Contribution Title

First Author[1][0000−1111−2222−3333], Second Author[2,3][1111−2222−3333−4444], and Third Author[3][2222−−3333−4444−5555]

[1] Princeton University, Princeton NJ 08544, USA
[2] Springer Heidelberg, Tiergartenstr. 17, 69121 Heidelberg, Germany
lncs@springer.com
http://www.springer.com/gp/computer-science/lncs
[3] ABC Institute, Rupert-Karls-University Heidelberg, Heidelberg, Germany
{abc,lncs}@uni-heidelberg.de

Abstract. The abstract should briefly summarize the contents of the paper in 150–250 words.

Keywords: Linear arithmetic · SMT · normal form · Lambdapi · reflection

1 Alethe proof

The Alethe proof trace format [1] for SMT solvers comprises two parts: the trace language based on SMT-LIB and a collection of proof rules. Traces witness proofs of unsatisfiability of a set of constraints. They are sequences $a_1 \dots a_m\ t_1 \dots t_n$ where the a_i corresponds to the constraints of the original SMT problem being refuted, each t_i is a clause inferred from previous elements of the sequence, and t_n is $\bot$ (the empty clause). In the following, we designate the SMT-LIB problem as the *input problem*.

```
(set-logic QF_LIA)
(declare-const x Int)
(declare-const y Int)
(assert (= 0 y))
(assert (= x 2))
(assert (or (< (+ x y) 1) (< 3 x)))
(get-proof)
```

↯

```
(assume a0 (or (< (+ x y) 1) (< 3 x)))
(assume a1 (= x 2))
(assume a2 (= 0 y))
(step t1 (cl (< (+ x y) 1) (< 3 x)) :rule or :premises (a0))
(step t2 (cl (not (< 3 x)) (not (= x 2))) :rule la_generic :args (1/1 1/1))
(step t3 (cl (not (< 3 x))) :rule resolution :premises (a1 t2))
(step t4 (cl (< (+ x y) 1)) :rule resolution :premises (t1 t3))
(step t5 (cl (not (< (+ x y) 1)) (not (= x 2)) (not (= 0 y))) :rule
la_generic :args (1/1 -1/1 1/1))
(step t6 (cl) :rule resolution :premises (t5 t4 a1 a2))
```

Listing 1.1. The following example is the proof for the unsatisfiability of $(x + y < 1) \lor (3 < x), x = 2$ and $0 = y$.

We will use the input problem shown in the top part of example 1 with its Alethe proof (found by cvc5) in the bottom part as a running example to provide an overview of Alethe concepts and to illustrate our reconstruction of linear arithmetic step in Lambdapi.

1.1 The Alethe Language

An Alethe proof trace inherits the declarations of its input problem. All symbols (sorts, functions, assertions, etc.) declared or defined in the input problem remain declared or defined, respectively. Furthermore, the syntax for terms, sorts, and annotations uses the syntactic rules defined in SMT-LIB [2, §3] and the SMT signature context defined in [2, §5.1 and §5.2]. In the following we will represent an Alethe step as

$$\underbrace{i}_{\text{index}}.\quad \underbrace{\Gamma}_{\text{context}} \triangleright \overbrace{l_1 \dots l_n}^{\text{clause}} \quad (\underbrace{\mathcal{R}}_{\text{rule}}\ \underbrace{p_1 \dots p_m}_{\text{premises}})\ \underbrace{[a_1 \dots a_r]}_{\text{arguments}} \tag{1}$$

A step consists of an index $i \in \mathbb{I}$ where $\mathbb{I}$ is a countable infinite set of indices (e.g. `a0`, `t1`), and a clause of formulae $l_1, \dots, l_n$ representing an n-ary disjunction. Steps that are not assumptions are justified by a proof rule $\mathcal{R}$ that depends on a possibly empty set of premises $\{p_1 \dots p_m\} \subseteq \mathbb{I}$ that only references earlier steps such that the proof forms a directed acyclic graph. A rule might also depend on a list of arguments $[a_1 \dots a_r]$ where each argument a_i is either a term or a pair (x_i, t_i) where x_i is a variable and t_i is a term. The interpretation of the arguments is rule specific. The context Γ of a step is a list $c_1 \dots c_l$ where each element c_j is either a variable or a variable-term tuple denoted $x_j \mapsto t_j$. Therefore, steps with a non-empty context contain variables x_j that appear in l_i and will be substituted by t_j. Proof rules $\mathcal{R}$ include theory lemmas and `resolution`, which corresponds to hyper-resolution on ground first-order clauses.

