SMT Linear Integer Arithmetic proof checking in Lambdapi

Coltellacci Alessio[1] [0009−0005−3580−2075]

University of Lorraine, CNRS, Inria, LORIA Nancy alessio.coltellacci@inria.fr

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

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

1 Background

1.1 An Overview of Lambdapi

Lambdapi is an implementation of $\lambda\Pi$ modulo theory ($\lambda\Pi/\equiv$) [10], an extension of the Edinburgh Logical Framework $\lambda\Pi$ [11], a simply typed λ-calculus with dependent types. $\lambda\Pi/\equiv$ adds user-defined higher-order rewrite rules. Its syntax is given by

$$\begin{array}{lrl}
\text{Universes} & u & ::= \texttt{TYPE} \mid \texttt{KIND} \\
\text{Terms} & t, v, A, B, C & ::= c \mid x \mid u \mid \Pi\, x : A,\, B \mid \lambda\, x : A,\, t \mid t\ v \\
\text{Contexts} & \Gamma & ::= \langle\rangle \mid \Gamma, x : A \\
\text{Signatures} & \Sigma & ::= \langle\rangle \mid \Sigma, c : C \mid \Sigma, c := t : C \mid \Sigma, t \hookrightarrow v
\end{array}$$

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 [11, §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 $\Gamma \vdash_\Sigma v : A$ [5].

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 $\lambda\Pi$ [11, §2], except for the rule (Conv) where it use the version of fig. 2 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 $\lambda\Pi/\equiv$

1.2 Alethe proof

The Alethe proof trace format [2] 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)
```

Listing 1.1. Input problem

ϟ

```
(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 1.2. The following example is the proof for the unsatisfiability of $(x + y < 1) \vee (3 < x), x = 2$ 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.

Alethe Trace Format Overview 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 [3, §3] and the SMT signature context defined in [3, §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.

Rule	Description
la_generic	Tautologous disjunction of linear inequalities.
lia_generic	Tautologous disjunction of linear integer inequalities.
la_disequality	$t_1 \approx t_2 \lor \neg(t_1 \geq t_2) \lor \neg(t_2 \geq t_1)$
la_totality	$t_1 \geq t_2 \lor t_2 \geq t_1$
la_tautology	A trivial linear tautology
la_mult_pos	$t_1 > 0 \land (t_2 \bowtie t_3) \rightarrow t_1 * t_2 \bowtie t_1 * t_3$ and $\bowtie \in \{<, >, \geq, \leq, =\}$
la_mult_neg	$t_1 < 0 \land (t_2 \bowtie t_3) \rightarrow t_1 * t_2 \bowtie_{inv} t_1 * t_3$
la_rw_eq	$(t \approx u) \approx (t \geq u \land u \geq t)$
comp_simplify	Simplification of arithmetic comparisons.

Table 1. Linear arithmetic rules in Alethe.

We now have the key components to explain the guiding proof in the bottom part of listing 1.2. The proofs starts with `assume` steps `a0`, `a1`, `a2` that restate the assertions from the *input problem* (listing 1.2). 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 $l_1, l_2, \dots, l_n$ represent linear inequalities. These logics use closed linear formulas over the `Real` signature and `Int` respectively. The `Real` terms in `LRA` logic are built over the Reals signature from SMT-LIB with free variables, but containing only linear atoms; that is atoms of the form `d`, `(* d x)`, or `(* x d)` where `x` is a free variable and `d` is an integer or rational constant. Similarly, the `Int` terms in `LIA` logic are closed formulas built over the Ints signature with free variables, but whose terms are also all linear, such that there is no occurrences of the function symbols `*` (except variable multiplied by an `Int` constant), `/`, `div`, `mod`, and `abs`. A linear inequality is of term of the form

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

where $\bowtie \in \{=, <, >, \leq, \geq\}$, where $m \geq n$, c_i, d_1, d_2 are either `Int` or `Real` constants, and for each i c_i and t_i have the same sort. Checking the clause validity of `t2` and `t5` in listing 1.2, amounts to checking the unsatisfiability of the system of linear equations (we provide more details in section 1.2) 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 $\perp$, concretely denoted as `(cl)` in listing 1.2. Notice that the contexts Γ of each step are all empty in this proof.

Linear arithmetic in Alethe 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 `*_simplify`, 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.

Following our method to encode Alethe described in [9], the linear arithmetic tautology rules `la_disequality`, `la_totality` and `la_mult_*` are encoded as lemmas in our embedding of Alethe in Lambdapi. The simplification rule `comp_simplify` is encoded as a lemma for each rewrite case and applied multiple times. We do not support the remaining `*_simplify` rules and the `la_tautology` rule in this work, primarily because cvc5 does not follow the Alethe standard for simplification step. Instead, it extends the Alethe format with the RARE simplification rules [13]. As a result, cvc5 does not generate proofs using these standard rules for the SMT-LIB benchmarks.

A different approach is taken for the primary rules `*_generic`,as they describe an algorithm. 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. 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 `Int` variables are assumed to be `Real` variables. Although the rule may introduce rational coefficients, they often reduce to integers—as shown in listing 1.2, where the coefficients are $(\frac{1}{1}, \frac{1}{1})$.

