx r_i)\,(s_i \approx r_i')$$

The term t' is equal to the term t 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.

Rule 114: bitblast_extract

$i.\ \rhd \qquad ((\mathbf{extract}\ j\ i)\ x) \approx (\mathbf{bbT}\ \varphi_i\ \ldots\ \varphi_j) \qquad$ (bitblast_extract)

where the formulas φ_k are $(\mathbf{bitOf}_k\ x)$ for $i \leq k \leq j$.

Alternatively, the rule may also be phrased as a “short-circuiting” of the above when x is a **bbT** application:

$i.\ \rhd\ ((\mathbf{extract}\ j\ i)\ (\mathbf{bbT}\ x_0\ \ldots\ x_i\ \ldots\ x_j\ \ldots\ x_n)) \approx (\mathbf{bbT}\ x_i\ \ldots\ x_j)$ (bitblast_extract)

This alternative is based on the validity of the equality

$$\mathbf{bitOf}_k\ (\mathbf{bbT}\ x_0\ \ldots\ x_i\ \ldots\ x_j\ \ldots\ x_n) \approx x_k$$

for any bit-vector x of size $n+1$, where $0 \leq k \leq n$.

Rule 115: bitblast_ult

$i.\ \rhd \qquad (\mathbf{bvult}\ x\ y) \approx \mathrm{res}_{n-1} \qquad$ (bitblast_ult)

in which both x and y must have the same type $(\mathbf{BitVec}\ n)$ and, for $i \geq 0$

$$\begin{array}{lcl}\mathrm{res}_0 & = & \neg(\mathbf{bitOf}_0\ x) \wedge (\mathbf{bitOf}_0\ y) \\ \mathrm{res}_{i+1} & = & (((\mathbf{bitOf}_{i+1}\ x) \approx (\mathbf{bitOf}_{i+1}\ y)) \wedge \mathrm{res}_i) \vee (\neg(\mathbf{bitOf}_{i+1}\ x) \wedge (\mathbf{bitOf}_{i+1}\ y))\end{array}$$

Alternatively, the rule may also be phrased as a “short-circuiting” of the above when x and y are “**bbT**” applications. So given that

$$\begin{array}{lcl}x & = & (\mathbf{bbT}\ x_0\ \ldots\ x_i\ \ldots\ x_j\ \ldots\ x_n) \\ y & = & (\mathbf{bbT}\ y_0\ \ldots\ y_i\ \ldots\ y_j\ \ldots\ y_n)\end{array}$$

then “res” can be defined, for $i \geq 0$, as

$$\begin{aligned} \mathrm{res}_0 &= \neg x_0 \wedge y_0 \\ \mathrm{res}_{i+1} &= ((x_{i+1} \approx y_{i+1}) \wedge \mathrm{res}_i) \vee (\neg x_{i+1} \wedge y_{i+1}) \end{aligned}$$

Rule 116: bitblast_add

$i.$ ▷ $(\mathbf{bvadd}\ x\ y) \approx A_1$ (bitblast_add)

in which both x and y must have the same type $(\mathbf{BitVec}\ n)$. The term “A_1” is

$$\begin{aligned} (\mathbf{bbT}\ & (((\mathbf{bitOf}_0\ x)\,\mathbf{xor}\,(\mathbf{bitOf}_0\ y))\,\mathbf{xor}\,\mathrm{carry}_0) \\ & (((\mathbf{bitOf}_1\ x)\,\mathbf{xor}\,(\mathbf{bitOf}_1\ y))\,\mathbf{xor}\,\mathrm{carry}_1) \\ & \ldots \\ & (((\mathbf{bitOf}_{n-1}\ x)\,\mathbf{xor}\,(\mathbf{bitOf}_{n-1}\ y))\,\mathbf{xor}\,\mathrm{carry}_{n-1})) \end{aligned}$$

and for $i \geq 0$

$$\begin{aligned} \mathrm{carry}_0 &= \bot \\ \mathrm{carry}_{i+1} &= ((\mathbf{bitOf}_i\ x) \wedge (\mathbf{bitOf}_i\ y)) \vee (((\mathbf{bitOf}_i\ x)\,\mathbf{xor}\,(\mathbf{bitOf}_i\ y)) \wedge \mathrm{carry}_i) \end{aligned}$$