We now have the key components to explain the guiding proof in the bottom part of listing 1.1. The proofs starts with `assume` steps `a0`, `a1`, `a2` that restate the assertions from the *input problem* (listing 1.1). Step `t1` transforms disjunction into clause by using the Alethe rule `or`. Steps `t2` and `t5` are tautologies introduced by the main rule `la_generic` in Linear Real Arithmetic (LRA) logic and used also in LIA logic, where $\neg l_1, \neg l_2, \dots, \neg l_n$ represent linear inequalities. These logics operate over closed linear over arbitrary expansions of the `Real` signature (resp. `Int`). The `Real` terms in `LRA` logic are Reals signature with free constant symbols, but containing only linear atoms, that is, atoms with no occurrences of the function symbols `*` and `/` except in coefficient multiplications—specifically, terms of the form c, `(* c x)`, or `(* x c)` where `x` is a free constant and `c` is an integer or rational coefficient. Similarly, the `Int` terms in `LIA` logic are closed formulas built over an arbitrary expansion of the Ints signature with free constant symbols, but whose terms of sort Int are all linear, that is, have no occurrences of

Rule	Description
la_generic	Tautologous disjunction of linear inequalities.
lia_generic	Tautologous disjunction of linear integer inequalities.
la_disequality	$t_1 \approx t_2 \vee \neg(t_1 \geq t_2) \vee \neg(t_2 \geq t_1)$
la_totality	$t_1 \geq t_2 \vee t_2 \geq t_1$
la_tautology	A trivial linear tautology
la_mult_pos	$t_1 > 0 \wedge (t_2 \bowtie t_3) \rightarrow t_1 * t_2 \bowtie t_1 * t_3$ and $\bowtie \in \{<, >, \geq, \leq, =\}$
la_mult_neg	$t_1 < 0 \wedge (t_2 \bowtie t_3) \rightarrow t_1 * t_2 \bowtie_{inv} t_1 * t_3$
la_rw_eq	$(t \approx u) \approx (t \geq u \wedge u \geq t)$
div_simplify	Simplification of division.
prod_simplify	Simplification of products.
unary_minus_simplify	Simplification of the unuary minus.
minus_simplify	Simplification of the substractions.
sum_simplify	Simplification of sums.
comp_simplify	Simplification of arithmetic comparisons.

Table 1. Linear arithmetic rules in Alethe.

the function symbols`*`, `/`, `div`, `mod`, and `abs` , except terms with coefficients are also allowed, that is, terms of the form `c`, `(* c x)`, or `(* x c)` where `x` is a free constant and `c` is a term of the form `n` or `(- n)` for some numeral `n`.

Checking the validity of this clause amounts to checking the unsatisfiability of the system of linear equations e.g. $x < 3$ and $x = 2$ in `t2`. A coefficient for each inequality are pass as arguments e.g. $(\frac{1}{1}, \frac{1}{1})$ in `t2`. Steps `t3` (and `t4`) applies the `resolution` rule to the premises `a1`, `t2` (respectively `t1 t3`). Finally, the step `t6` concludes the proof by generating the empty clause $\bot$, concretely denoted as `(cl)` in listing 1.1. Notice that the contexts Γ of each step are all empty in this proof.

1.2 Checking Alethe arithmetic step

Proofs for linear arithmetic steps use a number of straightforward rules listed in table 1, such as `la_totality` $(t_1 \leq t_2 \vee t_2 \geq t_1)$. Simplification rules such as `sum_simplify` transform arithmetic formulas by applying equivalence-preserving operations repeatedly until a fixed point is reached; these operations are no more complex than constant folding. The primary rules are `la_generic` and `lia_generic`, both introduce a tautologous disjunction of linear inequalities. The `lia_generic` is structurally similar to `la_generic`, but omits the coefficients. Since this rule can introduce a disjunction of arbitrary linear integer inequalities without any additional hints, proof checking is *NP-complete* [11]. While `la_generic rule` is primarily intended for LRA logic, it is also applied in LIA proofs when all variables in the (in)equalities are of integer sort. Although the rule may introduce rational coefficients, they often reduce to integers—as shown in listing 1.1, where the coefficients are $(\frac{1}{1}, \frac{1}{1})$. Cases where coefficients cannot be reduced to integers are rare in practice.

Although the sort `Real` is not currently defined in our Lambdapi environment, it is entirely feasible to define them in Lambdapi. However, in this work we focus exclusively on reconstructing proofs within the LIA (Linear Integer Arithmetic) logic. In LIA, rational numbers only appear solely as coefficients. Any finite set of such rational coefficients can be transformed into integers by multiplying each by the least common multiple (LCM) of their denominators. Thus, defining `Int` suffices to reconstruct proofs from LIA input problem.

2 Lia elaboration

Carcara considers `lia_generic` steps as holes in the proof, as verifying them is as difficult. Currently, Carcara relies on an external tool that generates Alethe proofs to formulate the steps by solving corresponding problems in a proof-producing manner. The proof is then imported, verified, and validated before replacing the original step. However, at present, Carcara only use cvc5 for performing this task. It is important to note that cvc5 has a limitation: its Alethe proofs may contain rewrite steps that are not yet modeled in the Alethe simplification rules, and as such, these steps are not supported by Carcara. While these steps are considered holes, they typically involve simple simplification rules and, therefore, have much less impact than the more complex `lia_generic` ones.