Cases where coefficients cannot be reduced to integers are rare in practice, however, we eliminate . Let $\varphi_1, \dots, \varphi_n$ be linear inequalities and $a_1, \dots, a_n$ rational numbers, then a `la_generic` step has the general form

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

The constants a_i are of sort `Real`. To check the unsatisfiability of the negation of $\varphi_1, \dots, \varphi_n$ one performs the following steps for each literal. For each i, let $\varphi := \varphi_i$, $a := a_i$ and we write $s1 \bowtie s2$ to denotes the left and right side of an inequality of eq. (1).

1. If $\varphi = \neg(s_1 < s_2)$ or $s_1 \geq s_2$, then let $\varphi := \neg(-s_1 \geq -s_2)$. If $\varphi = \neg(s_1 \leq s_2)$ or $s_1 > s_2$, then let $\varphi := \neg(-s_1 > -s_2)$. If $\varphi = s_1 < s_2$, then let $\varphi := \neg(s_1 \geq s_2)$. If $\varphi = s_1 \leq s_2$, then let $\varphi := \neg(s_1 > s_2)$. This negates the literal. We want a canonical form that use only the operators $>, \geq$ and $=$.
2. Replace $\varphi = \sum_{i=0}^{n} c_i \times t_i + d_1 \bowtie \sum_{i=n+1}^{m} c_i \times t_i + d_2$ by $\sum_{i=0}^{n} c_i \times t_i - \sum_{i=n+1}^{m} c_i \times t_i \bowtie d_2 - d_1$.
3. Now φ has the form $s_1 \bowtie d$. If all variables in s_1 are integer sorted then replace $\bowtie d$ by $\bowtie \lceil d \rceil$, otherwise replace by $\bowtie \lfloor d \rfloor + 1$.
4. If all variables of φ are Int and coefficient $a_1 \dots a_n \in \mathbb{Q}$, then $a_i := a \times lcd(a_1 \dots a_n)$ where lcd is the least common denominator of $[a_1 \dots a_n]$.
5. If $\bowtie$ is $=$, then replace φ by $\sum_{i=0}^{m} a \times c_i \times t_i = a \times d$, otherwise replace it by $\sum_{i=0}^{m} |a| \times c_i \times t_i \bowtie |a| \times d$.
6. Finally, the sum of the resulting literals is trivially contradictory. The sum

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

   where c_i^k and t_i^k are the constant and term from the literal φ_k, and d^k is the constant d of φ_k. The operator $\bowtie$ is $=$ if all operators are $=$, $>$ if all are either $=$ or $>$, and $\geq$ otherwise. Finally, the sum on the left-hand side is 0 and the right-hand side is > 0 (or ≥ 0 if $\bowtie$ is $>$).

The step 1 has been added by our own, as the subsequent steps in the original algorithm are designed for $>$ and $\geq$ and do not clearly address how to handle $<$ and $\leq$. Additionally, step 6 was added to ensure that our construction is independent of $\mathbb{Q}$.

Example 1. Consider the `la_generic` step in the logic `QF_UFLIA` with the uninterpreted function symbol `(f Int)`:

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

the algorithm run as follow in a natural deduction:

$$\begin{aligned}
&\vdash \neg(-f > 0),\ \neg(4f > 0) && \text{(Step 2)}\\
&\vdash \neg(-f > 0),\ \neg(4f \geq 1) && \text{(Step 3)}\\
&\text{Replace } a = [\frac{1}{1}, \frac{1}{4}] \text{ by } a = [4, 1] && \text{(Step 4)}\\
&\vdash \neg(|4| * -f > |4| * 0),\ \neg(|1| * 4f \geq |1| * 1) && \text{(Step 5)}\\
&-4f + 4f \geq 1 \vdash \texttt{False} && \text{(Step 6)}
\end{aligned}$$

Je met $\vdash$ car je trouve que la dérivation ce comprend mieu de mon point de vue. Mais je peux l'enlever si trop de confusion est ajouté.

Which sums to the contradiction $0 \geq 1$.

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* [15].

2 The approach to reconstruct lia_generic step

The `lia_generic` represent a challenge for reconstruction because the coefficients are not provided by the solver in the trace i.e. $[a_1 \dots a_r]$ is empty. 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 [14, 4] - and therefore reimplementing the work done by the solver.

Carcara considers `lia_generic` steps as holes in the proof, as "their checking is as hard as solving" [1, §3.2]. However, 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, and validated before replacing the original step. However, at present, Carcara only use cvc5 for performing this task. 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 $l_1 \wedge \dots \wedge l_n$ and invokes *cvc5* on it, expecting an Alethe proof $\pi : (l_1 \wedge \dots \wedge l_n) \to \bot$. Carcara will check this subproof and then replace step S in the original proof by a proof of the form:

```
(step S (cl (not l1) ... (not ln)) :rule lia_generic)
```

