# Carcara: A proof checker, elaborator and translator for Alethe

Bruno Andreotti

EuroProofNet Symposium at Orsay, France, 11 Sep 2025

Universidade Federal de Minas Gerais (UFMG)

# Introduction

# SMT solvers and trust

- SMT solvers are crucial tools in many formal methods applications, like proof assistants and program verification
- However, these solvers often have large and complex codebases, which makes detecting bugs difficult
  - Correctness bugs are often found in widely used SMT solvers

# SMT solvers and trust

- SMT solvers are crucial tools in many formal methods applications, like proof assistants and program verification
- However, these solvers often have large and complex codebases, which makes detecting bugs difficult
  - Correctness bugs are often found in widely used SMT solvers
- Then, how can we trust the correctness of their results?

# SMT proofs

- We can instrument the SMT solver to produce a proof
- An SMT proof is a certificate of the solver results, that formally justifies the logical reasoning it used to find a solution

# SMT proofs

- We can instrument the SMT solver to produce a proof
- 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 is usually simpler and quicker than solving

# Alethe format

- Alethe is a well established SMT proof format that aims to be usable by many different solvers
  - It is currently supported by the SMT solvers veriT and cvc5
- Alethe's syntax is very similar to that of SMT-LIB, the standard input language for SMT solvers

# Alethe format

- Alethe is a well established SMT proof format that aims to be usable by many different solvers
  - It is currently supported by the SMT solvers veriT and cvc5
- Alethe's syntax is very similar to that of SMT-LIB, the standard input language for SMT solvers
- The format allows proofs with varying levels of *granularity*
- This allows solvers to rely on powerful checkers and produce coarse-grained proofs, or take the effort to produce more fine-grained proofs

# Example of an Alethe proof

```
(set-logic LIA)
(declare-fun p (Int) Bool)
(assert (forall ((x Int)) (p x)))
(assert (not (forall ((y Int)) (p y))))
(check-sat)
```

```
(assume h1 (forall ((x Int)) (p x)))
(assume h2 (not (forall ((y Int)) (p y))))
(anchor :step t3 :args ((y Int) (:= x y)))
(step t3.t1 (cl (= x y)) :rule refl)
(step t3.t2 (cl (= (p x) (p y))) :rule cong :premises (t3.t1))
(step t3 (cl (= (forall ((x Int)) (p x)) (forall ((y Int)) (p y)))) :rule bind)
(step t4 (cl (not (forall ((x Int)) (p x))) (forall ((y Int)) (p y)))
    :rule equiv1 :premises (t3))
(step t5 (cl) :rule resolution :premises (t4 h1 h2))
```

# Checking Alethe proofs

- One way to check that an Alethe proof is valid is by *reconstructing* it in a proof assistant
- This approach allows a lot of trust in the correctness of the result, but has some drawbacks
  - performance, usability

# Checking Alethe proofs

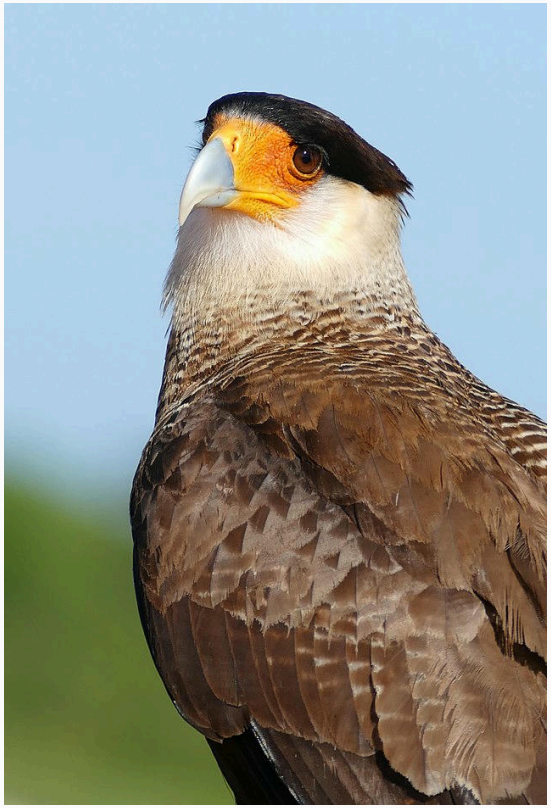
- One way to check that an Alethe proof is valid is by *reconstructing* it in a proof assistant
- This approach allows a lot of trust in the correctness of the result, but has some drawbacks
  - performance, usability
- Also, some coarse-grained steps might be very hard or slow to check, resulting in reconstruction failures

