Progress on the reconstruction of TLAPS proofs solved by SMT in Lambdapi

Alessio Coltellacci

Univ. Lorraine, CNRS, Inria, Loria

Fontainebleau

TLA$^+$ at a glance

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

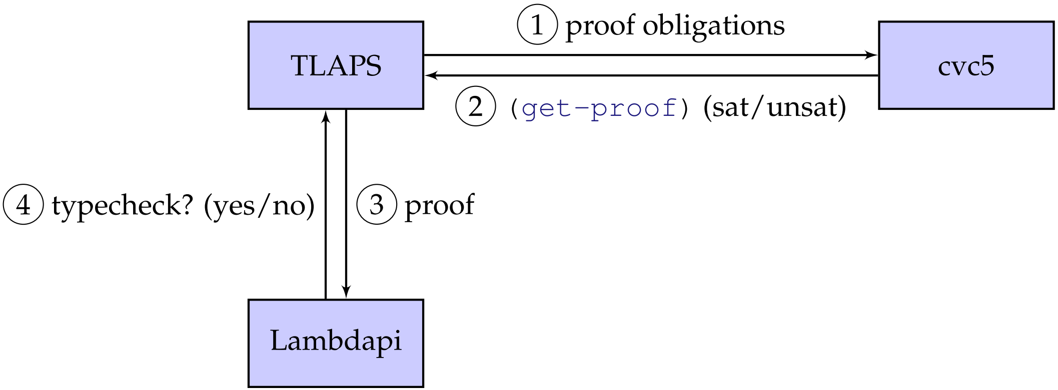
$VARIABLE\ x$
$CONSTANT\ N$
$ASSUME\ N \in Nat$

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

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

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

Targeted validation process



Cantor theorem proof with TLAPS

```
--------------- MODULE Cantor1 ------------------
THEOREM cantor ==
  \A S :
    \A f \in [S -> SUBSET S] :
      \E A \in SUBSET S :
        \A x \in S :
          f [x] # A
PROOF
  <1>1. TAKE S
  <1>2. TAKE f \in [S -> SUBSET S]
  <1>3. DEFINE T == { z \in S : z \notin f[z] }
  <1>4. WITNESS T \in SUBSET S
  <1>5. TAKE x \in S
  <1>6. QED BY x \in T \/ x \notin T
================================================
```

Proof script obtained from veriT

```
(assume |ExtTrigEqDef SetSt_flatnd_1| (forall ((x Idv) (y Idv))
    (= (TrigEq_Idv x y) (= x y))))
(assume h2 (Mem CONST_x_ CONST_S_))
(assume |SetStDef SetSt_flatnd_1|
    (forall ((a Idv) (x Idv)) (= (Mem x (SetSt_flatnd_1 a))
    (and (Mem x a) (not (Mem x (FunApp CONST_f_ x)))))))
(assume | Goal | (not (not (TrigEq_Idv (FunApp CONST_f_ CONST_x_)
    (SetSt_flatnd_1 CONST_S_)))))
(step t5 (cl (not (not (not (TrigEq_Idv (FunApp CONST_f_ CONST_x_)
    (SetSt_flatnd_1 CONST_S_)))))
    (TrigEq_Idv (FunApp CONST_f_ C CONST_x_)
    (SetSt_flatnd_1 CONST_S_))) :rule not_not)
...
(step t47 (cl (Mem CONST_x_ (FunApp CONST_f_ CONST_x_)))
    :rule resolution :premises (t46 t38 t41 t42))
...
(step t52 (cl) :rule resolution :premises (t51 t32 t49 t47))
```

Alethe format

- Alethe is a new SMT proof format that aims to be usable by many different solvers.
  - It is currently supported by the SMT solvers veriT and cvc5
- Alethe uses a term language that directly extend SMT-LIB.
- Alethe provides rules with varying levels of granularity
- This allows solver to rely on powerful checkers and produce coarse-grained proofs, or take the effort to produce more fine-grained proofs.

