Reconstruction of TLAPS proofs solved by SMT in Lambdapi

Alessio Coltellacci

Univ. Lorraine, CNRS, Inria, Loria

ICSPA

Outline

TLA$^+$ at a glance

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

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

$$\mathit{Init} \triangleq \quad \land\ x = 0$$

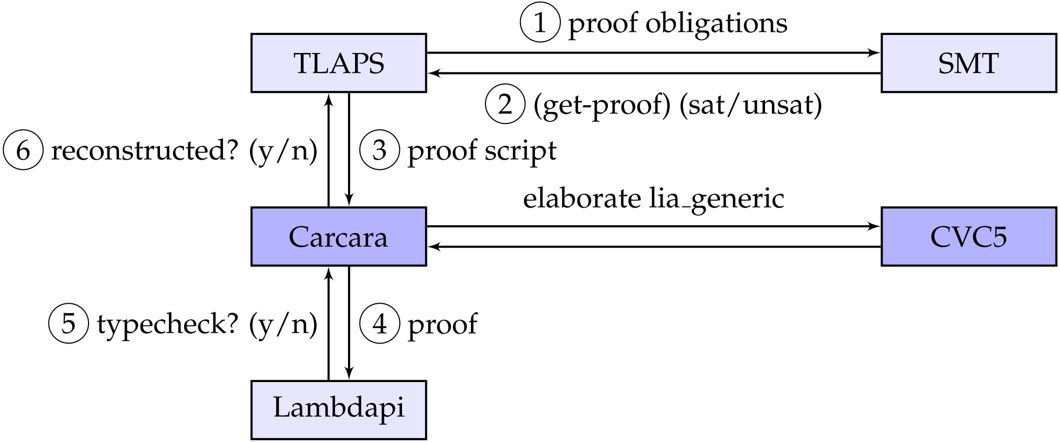
$$\mathit{Next} \triangleq \quad \begin{array}{l} \land\ x < N \\ \land\ x' = x + 1 \end{array}$$

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

TLAPS proof example

```
---------------- MODULE Cantor1 ----------------
THEOREM cantor ==
    ∀ S :
        ∀f ∈ [S → SUBSET S] :
            ∃A ∈ SUBSET S :
                ∀x ∈ S :
                    f [x] # A
PROOF
<1> 1.  TAKE S
<1> 2.  TAKE f ∈ [S → SUBSET S]
<1> 3.  DEFINE T == { z ∈ S : z ∉ f[z] }
<1> 4.  WITNESS T ∈ SUBSET S
<1> 5.  TAKE x ∈ S
<1> 6.  QED BY x ∈ T ∨ x ∉ T
```

Proposed solution



Simple example

```
(set-logic QF_UF)
(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)
```

SMT proof

```
(assume a0 (p a))
(assume a1 (= a b))
(assume a2 (not (p b)))
(step t1 (cl (not (= (p a) (p b))) (not (p a)) (p b)) :rule equiv_pos2)
(step t2 (cl (= (p a) (p b))) :rule cong :premises (a1))
(step t3 (cl (p b)) :rule resolution :premises (t1 t2 a0))
(step t4 (cl) :rule resolution :premises (a2 t3))
```

Alethe rules example

Definition (Alethe step notation)

A proof in the Alethe language is an indexed list of steps.

$$j. \quad \Delta \quad \vdash \varphi \quad (R; p_1 \dots p_n)[a_1, \dots, a_n]$$

With $i \in \mathbb{I}$ where $\mathbb{I}$ is a countable infinite set of valid indices,

a formula φ,

a rule name $\mathcal{R}$ from set of possible rules,

a possible empty sets $\{p_1 \dots p_n\} \subseteq \mathbb{I}$,

a possible empty list of arguments $[a_1 \dots a_n]$ where $a_i = (x_i, t_i)$
with x_i a variable and t_i a term,

and a context of the step Δ.