# Checking Alethe proofs

- One way to check that an Alethe proof is valid is by *reconstructing* it in a proof assistant
- This approach allows a lot of trust in the correctness of the result, but has some drawbacks
  - performance, usability
- Also, some coarse-grained steps might be very hard or slow to check, resulting in reconstruction failures
- Instead, one might prefer a stand-alone checker, focused on efficiency and usability

# Introducing Carcara

- *Carcara* is an efficient and independent proof checker for Alethe proofs, developed in Rust
- Available at *https://github.com/ufmg-smite/carcara*



*Caracara plancus*

# Checking

# Checking Alethe proofs

- The validity of each step depends on the rule used
- A checking procedure had to be implemented for each of the over 100 rules currently in the Alethe format

# Checking Alethe proofs

- The validity of each step depends on the rule used
- A checking procedure had to be implemented for each of the over 100 rules currently in the Alethe format
- As we'll see, the flexibility of the Alethe format means some steps can be tricky to check

# Checking `assume` commands

- An `assume` command introduces a premise of the proof, and they each must correspond to an `assert` in the original problem
- During parsing, the problem's assumptions are stored in a hash set

# Checking `assume` commands

- An `assume` command introduces a premise of the proof, and they each must correspond to an `assert` in the original problem
- During parsing, the problem's assumptions are stored in a hash set
- Generally, checking `assume` commands ammounts to simply accessing that set

# Checking `assume` commands

- However, the solver may implicitly reorder equalities when producing a proof
- This means an `assume` command may reference a problem premise while implicitly reordering an equality inside it

```
(set-logic QF_UF)
(declare-const a Bool)
(declare-const b Bool)
(assert (= a b))
(assert (not (= b a)))
(check-sat)
```

```
(assume h1 (= a b))
(assume h2 (not (= a b)))
(step t3 (cl)
    :rule resolution
    :premises (h1 h2))
```

# Checking `assume` commands

- In this case, the checker must iterate through all the premises and see if they are equal to the `assume` term, modulo the reordering of equalities
- This requires traversing the terms, possibly up to their depth
- We call this equality check a *Polyequal* check

```
(set-logic QF_UF)
(declare-const a Bool)
(declare-const b Bool)
(assert (= a b))
(assert (not (= b a)))
(check-sat)
```

```
(assume h1 (= a b))
(assume h2 (not (= a b)))
(step t3 (cl)
    :rule resolution
    :premises (h1 h2))
```

# Implicit reordering of equalities

- This reordering of equalities can happen not only in `assume` commands, but anywhere in a proof!

---

```
(step t1 (cl (= (= a b) (= b a))) :rule refl)
```

---

```
(step t1 (cl (= t u)) :rule ...)
(step t2 (cl (= (= a t) (= u a))) :rule cong :premises (t1))
```

---

```
(step t1 (cl
  (or (not (forall ((x Real) (y Real)) (= x y))) (= b a))
) :rule forall_inst :args (a b))
```

---

# Checking resolution steps

- SMT proofs usually make heavy use of the resolution inference rule
- In Alethe, this is modeled by the rule `resolution`

$$\frac{A \vee p_1 \qquad B \vee \neg p_1 \vee p_2 \qquad C \vee \neg p_2 \vee p_3 \qquad D \vee \neg p_3}{A \vee B \vee C \vee D}\ \texttt{resolution};\ p_1, p_2, p_3$$

- Here, $p_1$, $p_2$ and $p_3$ are the pivots of the resolution step

# Checking resolution steps

- In general, the pivots used in a `resolution` step are not provided in the Alethe proof
- The checker must then infer which pivot was used for each binary resolution step
- To do this, Carcara uses a greedy algorithm that looks at the conclusion clause to guess which terms are pivots and which terms should be kept

# Checking resolution steps: inferring the pivots

$$\frac{a \vee b \vee c \qquad \neg a \vee d \qquad \neg c \vee e \vee \neg f \qquad f}{b \vee d \vee e}\ \texttt{resolution;}\ ???$$

# Checking resolution steps: inferring the pivots

$$\frac{a \vee b \vee c \qquad \neg a \vee d \qquad \neg c \vee e \vee \neg f \qquad f}{b \vee d \vee e}\ \texttt{resolution};\ a, \ldots$$

# Checking resolution steps: inferring the pivots

$$\frac{a \vee b \vee c \qquad \neg a \vee d \qquad \neg c \vee e \vee \neg f \qquad f}{b \vee d \vee e}\ \texttt{resolution};\ a, c, \ldots$$

# Checking resolution steps: inferring the pivots

$$\frac{a \vee b \vee c \qquad \neg a \vee d \qquad \neg c \vee e \vee \neg f \qquad f}{b \vee d \vee e}\ \texttt{resolution};\ a, c, f$$

# Checking resolution steps: inferring the pivots

- This greedy algorithm is relatively efficient, but incomplete—some valid resolution steps will be rejected by it
- Carcara also uses a more complex algorithm based on *Reverse Unit Propagation* (that is complete) as a fallback

# Checking resolution steps: inferring the pivots

- This greedy algorithm is relatively efficient, but incomplete—some valid resolution steps will be rejected by it
- Carcara also uses a more complex algorithm based on *Reverse Unit Propagation* (that is complete) as a fallback
- If the pivots were provided, of course, checking would be both easy and complete!

