Reconstructing SMT Proofs in Lambdapi

Alessio Coltellacci

Univ. Lorraine, CNRS, Inria, Loria

September 10, 2025

On the Correctness of SMT solvers

Can we trust SMT solvers?

- SMT solvers are widely used in proof assistants and program verification, but their large codebases make bugs hard to catch.

- Even major solvers show correctness issues.
  - Every year SMT-COMP uncovers disagreements between solvers results.

Can we trust their results?

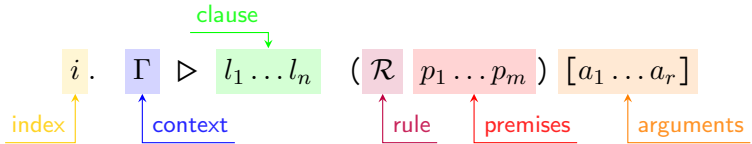
Why certifying solvers is impractical?

- A natural idea is to certify the solver itself.
  - Their extensive and complex codebases complicate certification.
  - Simplifying them for certification would sacrifice performance.
- In addition, once certified, a system becomes essentially frozen, which hinders the integration of new features and improvements.

SMT proofs

- Instead, the SMT solver can produce a proof (proof logging).
- An SMT proof is a certificate of the solver results, that formally justifies the logical reasoning it used to find a solution.
- Proofs can be checked independently, decoupling the confidence in the solver's results from the solver's implementation.
- Checking should be quicker than solving.

The Alethe format



(1)

- Many-Sorted First-Order Logic of SMT-LIB
- The proof forms a directed acyclic graph
- Proof rules $\mathcal{R}$ include theory lemmas

Example of an Alethe SMT Proof

```
1 (declare-sort U 0)
2 (declare-fun a () U)
3 (declare-fun b () U)
4 (declare-fun p (U) Bool)
5 (assert (p a))
6 (assert (= a b))
7 (assert (not (p b)))
8 (get-proof)
```

```
1 (assume a0 (p a))
2 (assume a1 (= a b))
3 (assume a2 (not (p b)))
4 (step t1 (cl (not (= (p a) (p b))) (not (p a)) (p b))
5       :rule equiv_pos2)
6 (step t2 (cl (= (p a) (p b))) :rule cong :premises (a1))
7 (step t3 (cl (p b)) :rule resolution :premises (t1 t2 a0))
8 (step t4 (cl) :rule resolution :premises (a2 t3))
```

Supported logics

Alethe support the SMT-LIB logics:

- Uninterpreted Function (**UF**)
- Linear Real Arithmetic (**LRA**)
- Linear Integer Arithmetic (**LIA**)
- Bitvectors[1] (**BV**)
- + Quantifier free formulas (**QF**)

[1] Work in progress

Classification of Alethe Rules

1. Special rules
   - $\triangleright\ \varphi$ (asssume)
   - $\triangleright\ \varphi$ (hole; $p_1 \dots p_n$)$[a_1 \dots a_n]$
   - $\varphi_1 \dots \varphi_n,\ \psi\ \triangleright\ \neg\varphi_1 \dots \neg\varphi_n \psi$ (subproof; $p_1 \dots p_n$)
2. Resolution rules
   - th_resolution,resolution
   - contraction, reordering
3. Introducing tautologies
   - $\triangleright\ \neg(\neg\neg\varphi),\ \varphi$ (not_not)
   - $\triangleright\ \neg(\varphi_1 \approx \varphi_2),\ \neg\varphi_1,\ \varphi_2$ (equiv_pos2)
   - $\triangleright\ \neg(\varphi_1 \wedge \dots \wedge \varphi_n),\ \varphi_k$ (and_pos)
4. Linear arithmetic
   - lia_generic,la_generic
   - $\triangleright\ t_1 \leq t_2 \vee t_2 \leq t_1$ (la_totality)
5. Quantifier handling
   - $\triangleright\ (\neg\forall \bar{x},\ \varphi) \vee (\varphi[\bar{t}])$ forall_inst
   - sko_ex
   - sko_forall
6. Simplification rules
   - and_simplify
   - bool_simplify
   - eq_simplify
   - sum_simplify

Challenges in validating Alethe proofs

- The ordering of clauses $l_1, \ldots, l_n$ is unspecified, which affects proof interpretation.
- Solvers may implicitly reorder equalities, introducing nondeterminism in proof structure.
- Proofs are often coarse-grained, lacking detailed justification for individual steps.
- Key information is sometimes omitted, especially for reasoning over linear integer arithmetic (`LIA`).

Carcara

- Carcara is an efficient and independent proof checker and elaborator for Alethe proofs.
- Carcara is written in Rust, a high performance language,
- implements elaboration procedures for a few important rules (ex: inferring pivots),
- it removes implicit transformations (ex: reordering clause).