$\lightning$

```
(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 1.3. Elaboration of `lia_generic`

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.

3 Reconstruction of `la_generic` step for LIA logic

3.1 An Overview of Lambdapi

Lambdapi is an implementation of $\lambda\Pi$ modulo theory ($\lambda\Pi/\equiv$) [10], an extension of the Edinburgh Logical Framework $\lambda\Pi$ [11], a simply typed λ-calculus with dependent types. $\lambda\Pi/\equiv$ adds user-defined higher-order rewrite rules. Its syntax is given by

$$
\begin{array}{llr}
\text{Universes} & u ::= \texttt{TYPE} \mid \texttt{KIND} \\
\text{Terms} & t, v, A, B, C ::= c \mid x \mid u \mid \Pi\, x : A,\, B \mid \lambda\, x : A,\, t \mid t\ v \\
\text{Contexts} & \Gamma ::= \langle\rangle \mid \Gamma, x : A \\
\text{Signatures} & \Sigma ::= \langle\rangle \mid \Sigma, c : C \mid \Sigma, c := t : C \mid \Sigma, t \hookrightarrow v
\end{array}
$$

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 [11, §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 $t = c\ v_1 \ldots v_n$ 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 $\Gamma \vdash_\Sigma v : A$ [5].

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 $\lambda\Pi$ [11, §2], except for the rule (Conv) where it use the version of fig. 2 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. 2. (Conv) rule in $\lambda\Pi/\equiv$

We now provide an overview of the encoding of Alethe linear integers arithmetic in Lambdapi.

3.2 Encoding of Integers in Lambdapi

$\mathbb{Z}$: `TYPE`	$\mathbb{P}$: `TYPE`	`Comp` : `TYPE`	$\mathbb{B}$: `TYPE`
∣ `Z0` : $\mathbb{Z}$	∣ `H` : $\mathbb{P}$	∣ `Eq` : `Comp`	∣ `true` : $\mathbb{B}$
∣ `ZPos` : $\mathbb{P} \to \mathbb{Z}$	∣ `O` : $\mathbb{P} \to \mathbb{P}$	∣ `Lt` : `Comp`	∣ `false` : $\mathbb{B}$
∣ `ZNeg` : $\mathbb{P} \to \mathbb{Z}$	∣ `I` : $\mathbb{P} \to \mathbb{P}$	∣ `Gt` : `Comp`	
`int` : `Set`	`pos` : `Set`	`comp` : `Set`	`bool` : `Set`
`El int` $\hookrightarrow \mathbb{Z}$	`El pos` $\hookrightarrow \mathbb{P}$	`El comp` $\hookrightarrow$ `Comp`	`El comp` $\hookrightarrow \mathbb{B}$

Fig. 3. Overview of sorts, constructors, constants, and element relations

The definition we use of integers in Lambdapi in fig. 3 follows a common encoding found in many other theories, including the one adopted in the Rocq standard library [16]. 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). As discussed in our previous work [9], $\lambda\Pi/\equiv$ does not support quantifiy over a variable of type `TYPE`. More precisely, it is not possible to assign the type ΠX : `TYPE`, ($X \to$ `Prop`) $\to$ `Prop` to the universal quantifier $\forall$, where `Prop` : `TYPE` is the type of proposition. To address this, we introduce a constant `Set` : `TYPE` for the types of object-terms, and a constant `El` to embed the terms of type `Set` into terms of type `TYPE` giving us the quantifier $\forall$: Πx : `Set`, (`El` $x \to$ `Prop`) $\to$ `Prop`. To enable quantification over types such as integers, positive binary numbers, booleans, and comparison results, we introduce a constant of type `Set` (e.g. `int` : `Set`) that represents codes for these types — similar to the Tarski-style universe [12, §Universes], where types are represented by elements of a base type and interpreted via the decoding function. In our setting, the decoding function `El` is realized through a rewriting rule that reduces the term to its corresponding type; for example, `El int` $\hookrightarrow \mathbb{Z}$. The comparison datatype `Comp` is utilized to define the decidable equality $\doteq$ between the $\mathbb{Z}$ and the function `cmp` for $\mathbb{P}$ (as defined in appendix B).

The arithmetic operator such as `add`, `sub`, and others, as presented in fig. 4 are constants defined by rewriting rules. In the following sections, we will refers to the rewriting rules for integers as $\to_{\mathbb{Z}}$ and positive binary numbers as $\to_{\mathbb{P}}$. The confluence of the rewriting rules for the arithmetic of $\mathbb{Z}$ and $\mathbb{P}$ has been proven

$$
\begin{array}{ll}
+ : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} & \doteq : \mathbb{Z} \to \mathbb{Z} \to \texttt{Comp} \\
\texttt{Z0} + y \hookrightarrow y & \texttt{Z0} \doteq \texttt{Z0} \hookrightarrow \texttt{Eq} \\
x + \texttt{Z0} \hookrightarrow \texttt{Z0} & \texttt{Z0} \doteq \texttt{Zpos}\ \_ \hookrightarrow \texttt{Lt} \\
(\texttt{Zpos x}) + (\texttt{Zpos y}) \hookrightarrow (\texttt{Zpos}\ (\texttt{add}\ x\ y)) & \texttt{Z0} \doteq \texttt{Zneg}\ \_ \hookrightarrow \texttt{Gt} \\
(\texttt{Zpos x}) + (\texttt{Zneg y}) \hookrightarrow (\texttt{sub}\ x\ y) & \texttt{Zpos}\ \_ \doteq \texttt{Z0} \hookrightarrow \texttt{Gt} \\
(\texttt{Zneg x}) + (\texttt{Zpos y}) \hookrightarrow (\texttt{sub}\ y\ x) & \texttt{Zpos}\ p \doteq \texttt{Zpos}\ q \hookrightarrow \texttt{cmp}\ p\ q \\
(\texttt{Zneg x}) + (\texttt{Zneg y}) \hookrightarrow \texttt{Zpos}(\texttt{add}\ x\ y) & \texttt{Zpos}\ \_ \doteq \texttt{Zneg}\ \_ \hookrightarrow \texttt{Gt} \\
 & \texttt{Zneg}\ \_ \doteq \texttt{Z0} \hookrightarrow \texttt{Lt} \\
 & \texttt{Zneg}\ \_ \doteq \texttt{Zpos}\ \_ \hookrightarrow \texttt{Lt} \\
 & \texttt{Zneg}\ p \doteq \texttt{Zneg}\ q \hookrightarrow \texttt{cmp}\ q\ p
