Efficient, Verified Checking of Propositional Proofs

Marijn J.H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Nathan D. Wetzler

http://www.cs.utexas.edu/users/moore/acl2

THE UNIVERSITY OF TEXAS AT AUSTIN

ACL2

ITP in Brasilia, Brazil September 27, 2017

ABSTRACT

We present a case study, consisting of a sequence of verified checkers that validate SAT proofs. These culminate in an efficient checker that can be used in SAT competitions and in industry. No background in SAT is assumed.

OUTLINE

INTRODUCTION
- The Problem
- Propositional Proofs
- Efficient Proof-checking

A SEQUENCE OF CHECKERS
- The ACL2 Theorem-Proving System
- The Input Format
- **[lrat-1]** to **[lrat-5]**

CONCLUSION
- Overview
- References

Outline

Introduction
- The Problem
- Propositional Proofs
- Efficient Proof-checking

A Sequence of Checkers
- The ACL2 Theorem-Proving System
- The Input Format
- **[lrat-1]** to **[lrat-5]**

Conclusion
- Overview
- References

The Problem

Boolean Satisfiability (SAT) solvers are proliferating and useful.

But how can we trust them?

Modern ones [3] emit proofs!

But how do we know that these "proofs" are valid?

We check them with software programs called checkers!

But how do we know that a checker is sound? Inspection?

- Checkers are typically simpler than solvers...
- ... but not *that* simple, and inspection is error-prone.

Propositional Proofs

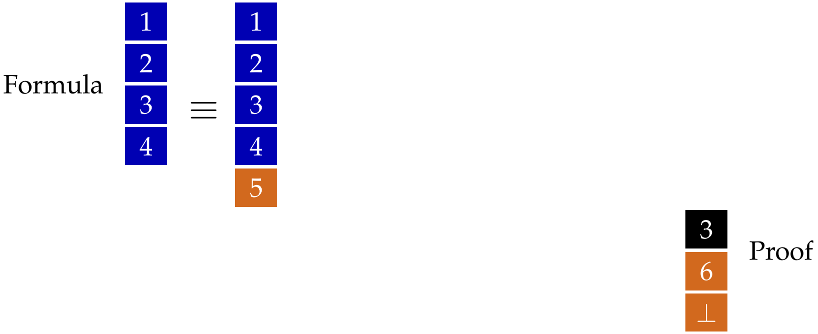
A *propositional proof* (or *clausal proof*, or *refutation*) for a formula F is a sequence $\Pi = \langle p_1, p_2, \ldots, p_k \rangle$ such that:

- Each p_i is $\langle b_i, c_i \rangle$, where b_i is a Boolean and c_i is a clause.
  Deletion step: b_i is true; Addition step: b_i is false.
- b_k is false and c_k is the empty clause, denoted by $\perp$.

Propositional Proofs

A *propositional proof* (or *clausal proof*, or *refutation*) for a formula F is a sequence $\Pi = \langle p_1, p_2, \ldots, p_k \rangle$ such that:

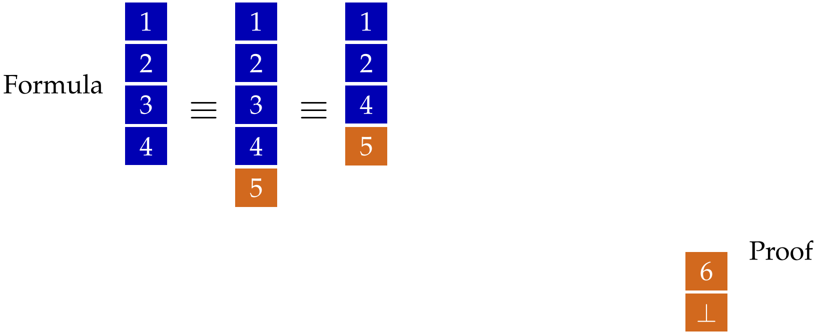
- Each p_i is $\langle b_i, c_i \rangle$, where b_i is a Boolean and c_i is a clause.
  Deletion step: b_i is true; Addition step: b_i is false.
- b_k is false and c_k is the empty clause, denoted by $\perp$.



PROPOSITIONAL PROOFS

A *propositional proof* (or *clausal proof*, or *refutation*) for a formula F is a sequence $\Pi = \langle p_1, p_2, \ldots, p_k \rangle$ such that:

- Each p_i is $\langle b_i, c_i \rangle$, where b_i is a Boolean and c_i is a clause.
  Deletion step: b_i is true; Addition step: b_i is false.
- b_k is false and c_k is the empty clause, denoted by $\perp$.