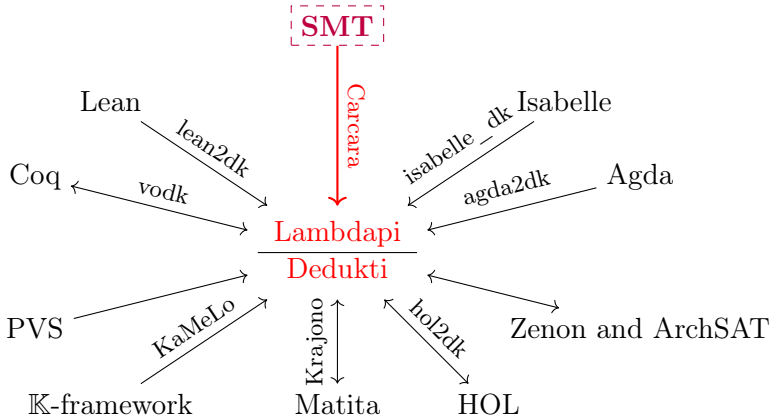
Elaborated proof with Carcara

Make pivot and resolution order explicit:

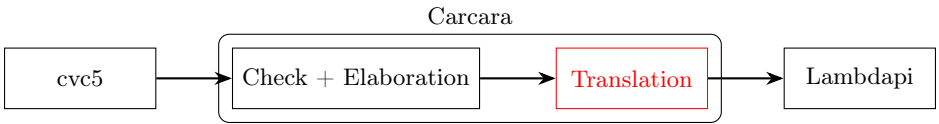
```
1 (assume a0 (p a))
2 (assume a1 (= a b))
3 (assume a2 (not (p b)))
4 (step t1 (cl (not (= (p a) (p b))) (not (p a)) (p b)) :rule equiv_pos2)
5 (step t2 (cl (= (p a) (p b))) :rule cong :premises (a1))
6 (step t3 (cl (p b)) :rule resolution :premises (t1 t2 a0)
7     :args ((= (p a) (p b)) false (p a) false))
8 (step t4 (cl) :rule resolution :premises (a2 t3)
9     :args ((p b) false))
```

Automated reconstruction of SMT proofs

Verifying SMT proofs in Lambdapi



Complete verification pipeline for Alethe proof



Lambdapi syntax

Based on the Edinburgh Logical Framework (LF):

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	$\Sigma ::= \langle\rangle \mid \Sigma, c : C \mid \Sigma, c := t : C \mid \Sigma, t \hookrightarrow v$

- Rewriting rules must be confluent and preserve typing (subject reduction).
- Confluence is not guaranteed and must be proved separately.
- Supports higher-order rewriting.
- Lacks meta-programming features such as type classes and records.
- Simple set of tactic i.e. no automatic solvers and programmable tactics

Lambdapi typing rules

Similar to Edinburgh Logical Framework (LF) but it uses $\equiv_{\beta\Sigma}$.

$$\frac{}{\vdash_\Sigma \langle\rangle}\ \text{(Empty)} \qquad \frac{\vdash_\Sigma \Gamma \qquad \Gamma \vdash_\Sigma A : s}{\vdash_\Sigma \Gamma, x : A}\ \text{(Decl)}\ x \notin \Gamma \qquad \frac{\vdash_\Sigma \Gamma \qquad \Gamma \vdash_\Sigma A : s}{\Gamma \vdash_\Sigma c : A}\ \text{(Const)}$$

$$\frac{\vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma \texttt{TYPE} : \texttt{KIND}}\ \text{(Sort)} \qquad \frac{\vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma x : A}\ \text{(Var)}\ x : A \in \Sigma$$

$$\frac{\Gamma, \vdash_\Sigma A : \texttt{TYPE} \qquad \Gamma, x : A \vdash_\Sigma B : s \qquad \Gamma, x : A \vdash_\Sigma t : B}{\Gamma \vdash_\Sigma \lambda x : A, t : \Pi x : A, B}\ \text{(Abs)}$$

$$\frac{\Gamma \vdash_\Sigma A : \texttt{TYPE} \qquad \Gamma, x : A \vdash_\Sigma B : s}{\Gamma \vdash_\Sigma \Pi x : A, B : s}\ \text{(Prod)} \qquad \frac{\Gamma \vdash t : \Pi x : A, B \qquad \Gamma \vdash u : A}{\Gamma \vdash_\Sigma t\ u : B[u \leftarrow x]}\ \text{(App)}$$