Overview of rules

1. Special rules
   * $\vdash\ \varphi$ `asssume`
   * $\vdash\ \varphi$ $(\texttt{hole}; p_1 \dots p_n)[a_1 \dots a_n]$
   * $\varphi_1 \dots \varphi_n, \psi\ i.\ \vdash\ \neg\varphi_1 \dots \neg\varphi_n \psi$ $(\texttt{subproof};\ p_1 \dots p_n)$
2. Resolution rules
   * `th_resolution`, `resolution`
   * `contraction`
3. Introducing tautologies
   * $\vdash\ \neg(\neg\neg\varphi) \vee \varphi$ (`not_not`)
   * $\vdash\ \neg(\varphi_1 \approx \varphi_2) \vee \neg\varphi_1 \vee \varphi_2$ (`equiv_pos2`)
   * $\vdash\ \neg(\varphi_1 \wedge \dots \wedge \varphi_n) \vee \varphi_k$ (`and_pos`)
4. Linear arithmetic
   * `lia_generic`, `la_generic`
   * $\vdash\ t_1 \leq t_2 \vee t_2 \leq t_1$ (`la_totality`)
5. Quantifier handling
   * $j.\Delta, x_i \mapsto y_i \vdash\ \varphi \approx \varphi'$
     $i. \vdash \forall x_1 \dots x_n, \varphi \approx \forall y_1 \dots y_n, \varphi'$ (`bind`)
   * `forall_inst`
6. Skolemization
   * `sko_ex`
   * `sko_forall`
7. Clausification rules
   * **let**
   * `distinct_elim`
8. Simplification rules
   * `and_simplify`
   * `bool_simplify`
   * `eq_simplify`
   * `sum_simplify`

Alethe proof as derivation tree

$$
t_i \; \dfrac{t_j \; \dfrac{\vdots}{\Delta'' \vdash p_1}\,\text{Rule}(\dots) \qquad \dots \qquad t_k \; \dfrac{\vdots}{\Delta' \vdash p_n}\,\text{Rule}(\dots)}{\Delta \cup \{p_1, \dots, p_n\} \vdash c_1, \dots c_n}\,\text{Rule}(a_1, \dots, a_n)
$$

SMT proof

```
(assume a0 (p a))
(assume a1 (= a b))
(assume a2 (not (p b)))
(step t1 (cl (not (= (p a) (p b))) (not (p a)) (p b)) :rule equiv_pos2)
(step t2 (cl (= (p a) (p b))) :rule cong :premises (a1))
(step t3 (cl (p b)) :rule resolution :premises (t1 t2 a0))
(step t4 (cl) :rule resolution :premises (a2 t3))
```

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: infering pivots),
- it remove implicit transformations (ex: reordering clause).

Elaborated proof with Carcara

Make pivot and resolution order explicit

```
(assume a0 (p a))
(assume a1 (= a b))
(assume a2 (not (p b)))
(step t1 (cl (not (= (p a) (p b))) (not (p a)) (p b)) :rule equiv_pos2)
(step t2 (cl (= (p a) (p b))) :rule cong :premises (a1))
(step t3 (cl (p b)) :rule resolution :premises (t1 t2 a0)
    :args ((= (p a) (p b)) false (p a) false))
(step t4 (cl) :rule resolution :premises (a2 t3)
    :args ((p b) false))
```

Corresponding proof tree

$$
\dfrac{
  \dfrac{}{\Delta \vdash p(a)}\, a_0
  \qquad
  \dfrac{
    \dfrac{}{\Delta \vdash \neg\,(p(a) = p(b)), \neg p(a), p(b)}\, t_1 \ \text{equiv\_pos2}
    \qquad
    \dfrac{}{\Delta \vdash p(a) = p(b)}\, t_2 \ \text{cong}(a_1)
  }{\Delta \vdash \neg p(a), p(b)}\, t_3' \ \text{Resolution}(t_1, t_2)
}{\Delta \vdash p(b)}\, t_3 \ \text{Resolution}(a_0, t3')
\qquad
\dfrac{}{\Delta \vdash \neg\, p(b)}\, a_2
$$

$$
t_4 \ \dfrac{\Delta \vdash p(b) \qquad \Delta \vdash \neg\, p(b)}{\Delta \vdash \bot}\ \text{Resolution}(a_2, t_3)
$$