# Elaboration

# Proof elaboration

- There are many rules in Alethe which allow very coarse-grained steps
- By breaking them down into smaller, finer-grained steps, we can produce a proof that is easier to check and contains less holes

# Proof elaboration

- There are many rules in Alethe which allow very coarse-grained steps
- By breaking them down into smaller, finer-grained steps, we can produce a proof that is easier to check and contains less holes
- Besides proof checking, Carcara is also able to do this proccess, called *proof elaboration*

# Proof elaboration

- While it doesn't make sense to elaborate a proof to check it with Carcara, it might be useful as a "post-processing" step before passing it to a different tool

# Proof elaboration

- While it doesn't make sense to elaborate a proof to check it with Carcara, it might be useful as a "post-processing" step before passing it to a different tool
- Notably, if you want to reconstruct the proof in a proof assistant, or translate the proof to a different format, elaboration can be very helpful

# Removing the implicit reordering of equalities

- Recall that SMT solvers may implicitly reorder equalities when producing Alethe proofs
- This makes checking more complicated and less efficient

# Removing the implicit reordering of equalities

- Recall that SMT solvers may implicitly reorder equalities when producing Alethe proofs
- This makes checking more complicated and less efficient
- An elaboration procedure was developed to remove this implicit transformation

# Removing the implicit reordering of equalities

- The proof would be simpler to check if `h2` was instead `(assume h2 (not (= b a)))`

```
(set-logic QF_UF)
(declare-const a Bool)
(declare-const b Bool)
(assert (= a b))
(assert (not (= b a)))
(check-sat)
```

```
(assume h1 (= a b))
(assume h2 (not (= a b)))
(step t3 (cl) :rule resolution
    :premises (h1 h2))
```

# Removing the implicit reordering of equalities

- The proof would be simpler to check if `h2` was instead `(assume h2 (not (= b a)))`
- However, step `t3` uses `h2` as a premise, so we can't just change the assumed term
- Instead, we need to add steps that reconstruct the original term

```
(set-logic QF_UF)
(declare-const a Bool)
(declare-const b Bool)
(assert (= a b))
(assert (not (= b a)))
(check-sat)
```

```
(assume h1 (= a b))
(assume h2 (not (= a b)))
(step t3 (cl) :rule resolution
    :premises (h1 h2))
```

# Removing the implicit reordering of equalities

- This is the correct elaboration of the proof:

```
(set-logic QF_UF)
(declare-const a Bool)
(declare-const b Bool)
(assert (= a b))
(assert (not (= b a)))
(check-sat)
```

```
(assume h1 (= a b))
(assume h2 (not (= b a)))
(step h2.t1 (cl (not (= a b)))
    :rule not_symm :premises (h2))
(step t3 (cl) :rule resolution
    :premises (h1 h2.t1))
```

# Removing the implicit reordering of equalities

- This is the correct elaboration of the proof:
- Now `h2` refers to the premise as it appeared in the original problem, and the added step reconstructs the original `h2` term
- The step `t3`, that used to reference `h2`, now references `h2.t1`

```
(set-logic QF_UF)
(declare-const a Bool)
(declare-const b Bool)
(assert (= a b))
(assert (not (= b a)))
(check-sat)
```

```
(assume h1 (= a b))
(assume h2 (not (= b a)))
(step h2.t1 (cl (not (= a b)))
    :rule not_symm :premises (h2))
(step t3 (cl) :rule resolution
    :premises (h1 h2.t1))
```

# Removing the implicit reordering of equalities

- A similar procedure is done for all rules that are affected by this implicit reordering of equalities
- Thus, checking an elaborated proof can be done without the use of polyequality checks

# Removing `reordering` steps

- The `reordering` rule takes a single clause as a premise and produces a permutation of that clause:

$$\frac{a \vee b \vee c \vee d}{c \vee b \vee d \vee a}\,\texttt{reordering}$$

- While not particularly difficult to check in Carcara, it is challenging to formalize in tools like proof assistants

# Removing `reordering` steps

- The `reordering` rule takes a single clause as a premise and produces a permutation of that clause:

$$\frac{a \vee b \vee c \vee d}{c \vee b \vee d \vee a}\,\texttt{reordering}$$

- While not particularly difficult to check in Carcara, it is challenging to formalize in tools like proof assistants

- To facilitate in translation to different proof formats, Carcara can also remove *all* `reordering` steps in a proof

# Removing `reordering` steps

- To do this, we
  1. Remove all `reordering` steps, replacing them with their premise
  2. Recompute every step that uses one of the modified steps as a premise
  3. Repeat, until the end of the proof

# Removing `reordering` steps

- To do this, we
  1. Remove all `reordering` steps, replacing them with their premise
  2. Recompute every step that uses one of the modified steps as a premise
  3. Repeat, until the end of the proof

- In Alethe, most rules only accept premises with unary clauses, with the only exceptions being the rules `weakening`, `contraction` and `resolution`
  - so we only need to recompute those!