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

Lambdapi prelude encoding example

$$\texttt{Set} : \texttt{TYPE}$$
$$\texttt{El} : \texttt{Set} \to \texttt{TYPE}$$
$$\rightsquigarrow : \texttt{Set} \to \texttt{Set} \to \texttt{Set}$$
$$\texttt{El}\,(x \rightsquigarrow y) \hookrightarrow \texttt{El}\,x \to \texttt{El}\,y$$
$$= : \Pi[a : \texttt{Set}], \texttt{El}\,a \to \texttt{El}\,a \to \texttt{Prop}$$

$$\texttt{Prop} : \texttt{TYPE}$$
$$\texttt{Prf} : \texttt{Prop} \to \texttt{TYPE}$$
$$o : \texttt{Set}$$
$$\texttt{El}\,o \hookrightarrow \texttt{Prop}$$

$$\texttt{Clause} : \texttt{TYPE}$$
$$\blacksquare : \texttt{Clause}$$
$$\vee : \texttt{Prop} \to \texttt{Clause} \to \texttt{Clause}$$
$$\texttt{Prf}^{\bullet} : \texttt{Clause} \to \texttt{TYPE}$$

$$+\!+ : \texttt{Clause} \to \texttt{Clause} \to \texttt{Clause}$$
$$\blacksquare +\!+ x \hookrightarrow x$$
$$(x \vee y) +\!+ z \hookrightarrow x \vee (y +\!+ z)$$

Constructive operators, quantifiers and classical axioms

Operators and quantifiers:

$$\top, \bot : \texttt{Prop} \tag{1}$$

$$\wedge, \vee, \Rightarrow : \texttt{Prop} \to \texttt{Prop} \to \texttt{Prop} \qquad \text{(written infix)} \tag{2}$$

$$\texttt{Prf}\ (a \Rightarrow b) \hookrightarrow \texttt{Prf}\ a \to \texttt{Prf}\ b \tag{3}$$

$$\neg\ a \coloneqq a \Rightarrow \bot \tag{4}$$

$$\forall : \Pi [a : \texttt{Set}], (\texttt{El}\ a \to \texttt{Prop}) \to \texttt{Prop} \tag{5}$$

$$\texttt{Prf}(\forall\ p) \hookrightarrow \Pi\, x, \texttt{Prf}(p\ x) \tag{6}$$

$$\exists : \Pi\, [a : \texttt{Set}], (\texttt{El}\ a \to \texttt{Prop}) \to \texttt{Prop} \tag{7}$$

Classical axioms:

$$\texttt{em} : \Pi\, p, \texttt{Prf}(p \vee \neg p) \tag{8}$$

$$\texttt{prop\_eq} : \Pi\, p\, q, \texttt{Prf}(p \Leftrightarrow q) \to \texttt{Prf}(p = q) \tag{9}$$

Encoding Alethe rules in Lambdapi

Alethe rule	**Lambdapi encoding**
R = equiv_pos2	
$i.\ \Gamma\ \rhd\ \neg(a \approx b), \neg a, b \quad (\mathtt{R})[]$	$i : \mathtt{Prf}^{\bullet}(\neg(a = b) \vee \neg a \vee b \vee \blacksquare)$
R = resolution	
$i_1.\ \rhd\ l_1^1, \ldots, l_{k^1}^1 \quad (\ldots)$ $i_n.\ \rhd\ l_1^n, \ldots, l_{k^n}^n \quad (\ldots)$ $\vdots$ $j.\ \rhd\ l_{s_1}^{r_1}, \ldots, l_{s_m}^{r_m} \quad (\mathtt{R}\ i_1 \ldots i_n)[]$	$\mathtt{resolution}\ (ps\ qs : \mathtt{Clause})\ (i\ j : \mathbb{N})$ $(hps : \mathtt{Prf}^{\bullet} ps)$ $(hqs : \mathtt{Prf}^{\bullet} qs)$ $(hi : \mathtt{Prf}(i < \mathtt{size}\ ps))$ $(hj : \mathtt{Prf}(j < \mathtt{size}\ qs))$ $(hij : \mathtt{Prf}((\mathtt{nth}\ ps\ i) = \neg(\mathtt{nth}\ qs\ j)))$ $: \mathtt{Prf}^{\bullet}(\mathtt{remove}\ ps\ i +\!\!+ \mathtt{remove}\ qs\ j)$

Translation of the input problem

```
(declare-sort U 0)
(declare-fun a () U)
(declare-fun b () U)
(declare-fun p (U) Bool)
(assert (p a))
(assert (= a b))
(assert (not (p b)))
(get-proof)
```

↯

```
symbol U : Set;
symbol a : El U;
symbol b : El U;
symbol p : El (U ⇝ o);
symbol a0 : Prf• (p_5 ∨ ■);
symbol a1 : Prf• ((a = b) ∨ ■);
symbol a2 : Prf• ((¬ ((p b))) ∨ ■);
symbol p_2 ≔ (p b);
symbol p4 ≔ ((p a) = (p b));
```