\end{array}
$$

$$
\begin{array}{lll}
\texttt{isEq} : \texttt{Comp} \to \mathbb{B} & \texttt{isLt} : \texttt{Comp} \to \mathbb{B} & \texttt{isGt} : \texttt{Comp} \to \mathbb{B} \\
\texttt{isEq Eq} \hookrightarrow \texttt{true} & \texttt{isLt Eq} \hookrightarrow \texttt{false} & \texttt{isGt Eq} \hookrightarrow \texttt{false} \\
\texttt{isEq Lt} \hookrightarrow \texttt{false} & \texttt{isLt Lt} \hookrightarrow \texttt{true} & \texttt{isGt Lt} \hookrightarrow \texttt{false} \\
\texttt{isEq Gt} \hookrightarrow \texttt{false} & \texttt{isLt Gt} \hookrightarrow \texttt{false} & \texttt{isGt Gt} \hookrightarrow \texttt{true}
\end{array}
$$

$$
\begin{array}{ll}
\leq : \mathbb{Z} \to \mathbb{Z} \to \texttt{Prop} \coloneqq \lambda x, \lambda y, \neg(\texttt{istrue}(\texttt{isGt}(x \doteq y))) & \texttt{istrue} : \mathbb{B} \to \texttt{Prop} \\
< : \mathbb{Z} \to \mathbb{Z} \to \texttt{Prop} \coloneqq \lambda x, \lambda y, (\texttt{istrue}(\texttt{isLt}(x \doteq y))) & \texttt{istrue true} \hookrightarrow \top \\
\geq : \mathbb{Z} \to \mathbb{Z} \to \texttt{Prop} \coloneqq \lambda x, \lambda y, \neg(x < y) & \texttt{istrue false} \hookrightarrow \bot \\
> : \mathbb{Z} \to \mathbb{Z} \to \texttt{Prop} \coloneqq \lambda x, \lambda y, \neg(x \leq y) &
\end{array}
$$

Fig. 4. Decidable equality, + operator and inequalities relations definitions for $\mathbb{Z}$

using CSI [17]. A detailed proof of confluence can be found in appendix B.1. The inequality symbols for $\mathbb{Z}$ are binary predicates defined by rewriting rules over the decidable equality $\doteq$. They reduce to $\top$, $\bot$ (or negated) by $\equiv_{\beta\Sigma}$ with rules of $\rightarrow_{\mathbb{Z}}$ and $\rightarrow_{\mathbb{P}}$. For example, $1 < 2 \hookrightarrow \texttt{istrue}(\texttt{isLt}(1 \doteq 2)) \hookrightarrow \texttt{istrue}(\texttt{isLt}(\texttt{Lt})) \hookrightarrow \texttt{istrue}(\texttt{true}) \hookrightarrow \top$.

3.3 Functions used in the translation

We now provide an overview of how input problems expressed in a given SMT-LIB signature [3, §5.2.1] are encoded. A comprehensive description of the encoding can be found in [9], we will focus here on the arithmetic. 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 $\Theta^{\mathcal{X}}$. 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 $i.\ \Gamma \triangleright\ \varphi\ (\mathcal{R}\ P)[A]$ to typing judgments $\Gamma \vdash_{\Sigma} i := M : N$.

Definition 1 (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 $s : \texttt{Set} \rightarrow \cdots \rightarrow \texttt{Set}$. For each function symbol $f\ \sigma^{+}$ in $\Theta^{\mathcal{F}}$ we create a constant $f : \mathcal{S}(\sigma^{+})$.*

Definition 2 (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 $s = \sigma_1\, \sigma_2 \dots \sigma_n$ then $\mathcal{S}(s) = \texttt{El}(\mathcal{S}(\sigma_1) \rightsquigarrow \cdots \rightsquigarrow \mathcal{S}(\sigma_n))$,*
- *otherwise $\mathcal{S}(s) = \texttt{El}\, \mathcal{D}(s)$.*

with the constant $o : \texttt{Set}$ and $\texttt{El}\, o \hookrightarrow \texttt{Prop}$ to quantify over propositions.

Definition 3 (Function $\mathcal{E}$ translating SMT expressions). *The definition of $\mathcal{E}(e)$ is as follows.*

- *Case $e = (p\ t_1\ t_2 \dots\ t_n)$ and p a logical connector, then $\mathcal{E}(e) = \mathcal{E}(t_1)\ p^c\ \dots\ p^c\ \mathcal{E}(t_n)$.*
- *Case $e = (+\ t_1 \dots\ t_n)$, then $\mathcal{E}(e) = \mathcal{E}(t_1) +\ \dots\ + \mathcal{E}(t_n)$.*
- *Case $e = (*\ t_1 \dots\ t_n)$, then $\mathcal{E}(e) = \mathcal{E}(t_1) *\ \dots\ * \mathcal{E}(t_n)$.*
- *Case $e = (-t)$, then $\mathcal{E}(e) = \sim \mathcal{E}(t_1)$.*
- *Case $e = (-\ t_1 \dots\ t_n)$, then $\mathcal{E}(e) = \mathcal{E}(t_1) -\ \dots\ - \mathcal{E}(t_n)$.*
- *Case $e = (\approx\ t_1\ t_2)$ then $\mathcal{E}(e) = (\mathcal{E}(t_1) = \mathcal{E}(t_2))$.*
- *Case $e = (Q\ x_1 : \sigma_1 \dots x_n : \sigma_n\ t)$ where $Q \in \{\texttt{forall}, \texttt{exists}\}$, then $\mathcal{E}(e) = Q^c x_1 : \mathcal{S}(\sigma_1), \dots, Q^c x_n : \mathcal{S}(\sigma_n), \mathcal{E}(t)$.*

– *Case* $e = (x : \sigma)$ *with* $x \in \Theta^{\mathcal{X}}$ *a sorted variable, then* $\mathcal{E}(e) = x : \mathcal{S}(\sigma)$.

Definition 4. *We define the function* $\mathcal{C}$ *that translates an Alethe step of the form* $i.\ \Gamma \rhd l_1 \ldots l_n\,(R\,P)[A]$ *into a constant* $i : \texttt{Prf}^{\bullet}(\mathcal{E}(l_1) \triangledown \ldots \triangledown \mathcal{E}(l_n) \triangledown \blacksquare) \coloneqq M$ *where* M *is a proof term of appropriate type. The function* $\mathcal{C}$ *is defined by cases on the rule* R.

We introduce an embedding $\texttt{Prf}^{\bullet} : \texttt{Clause} \rightarrow \texttt{TYPE}$ of clause into types, mapping each clause C to the type $\texttt{Prf}^{\bullet}\ C$ of its proofs. Similarly, the constant $\texttt{Prf}^{c} : \texttt{Prop} \rightarrow \texttt{TYPE}$ maps each proposition A. The type $\texttt{Clause} : \texttt{TYPE}$ represent the type of clause encoded as list [9, §3] with the constructor $\triangledown : \texttt{Prop} \rightarrow \texttt{Clause} \rightarrow \texttt{Clause}$ and the empty clause $\blacksquare : \texttt{clause}$.

Example 2. Translation of the proof goal of the prelude, step t2 and t5 in listing 1.2