Translate prelude

```
(declare-sort U 0)
(declare-fun a () U)
(declare-fun b () U)
(declare-fun p (U) Bool)
```

$\rightsquigarrow$

```
symbol U : TYPE;
rule U ↪ τ o;

symbol a : U;
symbol b : U;
symbol p : U → Prop;
```

Translate assert/assume

```
(assert (p a))
(assert (= a b))
(assert (not (p b)))
```

```
(assume a0 (p a))
(assume a1 (= a b))
(assume a2 (not (p b)))
```

constant symbol a0 : $\dot{\pi}$ (p a $\dot{\vee}$ $\square$);
constant symbol a1 :
    $\dot{\pi}$ (a $\overset{c}{\iff}$ b $\dot{\vee}$ $\square$);
constant symbol a2 :
    $\dot{\pi}$ ($\overset{c}{\neg}$ (p b) $\dot{\vee}$ $\square$);

Translation of step t1 and t2

```
(step t1
    (cl (not (= (p a) (p b)))
        (not (p a))
        (p b))
    :rule equiv_pos2)
(step t2 (cl (= (p a) (p b)))
    :rule cong :premises (a1))
...
```

⇝

```
opaque symbol pb :
begin
have t1: π̇ (
    ¬ᶜ ((p a) ⟺ᶜ (p b))
    ∨̇ (p a)
    ∨̇ (p b)
    ∨̇ □)
{
    apply equiv_pos2;
};
have t2: π̇ (
    p a ⟺ᶜ p b
    ∨̇ □)
{
    apply ;
    apply cong p ( a1);
};
```

Translation of step t3

```
(step t3 (cl (p b))
    :rule resolution
    :premises (t1 t2 a0))
```

$\rightsquigarrow$

```
...
have t3 : π̇ ((p b) ∨̇ □) {
    have t1_t2 : π̇ (
        (¬ᶜ ((p a)))
        ∨̇ (p b) ∨̇ □)
    {
        apply resolution t1 t2;
    };
    have t1_t2_a0 : π̇ ((p b) ∨̇ □)
    {
        apply resolution t1_t2 a0;
    };
    apply t1_t2_a0;
};
```

Translation of step t4

```
(step t4 (cl)
    :rule resolution
    :premises (a2 t3))
```

$\rightsquigarrow$

```
...
have t4 : π̇ □ {
    have a2_t3 : π̇ □  {
        apply resolution a2 t3;
    };
    apply a2_t3;
};
apply t4;
proofterm;
end;
```

Corresponding proof term

```
resolution_r (p b) _ _
    a2
    (resolution_r (p a) _ _
        (resolution_r (p a ⟺^c p b) _ _
            equiv_pos2
            (∨_i1^c (cong p (π a1)))
        )
        a0
    )
```

Supported rules overview

1. Special rules ✓
   * $\vdash \varphi$ `asssume`
   * $\vdash \varphi$ $(\texttt{hole}; p_1 \dots p_n)[a_1 \dots a_n]$
   * $\varphi_1 \dots \varphi_n, \psi\ i.\ \vdash \neg\varphi_1 \dots \neg\varphi_n \psi$
     $(\texttt{subproof};\ p_1 \dots p_n)$
2. Resolution rules ✓
   * `th_resolution,resolution`
   * `contraction`
3. Introducing tautologies ✓
   * $\vdash \neg(\neg\neg\varphi) \mathbin{\dot\vee} \varphi$ (`not_not`)
   * $\vdash \neg(\varphi_1 \approx \varphi_2) \mathbin{\dot\vee} \neg\varphi_1 \mathbin{\dot\vee} \varphi_2$ (`equiv_pos2`)
   * $\vdash \neg(\varphi_1 \wedge \cdots \wedge \varphi_n) \mathbin{\dot\vee} \varphi_k$ (`and_pos`)
4. Linear arithmetic ×
   * `lia_generic,la_generic`
   * $\vdash t_1 \leq t_2 \vee t_2 \leq t_1$ (`la_totality`)
5. Quantifier handling (WIP)
   * $j.\Delta, x_i \mapsto y_i \vdash \varphi \approx \varphi'$
     $i. \vdash \forall x_1 \dots x_n, \varphi \approx \forall y_1 \dots y_n, \varphi'$ (`bind`)
   * `forall_inst`
6. Skolemization (WIP)
   * `sko_ex`
   * `sko_forall`
7. Clausification rules (WIP)
   * **let**
   * `distinct_elim`
8. Simplification rules ×
   * `and_simplify`
   * `bool_simplify`
   * `eq_simplify`
   * `sum_simplify`