Reminder: proof of the guiding example

```
1 (assume a0 (p a))
2 (assume a1 (= a b))
3 (assume a2 (not (p b)))
4 (step t1 (cl (not (= (p a) (p b))) (not (p a)) (p b))
5       :rule equiv_pos2)
6 (step t2 (cl (= (p a) (p b))) :rule cong :premises (a1))
7 (step t3 (cl (p b)) :rule resolution :premises (t1 t2 a0))
8 (step t4 (cl) :rule resolution :premises (a2 t3))
```

Automated translation of the proof into Lambdapi via our tool

```
1  opaque symbol t0 : Prf• ((¬ ((p_5 = p_2))) ∨ (¬ (p_5)) ∨ p_2 ∨ ■) ≔
2  begin apply equiv_pos2; end;
3
4  opaque symbol t1 : Prf• (p_4 ∨ ■) ≔
5  begin apply ∨i1; apply feq (p) (Prf•l a1) end;
6
7  opaque symbol t2 : Prf• (p_2 ∨ ■) ≔
8  begin
9  have t0_t1 : Prf• ((¬ (p_5))  ∨ p_2 ∨ ■) {
10   apply resolution 0 0 t0 t1 ⊤i ⊤i (eq_refl _)
11 };
12 have t0_t1_a0 : Prf• (p_2 ∨ ■) {
13     apply resolution 0 0 t0_t1 a0 ⊤i ⊤i (eq_refl _)
14 }; refine t0_t1_a0;
15 end;
16
17 opaque symbol t3 : Prf• ■ ≔
18 begin
19   apply resolution 0 0 a2 t2 ⊤i ⊤i (eq_refl _);
20 end;
```

Focus: reconstructing arithmetic proofs

An example in LIA logic

```
1 (set-logic LIA)
2 (declare-const x Int)
3 (declare-const y Int)
4 (assert (= x 2))
5 (assert (= 0 y))
6 (assert (or (< (+ x y) 1) (< 3 x)))
7 (check-sat)
8 (get-proof)
```



```
1  (assume a0 (or (< (+ x y) 1) (< 3 x)))
2  (assume a2 (= 0 y))
3  (assume a1 (= x 2))
4  (step t1 (cl (< (+ x y) 1) (< 3 x)) :rule or :premises (a0))
5  (step t2 (cl (not (< 3 x)) (not (= x 2))) :rule la_generic :args (1/1 -1/1))
6  (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)))
9        :rule la_generic :args (1/1 1/1 -1/1))
10 (step t6 (cl) :rule resolution :premises (t5 t4 a1 a2))
```

Linear arithmetic rules in Alethe supported in our encoding.

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 $\bowtie \in \{<, >, \geq, \leq, \approx\}$
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
arith-int-eq-elim	$(t \approx s) \rightarrow t \geq s \wedge t \leq s$
arith-leq-norm	$t \leq s \rightarrow \neg(t \geq s + 1)$
arith-geq-norm1	$t \geq s \rightarrow (t - s) \geq 0$
arith-geq-norm2	$t \geq s \rightarrow -t \leq -s$
arith-geq-tighten	$\neg(t \geq s) \rightarrow s \geq t + 1$
arith-poly-norm	polynomial normalization
evaluate	evaluate constant terms

The (refactored) `la_generic` description

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

1. If φ_i is of the form $s_1 \geq s_2$ or $\neg(s_1 < s_2)$, then let $\psi_i = s_2 > s_1$. If φ_i is of the form $s_1 > s_2$ or $\neg(s_1 \leq s_2)$, then let $\psi_i = s_2 \geq s_1$. If φ_i is of the form $s_1 < s_2$ or $\neg(s_1 \geq s_2)$, then let $\psi_i = s_1 \geq s_2$. If φ_i is of the form $s_1 \leq s_2$ or $\neg(s_1 > s_2)$, then let $\psi_i = s_1 > s_2$. If φ_i is of the form $\neg(s_1 \approx s_2)$, then let $\psi_i = s_1 \approx s_2$. This step produces a positive literal that is equivalent to $\neg\varphi_i$ and that only contains the operators $>$, $\geq$, and $\approx$.
2. Replace $\psi_i = \sum_{j=0}^{k_i} c_j^i \times t_j^i + d_1^i \bowtie \sum_{j=k_i+1}^{m_i} c_j^i \times t_j^i + d_2^i$ by the literal $\left(\sum_{j=0}^{k_i} c_j^i \times t_j^i\right) - \left(\sum_{j=k_i+1}^{m_i} c_j^i \times t_j^i\right) \bowtie d_2^i - d_1^i$.
3. Now ψ_i has the form $s_1^i \bowtie d^i$. If all variables in s_1^i are integer-sorted then replace $s_1^i > d^i$ by $s_1^i \geq \lfloor d^i \rfloor + 1$, respectively, replace $s_1^i \geq d^i$ by $s_1^i \geq \lfloor d^i \rfloor + 1$ if d is not an integer.
4. If all variables of ψ_i are integer-sorted and the coefficients $a_1 \ldots a_n$ are in $\mathbb{Q}$, then $a_i := a_i \times lcd(a_1 \ldots a_n)$ where lcd is the least common denominator of $\{a_1, \ldots, a_n\}$.
5. If $\bowtie$ is $\approx$, then replace ψ_i by $\sum_{j=0}^{m_i} a_i \times c_j^i \times t_j^i = a_i \times d^i$, otherwise replace ψ_i by $\sum_{j=0}^{m_i} |a_i| \times c_j^i \times t_j^i \bowtie |a_i| \times d^i$.
6. Finally, the sum of the resulting literals is trivially contradictory,