```
symbol x: El int;
symbol y: El int;
...
opaque symbol t2: Prf• (¬ (3 < x)  ▽ ¬ (x = 2) ▽ ■) ≔ begin ... end;
opaque symbol t5: Prf• (¬ (x + y < 1) ▽ (¬ (x = 2))  ▽ (¬ (0 = y)) ▽ ■) ≔
begin ... end;
...
```

The proof terms generated by $\mathcal{C}$ for steps `t2` and `t5` must faithfully represent the algorithm presented in **??**. While steps 1 through 7 of the algorithm correspond to explicit rewriting steps, the final step (step 8) — which involves summing all inequalities represents a multi-step rewriting sequence. This sequence reduced the initial sum to a decidable comparison between constants (e.g. $0 > 1 \hookrightarrow \bot$), which serves as the conclusion of the reduction. The reflection technique introduced by [8] leverages the reduction system of the proof assistant to produce an efficient decidable automatic procedure for comparing *Ring* terms. We decided to follow this approach to implement our decision procedure for evaluating inequality. In the following section, we describe how we implemented this procedure for our case, and how we extended the definition of $\mathcal{C}$.

4 Reconstruction of linear arithmetic for LIA logic

Proof by reflection is a technique used to write certified automation procedure by reducing the validity of a logical statement to a symbolic computation. It relies on the following definitions: let $P : Z \rightarrow \texttt{Prop}$ be a predicate over a data type Z and we have a function $f : Z \rightarrow \texttt{bool}$ such that the following theorem holds:

$$\texttt{f\_correct} : \forall z : Z, (f\ z = \texttt{true}) \rightarrow (P\ z)$$

If f is defined so that it reduces to fz to `true` for a broad class of expressions z, then the following proof term `f_correct` z (`relf true`) constitutes a proof of predicate $(P\ z)$. In step 8 of the `la_generic`, the primary challenge lies in reasoning modulo associativity and commutativity when manipulating expressions over $\mathbb{Z}$. Since arithmetic operators and inequality relations are defined by rewrite rules by rewrites rules as shown fig. 4, the key idea is to provide a normalization function that transforms a $\mathbb{Z}$ expression into a canonical form, such that it can be reduce to a constant as variable terms of a linear polynomial cancel each other e.g. x and y in listing 1.2.

Add an example of the reduction step by step in Sec 1.

To handle associative and commutative, Lambdapi provides the modifiers `associative commutative symbol`, ensuring that terms are systematically placed into a canonical form given a builtin ordering relation, following the technique described in [7] and [6, §5]. To avoid polluting the $\mathbb{Z}$ type and its associated rewrites rules, we introduce a separate type dedicated to representing linear polynomials. By applying the `associative commutative` modifiers within this specialized type, we ensure canonicalization of expressions while preserving the original structure and semantics of $\mathbb{Z}$.

4.1 Representation

The procedure is based on a group structure, denoted as $\mathbb{G}$ defined in fig. 5 which represents linear polynomials. The base type for the elements of this group is specified as $\mathbb{G}$: TYPE. The unary operator `cst` denotes a constant from $\mathbb{Z}$. A `var` constructoris a "catch-all" case for subexpressions that we cannot model. These subexpressions will correspond to actual variables in $\mathbb{Z}$. The constructor `mul` represents the multiplication of an element of $\mathbb{G}$ by a constant. The constructor `opp` corresponds to the inverse operator within the group. Lastly, the constructor `add` represent the addition between two elements of fig. 5 and it is characterized by the modifiers `associative commutative`.

$\mathbb{G}$: TYPE	$\Uparrow : \mathbb{Z} \to \mathbb{G}$	$\Downarrow : \mathbb{G} \to \mathbb{Z}$