Alethe format through examples

Proof statement examples:

```
(assume h1 (not (p a)))

(step t1 (cl (= z2 vr4)) :rule refl)

(step t4 (cl (= (P x) (P y))) :rule cong :premises (t3))

(step t7 (cl) :rule resolution :premises (h1 h2 t5 t6))
```

Alethe rules example

- $\neg(\varphi_1 \approx \varphi_2),\ \varphi_1,\ \neg\varphi_2$ **(equiv_pos2)**
- $t \approx t$ **(equiv_refl)**
- $\bot \vee \cdots \vee \bot \Rightarrow \bot$ **(or_simplify)**

  $\varphi_1 \vee \cdots \vee \top \vee \cdots \vee \varphi_n \Rightarrow \top$

$$\frac{A \vee B \vee x \qquad C \vee \neg x}{A \vee B \vee C}$$ **(resolution)**

Alethe rule bind

Renaming of bound variables with **(bind)**

$$
\begin{array}{llcll}
j. & \Gamma,\, y_1 \ldots y_n,\, x_1 \mapsto y_1,\ \ldots,\, x_n \mapsto y_n & \rhd & \varphi \approx \varphi' & (\ldots) \\
k. & & \rhd & \forall x_1, \ldots, x_n . \varphi \approx \forall y_1, \ldots, y_n . \varphi' & \textbf{(bind)}
\end{array}
$$

Alethe format through examples

Sub proof example:

```
...
(anchor :step t9 :args ((:= z2 vr4)))
(step t9.t1 (cl (= z2 vr4)) :rule refl)
(step t9.t2 (cl (= (p z2) (p vr4)))
        :rule cong :premises (t9.t1))
(step t9 (cl (= (forall ((z2 U)) (p z2))
                (forall ((vr4 U)) (p vr4))))
        :rule bind)
...
```

Main difficulties

1. SMT solvers produce very coarse-grained proofs, which can be very hard to check.

    - Operation on clause are modulo associativity and commutativity
    - Implicit contraction of literals in clause
    - Implicit reordering of literals in clause
    - Implicit application of symmetry on equality

2. Fine-grained proofs are necessary due to the lack of automation in Lambdapi.

Carcara

- Carcara [ALB23] is an efficient and independent proof checker and elaborator for Alethe proofs
- It is written in Rust, a high performance language

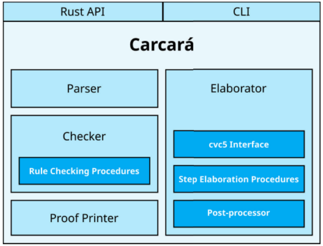
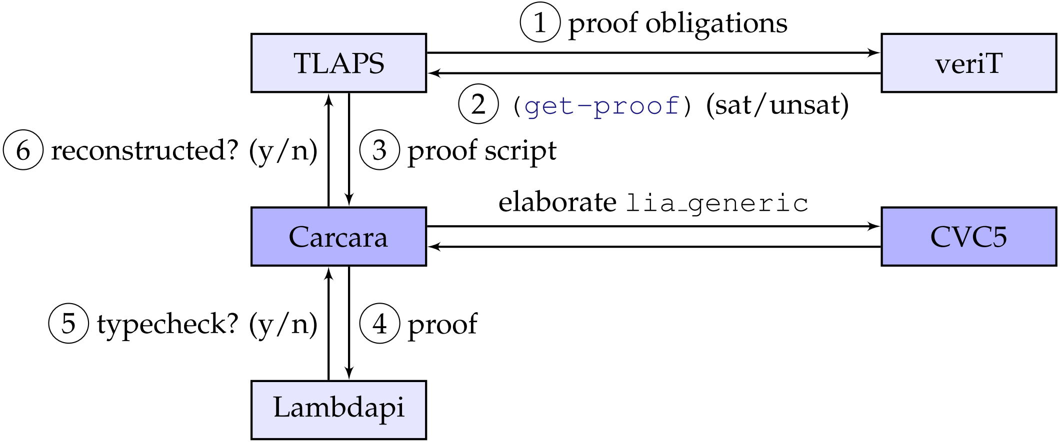


Fig. 2: Overview of the architecture of CARCARA.