$$\sum_{i=1}^{n} \sum_{j=1}^{m_i} c_j^i * t_j^i \bowtie \sum_{i=1}^{n} d^i$$

An example of `la_generic`

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

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

The algorithm then performs the following steps:

$-f \geq 0,\ 4 \times f > 0$ (Steps 1 and 2)

$-f \geq 0,\ 4 \times f \geq 1$ (Step 3)

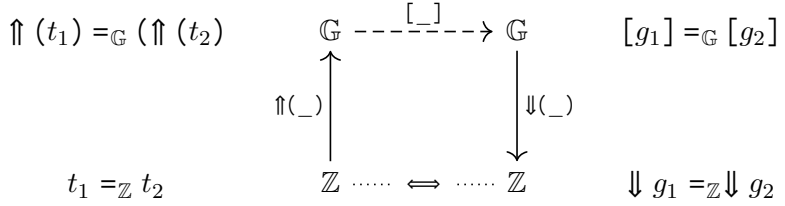
Replace arguments $[\frac{1}{1}, \frac{1}{4}]$ by $[4, 1]$ due to clearing denominators (Step 4)

$|4| \times (-f) \geq |4| \times 0,\ |1| \times 4 \times f \geq |1| \times 1)$ (Step 5)

$-4 \times f + 4 \times f \geq 1$ (Step 6)

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

A Scheme for proof by reflection - Inner version



with:

- $\mathbb{G}$ an Algebra to represent integers
- $\Uparrow : \mathbb{Z} \to \mathbb{G}$ the reify function
- $\Downarrow : \mathbb{G} \to \mathbb{Z}$ the denotation function
- $[\_] : \mathbb{G} \to \mathbb{G}$ the normalization function
- $_{\mathbb{G}} : \mathbb{G} \to \mathbb{G} \to \mathbb{B}$ a (decidable) equivalence relation

Two methods for normalising terms

1. Inner normalisation of terms performed in the Lambdapi *kernel* using `associative commutative`
   Based on: *Encoding Type Universes Without Using Matching Modulo Associativity and Commutativity. Frédéric Blanqui. (FSCD 2022).*

2. Outer normalisation function with user-defined rewrite rules and symbolic execution.

The $\mathbb{G}$ Algebra to represent integers

$\mathbb{G} : \texttt{TYPE}$
$\mid \oplus : \mathbb{G} \to \mathbb{G} \to \mathbb{G}$
$\mid \texttt{var} : \mathbb{Z} \to \mathbb{Z} \to \mathbb{G}$
$\mid \texttt{mul} : \mathbb{Z} \to \mathbb{G} \to \mathbb{G}$
$\mid \texttt{opp} : \mathbb{G} \to \mathbb{G}$
$\mid \texttt{cst} : \mathbb{Z} \to \mathbb{G}$
$\texttt{grp} : \texttt{Set}$
$\texttt{El grp} \hookrightarrow \mathbb{G}$

