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. Modern SMT solvers can generate proofs of unsatisfiability to ensure correctness independently of their implementation. A dependable approach to verify these proofs is to reconstruct them within a proof assistant. In previous work, the SMT checker Carcara was extended to reconstruct SMT proofs in Lambdapi — a proof assistant designed for interoperability, supporting the import and export of proofs for integration with other proof assistants such as Rocq, Lean, or HOL-Light. Nevertheless, that work was limited to non-arithmetic fragments. In this paper, we present an extension that enables the reconstruction, and thus independant verification of SMT proofs involving linear integer arithmetic.

Keywords: SMT · Alethe · LIA · Lambdapi · normal form · Proof by Reflection

1 Introduction

SMT solvers have become capable of producing proofs alongside their satisfiability results, enabling independent verification of correctness without relying on the solver's implementation. This development addresses the growing need for trustworthy verification in safety-critical applications and SMT solver usage in proof assistant, where the solver cannot be assumed to be fully trusted. The Alethe [21, 2] is an established SMT proof format generated by the solvers cvc5 and veriT. In our previous work [9], we extended the Alethe proof checker and elaborator Carcara [1] to reconstruct Alethe SMT proofs within the Lambdapi proof assistant, thereby ensuring their validity. Lambdapi is an interactive proof assistant based on the $\lambda\Pi$-calculus modulo rewriting, a logical framework that extends the λ-calculus with dependent types and user-defined rewrite rules. This foundation allows Lambdapi to serve as a framework for formalizing various logical systems. Designed with interoperability in mind, Lambdapi can import and export proofs, facilitating integration with other proof assistants such as Rocq, Lean, and HOL-Light.

However, our prior work was limited to proofs expressed within the logic of Uninterpreted Functions (UF) and Quantifier-Free (QF) fragments. In the present work, we extend this approach to support the reconstruction of proof steps involving linear integer arithmetic (LIA). When the propositional model is unsatisfiable in the theory of linear real/integer arithmetic, the solver creates a proof certificate. The conclusion is a tautological clause of linear inequalities

and equations and the justification of the step is a list of coefficients so that the linear combination is a trivially contradictory inequality after simplification (e.g., $0 \geq 1$). Most SMT solvers, including cvc5 and veriT, use the simplex method [11] to handle linear arithmetic. To verify these certificates, we implemented an automatic decision procedure based on proof by reflection [8, 13].

In section 2, we provide an overview of Lambdapi and present the structure of Alethe proof certificates for linear arithmetic. Then, in section 3, we describe how we leverage Carcara's elaboration process to reconstruct linear integer arithmetic steps, even when coefficient annotations are missing. In section 5 introduces our encoding of SMT linear arithmetic within Lambdapi, while in section 4 presents an automatic decision procedure, based on proof by reflection, to verify these arithmetic steps. An empirical evaluation of our approach is provided in section 6. We review related work in section 7, and we conclude in section 8.

2 Background

2.1 An Overview of Lambdapi

Lambdapi is an implementation of $\lambda\Pi$ modulo theory ($\lambda\Pi/\equiv$) [10], an extension of the Edinburgh Logical Framework $\lambda\Pi$ [14], a simply typed λ-calculus with dependent types. $\lambda\Pi/\equiv$ adds user-defined higher-order rewrite rules. Its syntax is given by

Universes	$u ::= \texttt{TYPE} \mid \texttt{KIND}$
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 [14, §2.1]. $\Pi\, x : A,\, B$ is the dependent product, and we write $A \to B$ when B does not depend on x, $\lambda\, x : A.\, t$ 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

$\lambda\Pi$ [14, §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 $\lambda\Pi/\equiv$

In our encoding presented [9] we employ Tarski-style universe [17, §Universes] where types are represented by elements of a base type and interpreted via the decoding function. We defined the constant `Prop` : `TYPE` for the type of proposition and the decoding function $\texttt{Prf}^c$: `Prop` $\to$ `TYPE` that maps each proposition to `TYPE`. Since Lambdapi 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 $\forall$. To circumvent this, we have the constant `Set` : `TYPE` for the types of object-terms, and a decoding function `El` : `Set` $\to$ `TYPE` that embeds `Set` into `TYPE`. This permits us to define the quantifier as $\forall : \Pi x :$ `Set`, (`El` $x \to$ `Prop`$) \to$ `Prop`. To quantify over propositions, we further define a constant o : `Set` and add the rewrite rule `El` $o \hookrightarrow$ `Prop`.