Alternatively, the rule may also be phrased as a “short-circuiting” of the above when x and y are “**bbT**” applications. So given that

$$\begin{aligned} x &= (\mathbf{bbT}\ x_0\ \ldots\ x_i\ \ldots\ x_j\ \ldots\ x_n) \\ y &= (\mathbf{bbT}\ y_0\ \ldots\ y_i\ \ldots\ y_j\ \ldots\ y_n) \end{aligned}$$

then the rule can be alternatively phrased as

$i.$ ▷ $(\mathrm{bvadd}\ x\ y) \approx A_2$ (bitblast_add)

with $A_2 := (\mathbf{bbT}\ (x_0\,\mathbf{xor}\,y_0)\,\mathbf{xor}\,\mathrm{carry}_0\ \ldots\ (x_{n-1}\,\mathbf{xor}\,y_{n-1})\,\mathbf{xor}\,\mathrm{carry}_{n-1})$ and “carry” being defined, for $i \geq 0$, as

$$\begin{aligned} \mathrm{carry}_0 &= \bot \\ \mathrm{carry}_{i+1} &= (x_i \wedge y_i) \vee ((x_i\,\mathbf{xor}\,y_i) \wedge \mathrm{carry}_i) \end{aligned}$$

Rule 117: symm

$i.$ ▷ $\varphi \approx \psi$ (...)

$j.$ ▷ $\psi \approx \varphi$ (symm i)

If $\varphi \neq \psi$ then the conclusion *must not* be $\varphi \approx \psi$.

Note that since Alethe allows the implicit reordering of equalities, this rule is technically superfluous. However, the rule is useful to indicate an explicit usage of the symmetry of equality to aid proof reconstruction.

Example 117.1:. The side condition ensures that the following example is not a valid application of the rule. Without this condition, this derivation could be obtained by applying symmetry of equality implicitly to the conclusion.

10. ▷ $P(a) \approx Q(b)$ (...)

11. ▷ $P(a) \approx Q(b)$ (symm 10)

Rule 118: not_symm

$i.$ ▷	$\neg(\varphi \approx \psi)$	(...)
$j.$ ▷	$\neg(\psi \approx \varphi)$	(not_symm i)

If $\varphi \neq \psi$ then the conclusion *must not* be $\neg(\varphi \approx \psi)$.

See symm for an explanation of this rule.

Rule 119: eq_symmetric

$i.$ ▷	$(t_1 \approx t_2) \approx (t_2 \approx t_1)$	(eq_symmetric)

Note that since Alethe allows the implicit reordering of equalities, this rule is technically superfluous. However, the rule is useful to state an explicit usage of symmetry of equality reasoning to aid proof reconstruction.

5.3 Index of Rules

Changelog

Unreleased

Proof rules:

- Addition of a section describing bitvector proofs.
- Bitblasting rules: bitblast_extract, bitblast_add, bitblast_ult.
- Addition of rules la_mult_pos and la_mult_neg to describe multiplication with a positive or negative factor.
- Addition of rules symm and not_symm to express symmetry of equality explicitly.
- Addition of the rule eq_symmetric to express symmetry of equality explicitly but as an equivalence. Note that in principle this could be done with the rule symm above, but would require a long and tedious use of subproof for each direction of the equivalence.
- Addition of the rule weakening to express weakening of a clause.
- Addition of the reordering rule to represent reordering of the literals in a clause.
- Addition of the bind_let rule. This rule can be used to preprocess `let` expressions similar to the bind rule used with ordinary quantifiers.

Breaking changes:

- Allow arbitrary extra annotations in `assume` commands.
- Add the sort to all variables in contexts. Before, the context of a bind could be `(x S) (:= y x)`. Now it must be `(x S) (:= (y S) x)`.
- The arguments for forall_inst have been changed to no longer take the shape of bindings using `(:= x c)`. Instead, the list of instantiation terms must follow the variable order and cover all the respective bound variables.
- The rules and_pos, or_neg, and, not_or now have one argument that indicates which subterm they use.
- Add new syntax for decimal and negative numbers and use it for la_generic.
- Restrict sorting of numbers such that the sort of a constant is only determined by its syntactic category.
- Add new syntax for decimal and negative numbers and clarify that the set of allowed symbols is restricted to not conflict with this new syntax.
- Always use decimals for the constants in la_generic.