$\Uparrow : \mathbb{Z} \to \mathbb{G}$
$\Uparrow \texttt{Z0} \hookrightarrow (\textsc{cst}\ \texttt{Z0})$
$\Uparrow \texttt{ZPos}\, c \hookrightarrow (\textsc{cst}\ (\texttt{ZPos}\, c))$
$\Uparrow \texttt{ZNeg}\, c \hookrightarrow (\textsc{cst}\ (\texttt{ZNeg}\, c))$
$\Uparrow (x + y) \hookrightarrow (\Uparrow x) \oplus (\Uparrow y)$
$\Uparrow (\sim x) \hookrightarrow \textsc{opp}\ \Uparrow x$
$\Uparrow ((\texttt{ZPos}\, c) * x) \hookrightarrow \textsc{mul}\ (\texttt{ZPos}\, c)\ (\Uparrow x)$
$\Uparrow ((\texttt{ZNeg}\, c) * x) \hookrightarrow \textsc{mul}\ (\texttt{ZNeg}\, c)\ (\Uparrow x)$
$\Uparrow (x * (\texttt{ZPos}\, c)) \hookrightarrow \textsc{mul}\ (\texttt{ZPos}\, c)\ (\Uparrow x)$
$\Uparrow (x * (\texttt{ZNeg}\, c)) \hookrightarrow \textsc{mul}\ (\texttt{ZNeg}\, c)\ (\Uparrow x)$
$\Uparrow (x * \texttt{Z0}) \hookrightarrow (\textsc{cst}\ 0)$
$\Uparrow (\texttt{Z0} * x) \hookrightarrow (\textsc{cst}\ 0)$
$\Uparrow x \hookrightarrow (\textsc{var}\ 1\ x)$

$\Downarrow : \mathbb{G} \to \mathbb{Z}$
$\Downarrow (\textsc{cst}\ c) \hookrightarrow c$
$\Downarrow \textsc{opp}\ x \hookrightarrow \sim (\Downarrow x)$
$\Downarrow \textsc{mul}\ c\ x \hookrightarrow c \times (\Downarrow x)$
$\Downarrow x \oplus y \hookrightarrow (\Downarrow x) + (\Downarrow y)$
$\Downarrow (\textsc{var}\ c\ x) \hookrightarrow c \times x$

with $\oplus$ declared `associative commutative`, and $\Uparrow$ `sequential`.

Normalisation with associative commutative modifier

Definition

The $\leq$ builtin total order on $\mathbb{G}$-terms is defined as follows: Terms are ordered such that $\texttt{cst}(c_1) \leq \texttt{cst}(c_2) < (\textsc{var}\ c\ x)$ for any constants $c_1 \leq c_2$ and any variable term $(\textsc{var}\ c\ x)$. For variable terms, $(\textsc{var}\ c\ x) \leq (\textsc{var}\ d\ y)$ if either $x < y$, or $x = y$ and $c \leq d$.

Example

Consider the term below not in normal form:

$$(\textsc{var}\ c_1\ x) \oplus (\textsc{cst}\ k_1) \oplus (\textsc{var}\ c_2\ y) \oplus (\textsc{cst}\ k_m) \oplus (\textsc{var}\ c_3\ x) \oplus (\textsc{var}\ c_4\ y)$$

It will be then normalise into:

$$(\textsc{cst}\ k_1) \oplus (\textsc{cst}\ k_2) \oplus (\textsc{var}\ c_1\ x) \oplus (\textsc{var}\ c_3\ x) \oplus (\textsc{var}\ c_2\ y) \oplus (\textsc{var}\ c_4\ y)$$

Theory rules for $\mathbb{G}$

Group theory axioms

$$(\textsc{var}\ c_1\ x) \oplus (\textsc{var}\ c_2\ x) \hookrightarrow (\textsc{var}\ (c_1 + c_2)\ x) \tag{10}$$

$$(\textsc{var}\ c_1\ x) \oplus ((\textsc{var}\ c_2\ x) \oplus y) \hookrightarrow (\textsc{var}\ (c_1 + c_2)\ x) \oplus y \tag{11}$$

$$(\textsc{cst}\ c_1) \oplus (\textsc{cst}\ c_2) \hookrightarrow (\textsc{cst}\ (c_1 + c_2)) \tag{12}$$

$$(\textsc{cst}\ c_1) \oplus ((\textsc{cst}\ c_2) \oplus y) \hookrightarrow (\textsc{cst}\ (c_1 + c_2)) \oplus y \tag{13}$$

$$(\textsc{cst}\ 0) \oplus x \hookrightarrow x \tag{14}$$

$$x \oplus (\textsc{cst}\ 0) \hookrightarrow x \tag{15}$$

$$\textsc{opp}\ (\textsc{var}\ c\ x) \hookrightarrow (\textsc{var}\ (-c)\ x) \tag{16}$$

$$\textsc{opp}\ (\textsc{cst}\ c) \hookrightarrow (\textsc{cst}\ (-c)) \tag{17}$$

$$\textsc{opp}\ (\textsc{opp}\ x) \hookrightarrow x \tag{18}$$

$$\textsc{opp}\ (x \oplus y) \hookrightarrow (\textsc{opp}\ x) \oplus (\textsc{opp}\ y) \tag{19}$$

$$\textsc{opp}\ (\textsc{mul}\ k\ x) \hookrightarrow \textsc{mul}\ (-k)\ x \tag{20}$$

$$\textsc{mul}\ k\ (\textsc{var}\ c\ x) \hookrightarrow (\textsc{var}\ (k * c)\ x) \tag{21}$$

$$\textsc{mul}\ k\ (\textsc{opp}\ x) \hookrightarrow \textsc{mul}\ (-k)\ x \tag{22}$$