Contexts in Alethe

Definition (Proof context)

We denote Δ_c as the Alethe proof context. It is use to reason about bound variable and previous **step**.

$$\text{If} \quad j. \quad x_1 \mapsto y_1, \ldots, x_n \mapsto y_n \quad \vdash \ \varphi\, y_1 \ldots y_n \quad (R, p_1 \ldots p_n)[x_1, \ldots, x_n] \quad \text{proved}$$

Then $j(\varphi\, y_1 \ldots y_n;\ R;\ p_1 \ldots p_n;\ a_1 \ldots a\_n) \in \Delta$
And $x_1 \mapsto y_1, \ldots, x_n \mapsto y_n \in \Delta$

Definition (Definition context)

We denote Δ_{def} as the Alethe definition context. It is use to store user declarations `declare-sort` and `declare-fun`.

Definition (Alethe context)

We set $\Delta = \Delta_c \cup \Delta_{def}$

Alethe encoding in Lambdapi

We denote $\Gamma_{\mathcal{A}}$ as the Lambdapi context with Alethe definitions.

Encoding of classical logic in $\Gamma_{\mathcal{A}}$ [1]

- The set of terms `Set : TYPE`
  function symbol `Set →... →Set`,
- the set of propositions `Prop : TYPE`
  predicate symbol `Set →... →Prop`,
- and the classical connectives $\forall^c \mid \exists^c \mid \wedge^c \mid \neg^c \mid \vee^c \mid \Rightarrow^c \mid \iff^c$.
- $\pi^c := \neg\neg p$ the definition of classical proofs of proposition p,
- the axioms of classical natural deduction system $\mathcal{NK}$ and some lemmas.
- quantification on propositions/impredicativity ($e.g. \forall p, p \Rightarrow p$) :

```
symbol o : Set;
rule τo ↪Prop;
```

[1] Classical logic definitions is based on Lambdapi Stdlib.

Alethe encoding in Lambdapi

We encode Alethe rule $\mathcal{R}$ as a corresponding symbol.

Clause

Alethe treats clause as a set. To solve canonical representation of clause, we define clause as list in a Church encoding style:

```
constant symbol Clause : TYPE;
symbol □ : Clause; // Nil
injective symbol ∨̇ : Prop → Clause →  Clause; // Cons x l
sequential symbol ++ : Clause → Clause → Clause;
rule □ ++  $m ↪ $m
with ( $x ∨̇  $l) ++  $m ↪ $x ∨̇ ( $l ++  $m);

symbol Clause_ind:  Π P: (Clause → Prop), Π l,
π (P □)  → (Π x: Prop, Π l: Clause, π (P l)  →
π (P (x ∨̇ l))) → π (P l);
```

Alethe rule encoding example

The rule *not_implies*1 in Alethe set of rules

$$\begin{array}{lll} i. & \vdash \neg(\varphi_1 \to \varphi_2) & (\ldots) \\ j. & \vdash \varphi_1 & (\mathit{not\_implies}1; i) \end{array}$$

is translated into:

```
opaque symbol not_implies_1 [φ1 φ2] : π(¬ᶜ(φ1 ⇒ᶜ φ2)) → π̇ (φ1 ∨̇ □) ≔
begin
    assume φ1 φ2 H;
    apply ∨ᶜᵢ₁;
    apply ∧ᶜₑ₁ (imply_to_and H);
end;
```