2.2 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

↯

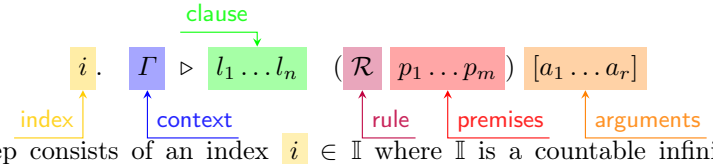
```
(assume a0 (or (< (+ x y) 1) (< 3 x)))
(assume a1 (= x 2))
(assume a2 (= 0 y))
(step t1 (cl (< (+ x y) 1) (< 3 x)) :rule or :premises (a0))
(step t2 (cl (not (< 3 x)) (not (= x 2))) :rule la_generic :args (1/1 1/1))
(step t3 (cl (not (< 3 x))) :rule resolution :premises (a1 t2))
```

```
7 (step t4 (cl (< (+ x y) 1)) :rule resolution :premises (t1 t3))
8 (step t5 (cl (not (< (+ x y) 1)) (not (= x 2)) (not (= 0 y))) :rule
  la_generic :args (1/1 -1/1 1/1))
9 (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 listing 1.1 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



(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.

add RARE rules in table 1

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

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_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	$(t \approx u) \approx (t \geq u \wedge u \geq t)$
comp_simplify	Simplification of arithmetic comparisons.

Table 1. Linear arithmetic rules in Alethe supported.

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 2.2) 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 $\bot$, 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`: $(t_1 \leq t_2 \vee t_2 \geq t_1)$.

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.

mention RARE rules

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, how-

ever, we eliminate . Let $\varphi_1, \ldots, \varphi_n$ be linear inequalities and $a_1, \ldots, a_n$ rational numbers, then a `la_generic` step has the general form

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

The constants a_i are of sort `Real`. 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 $\varphi := \varphi_i$, $a := a_i$ 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 $a_1 \ldots a_n \in \mathbb{Q}$, then $a_i := a \times lcd(a_1 \ldots a_n)$ where lcd is the least common denominator of $[a_1 \ldots a_n]$.
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:

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

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

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.

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* [20].

3 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 [19, 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 $\pi : (l_1 \wedge \dots \wedge l_n) \to \bot$ that do not use `lia_generic`. 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)
```