$\mid$ `add` $: \mathbb{G} \to \mathbb{G} \to \mathbb{G}$	$\Uparrow\ Z0 \hookrightarrow (\texttt{cst}\ Z0)$	$\Downarrow\ (\texttt{cst}\ c) \hookrightarrow c$
$\mid$ `var` $: \mathbb{Z} \to \mathbb{Z} \to \mathbb{G}$	$\Uparrow\ (x + y) \hookrightarrow \texttt{add}\ (\Uparrow\ x\ )(\Uparrow\ y)$	$\Downarrow\ \texttt{opp}\ x \hookrightarrow \sim (\Downarrow\ x)$
$\mid$ `mul` $: \mathbb{Z} \to \mathbb{R} \to \mathbb{G}$	$\Uparrow\ (\sim x) \hookrightarrow \texttt{opp}\ \Uparrow\ x$	$\Downarrow\ \texttt{mul}\ c\ x \hookrightarrow c \times (\Downarrow\ x)$
$\mid$ `opp` $: \mathbb{R} \to \mathbb{G}$	$\Uparrow\ x \hookrightarrow (\texttt{var}\ x\ 1)$	$\Downarrow\ \texttt{add}\ x\ y \hookrightarrow (\Downarrow\ x) + (\Downarrow\ y)$
$\mid$ `cst` $: \mathbb{Z} \to \mathbb{G}$		$\Downarrow\ (\texttt{var}\ c\ x) \hookrightarrow c \times x$
`grp` : `Set`		
`El grp` $\hookrightarrow \mathbb{G}$		

Fig. 5. Definition of $\mathbb{G}$ Algebra and its reification and denotation functions

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

4.2 Normalization

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

Definition 5. *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* $n \geq 2, comb_{\sqcup}[t_1, \dots, t_n] = t_1 \sqcup comb_{\sqcup}[t_2, \dots, t_n]$.

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

Definition 6 (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* $[t]_{AC} = comb_{\sqcup}[\mathtt{sort}(aliens_{\sqcup}(t))]$, *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 3. Assuming that the terms x and y are ordered $x < y$, the AC-canonical form of XXX is XXX.

Definition 7 (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.

4.3 Generation of the proof term for `la_generic` in `LIA`

5 Evaluation

References

1. Andreotti, B., Lachnitt, H., Barbosa, H.: Carcara: An efficient proof checker and elaborator for SMT proofs in the Alethe format. In: Tools and Algorithms for the Construction and Analysis of Systems: 29th International Conference, TACAS 2023. p. 367–386. Springer-Verlag, Cham (2023)
2. 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
3. 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
4. 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)
5. 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)
6. Blanqui, F.: Encoding Type Universes Without Using Matching Modulo Associativity and Commutativity. In: Felty, A.P. (ed.) 7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 228, pp. 24:1–24:14 (2022)
7. Blanqui, F., Hardin, T., Weis, P.: On the implementation of construction functions for non-free concrete data types. In: De Nicola, R. (ed.) Programming Languages and Systems. pp. 95–109. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
8. Boutin, S.: Using reflection to build efficient and certified decision procedures. In: Abadi, M., Ito, T. (eds.) Theoretical Aspects of Computer Software. pp. 515–529. Springer Berlin Heidelberg, Berlin, Heidelberg (1997)

9. 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)
10. 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)
11. Harper, R., Honsell, F., Plotkin, G.D.: A framework for defining logics. J. ACM **40**, 143–184 (1993), https://api.semanticscholar.org/CorpusID:13375103
12. Martin-Löf, P.: Intuitionistic type theory (1980),
13. Nötzli, A., Barbosa, H., Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Reconstructing fine-grained proofs of rewrites using a domain-specific language. In: FMCAD 2022. pp. 65–74 (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_12
14. 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)
15. Schrijver, A.: Theory of linear and integer programming. John Wiley & Sons, Inc., USA (1986)
16. The Rocq Development Team: The Rocq reference manual – release 9.0.0. https://rocq-prover.org/doc/V9.0.0/refman/index.html (2025)
17. Zankl, H., Felgenhauer, B., Middeldorp, A.: CSI - A confluence tool. In: Bjørner, N.S., Sofronie-Stokkermans, V. (eds.) 23rd Intl. Conf. Automated Deduction (CADE-23). LNCS, vol. 6803, pp. 499–505. Springer, Wroclaw, Poland (2011)