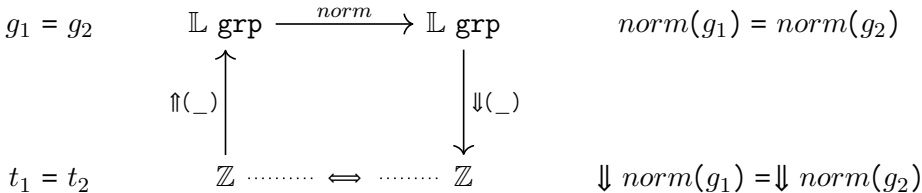
$$\textsc{mul}\ k\ (x \oplus y) \hookrightarrow (\textsc{mul}\ k\ x) \oplus (\textsc{mul}\ k\ y) \tag{23}$$

$$\textsc{mul}\ k\ (\textsc{cst}\ c) \hookrightarrow (\textsc{cst}\ (k * c)) \tag{24}$$

$$\textsc{mul}\ c_1\ (\textsc{mul}\ c_2\ x) \hookrightarrow \textsc{mul}\ (c_1 * c_2)\ x \tag{25}$$

```
opaque symbol t2: π ((¬ (3 < x) ∨ ¬ (x = 2)) ∨ ■) {
    apply ∨ᵢ1;
    rewrite Zinv_lt_eq;
    rewrite Z_diff_gt_Z0_eq (- 3) (- x);
    rewrite Z_diff_eq_Z0_eq (x) 2;
    rewrite Zgt_le_succ_r_eq ((- 3) - (- x)) 0;
    rewrite Zmult_ge_compat_eq 1 ((- 3) - (- x)) ((0 + 1));
    rewrite Zmult_eq_compat_eq (- 1) (x - 2) 0;
    rewrite imp_eq_or; apply ⇒ᵢ; assume H0; apply ¬ᵢ; assume H1;
    set H0l' ≔ (1 * ((- 3) - (- x))); set H0r' ≔ (1 * (0 + 1));
    set H1l' ≔ ((- 1) * (x - 2)); set H1r' ≔ ((- 1) * 0);
    have H1': π (H1l' ≥ H1r') { refine Z_eq_implies_ge H1 }; remove H1;
    have contra : π ((⇓ (⇑ (H0l' + H1l'))) ≥ (⇓ (⇑ (H0r' + H1r')))){
        rewrite reify_correct; rewrite reify_correct;
        apply (Zsum_geq_s H0l' H0r' H1l' H1r' H0 H1');
    };
    apply contra; apply ⊤ᵢ;
};
```

A Scheme for Proof by Reflection - Outer version



with:

- $\mathbb{G}$ an Algebra to represent integers
- $\Uparrow$: $\mathbb{Z} \to \mathbb{G} \times (\mathbb{L}$ `int`$)$ the reify function
- $\Downarrow$: $\mathbb{G} \times (\mathbb{L}$ `int`$) \to \mathbb{Z}$ the denotation function
- $norm$: $\mathbb{G} \to \mathbb{G}$ the normalization function

Normalization: an overview

We redefine the type $\mathbb{G}$ and we reify into a $\mathbb{L}$ `int`:

$$\begin{array}{l} \mathbb{G} : \texttt{TYPE} \\ \mid \texttt{var} : \mathbb{N} \to \mathbb{Z} \to \mathbb{G} \\ \mid \texttt{cst} : \mathbb{Z} \to \mathbb{G} \\ \mid \texttt{mul} : \mathbb{Z} \to \mathbb{G} \to \mathbb{G} \end{array} \qquad \begin{array}{l} \texttt{grp} : \texttt{Set} \\ \texttt{El grp} \hookrightarrow \mathbb{G} \end{array}$$

We then define the normalization function as follows:

$$norm\,(x : \mathbb{L}\ \texttt{grp}) \coloneqq \underbrace{remove0}_{\text{cancel and removes neutral elements}}\Big(\underbrace{mergesort}_{\text{sorts a list of grp}}\ x\Big) \tag{2}$$

Current evaluation of the two methods

- The inner approach is easier to implement but requires trusting the Lambdapi kernel.

- The outer approach is still slow on large examples and poses major challenges for further optimisation.