In detail, the elaboration method, when encountering a `lia_generic` step S concluding the negated inequalities $\neg l_1 \vee \dots \neg l_n$, generates an SMT-LIB problem asserting $l_1 \wedge \cdots \wedge l_n$ and invokes *cvc5* on it, expecting an Alethe proof π : $(l_1 \wedge \cdots \wedge l_n) \rightarrow \bot$. Carcara will check each step in π and, if they are not invalid, will replace step S in the original proof by a proof of the form:

```
(anchor :step S.t_m+1)
(assume S.h_1 l1)
...
(assume S.h_n ln)
...
(step t.t_m (cl false) :rule ...)
(step t.t_m+1 (cl (not l1) ... (not ln) false) :rule subproof)
(step t.t_m+2 (cl (not false)) :rule false)
(step S (cl (not l1) ... (not ln)) :rule resolution :premises (S.t_m+1 S.t_m
+2))
```

Listing 1.2. Elaboration of `lia_generic`

We decided to leverage the elaboration process of `lia_generic` performed by Carcara, as doing otherwise would require implementing Fourier-Motzkin elimination for integers, as demonstrated in [10, 3] - and therefore reimplementing the work done by the solver.

3 Linear Arithmetic in Alethe

All linear arithmetic tautology rules, such as `la_disequalities`, `la_totality`, and simplification rules like `comp_simplify`, are encoded as lemmas in our embedding of Alethe in Lambdapi, as presented in section 4. The `la_generic` rule, however, must be reconstructed using a different approach, as it involves following the algorithm described below.

A step of the rule `la_generic` represents a tautological clause of linear disequalities. It can be checked by showing that the conjunction of the negated disequalities is unsatisfiable. After the application of some strengthening rules, the resulting conjunction is unsatisfiable, even if integer variables are assumed to be real variables.

A linear inequality is of term of the form

$$\sum_{i=0}^{n} c_i \times t_i + d_1 \bowtie \sum_{i=n+1}^{m} c_i \times t_i + d_2$$

where $\bowtie \in \{=, <, >, \leq, \geq\}$, where $m \geq n$, c_i, d_1, d_2 are either integer or real constants, and for each i c_i and t_i have the same sort. We will write $s_1 \bowtie s_2$.

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

$$i. \triangleright \varphi_1, \ldots, \varphi_n \; \texttt{la\_generic} \; [a_1, \ldots, a_n]$$

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 $\varphi_1, \ldots, \varphi_n$ 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. If $\varphi = s_1 < s_2$, then let $\varphi := -s_1 > -s_2$. If $\varphi = s_1 \leq s_2$, then let $\varphi := s_1 \geq -s_2$.
4. 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 $d := d_2 - d_1$.
5. 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.
6. If $\bowtie$ is $=$ replace l by $\sum_{i=0}^{m} a \times c_i \times t_i = a \times d$, otherwise replace it by $\sum_{i=0}^{m} |a| \times c_i \times t_i = |a| \times d$. Coefficients are put on the same denominator to keep whole numbers as coefficients.

The replacements that can be performed by step 5 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 $=$ if all operators are $=$, $>$ if all are either $=$

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 $>$).

The step 3 has been added by our own since the following steps in the original algorithm work with $>$ and $\geq$ and does not mention clearly what to do with $<$ and $\leq$.

Example 1. Consider the `la_generic` step in the logic `LIA`:

```
(step t11 (cl (not (<= f 0)) (<= (+ 1 (* 4 f)) 1))
  :rule la_generic :args (1.0 1/4))
```

After step 4, we get $-f > 0 \wedge 4 \times f > 0$. The step 5 applied and we can strengthen this to $-f \geq 0 \wedge 4 \times f \geq 1$ and after multiplication of the normalized coefficients we get $4 \times (-f) \geq 0 \ \wedge \ 4 \times f \geq 1$. Which sums to the contradiction $0 \geq 1$.

In the next section, we first present an overview of our embedding of Alethe in Lambdapi, and then our automatic procedure to reconstruct `la_generic` step that appear in LIA problem.

4 Reconstruction of `la_generic` step

4.1 Lambdapi

Lambdapi is an implementation of $\lambda\Pi$ modulo theory ($\lambda\Pi/\equiv$) [6], an extension of the Edinburgh Logical Framework $\lambda\Pi$ [8], a simply typed λ-calculus with dependent types. $\lambda\Pi/\equiv$ adds user-defined higher-order rewrite rules. Its syntax is given by