A Alethe

A.1 The Syntax

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

Fig. 6. Alethe grammar

B Lambdapi Formalizations for Integer and Binary Number Operations

$$
\begin{array}{ll}
\texttt{add} : \mathbb{P} \to \mathbb{P} \to \mathbb{P} & \texttt{add\_c} : \mathbb{P} \to \mathbb{P} \to \mathbb{P}\\
\texttt{add}\ (\texttt{I}\ x)\ (\texttt{I}\ q) \hookrightarrow \texttt{O}\ (\texttt{add\_c}\ x\ q) & \texttt{add\_c}\ (\texttt{I}\ x)\ (\texttt{I}\ q) \hookrightarrow \texttt{I}\ (\texttt{add\_c}\ x\ q)\\
\texttt{add}\ (\texttt{I}\ x)\ (\texttt{O}\ q) \hookrightarrow \texttt{I}\ (\texttt{add}\ x\ q) & \texttt{add\_c}\ (\texttt{I}\ x)\ (\texttt{O}\ q) \hookrightarrow \texttt{O}\ (\texttt{add\_c}\ x\ q)\\
\texttt{add}\ (\texttt{O}\ x)\ (\texttt{I}\ q) \hookrightarrow \texttt{I}\ (\texttt{add}\ x\ q) & \texttt{add\_c}\ (\texttt{O}\ x)\ (\texttt{I}\ q) \hookrightarrow \texttt{O}\ (\texttt{add\_c}\ x\ q)\\
\texttt{add}\ (\texttt{O}\ x)\ (\texttt{O}\ q) \hookrightarrow \texttt{O}\ (\texttt{add}\ x\ q) & \texttt{add\_c}\ (\texttt{O}\ x)\ (\texttt{O}\ q) \hookrightarrow \texttt{I}\ (\texttt{add}\ x\ q)\\
\texttt{add}\ x\ \texttt{H} \hookrightarrow \texttt{succ}\ x & \texttt{add\_c}\ x\ \texttt{H} \hookrightarrow \texttt{add}\ x\ (\texttt{O}\ \texttt{H})\\
\texttt{add}\ \texttt{H}\ y \hookrightarrow \texttt{succ}\ y & \texttt{add\_c}\ \texttt{H}\ y \hookrightarrow \texttt{add}\ (\texttt{O}\ \texttt{H})\ y
\end{array}
$$

$$
\begin{array}{l}
\mathtt{sub} : \mathbb{P} \to \mathbb{P} \to \mathbb{Z} \\
\mathtt{sub}\ (\mathtt{I}\ p)\ (\mathtt{I}\ q) \hookrightarrow \mathtt{double}\ (\mathtt{sub}\ p\ q) \\
\mathtt{sub}\ (\mathtt{I}\ p)\ (\mathtt{O}\ q) \hookrightarrow \mathtt{succ\_double}\ (\mathtt{sub}\ p\ q) \\
\mathtt{sub}\ (\mathtt{I}\ p)\ \mathtt{H} \hookrightarrow \mathtt{Zpos}\ (\mathtt{O}\ p) \\
\mathtt{sub}\ (\mathtt{O}\ p)\ (\mathtt{I}\ q) \hookrightarrow \mathtt{pred\_double}\ (\mathtt{sub}\ p\ q) \\
\mathtt{sub}\ (\mathtt{O}\ p)\ (\mathtt{O}\ q) \hookrightarrow \mathtt{double}\ (\mathtt{sub}\ p\ q) \\
\mathtt{sub}\ (\mathtt{O}\ p)\ \mathtt{H} \hookrightarrow \mathtt{Zpos}\ (\mathtt{pos\_pred\_double}\ p) \\
\mathtt{sub}\ \mathtt{H}\ (\mathtt{I}\ q) \hookrightarrow \mathtt{Zneg}\ (\mathtt{O}\ q) \\
\mathtt{sub}\ \mathtt{H}\ (\mathtt{O}\ q) \hookrightarrow \mathtt{Zneg}\ (\mathtt{pos\_pred\_double}\ q) \\
\mathtt{sub}\ \mathtt{H}\ \mathtt{H} \hookrightarrow \mathtt{Z0}
\end{array}
$$

$$
\begin{array}{l}
\mathtt{compare\_acc} : \mathbb{P} \to \mathtt{Comp} \to \mathbb{P} \to \mathtt{Comp} \\
\mathtt{compare\_acc}\ (\mathtt{I}\ x)\ c\ (\mathtt{I}\ q) \hookrightarrow \mathtt{compare\_acc}\ x\ c\ q \\
\mathtt{compare\_acc}\ (\mathtt{I}\ x)\ \_\ (\mathtt{O}\ q) \hookrightarrow \mathtt{compare\_acc}\ x\ \mathtt{Gt}\ q \\
\mathtt{compare\_acc}\ (\mathtt{I}\ \_)\ \_\ \mathtt{H} \hookrightarrow \mathtt{Gt} \\
\mathtt{compare\_acc}\ (\mathtt{O}\ x)\ \_\ (\mathtt{I}\ q) \hookrightarrow \mathtt{compare\_acc}\ x\ \mathtt{Lt}\ q \\
\mathtt{compare\_acc}\ (\mathtt{O}\ x)\ c\ (\mathtt{O}\ q) \hookrightarrow \mathtt{compare\_acc}\ x\ c\ q \\
\mathtt{compare\_acc}\ (\mathtt{O}\ \_)\ \_\ \mathtt{H} \hookrightarrow \mathtt{Gt} \\
\mathtt{compare\_acc}\ \mathtt{H}\ \_\ (\mathtt{I}\ \_) \hookrightarrow \mathtt{Lt} \\
\mathtt{compare\_acc}\ \mathtt{H}\ \_\ (\mathtt{O}\ \_) \hookrightarrow \mathtt{Lt} \\
\mathtt{compare\_acc}\ \mathtt{H}\ c\ \mathtt{H} \hookrightarrow c
\end{array}
$$

$$\mathtt{compare}\ x\ y \coloneqq \mathtt{compare\_acc}\ x\ \mathtt{Eq}\ y$$

B.1 Confluence of the rewriting rules of integers and positive binary number

The rules presented below represent the relations $\to_{\mathbb{Z}}$ and $\to_{\mathbb{P}}$ encoded in the TRS[1] format accepted by the [17] tool. These rules can be used to rerun the tool in order to verify the confluence property.