Clause concatenation and canonical form

With clause as disjunction

$$((x_1 \dot{\vee} x_2) \vee x_3) \quad \vee \quad (y_1 \vee y_2 \vee y_3) \rightsquigarrow (((x_1 \vee x_2) \vee x_3) \vee (y_1 \vee y_2 \vee y_3))$$

With clause with type Clause

$$((x_1 \dot{\vee} x_2) \dot{\vee} x_3 \dot{\vee} \Box) \quad ++ \quad (y_1 \dot{\vee} y_2 \dot{\vee} y_3 \dot{\vee} \Box) \rightsquigarrow x_1 \dot{\vee} x_2 \dot{\vee} x_3 \dot{\vee} y_1 \dot{\vee} y_2 \dot{\vee} y_3 \dot{\vee} \Box$$

Clause representation in Lambdapi

Proof of clause (cl $\varphi_1, \ldots, \varphi_n$) with $(\mathcal{R}; p_1 \ldots p_n)[a_1 \ldots a_n]$ in a step such as

$$j. \quad \Delta \quad \vdash (\mathrm{cl}\ \varphi_1, \ldots, \varphi_n) \quad (R; p_1 \ldots p_n)[a_1, \ldots, a_n]$$

are encoded as proof:

```
injective symbol π̇ c: TYPE ≔ π (ℰ c);

sequential symbol ℰ: Clause → Prop;
rule ℰ ( $x ∨̇ $y) ↪ $x ∨ᶜ (ℰ $y)
with ℰ □ ↪ ⊥;
```

Clause conversion lemma

Lemma (Clause equivalence with disjunctions)

Given a b Clause we have the equivalence:

$\forall a\ b : Clause.\ [\![a\ ++\ b]\!] \iff^{c} [\![a]\!] \vee^{c} [\![a]\!]$

Clause resolution rule translation

Lemma (Resolution)

Given a b: Clause and a privot x: Prop, then a premise of $(x \mathrel{\dot\vee} a) \in \Gamma_{\mathcal{A}}$ *and a premise of* $(\neg^c\ x \mathrel{\dot\vee} b) \in \Gamma_{\mathcal{A}}$ *implies a clause* $a + +b$.

Lambdapi encoding:

```
opaque symbol resolution x a b : π̇ (x ∨̇ a) →π̇ (¬ᶜx ∨̇ b) →π̇ (a ++ b) ≔
```

Translation functions

The embedding uses four functions:

- $\mathcal{F}$ which translates first order formulas to $\Gamma_{\mathcal{A}}$-propositions,
- $\mathcal{S}$ which translates SMTLib sort from theory to $\Gamma_{\mathcal{A}}$-type,
- $\mathcal{T}$ which translates first order individual terms to $\Gamma_{\mathcal{A}}$-terms,
- $\mathcal{C}(\Gamma, c_1 \dots c_n)$ which translates a non-empty set of commands $c_1 \dots c_n$ to typing goal $\Gamma \vdash M : N$ and a term of type $M : N$ being proof term.

Function $\mathcal{F}$

translates first order formulas to $\Gamma_{\mathcal{A}}$-propositions

Definition ($\mathcal{F}$)

The definition of $\mathcal{F}$(f) is as follows.

- if f = $cl\ x_1 \ldots x_n$, then $\mathcal{F}_\Delta(cl\ x_1 \ldots x_n) = x_1 \ddot{\vee} \cdots \ddot{\vee}\ x_n \ddot{\vee} \Box$,
- if f = $a_1 \wedge \cdots \wedge a_n$, then $\mathcal{F}_\Delta(a_1 \wedge \cdots \wedge a_2) = a_1 \wedge^c \cdots \wedge^c a_2 \wedge^c \top$,
- if f = $a_1 \vee \cdots \vee a_n$, then $\mathcal{F}_\Delta(a_1 \vee \cdots \vee a_2) = a_1 \vee^c \cdots \vee^c a_2 \vee^c \bot$,
- if f = $a \approx b$ and $a\ b \in$ **Bool**, then $\mathcal{F}_\Delta(a \approx b) = a \iff^c b$,
- if f = $a \approx b$ and $a\ b \notin$ **Bool**, then $\mathcal{F}_\Delta(a \approx b) = (a = b)$,
- otherwise we are in the case $\mathcal{F}_\Delta(f) = f$ with all connector changes for their Corresponding classical connector $\star^c$.