Universes	$u ::= \mathtt{TYPE} \mid \mathtt{KIND}$
Terms	$t, v, A, B, C ::= c \mid x \mid u \mid \Pi\, x : A,\, B \mid \lambda\, x : A,\, t \mid t\ v$
Contexts	$\Gamma ::= \langle\rangle \mid \Gamma, x : A$
Signatures	$\Sigma ::= \langle\rangle \mid \Sigma, c : C \mid \Sigma, c := t : C \mid \Sigma, t \hookrightarrow v$

where c is a constant and x is a variable (ranging over disjoint sets), C is a closed term. *Universes* are constants used to verify if a type is well-formed – more details can be found in [8, §2.1]. $\Pi\, x : A.\, B$ is the dependent product, and we write $A \to B$ when x does not appear free in B, $\lambda\, x : A.\, t$ is an abstraction, and $t\ v$ is an application. A *(local) context* Γ is a finite sequence of variable declarations $x : A$ introducing variables and their types. A *signature* Σ representing the global context is a finite sequence of *assumptions* $c : C$, indicating that constant c is of type C, *definitions* $c := t : C$, indicating that c has the value t and type C, and *rewrite rules* $t \hookrightarrow v$ such that $t = c\ v_1 \dots v_n$ where c is a constant.

The relation $\hookrightarrow_{\beta\Sigma}$ is generated by β-reduction and by the rewrite rules of Σ. The relation $\hookrightarrow^*_{\beta\Sigma}$ denotes the reflexive and transitive closure of $\hookrightarrow_{\beta\Sigma}$, and the relation $\equiv_{\beta\Sigma}$ (called *conversion*) the reflexive, symmetric, and transitive closure of $\hookrightarrow_{\beta\Sigma}$. The relation $\hookrightarrow_{\beta\Sigma}$ must be confluent, i.e., whenever $t \hookrightarrow^*_{\beta\Sigma} v_1$ and $t \hookrightarrow^*_{\beta\Sigma} v_2$, there exists a term w such that $v_1 \hookrightarrow^*_{\beta\Sigma} w$ and $v_2 \hookrightarrow^*_{\beta\Sigma} w$, and it must preserve typing, i.e., whenever $\Gamma \vdash_\Sigma t : A$ and $t \hookrightarrow_{\beta\Sigma} v$ then $\Gamma \vdash_\Sigma v : A$ [4].

A Lambdapi typing judgment $\Gamma \vdash_\Sigma t : A$ asserts that term t has type A in the context Γ and the signature Σ. The typing rules of $\lambda\Pi/\equiv$ are the one of $\lambda\Pi$ [8, §2], except for the rule (Conv) where it use the version of fig. 1 that identifies types modulo $\equiv_{\beta\Sigma}$ instead of just modulo β-reduction.

$$\frac{\Gamma, \vdash_\Sigma B : u \qquad \Gamma \vdash_\Sigma t : A \qquad A \equiv_{\beta\Sigma} B}{\Gamma \vdash_\Sigma t : B} \text{ (Conv)}$$

Fig. 1. (Conv) rule in $\lambda\Pi/\equiv$

We now provide an overview of our encoding of Alethe in Lambdapi. A more comprehensive version of the encoding is available in [5].

4.2 A Prelude Encoding for Alethe

Definition 1 (Prelude Encoding). *The signature Σ of our encoding contains the following definitions and rewrite rules provided by the standard library of Lambdapi that we use to encode Alethe proofs:*

$\texttt{Set} : \texttt{TYPE}$	$\texttt{Prop} : \texttt{TYPE}$
$\texttt{El} : \texttt{Set} \to \texttt{TYPE}$	$\texttt{Prf} : \texttt{Prop} \to \texttt{TYPE}$
$\rightsquigarrow : \texttt{Set} \to \texttt{Set} \to \texttt{Set}$	$o : \texttt{Set}$
$\texttt{El}\,(x \rightsquigarrow y) \hookrightarrow \texttt{El}\,x \to \texttt{El}\,y$	$\texttt{El}\,o \hookrightarrow \texttt{Prop}$

The constants `Set` and `Prop` (lines 1 and 6) are type universes "à la Tarski" [9, §Universes] in $\lambda\Pi/\equiv$. The type `Set` represents the universe of *small types*, i.e. a subclass of types for which we can define equality. SMT sorts are represented in $\lambda\Pi/\equiv$ as elements of type `Set`. Since elements of type `Set` are not types themselves, we also introduce a decoding function $\texttt{El} : \texttt{Set} \to \texttt{TYPE}$ that interprets SMT sorts as $\lambda\Pi/\equiv$ types. Thus, we represent the terms of sort `Bool` of SMT by elements of type $\texttt{El}\,o$. The constructor $\rightsquigarrow$ is used to encode SMT functions and predicates. The type `Prop` represents the universe of propositions in $\lambda\Pi/\equiv$. Like `Set`, elements of type `Prop` are not types themselves but are mapped to types by the decoding function $\texttt{Prf} : \texttt{Prop} \to \texttt{TYPE}$. By analogy with the Curry-de-Brujin-Howard isomorphism, it embeds propositions into types, mapping each proposition A to the type $\texttt{Prf}\ A$ of its proofs.