```
(VAR
  a: Z
  b: Z
  x : P
```

[1] http://www.lri.fr/ marche/tpdb/format.html

```
  q : P
  y : P
)
(RULES
  ~(Z0) -> Z0
  ~(Zpos(p)) -> Zneg(p)
  ~(Zneg(p)) -> Zpos(p)
  ~(~(a)) -> a

  double(Z0) -> Z0
  double(Zpos(p)) -> Zpos(O(p))
  double(Zneg(p)) -> Zneg(O(p))

  succ_double(Z0) -> Zpos(H)
  succ_double(Zpos(p)) -> Zpos(I(p))
  succ_double(Zneg(p)) -> Zneg(pos_pred_double(p))

  pred_double(Z0) -> Zneg(H)
  pred_double(Zpos(p)) -> Zpos(pos_pred_double(p))
  pred_double(Zneg(p)) -> Zneg(I(p))

  sub(I(p), I(q)) -> double(sub(p, q))
  sub(I(p), O(q)) -> succ_double(sub(p, q))
  sub(I(p), H) -> Zpos(O(p))
  sub(O(p), I(q)) -> pred_double(sub(p, q))
  sub(O(p), O(q)) -> double(sub(p, q))
  sub(O(p), H) -> Zpos(pos_pred_double(p))
  sub(H, I(q)) -> Zneg(O(q))
  sub(H, O(q)) -> Zneg(pos_pred_double(q))
  sub(H, H) -> Z0

  +(Z0,a) -> a
  +(a,Z0) -> a
  +(Zpos(x), Zpos(y)) -> Zpos(add(x, y))
  +(Zpos(x), Zneg(y)) -> sub(x, y)
  +(Zneg(x), Zpos(y)) -> sub(y, x)
  +(Zneg(x), Zneg(y)) -> Zneg(add(x, y))

  mult(Z0, a) -> Z0
  mult(a, Z0) -> Z0
  mult(Zpos(x), Zpos(y)) -> Zpos(mul(x, y))
  mult(Zpos(x), Zneg(y)) -> Zneg(mul(x, y))
  mult(Zneg(x), Zpos(y)) -> Zneg(mul(x, y))
  mult(Zneg(x), Zneg(y)) -> Zpos(mul(x, y))


  succ(I(x)) -> O(succ(x))
  succ(O(x)) -> I(x)
  succ(H) -> O(H)
  add(I(x), I(q)) -> O(addcarry(x, q))
  add(I(x), O(q)) -> I(add(x, q))
  add(O(x), I(q)) -> I(add(x, q))
  add(O(x), O(q)) -> O(add(x, q))
  add(x, H) -> succ(x)
  add(H, y) -> succ(y)

  addcarry(I(x), I(q)) -> I(addcarry(x, q))
  addcarry(I(x), O(q)) -> O(addcarry(x, q))
  addcarry(O(x), I(q)) -> O(addcarry(x, q))
  addcarry(O(x), O(q)) -> I(add(x, q))
  addcarry(x, H) -> add(x, O(H))
  addcarry(H, y) -> add(O(H), y)

  pos_pred_double(I(x)) -> I(O(x))
  pos_pred_double(O(x)) -> I(pos_pred_double(x))
  pos_pred_double(H) -> H
```

```

  mul(I(x), y) -> add(x, O(mul(x,y)))
  mul(O(x), y) -> O(mul(x, y))
  mul(H, y) -> y
)
```

Listing 1.4. Rewriting rule of $\mathbb{Z}$ and $\mathbb{P}$ in the TRS format

Lemma 1 (Confluence).

Proof. CSI automatically proves the confluence of $\rightarrow_{\mathbb{Z}}$ *and* $\rightarrow_{\mathbb{P}}$ *by giving the polynomial interpretation:*

$$[\texttt{succ}(x)] = 4 * x \qquad [\texttt{add}(x, y)] = 4 * x + 4 * y + 2 \qquad [\texttt{H}] = 4$$