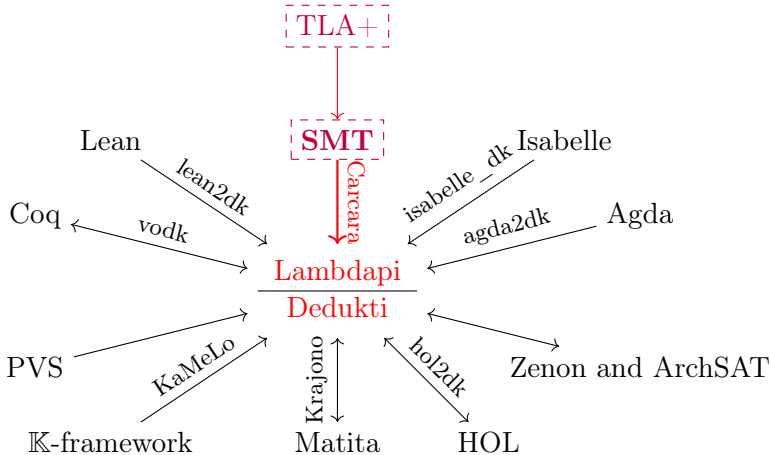
Evaluation and practical applications

Evaluation

Logic	Bench	Samples	Check
LIA	tptp	36	28
	Ultimate	153	50
QFLIA	SMPT	1568	804
	rings	294	7
	CAV2009	85	19
UFLIA	sledgeh	1521	713
	tokeneer	1732	1482
UF	sledgehammer	1403	994
QF_UF	eq_diamond	100	74
	2018-Goel-hwbench	229	160
	20170829-Rodin	20	16
UFNIA (TLA^+)	allocator	38	38
	EWD840	19	11

Table: Benchmark results.

Verifying SMT Proofs in Lambdapi



TLA$^+$ at a glance

- Specification language to design and verify reactive systems
- Systems are described as state machines

$$
\begin{array}{l}
VARIABLE\ x \\
CONSTANT\ N \\
ASSUME\ N \in Nat
\end{array}
$$

$$Init \triangleq \quad \land\ x = 0$$

$$Next \triangleq \quad \land\ x < N \\ \qquad\quad \land\ x' = x + 1$$

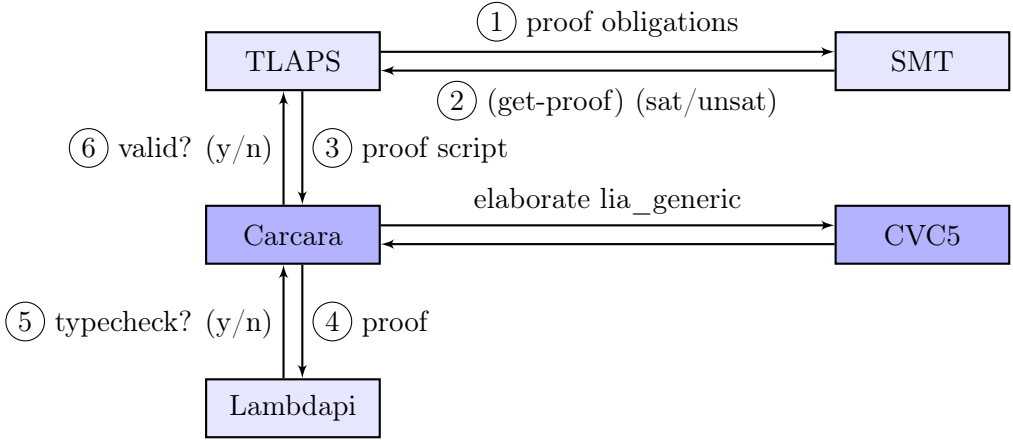
$$Spec \triangleq Init \land \Box[Next]_{\langle x \rangle}$$

TLAPS proof example

– – – – – – – – – – – – – – – – MODULE Cantor1 – – – – – – – – – – – – – – – –

THEOREM cantor ==
$\forall\ S\ :$
$\forall\ f \in [S \to \text{SUBSET}\ S]\ :$
$\exists\ A \in \text{SUBSET}\ S\ :$
$\forall\ x \in S\ :$
$f\ [x] \#\ A$
PROOF
<1> 1. TAKE S
<1> 2. TAKE $f \in [S \to \text{SUBSET}\ S]$
<1> 3. DEFINE $T\ ==\ \{\ z \in S\ :\ z \notin f[z]\ \}$
<1> 4. WITNESS $T \in \text{SUBSET}\ S$
<1> 5. TAKE $x \in S$
<1> 6. QED BY $x \in T \vee x \notin T$

TLA pipeline



Future works

1. Add support for bitvectors (**BV**).
2. Add support for rationals and reals (**LRA**).
3. Rocq export with user-defined rewrite rules.