Alethe distinguishes between clauses that appear in steps, $(\mathrm{cl}\, l_1 \ldots l_n)$ in eq. (1), and ordinary disjunction [1, §4]. The syntax for clauses uses the `cl` operator, while disjunction is represented as the standard SMT-LIB `or`. We define the type `Clause` that encodes an Alethe clause as a list of propositions. The

$$
\begin{array}{ll}
\texttt{Clause} : \texttt{TYPE} & \mathcal{F} : \texttt{Clause} \to \texttt{Prop} \\
\blacksquare : \texttt{Clause} & \mathcal{F}\ \blacksquare \hookrightarrow \bot \\
\forall : \texttt{Prop} \to \texttt{Clause} \to \texttt{Clause} & \mathcal{F}\ x \forall y \hookrightarrow x \vee^c (\mathcal{F}\ y) \\
\texttt{Prf}^{\bullet}(c : \texttt{Clause}) \coloneqq \texttt{Prf}^{c}(\mathcal{F}\ c) &
\end{array}
$$

Fig. 2. The `Clause` type and operations on clauses.

constructor $\forall$ prepends an element to a list, and $\blacksquare$ is the empty list. Logically, clauses are interpreted as disjunctions via the function $\mathcal{F}$ defined by rewriting rules. For convenience, we also introduce the predicate $\texttt{Prf}^{\bullet}$ asserting that a clause is provable.

4.3 Classical connectives, quantifiers and facts

Since SMT solvers are based on classical logic, we use the constructive connectives and quantifiers from the Lambdapi standard library and define the classical ones from them using the double-negation translation [7] as a definition.

$$
\begin{array}{l}
\texttt{Prf}^{c}\, p \coloneqq \texttt{Prf}(\neg\neg p) \\
= : \Pi[a : \texttt{Set}], \texttt{El}\, a \to \texttt{El}\, a \to \texttt{Prop} \\
p \vee^c q \coloneqq \neg\neg p \vee \neg\neg q \\
\forall^c \coloneqq \Pi[a : \texttt{Set}], \Pi p : (\texttt{El}\, a \to \texttt{Prop}), \forall x. \neg\neg p\, x \\
\texttt{classic} :\ \Pi[p : \texttt{Prop}], \texttt{Prf}^{c}(p \vee^c \neg p) \\
\texttt{prop\_ext} :\ \Pi[p\, q : \texttt{Prop}], \texttt{Prf}^{c}(p \Leftrightarrow^c q) \to \texttt{Prf}^{c}(p = q)
\end{array}
$$

Therefore, a step in an Alethe proof trace is represented as a proposition $\texttt{Prf}^{c}\ p$, defined as the intuitionistic proof `Prf` of the doubly negated predicate. Equality over small types is parameterized over types $\texttt{El}\, a$ for the type parameter $[a : \texttt{Set}]$ (the square brackets indicate that this parameter need not be given explicitly). We also define classical connectives, quantifiers, and the choice operator ϵ ([1, §2.1]) as illustrated above. We prove the law of excluded middle and add the proposition of Boolean extensionality stating that classical equivalence coincides with equality over Booleans. SMT logic enjoys the property of propositional completeness (also referred to as propositional degeneracy) asserting that $\forall^c A,\ (A = \top) \vee^c (A = \bot)$. Moreover, propositionally equivalent formulas are equal. We thus obtain the theorems `classic` and `prop_ext`.

4.4 Encoding of Integers in Lambdapi

The definition we use of integers in Lambdapi follows a common encoding found in many other theories, such as Rocq. First, the type $\mathbb{P}$ is an inductive type representing strictly positive integers in binary form. Starting from 1 (represented by constructor `H`), one can add a new least significant digit via the constructor `O` (digit 0) or constructor `I` (digit 1). The type $\mathbb{Z}$ is an inductive type representing integers in binary form. An integer is either zero (with constructor `Z0`) or a strictly positive number `Zpos` (coded as a $\mathbb{P}$) or a strictly negative number `Zneg` (whose opposite is stored as a $\mathbb{P}$ value).

$$
\begin{array}{ll}
\mathbb{Z} : \mathtt{TYPE} & \mathbb{P} : \mathtt{TYPE} \\
\mid \mathtt{Z0} : \mathbb{Z} & \mid \mathtt{H} : \mathbb{P} \\
\mid \mathtt{ZPos} : \mathbb{P} \to \mathbb{Z} & \mid \mathtt{O} : \mathbb{P} \to \mathbb{P} \\
\mid \mathtt{ZNeg} : \mathbb{P} \to \mathbb{Z} & \mid \mathtt{I} : \mathbb{P} \to \mathbb{P} \\
\mathtt{int} : \mathtt{Set} & \mathtt{pos} : \mathtt{Set} \\
\mathtt{El}\ \mathtt{int} \hookrightarrow \mathbb{Z} & \mathtt{El}\ \mathtt{pos} \hookrightarrow \mathbb{P}
\end{array}
$$