# Removing `reordering` steps

$$\dfrac{\dfrac{\dfrac{\dots}{a \lor b \lor c \lor d}}{c \lor b \lor d \lor a}\texttt{reordering} \qquad \dfrac{\dots}{\neg b}}{c \lor d \lor a}\texttt{resolution}$$

# Removing `reordering` steps

$$\dfrac{\dfrac{\dfrac{\dots}{a \lor b \lor c \lor d}}{\sout{c \lor b \lor d \lor a}}\,\sout{\texttt{reordering}} \qquad \dfrac{\dots}{\neg b}}{c \lor d \lor a}\,\texttt{resolution}$$

# Removing `reordering` steps

$$\dfrac{\dfrac{\dots}{a \vee b \vee c \vee d} \qquad \dfrac{\dots}{\neg b}}{c \vee d \vee a}\,\texttt{resolution}$$

# Removing `reordering` steps

$$\dfrac{\dfrac{\dots}{a \vee b \vee c \vee d} \qquad \dfrac{\dots}{\neg b}}{a \vee c \vee d}\ \texttt{resolution}$$

# Other elaboration procedures

- Simplifying some transitivity steps
  - this can make steps that take $O(n^2)$ time to check be checkable in $O(n)$

# Other elaboration procedures

- Simplifying some transitivity steps
  - this can make steps that take $O(n^2)$ time to check be checkable in $O(n)$
- Finding resolution pivots, and performing *"uncrowding"*
  - normally, resolution can include the implicit reordering of clauses and the implicit removal of duplicates

# Other elaboration procedures

- Simplifying some transitivity steps
  - this can make steps that take $O(n^2)$ time to check be checkable in $O(n)$
- Finding resolution pivots, and performing *"uncrowding"*
  - normally, resolution can include the implicit reordering of clauses and the implicit removal of duplicates
- Filling in "holes" using external tools
  - e.g., call the cvc5 SMT solver to elaborate hard-to-check `lia_generic` steps

# Ongoing and future work

# Proof translation

- We are working on adding support for translating Alethe proofs into different formats
  - Lambdapi/Dedukti
  - Eunoia

# Proof translation

- We are working on adding support for translating Alethe proofs into different formats
  - Lambdapi/Dedukti
  - Eunoia

- This effort will help the integration of SMT solvers, and SMT proofs, into many more tools

# Proof translation

- We are working on adding support for translating Alethe proofs into different formats
  - Lambdapi/Dedukti
  - Eunoia
- This effort will help the integration of SMT solvers, and SMT proofs, into many more tools
- Elaborating the proof before translation makes this a lot easier

# And many other things!

- Expanding support for other theories
  - bitvectors
  - strings
  - datatypes

# And many other things!

- Expanding support for other theories
  - bitvectors
  - strings
  - datatypes
- Proof compression

# And many other things!

- Expanding support for other theories
  - bitvectors
  - strings
  - datatypes
- Proof compression
- Improving *proof slicing*
  - extracting a specific step of interest from a large proof

# Conclusion

# Conclusion

- Carcara is currently used by developers of SMT solvers and other tools who want to support the Alethe format

# Conclusion

- Carcara is currently used by developers of SMT solvers and other tools who want to support the Alethe format
- The tool has served as testing grounds for changes in Alethe, and has uncovered inconsistencies that have since been fixed

# Conclusion

- Carcara is currently used by developers of SMT solvers and other tools who want to support the Alethe format
- The tool has served as testing grounds for changes in Alethe, and has uncovered inconsistencies that have since been fixed
- More than just a proof checker, Carcara is a becoming a multi-purpose tool to work with Alethe proofs

# Conclusion

Thank you to *Átila Augusto, Haniel Barbosa, Bernardo Borges, Vinícius Braga, Tiago Campos, Alessio Coltellacci, Vinicius Gomes, Jibiana Jakpor, Hanna Lachnitt, Guilherme Luiz, José Neto, Hans-Jörg Schurr and Mallku Soldevila*, who have worked and/or are working on Carcara.

And thank you for listening!