Why this $\mathcal{F}$ definitions for conjunctions and disjunctions ?

N-ary rules are proved with "reflexivity" proof. For example

$i.\quad \Delta \quad \vdash \neg(\varphi_1 \wedge \cdots \wedge \varphi_n), \varphi_k \quad (\mathit{and\_pos})$
with $1 \leq k \leq n$

where we have the reflexivity proof:

```
sequential symbol In_∧ᶜ: Prop → Prop → 𝔹;
rule In_∧ᶜ $x ( $h ∧ᶜ $tl) ↪ (eq $x $h) Bool.or (In_∧ᶜ $x $tl)
with In_∧ᶜ $x ⊤ ↪ false;

symbol and_pos [φ1‥φn φk]:
    π ((In_∧ᶜ φk φ1‥φn) = true) → π̇ (¬ᶜ φ1‥φn ∨̇ φk ∨̇ □);
```

Function $\mathcal{S}$

translates SMTLib sort from theory to $\Gamma_{\mathcal{A}}$-type

The definition of $\mathcal{S}$(t) is as follows.

- if t = **Bool**, then $\mathcal{S}(\textbf{Bool}) = \mathit{Prop}$,
- if $t \neq$ **Bool**, then $\mathcal{S}(t) = \tau\ o$,
- if $t = f(a_1 \ldots a_n)$ and $\mathit{codomain}(f) =$ **Bool**, then $\mathcal{S}(f(a_1 \ldots a_n)) = f : \mathcal{S}(a_1) \rightarrow \cdots \rightarrow \mathcal{S}(a_1) \rightarrow \textbf{Prop}$,
- if $t = f(a_1 \ldots a_n)$ and $\mathit{codomain}(f) \neq$ **Bool**, then $\mathcal{S}(f(a_1 \ldots a_n)) = f : \mathcal{S}(a_1) \rightarrow \cdots \rightarrow \mathcal{S}(a_1) \rightarrow \textbf{Set}$,

Function $\mathcal{T}(t)$

Definition

The definition of $\mathcal{T}(t)$ is a direct shallow embedding of t in corresponding term t in $\Gamma_{\mathcal{A}}$ (variables, functions, constants).

Function $\mathcal{C}(\Gamma, -)$

translates Alethe commands

Definition

The function $\mathcal{C}(\Gamma, i.\ \Delta\ \vdash \varphi \quad (R; p_1 \ldots p_n)[a_1 \ldots a_n]) \rightarrow \Gamma'$ translates, in a given context Γ, a step i into a judgement (typing goal) $\Gamma \vdash i : \mathcal{F}(\varphi)$ with a term M along satisfying the goal. It returns a new context Γ' with $i \in \Gamma'$. The definition of $\mathcal{C}$ is defined recursively on R.

Notation

The notation $tac(A)[\Gamma \vdash \varphi]$ means applying the Lambdapi tactic tac (with argument A) to the judgement $\Gamma \vdash \varphi$ and making the judgements (subgoals) generated by the tactic be the premises of the rule.

Example 1: *equiv_pos2* case

We translate a step i using the alethe rule *equiv_pos2*:

$$i. \qquad \vdash \neg(\varphi_1 \approx \varphi_2), \neg\varphi_1, \varphi_2 \qquad\qquad (equiv\_pos2)$$

with given a context Γ as

$\mathcal{C}(\Gamma,\ i.\ \vdash \neg(\varphi_1 \approx \varphi_2), \neg\varphi_1, \varphi_2\ (equiv\_pos2)) = apply(equiv\_pos2)[\Gamma \vdash i : \mathcal{F}(\neg(\varphi_1 \approx \varphi_2), \neg\varphi_1, \varphi_2)]$