4.5 Functions used in the translation

We now provide an overview of how input problems expressed in a given SMT-LIB signature [2, §5.2.1] are encoded. In order to avoid a notational clash with the Lambdapi signature Σ, we denote the set of SMT-LIB sorts as $\Theta^{\mathcal{S}}$, the set of function symbols $\Theta^{\mathcal{F}}$, and the set of variables $\Theta^{\mathcal{X}}$. Alethe does not support the sorts `Array` and `String`. Moreover, we do not yet provide support for `Bitvector` and `Real`. Our translation is based on the following functions:

- $\mathcal{D}$ translates declarations of sorts and functions in $\Theta^{\mathcal{S}}$ and $\Theta^{\mathcal{F}}$ into constants,
- $\mathcal{S}$ maps sorts to Σ types,
- $\mathcal{E}$ translates SMT expression to $\lambda\Pi/\equiv$ terms,
- $\mathcal{C}$ translates a list of commands $c_1 \dots c_n$ of the form $i.\ \Gamma \rhd\ \varphi\ (\mathcal{R}\ P)[A]$ to typing judgments $\Gamma \vdash_{\Sigma} i := M : N$.

Definition 2 (Function $\mathcal{D}$ translating SMT sort and function symbol declarations). *For each sort symbol s with arity n in $\Theta^{\mathcal{S}}$ we create a constant $s : \mathtt{Set} \to \cdots \to \mathtt{Set}$. For each function symbol $f\ \sigma^{+}$ in $\Theta^{\mathcal{F}}$ we create a constant $f : \mathcal{S}(\sigma^{+})$.*

In other words, all SMT sorts used in the Alethe proof trace will be defined as constants that inhabit the type `Set` in the signature context Σ. For every function declared in the SMT prelude, we define a constant whose arity follows the sort declared in the SMT prelude. The translation of sorts is formally defined as follows.

Definition 3 (Function $\mathcal{S}$ translating sorts of expression). *The definition of $\mathcal{S}(s)$ is as follows.*

- *Case* $s = \textbf{\textit{Bool}}$, *then* $\mathcal{S}(s) = \texttt{El}\, o$,
- *Case* $s = \textbf{\textit{Int}}$, *then* $\mathcal{S}(s) = \texttt{El}\ \textbf{\textit{int}}$,
- *Case* $s = \sigma_1\, \sigma_2 \ldots \sigma_n$ *then* $\mathcal{S}(s) = \texttt{El}(\mathcal{S}(\sigma_1) \rightsquigarrow \cdots \rightsquigarrow \mathcal{S}(\sigma_n))$,
- *otherwise* $\mathcal{S}(s) = \texttt{El}\, \mathcal{D}(s)$.

Definition 4 (Function $\mathcal{E}$ translating SMT expressions). *The definition of $\mathcal{E}(e)$ is as follows.*

- *Case* $e = (p\ t_1\ t_2 \ldots\ t_n)$ *and* p *a logical operator, then* $\mathcal{E}(e) = \mathcal{E}(t_1)\ p^c\ \ldots\ p^c\ \mathcal{E}(t_n)$.
- *Case* $e = (g\ t_1 \ldots\ t_n)$ *with* $g \in \Theta^{\mathcal{F}}$, *then* $\mathcal{E}(e) = (\mathcal{D}(g)\ \mathcal{E}(t_1)\ \ldots\ \mathcal{E}(t_n))$.
- *Case* $e = (\approx\ t_1\ t_2)$ *then* $\mathcal{E}(e) = (\mathcal{E}(t_1) = \mathcal{E}(t_2))$.
- *Case* $e = (Q\ x_1 : \sigma_1 \ldots x_n : \sigma_n\ t)$ *where* $Q \in \{\texttt{forall}, \texttt{exists}\}$, *then* $\mathcal{E}(e) = Q^c x_1 : \mathcal{S}(\sigma_1), \ldots, Q^c x_n : \mathcal{S}(\sigma_n), \mathcal{E}(t)$.
- *Case* $e = (x : \sigma)$ *with* $x \in \Theta^{\mathcal{X}}$ *a sorted variable, then* $\mathcal{E}(e) = x : \mathcal{S}(\sigma)$.