↯

```
(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)
```

```
9 (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.

4 Reconstruction of `la_generic` step for LIA logic

4.1 Encoding of Integers in Lambdapi

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

Fig. 2. Overview of inductive types for integers and manipulating them

The definition we use of integers in Lambdapi in fig. 2 follows a common encoding found in many other theories, including the one adopted in the Rocq standard library [22]. 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). To enable quantification over integers, positive binary numbers, booleans, and comparison results, we introduce constants such as $\texttt{int} : \texttt{Set}$ that represents codes for these types along with a rewrite rule that rewrite code to its corresponding type, for example $\texttt{El int} \hookrightarrow \mathbb{Z}$. The comparison inductive type `Comp` is used to specify comparison function. The infix function $\doteq$ define a decidable order comparison between two terms of type $\mathbb{Z}$. Similarly for $\mathbb{P}$ with the infix function $\texttt{cmp} : \mathbb{P} \to \mathbb{P} \to \texttt{Comp}$.

The arithmetic operator such as $+$ 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 using CSI [23]. A detailed proof of confluence can be found in appendix B.1. The inequality symbols for $\mathbb{Z}$ are

$$
\begin{array}{l}
+ : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} \\
\texttt{Z0} + y \hookrightarrow y \\
x + \texttt{Z0} \hookrightarrow x \\
(\texttt{Zpos x}) + (\texttt{Zpos y}) \hookrightarrow (\texttt{Zpos (add } x\ y)) \\
(\texttt{Zpos x}) + (\texttt{Zneg y}) \hookrightarrow (\texttt{sub } x\ y) \\
(\texttt{Zneg x}) + (\texttt{Zpos y}) \hookrightarrow (\texttt{sub } y\ x) \\
(\texttt{Zneg x}) + (\texttt{Zneg y}) \hookrightarrow \texttt{Zpos(add } x\ y)
\end{array}
$$

$$
\begin{array}{l}
\doteq : \mathbb{Z} \to \mathbb{Z} \to \texttt{Comp} \\
\texttt{Z0} \doteq \texttt{Z0} \hookrightarrow \texttt{Eq} \\
\texttt{Z0} \doteq \texttt{Zpos \_} \hookrightarrow \texttt{Lt} \\
\texttt{Z0} \doteq \texttt{Zneg \_} \hookrightarrow \texttt{Gt} \\
\texttt{Zpos \_} \doteq \texttt{Z0} \hookrightarrow \texttt{Gt} \\
\texttt{Zpos } p \doteq \texttt{Zpos } q \hookrightarrow \texttt{cmp } p\ q \\
\texttt{Zpos \_} \doteq \texttt{Zneg \_} \hookrightarrow \texttt{Gt} \\
\texttt{Zneg \_} \doteq \texttt{Z0} \hookrightarrow \texttt{Lt} \\
\texttt{Zneg \_} \doteq \texttt{Zpos \_} \hookrightarrow \texttt{Lt} \\
\texttt{Zneg } p \doteq \texttt{Zneg } q \hookrightarrow \texttt{cmp } q\ p
\end{array}
$$

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

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

Fig. 3. Decidable equality, + operator and inequalities relations definitions for $\mathbb{Z}$

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 $\to_{\mathbb{Z}}$ and $\to_{\mathbb{P}}$. For example, $1 < 2 \hookrightarrow \texttt{istrue}(\texttt{isLt}(1 \doteq 2)) \hookrightarrow \texttt{istrue}(\texttt{isLt}(\texttt{Lt})) \hookrightarrow \texttt{istrue}(\texttt{true}) \hookrightarrow \top$.

4.2 Functions used in the translation

We now outline the encoding of arithmetic expressions from SMT-LIB [3, §5.2.1]. This extends the approach introduced in [9] to handle arithmetic constructs. To avoid notational conflicts 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 $s : \texttt{Set}^0 \to \cdots \to^{n-1} \texttt{Set}$. 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 $s =$ **Bool**, then $\mathcal{S}(s) = \texttt{El}\ o$,*
- *Case $s =$ **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 $\mathcal{S}(s) = \texttt{El}\ \mathcal{D}(s)$.*

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

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

We defined the function $\texttt{Prf}^{\bullet} : \texttt{Clause} \to \texttt{TYPE}$, mapping each clause c to the type $\texttt{Prf}^{\bullet}\ c$ of its proofs. The type $\texttt{Clause} : \texttt{TYPE}$ represent the type of clause encoded as list [9, §3] with the constructor $\forall : \texttt{Prop} \to \texttt{Clause} \to \texttt{Clause}$ and the empty clause $\blacksquare : \texttt{clause}$. The function $\mathcal{C}$ encodes each step by invoking the functions $\mathcal{D}, \mathcal{S}$ and $\mathcal{E}$, and provides a proof term corresponding to the Alethe rule applied at that step.

Example 2. The translation of the steps t2 and t5 in listing 1.2 with the input problem definitions give us:

```
1 symbol x: El int;
2 symbol y: El int;
3 ...
4 opaque symbol t2: Prf• (¬ (3 < x)  ∨ ¬ (x = 2) ∨ ■) ≔ begin ... end;
5 opaque symbol t5: Prf• (¬ (x + y < 1) ∨ (¬ (x = 2))  ∨ (¬ (0 = y)) ∨ ■) ≔
  begin ... end;
6 ...
```

The proof terms generated by $\mathcal{C}$ for steps `t2` and `t5` must faithfully represent the algorithm presented in section 2.2. 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 solving arithmetic goals over $\mathbb{Q}, \mathbb{R}$ and $\mathbb{Z}$. 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 $\mathcal{C}$.

5 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 $P : Z \to$ `Prop` be a predicate over a data type Z and we have a function $f : Z \to$ `bool` such that the following theorem holds:

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

If $f\,z$ reduces to `true`, then the proof term `f_correct` z (`relf bool true`) with `refl` : ΠA : `Set`, Πx : `El` A, `Prf`$^c(x = x)$, constitutes a proof of predicate $(P\ z)$. In step 6 of the `la_generic`, the primary challenge lies in reasoning modulo associativity and commutativity when manipulating expressions over $\mathbb{Z}$. 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 because variables will cancel each other, as is the case with f in example 1.

5.1 Representation

The procedure is based on an algebraic group structure, denoted as $\mathbb{G}$ defined in fig. 4 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` constructor for "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. 4.

To support associative and commutative operations, Lambdapi provides the modifiers `associative commutative symbol`, ensuring that terms are systematically placed into a canonical form given a builtin ordering relation, as described in [7] and [6, §5]. Applying the `associative commutative` modifiers to the infix constructor $\oplus$, ensures that expressions involving sums of products are systematically canonicalized. Thus, equal variables are placed next to each other, facilitating simplification.

$$
\begin{array}{l}
\mathbb{G} : \texttt{TYPE} \\
\mid \oplus : \mathbb{G} \to \mathbb{G} \to \mathbb{G} \\
\mid \texttt{var} : \mathbb{Z} \to \mathbb{Z} \to \mathbb{G} \\
\mid \texttt{mul} : \mathbb{Z} \to \mathbb{G} \to \mathbb{G} \\
\mid \texttt{opp} : \mathbb{G} \to \mathbb{G} \\
\mid \texttt{cst} : \mathbb{Z} \to \mathbb{G} \\
\texttt{grp} : \texttt{Set} \\
\texttt{El grp} \hookrightarrow \mathbb{G}
\end{array}
\qquad
\begin{array}{l}
\Uparrow : \mathbb{Z} \to \mathbb{G} \\
\Uparrow \texttt{Z0} \hookrightarrow (\texttt{cst Z0}) \\
\Uparrow \texttt{ZPos}\, c \hookrightarrow (\texttt{cst}\ c) \\
\Uparrow \texttt{ZNeg}\, c \hookrightarrow (\texttt{cst}\ c) \\
\Uparrow (x + y) \hookrightarrow (\Uparrow x) \oplus (\Uparrow y) \\
\Uparrow (\sim x) \hookrightarrow \texttt{opp}\ \Uparrow x \\
\Uparrow (\texttt{Zpos}\, c) * x \hookrightarrow \texttt{mul}\ c\ (\Uparrow x) \\
\Uparrow (\texttt{ZNeg}\, c) * x \hookrightarrow \texttt{mul}\ c\ (\Uparrow x) \\
\Uparrow x * (\texttt{Zpos}\, c) \hookrightarrow \texttt{mul}\ c\ (\Uparrow x) \\
\Uparrow x * (\texttt{ZNeg}\, c) \hookrightarrow \texttt{mul}\ c\ (\Uparrow x) \\
\Uparrow x \hookrightarrow (\texttt{var}\ 1\ x)
\end{array}
\qquad
\begin{array}{l}
\Downarrow : \mathbb{G} \to \mathbb{Z} \\
\Downarrow (\texttt{cst}\ c) \hookrightarrow c \\
\Downarrow \texttt{opp}\ x \hookrightarrow \sim (\Downarrow x) \\
\Downarrow \texttt{mul}\ c\ x \hookrightarrow c \times (\Downarrow x) \\
\Downarrow x \oplus y \hookrightarrow (\Downarrow x) + (\Downarrow y) \\
\Downarrow (\texttt{var}\ c\ x) \hookrightarrow c \times x
\end{array}