Proposed solution



Elaborated proof with Carcara

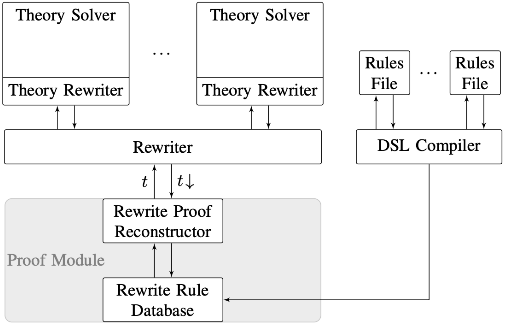
- Replace `lia_generic` by finer-grained steps by using SMT solver
- Adds pivots in resolution

```
(step t1 (cl a b c) :rule ...)
(step t2 (cl (not a) d) :rule ...)
(step t3 (cl (not c) e (not f)) :rule ...)
(step t4 (cl f) :rule ...)
(step t5 (cl b d e) :rule resolution :premises (t1 t2 t3 t4)
    :args (a true c true f false))
```

- Removing the implicit reordering of equalities

Reconstructing Fine-Grained Proofs of Rewrites

RARE, the reconstruction proofs of rewrite in cvc5 [Nöt+22].



```
(define-rule arith-plus-zero ((t ? :list) (s ? :list)) (+ t 0 s) (+ t s))

(define-rule arith-mul-zero ((t ? :list) (s ? :list)) (* t 0 s) 0)

(define-rule* bool-or-false
    ((xs Bool :list) (ys Bool :list)) (or xs false ys) (or xs ys))

(define-cond-rule ite-neg-branch ((c Bool) (x Bool) (y Bool))
    (= (not y) x) (ite c x y) (= c x))
```

Translation into Lambdapi

Classical logic

$$\top \mid \bot \mid \wedge^c \mid \vee^c \mid \forall^c \mid \exists^c \mid \neg^c \mid \Rightarrow^c \mid \leftrightarrow^c$$
$$\dot{\forall} \mid \pi^c(\_) \mid = \mid \leq$$

Translation overview

Alethe	Lambdapi
`(assume h1 (forall ((x S)) (P x)))`	`have h1: π^c (∀^c (x: S), P x)` `{ admit }`
`(step t1 (cl (= (P x) (P y)))` `    :rule cong :premises (t3))`	`have t1: π^c (P x = P y)` `{ apply feq P t3 }`

`(anchor :step t9 :args ((:= z2 vr4)))`
`(step t9.t1 (cl (X)) :rule ...)`
`...`
`(step t9.tn (cl (Y)) :rule ...)`
`(step t9 (cl Y) :rule subproof)`
	`opaque symbol t9 z2 vr4`
	`(p: π^c (z2 = vr4)): π (Y) :=`
	`begin`
	`    have t9.1: π^c (X) { ... };`
	`    have t9.n: π^c (X) { ... };`
	`    apply t9.n;`
	`end;`

Translation overview

Resolution

<table>
<tr><th>Alethe</th><th>Lambdapi</th></tr>
<tr><td>
<pre>
(step tn (cl b d e) :rule
resolution :premises (t1 t2 t3)
:args (a true c true))
</pre>
</td><td>
<pre>
have tn: π^c(b, d, e)
{
  have t1': π^c(c, b, a)
  -> π(a, c, b)
  { ... };
  have t1_t2: π^c(...)
  { apply resolution (t1' t1) t2};
  have t2_t3: π^c(...)
  { apply resolution t1_t2 t2};
   apply t2_t3
}
</pre>
</td></tr>
</table>

Conclusion

- The `_-simplify` step can be reconstructed with the rewrite rules of Lambdapi and RARE.
- Elaborated proof produced by Carcara allows us to reconstruct `resolution` and tautologies step.
- We do not yet know how to reconstruct arithmetic proof.

References I

[Nöt+22] Andres Nötzli et al. "Reconstructing Fine-Grained Proofs of Rewrites Using a Domain-Specific Language". In: *2022 Formal Methods in Computer-Aided Design (FMCAD)*. 2022, pp. 65–74. DOI: 10.34727/2022/isbn.978-3-85448-053-2_12.

[ALB23] Bruno Andreotti, Hanna Lachnitt, and Haniel Barbosa. "Carcara: An Efficient Proof Checker and Elaborator for SMT Proofs in the Alethe Format". In: *TACAS 2023, April 22–27, 2023*. Springer-Verlag, 2023. DOI: 10.1007/978-3-031-30823-9_19.