5 Reconstruction of linear integer arithmetic

$$\begin{tikzcd} reify(t_1) =_{\mathcal{R}} reify(t_2) & \mathcal{R} \arrow[r, dotted, "\rightarrow_{AC}"] & \mathcal{R} \arrow[d, "denote"] & t_1 \downarrow_{AC} =_{\mathcal{R}} t_2 \downarrow_{AC} \\ t_1 =_{\mathbb{Z}} t_2 & \mathbb{Z} \arrow[u, "reify"] \arrow[r, phantom, "\Longleftrightarrow"] & \mathbb{Z} & denote(t_1 \downarrow_{AC}) =_{\mathbb{Z}} denote(t_2 \downarrow_{AC}) \end{tikzcd}$$

Definition 5 ($\mathcal{R}$).

$$\begin{aligned}
\texttt{add}\ (\texttt{var}\ x\ c_1)\ (\texttt{var}\ x\ c2) &\hookrightarrow \texttt{var}\ x\ (c_1 + c_2) \\
\texttt{add}\ (\texttt{var}\ x\ c_1)\ (\texttt{add}\ (\texttt{var}\ x\ c_2)\ y) &\hookrightarrow \texttt{add}\ (\texttt{var}\ x\ c_1 + c_2)\ y \\
\texttt{add}\ (\texttt{cst}\ c_1)\ (\texttt{cst}\ c_2) &\hookrightarrow (\texttt{cst}\ c_1 + c_2) \\
\texttt{add}\ (\texttt{cst}\ c_1)\ (\texttt{add}\ (\texttt{cst}\ c_2)\ y) &\hookrightarrow \texttt{add}\ (\texttt{cst}\ c_1 + c_2)\ y \\
\texttt{add}\ (\texttt{cst}\ 0)\ x &\hookrightarrow x \\
\texttt{add}\ x\ (\texttt{cst}\ 0) &\hookrightarrow x \\
s\texttt{opp}\ (\texttt{var}\ x\ c) &\hookrightarrow (\texttt{var}\ x\ (-c)) \\
\texttt{opp}\ (\texttt{cst}\ c) &\hookrightarrow (\texttt{cst}\ (-c)) \\
\texttt{opp}\ \texttt{opp}\ x &\hookrightarrow x \\
\texttt{opp}\ \texttt{add}\ x\ y &\hookrightarrow \texttt{add}\ (\texttt{opp}\ x)\ (\texttt{opp}\ y) \\
\texttt{mul}\ k\ (\texttt{var}\ x\ c) &\hookrightarrow (\texttt{var}\ x\ (k \times c)) \\
\texttt{mul}\ k\ \texttt{opp}\ x &\hookrightarrow \texttt{mul}\ (-k)\ x \\
\texttt{mul}\ k\ (\texttt{add}\ x\ y) &\hookrightarrow \texttt{add}\ (\texttt{mul}\ k\ x)\ (\texttt{mul}\ k\ y) \\
\texttt{mul}\ k\ (\texttt{cst}\ c) &\hookrightarrow (\texttt{cst}\ k \times c) \\
\texttt{mul}\ c_1\ (\texttt{mul}\ c_2\ x) &\hookrightarrow \texttt{mul}\ (c_1 \times c_2)\ x
\end{aligned}$$

Definition 6 (reify).

$$\begin{aligned}
\texttt{reify}\ 0 &\hookrightarrow (\texttt{cst}\ 0)\\
\texttt{reify}\ (-x) &\hookrightarrow \texttt{opp}\ \texttt{reify}\ x\\
\texttt{reify}\ (x+y) &\hookrightarrow \texttt{add}\ \texttt{reify}\ x\ \texttt{reify}\ y\\
\texttt{reify}\ x &\hookrightarrow (\texttt{var}\ x\ 1)
\end{aligned}$$

Definition 7 (denote).

$$\begin{aligned}
\texttt{den}\ (\texttt{var}\ c\ x) &\hookrightarrow c \times x\\
\texttt{den}\ (\texttt{cst}\ c) &\hookrightarrow c\\
\texttt{den}\ \texttt{opp}\ x &\hookrightarrow -(\texttt{den}\ x)\\
\texttt{den}\ \texttt{mul}\ c\ x &\hookrightarrow c \times \texttt{den}\ x\\
\texttt{den}\ \texttt{add}\ x\ y &\hookrightarrow \texttt{den}\ x + \texttt{den}\ y
\end{aligned}$$

Definition 8. *Let $aliens_\sqcup : \mathcal{C} \to \mathcal{C}^+$ be the function mapping every term in $\mathcal{C}$ to a non-empty list of terms such that $aliens_\sqcup(t) = aliens_\sqcup(u) \circ aliens_\sqcup(v)$ if $t = u \sqcup v$, and $aliens_\sqcup(t) = [t]$ otherwise.*

Conversely, let $comb_\sqcup : \mathcal{C}^+ \to \mathcal{C}$ be the function mapping a non-empty list of $\mathcal{C}$-terms to a term such that $comb_\sqcup[t] = t$ and for all $n \geq 2, comb_\sqcup[t_1, \ldots, t_n] = t_1 \sqcup comb_\sqcup[t_2, \ldots, t_n]$.

For example $aliens_\sqcup((x \sqcup y) \sqcup z) = [x, y, z]$ and $comb_\sqcup[x, y, z] = ((x \sqcup y) \sqcup z)$.