$$

Fig. 4. Definition of $\mathbb{G}$ Algebra and its reification ($\Uparrow$) and denotation ($\Downarrow$) functions

5.2 Normalization

For `associative commutative` symbols, Lambdapi does not use matching modulo AC [18, 16] as this problem is NP-complete [15]. Instead, Lambdapi transforms sum expression of the form $(\texttt{var}\ c_1\ a_1) \oplus (\texttt{var}\ c_2\ a_2) \oplus \ldots \oplus (\texttt{var}\ c_n\ a_n)$ into a canonical form, ensuring that any pair of terms $(\texttt{var}\ p\ x)$ and $(\texttt{var}\ q\ x)$ involving the same variable x are placed adjacent to each other. In this way, we will use the rewriting rules *fig.* 5 to reduce expressions in canonical form. Notably, the resulting normal forms do not contain the constructors `mul` and `opp`, as the associated rewrite rules eliminate them in favor of `var`, `add` and `cst`.

Definition 4 (AC-canonical form). *Let $\leq$ be a total order on $\mathbb{G}$-terms defined as follows: Constants are ordered such that $\texttt{cst}(c_1) \leq \texttt{cst}(c_2) < (\texttt{var}\ p\ x)$ for any constants c_1, c_2 and any variable term $(\texttt{var}\ p\ x)$, with $c_1 \leq c_2$. For variable*

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

Fig. 5. Rewrite system on canonical forms

terms, $(\texttt{var}\ p\ x) \leq (\texttt{var}\ q\ y)$ *if either* $x < y$*, or* $x = y$ *and* $p \leq q$*. Let* $\twoheadrightarrow^{AC}$ *be the relation mapping every term* t *to its unique AC-canonical form denoted* $[t]$*.*

Two terms are AC-equivalent iff their AC-canonical forms are equal.

Definition 5 (Rewriting modulo AC-canonization). *Let* $\longrightarrow_{\Sigma}^{AC} = \hookrightarrow_{\Sigma} \twoheadrightarrow^{AC}$*, where* Σ *contains the rewrite rules of figs. 3 and 5.*

An $\longrightarrow_{\Sigma}^{AC}$ step is a standard $\hookrightarrow_{\Sigma}$ step with syntactic matching followed by ACcanonization. We now prove that the relation $\longrightarrow_{\Sigma}^{AC}$ terminates and is confluent.

Lemma 1. *The matching modulo AC relation* $\longrightarrow_{\Sigma/AC} = \simeq_{AC} \hookrightarrow_{\Sigma} \simeq_{AC}$*, which contains* $\longrightarrow_{\Sigma}^{AC}$*, terminates.*

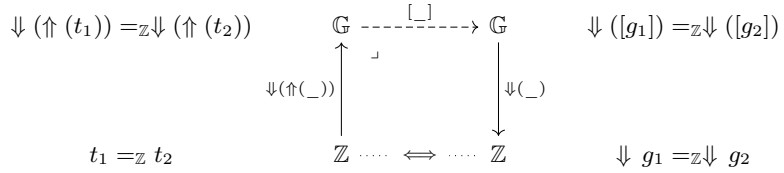
Proof. AProVE [12] automatically proves the termination of $\longrightarrow_{\Sigma/AC}$.

Lemma 2. $\longrightarrow_{\Sigma}^{AC}$ *is locally confluent on AC-canonical.*

Proof. We show that every critical pair is joinable using $\longrightarrow_{\Sigma}^{AC}$ *and confluence of* $\rightarrow_{\mathbb{Z}}$ *and* $\rightarrow_{\mathbb{P}}$*.*

To compare two $\mathbb{Z}$-terms, t_1 and t_2 wrt $\longrightarrow_{\Sigma}^{AC}$, the procedure involves reifying them into their corresponding $\mathbb{G}$-terms, denoted $[g_1]$ and $[g_2]$, using the reification function $\Uparrow (\_)$, and put in normal forms. Following the reduction rules specified

in fig. 5, we can then compare their corresponding $\mathbb{Z}$-terms by applying the denotation function $\Downarrow$ (_). To validate this procedure, it is necessary to establish the correctness of the following diagram:



by the correctness theorem:

Theorem 1 (Normalization correctness). *For all $\mathbb{G}$-terms t, we have $(\Downarrow [t]) = (\Downarrow t)$ where $[t]$ is the AC-canonical form of t with respect to $\longrightarrow_{\Sigma}^{AC}$.*

We make use of lemma 3 to embed $\mathbb{Z}$-terms into $\mathbb{G}$ in order to normalize and subsequent comparison.

Lemma 3 (Conversion). *For all $x : \mathbb{Z}$, we have $x = (\Downarrow (\Uparrow x))$.*

Example 3 (Lambdapi proof term for step `t5`).