Example 2: *cong* case

We translate a step k using the alethe rule *cong*

$$
\begin{array}{llll}
i. & \Delta & \vdash t_1 \approx u_1 & (\dots) \\
 & \vdots & & \\
j. & \Delta & \vdash t_n \approx u_n & (\dots) \\
k. & \Delta & \vdash f\ t_1 \dots t_n \approx f\ u_1 \dots u_n & (cong; p_1 \dots p_n)
\end{array}
$$

as:

if codomain(f) $\in$ **Bool**, then $\mathcal{C}(\Gamma,\ i.\ \Delta \ \ \vdash f\ t_1 \dots t_n \approx f\ u_1 \dots u_n\ (cong; p_1 \dots p_n))$

$= \ cong2_f(f\ p_1 \dots cong2_f(f\ p_{n-1}\ p_n))[\Gamma \vdash i : \mathcal{F}(f\ t_1 \dots t_n \approx f\ u_1 u_n)]$

otherwise $f\_equal_n(f\ p_1\ \ \dots\ \ p_n)[\Gamma \vdash i : \mathcal{F}(f\ t_1 \dots t_n \approx f\ u_1 u_n)]$,

with $\mathcal{C}(\Gamma \backslash \{i\},\ i.\ \Delta \ \ \vdash t_1 \approx u_1\ (\dots)) \in \Gamma$ and

$\vdots$

with $\mathcal{C}(\Gamma \backslash \{j\},\ j.\ \Delta \ \ \vdash t_n \approx u_n\ (\dots)) \in \Gamma$

Soundness argument

Theorem (soundness)

We define the soundness as for a Alethe context Δ and a first order formula φ in a step $i.\ \Delta \vdash \varphi\ (\mathcal{R}; p_1 \dots p_n)[a_1 \dots a_n]$ proved by rule $\mathcal{R}$, the translation $\mathcal{C}(\Gamma, i.\Delta \vdash_{FOL} \varphi(\mathcal{R}; p_1 \dots p_n)[a_1 \dots a_n])$ give a term $i : \tau$ such that $\Gamma_{\mathcal{A}} \vdash i : \tau$.

Proof.

(proof intuition) By induction on $\mathcal{R}$. $\square$

Evaluation with TLA+ example

Stephan Merz. TLA+ Case Study: A Resource Allocator.

- Use set theory only,
- no skolemization,
- 25 proofs obligations,
- size of the Alethe proofs varies between 4 and 288 steps.

Evaluation results

Proofs obligations

- 16 on the 25 proofs obligations passed,
- some proofs obligations do not passed due to rules not supported yet.

Bug founds

- 2 importants bugs in Carcara elaboration process found,
- 1 bug found in CVC5 with `and_not` rule,
- and a related bug found in the new TLAPS SMT encoding.

Issues for reconstructing simplification rules

Transformation case is implicit, and multiple transformations in a step are possible.

Rules example

$j.\quad \Delta \quad \vdash \varphi_1 \vee \cdots \vee \varphi_n \quad$ (**or_simplify**)

- $\bot \vee \cdots \vee \bot \Rightarrow \bot$
- $\varphi_1 \vee \cdots \vee \varphi_n \Rightarrow \varphi_1 \vee \cdots \vee \varphi_{n'}$ where all $\bot$ literals removed
- $\varphi_1 \vee \cdots \vee \top \vee \cdots \vee \varphi_n \Rightarrow \top$
- ...

$j.\quad \Delta \quad \vdash (\varphi_1 \approx \varphi_2) \approx \psi \quad$ (**equiv_simplify**)

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

...

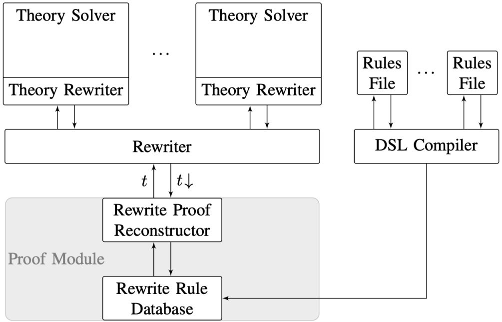
$j.\quad \Delta \quad \vdash \varphi \approx \psi \quad$ (**bool_simplify**)

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

$j.\quad \Delta \quad \vdash \varphi_1 \bowtie \varphi_n \approx \psi \quad$ (**comp_simplify**)

- $t < t \Rightarrow \bot$
- $t_1 < t_2 \Rightarrow \neg(t_2 \leq t_1)$
- $t_1 \leq t_2 \Rightarrow t_2 \leq t_1$
- ...