- Restrict sorting of numbers such that the sort of a constant is only determined by its syntactic category.

Clarifications and corrected errors:

- Clarify that the `:args` annotation in `anchor` can be omitted if the list is empty.
- Clarify that bind can work on existential and universal quantifiers.
- Fix mistake in proof grammar. It now uses the `context_annotation` non-terminal in the rule for the `anchor` command.
- Fix the example of onepoint.
- Add the missing context to the conclusion of bind, sko_ex, sko_forall, onepoint.

0.3 — 2023-02-10

This release overhauls the entire document, but introduces only few changes to the proof format itself.

The standard now specifies that `assume` commands can only be issued at the start of the proof or right after an `anchor` command.

Beyond many smaller clarifications and typographic improvements, the following changes are implemented in this release.

- Add an abstract proof checking procedure to clarify the semantics of the proof format.
- Add a description of the semantics of contexts based on λ-terms.
- List all transformations that are implicit in Alethe proofs.
- Change the notation used for terms from first-order style (e.g., $f(x, g(y))$) to higher-order style (e.g., $(f\ x\ (g\ y))$). This is only a change in notation – the used logic remains many-sorted first-order logic.
- Eliminate the distinction between if-and-only-if and equality. Instead, use equality (the symbol $\approx$) with appropriate sorts.
- Add an index that lists all proof rules.

Proof rules:

- The rule implies_simplify is no longer allowed to perform the simplification $(\varphi_1 \rightarrow \varphi_2) \rightarrow \varphi_2 \Rightarrow \varphi_1 \vee \varphi_2$. This is now covered by bool_simplify.

0.2 — 2022-12-19

This is an intermediate release. It collects all changes to the original specification document before the major changes that were implemented as part of Hans-Jörg Schurr's PhD thesis. These changes will be reflected in release 0.3.

This release implements major changes to the structure of the document to clarify the difference between the *language* and the *rules*. The language has a formal definition and a proof of soundness. The syntax describes how proofs are encoded in the text file.

The syntax was extended to allow extra annotations. Tools consuming Alethe proofs must be able to ignore such extra annotations.

List of rules:

- Improve description of sko_ex.
- Add hole rule to allow holes. A proof that contains steps that use this rule is not valid.

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.
- Change syntax of abstract proof steps to be clearer.

0.1 – 2021-07-10

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

Index

References

[1] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *TACAS 28*, pages 415–442, Cham, 2022. Springer International Publishing.

[2] Haniel Barbosa, Jasmin C. Blanchette, Mathias Fleury, and Pascal Fontaine. Scalable fine-grained proofs for formula processing. *Journal of Automated Reasoning*, 64(3):485–510, January 2019.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[4] Frédéric Besson, Pascal Fontaine, and Laurent Théry. A flexible proof format for SMT: A proposal. In Pascal Fontaine and Aaron Stump, editors, *PxTP 1*, pages 15–26, August 2011.

[5] David Déharbe, Pascal Fontaine, and Bruno Woltzenlogel Paleo. Quantifier inference rules for SMT proofs. In Pascal Fontaine and Aaron Stump, editors, *PxTP 1*, pages 33–39, August 2011.

[6] 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 29*, volume 10426 of *Lecture Notes in Computer Science*, pages 126–133. Springer, 2017.

[7] Mathias Fleury and Hans-Jörg Schurr. Reconstructing veriT proofs in Isabelle/HOL. In Giselle Reis and Haniel Barbosa, editors, *PxTP 6*, volume 301 of *EPTCS*, pages 36–50, 2019.

[8] 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*, Lecture Notes in Computer Science, pages 450–467, Cham, 2021. Springer International Publishing.

[9] Wikipedia contributors. Alethe (bird) – Wikipedia, the free encyclopedia, 2022. Online; accessed 2022-09-02.