```
1 symbol x: El int;
2 symbol y: El int;
3
4 opaque symbol t5: Prf• (¬ (x + y < 1) ∨ (¬ (x = 2))  ∨ (¬ (0 = y)) ∨ ■) ≔
  begin
5
6 end;
```

6 Evaluation

7 Related work

8 conclusion

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. Dutertre, B., de Moura, L.: Integrating simplex with dpll(t). Tech. Rep. CSL-06-01, SRI International (May 2006), https://yices.csl.sri.com/papers/sri-csl-06-01.pdf
12. Giesl, J., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Proving termination of programs automatically with aprove. In: Automated Reasoning. pp. 184–191. Springer International Publishing, Cham (2014)
13. Grégoire, B., Mahboubi, A.: Proving equalities in a commutative ring done right in coq. In: Hurd, J., Melham, T. (eds.) Theorem Proving in Higher Order Logics. pp. 98–113. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
14. Harper, R., Honsell, F., Plotkin, G.D.: A framework for defining logics. J. ACM **40**, 143–184 (1993), https://api.semanticscholar.org/CorpusID:13375103
15. Kapur, D., Narendran, P.: Np-completeness of the set unification and matching problems. In: 8th International Conference on Automated Deduction. pp. 489–495. Springer Berlin Heidelberg, Berlin, Heidelberg (1986)
16. Kirchner, H.: Rewriting systems and proofs. https://wiki.bordeaux.inria.fr/Helene-Kirchner/lib/exe/fetch.php?media=wiki:rsp.pdf (nd), accessed: 2025-04-30
17. Martin-Löf, P.: Intuitionistic type theory (1980),
18. Peterson, G.E., Stickel, M.E.: Complete sets of reductions for some equational theories. J. ACM **28**(2), 233–264 (Apr 1981)
19. 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)

20. Schrijver, A.: Theory of linear and integer programming. John Wiley & Sons, Inc., USA (1986)
21. Schurr, H.J., Fleury, M., Barbosa, H., Fontaine, P.: Alethe: Towards a generic smt proof format (extended abstract). Electronic Proceedings in Theoretical Computer Science **336**, 49–54 (07 2021)
22. The Rocq Development Team: The Rocq reference manual – release 9.0.0. https://rocq-prover.org/doc/V9.0.0/refman/index.html (2025)
23. 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

```
⟨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. 6. Alethe grammar

B Lambdapi Formalizations for Integer and Binary Number Operations

```
add : ℙ → ℙ → ℙ
add (I x) (I q) ↪ O (add_c 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
```

```
add_c : ℙ → ℙ → ℙ
add_c (I x) (I q) ↪ I (add_c x q)
add_c (I x) (O q) ↪ O (add_c x q)
add_c (O x) (I q) ↪ O (add_c x q)
add_c (O x) (O q) ↪ I (add x q)
add_c x H ↪ add x (O H)
add_c H y ↪ add (O H) y
```

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

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

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

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 [23] 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
```

```
71
72    mul(I(x), y) -> add(x, 0(mul(x,y)))
73    mul(0(x), y) -> 0(mul(x, y))
74    mul(H, y) -> y
75  )
```

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

Lemma 4 (Confluence).

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

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