Definition 9 (AC-canonical form). *Let $\leq$ be any total order on $\mathcal{C}$-terms with ϵ the least element such that for all x and b we have $\epsilon < (\texttt{var}\ b\ x)$, and $(\texttt{var}\ b\ x) \leq (\texttt{var}\ b'\ y)$ iff $x < y$ or else $x = y$ and $b \leq b'$ with the order $\texttt{false} < \texttt{true}$. The AC-canonization of a term t of $\mathcal{C}$ is defined as $[t]_{AC} = comb_\sqcup[\texttt{sort}(aliens_\sqcup(t))]$, where `sort(l)` is the list of the elements of l in increasing order with respect to $\leq$. The relation associating every term t with its AC-canonization $[t]_{AC}$ is denoted $\twoheadrightarrow^{AC}$. Two terms t and t' are AC-equivalent if $[t]_{AC} = [t']_{AC}$. The term t is in AC-canonical form if $t = [t]_{AC}$ and if every strict subterm of t is in AC-canonical form.*

Example 2. Assuming that the terms x and y are ordered $x < y$, the AC-canonical form of XXX is XXX.

Definition 10 (Rewriting modulo AC-canonization). *Let $\longrightarrow_{\mathcal{R}}^{AC} = \twoheadrightarrow^{AC} \longrightarrow_{\mathcal{R}}$, where $\mathcal{R}$ is defined by the rewrite rules of ??.*

An $\longrightarrow_{\mathcal{R}}^{AC}$ step is an AC-canonization followed by a standard $\longrightarrow_{\mathcal{R}}$ step with syntactic matching.

6 Evaluation

References

1. Barbosa, H., Fleury, M., Fontaine, P., Schurr, H.J.: The Alethe Proof Format An Evolving Specification and Reference (Mar 2024), https://verit.gitlabpages.uliege.be/alethe/specification.pdf
2. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017–2024), https://smt-lib.org/papers/smt-lib-reference-v2.6-r2024-09-20.pdf, available at www.SMT-LIB.org
3. Besson, F.: Fast reflexive arithmetic tactics the linear case and beyond. In: Altenkirch, T., McBride, C. (eds.) Intl. Workshop Types for Proofs and Programs (TYPES 2006). pp. 48–62. LNCS, Springer, Berlin, Heidelberg (2006)
4. Blanqui, F.: Type Safety of Rewrite Rules in Dependent Types. In: 5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020). Leibniz International Proceedings in Informatics (LIPIcs), vol. 167, pp. 13:1–13:14 (2020)
5. Coltellacci, A., Merz, S., Dowek, G.: Reconstruction of SMT proofs with Lambdapi. In: Proc. 22nd Intl. Wsh. Satisfiability Modulo Theories co-located with the 36th Intl. Conf. Computer Aided Verification (CAV 2024). CEUR Workshop Proceedings, vol. 3725, pp. 13–23. CEUR-WS.org, Montreal, Canada (2024)
6. Cousineau, D., Dowek, G.: Embedding pure type systems in the Lambda-Pi-Calculus Modulo. In: Typed Lambda Calculi and Applications. pp. 102–117. Springer, Berlin, Heidelberg (2007)
7. Dowek, G.: On the definition of the classical connectives and quantifiers (2016)
8. Harper, R., Honsell, F., Plotkin, G.D.: A framework for defining logics. J. ACM **40**, 143–184 (1993), https://api.semanticscholar.org/CorpusID:13375103
9. Martin-Löf, P.: Intuitionistic type theory (1980),
10. Pugh, W.: The omega test: A fast and practical integer programming algorithm for dependence analysis. In: Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing. pp. 4–13 (1991)
11. Schrijver, A.: Theory of linear and integer programming. John Wiley & Sons, Inc., USA (1986)

A Alethe

A.1 The Syntax

```
⟨proof⟩ = ⟨proof_command⟩*
⟨proof_command⟩ = (assume ⟨symbol⟩ ⟨proof_term⟩ )
                | (step ⟨symbol⟩ ⟨clause⟩ :rule ⟨symbol⟩
                    ⟨premises_annotation⟩?
                    ⟨context_annotation⟩? ⟨attribute⟩* )
                | (anchor :step ⟨symbol⟩
                    ⟨args_annotation⟩? ⟨attribute⟩* )
                | (define-fun ⟨function_def⟩ )
⟨clause⟩ = (cl ⟨proof_term⟩* )
⟨proof_term⟩ = ⟨term⟩ extended with
               (choice ( ⟨sorted_var⟩ ) ⟨proof_term⟩ )
⟨premises_annotation⟩ = :premises ( ⟨symbol⟩+)
⟨args_annotation⟩ = :args ( ⟨step_arg⟩+ )
⟨step_arg⟩ = ⟨symbol⟩|( ⟨symbol⟩ ⟨proof_term⟩ )
⟨context_annotation⟩ = :args ( ⟨context_assignment⟩+ )
⟨context_assignment⟩ = ( ⟨sorted_var⟩ )
                     | (:= ⟨symbol⟩ ⟨proof_term⟩ )
```

Fig. 3. Alethe grammar