Reconstruction with RARE



[RARE] Schurr, HJ., et.al. Reliable Reconstruction of Fine-grained Proofs in a Proof Assistant. CADE 2021. Springer, Cham.

Checking linear arithmetic steps

- The `la_generic` rule models linear arithmetic reasoning
- For example, consider this `la_generic` step:

```
(step t1
    (cl (<= (- x) 1) (<= (+ (* 2 x) (* (- 3) y)) 2) (<= y (- 1)))
    :rule la_generic :args (2 1 3))
```

- It introduces the following tautology:

$$(-x \leq 1) \vee (2x - 3y \leq 2) \vee (y \leq -1)$$

Checking linear arithmetic steps

```
(step t1
    (cl (<= (- x) 1) (<= (+ (* 2 x) (* (- 3) y)) 2) (<= y (- 1)))
    :rule la_generic :args (2 1 3))
```

- Checking that this clause is true is equivalent to proving that its negation, the following three inequalites, are contradictory
- Since `la_generic` steps provide the needed coefficients as arguments, checking them is simple
- Computing $2 \cdot (a) + 1 \cdot (b) + 3 \cdot (c)$, we get $0 > 1$, so the step must be true

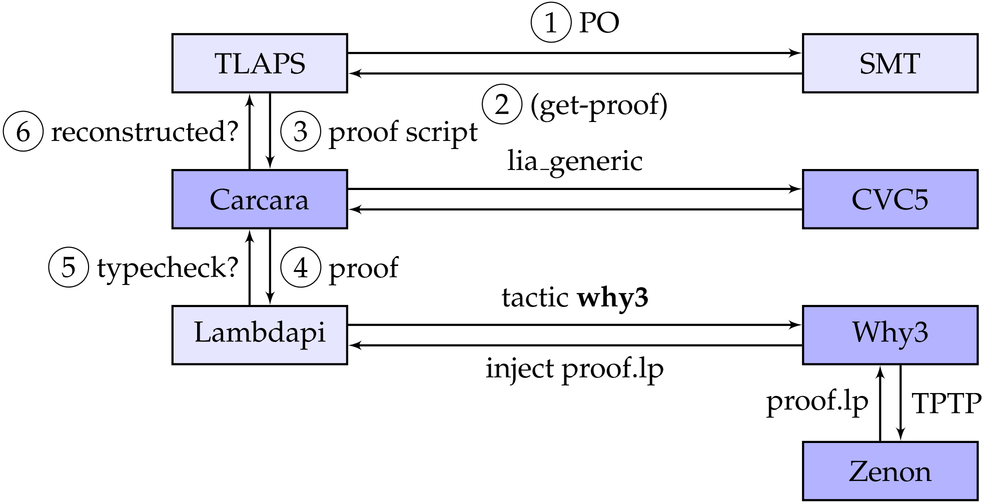
Checking linear arithmetic steps

- The `lia_generic` rule is very similar to `la_generic`, but it does not provide the coefficients as arguments:

```
(step t1
    (cl (<= (- x) 1) (<= (+ (* 2 x) (* (- 3) y)) 2) (<= y (- 1)))
    :rule lia_generic)
```

- In this case, the checker would need to search for the coefficients, which is an NP-hard problem
- Instead, occurences of this rule are not checked, and are considered holes by Carcara

Proposal for reconstructing arithmetic steps



Future perspectives

- Finish to validate *Allocator.tla*,
- add support for arithmetic steps,
- support for simplification steps,
- connect lambdapi TLA$^+$ encoding (*tla-lambdapi*) with TLA$^+$ SMT encoding,
- link Event-B encoding with tla-lambdapi (shared set theory library).