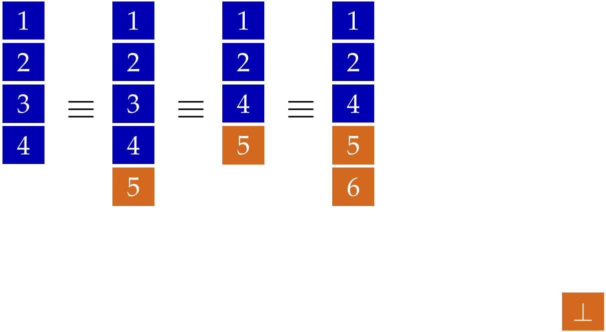


PROPOSITIONAL PROOFS

A *propositional proof* (or *clausal proof*, or *refutation*) for a formula F is a sequence $\Pi = \langle p_1, p_2, \ldots, p_k \rangle$ such that:

- Each p_i is $\langle b_i, c_i \rangle$, where b_i is a Boolean and c_i is a clause. Deletion step: b_i is true; Addition step: b_i is false.
- b_k is false and c_k is the empty clause, denoted by $\perp$.

Formula

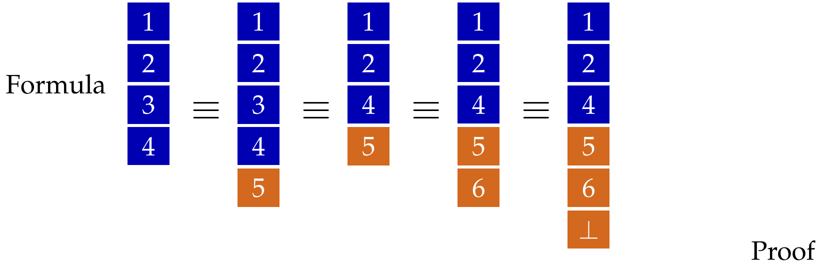


Proof

Propositional Proofs

A *propositional proof* (or *clausal proof*, or *refutation*) for a formula F is a sequence $\Pi = \langle p_1, p_2, \ldots, p_k \rangle$ such that:

- Each p_i is $\langle b_i, c_i \rangle$, where b_i is a Boolean and c_i is a clause.
  Deletion step: b_i is true; Addition step: b_i is false.
- b_k is false and c_k is the empty clause, denoted by $\perp$.



PROPOSITIONAL PROOF CHECKING

For $\Pi = \langle p_1, p_2, ..., p_k \rangle$ as above, recursively define formulas $\langle F_0 = F, F_1, ..., F_k \rangle$ by executing the p_i:

- For $i > 0$ and b_i true, delete c_i from F_{i-1} to get F_i.
- For $i > 0$ and b_i false, add c_i to F_{i-1} to get F_i.

For each addition step p_i we require:

- If F_{i-1} is satisfiable then F_i is satisfiable;
- This property must be checkable in polynomial time.

A popular proof system of propositional proofs is DRAT:

- DRAT allows the addition of so-called resolution asymmetric tautologies (RATs) — whatever that means.
- It can be efficiently checked if a clause is a RAT.
- RATs are not necessarily implied by the formula.

Formalizing Soundness

The following is trivial by induction.

Lemma. Suppose that $\Pi = \langle p_1, p_2, ..., p_k \rangle$ is a proof and F_0 is satisfiable. Then each F_i is satisfiable.

Soundness argument for DRAT proofs:

1. Deletion steps clearly preserve satisfiability.
2. Addition of RAT clauses preserves satisfiability.
3. By the lemma, if F_0 is satisfiable then F_k is satisfiable.
4. Since p_k adds the empty clause, F_k is unsatisfiable.
5. It follows immediately that F_0 is unsatisfiable.

EFFICIENT PROOF-CHECKING

HOWEVER: Our ITP 2013 checker, discussed above, was intended to be a proof of concept, not an efficient tool.

On one example:

- DRAT-trim checker [2]: 1.5 seconds
- Our ITP 2013 checker: 1 week

The flow for efficient, verified SAT proof-checking:

1. SAT solver verifies unsatisfiability of formula F; generates alleged proof, Π_0.
2. *DRAT-trim* takes inputs Π_0 and F; outputs alleged proof Π_1 for checker, in a format amenable to efficient checking.
3. A verified checker validates that Π_1 is a proof for F [1, 4].

OUTLINE

INTRODUCTION
The Problem
Propositional Proofs
Efficient Proof-checking

A SEQUENCE OF CHECKERS
The ACL2 Theorem-Proving System
The Input Format
[lrat-1] to **[lrat-5]**

CONCLUSION
Overview
References

ACL2: AN EFFICIENT PROGRAMMING AND PROOF SYSTEM

- Project began in 1989 but goes back to earliest Boyer-Moore provers from the early 1970s.
- Programming language supports efficient execution via any of six Common Lisp compilers.
- Remains under active development (maintaining extensive libraries, documentation, proof debugging capabilities, etc.).

Some organizations using ACL2:

A SEQUENCE OF CHECKERS

Table: Proof checking times in seconds on various inputs

Benchmark	[lrat-1] *(fast-alist)*	[lrat-3] *(shrink)*	[lrat-4] *(stobjs)*	[lrat-5] *(incremental)*
uuf-100-3	0.09	0.03	0.05	0.01
tph6[-dd]	3.08	0.57	0.33	0.33
R_4_4_18	164.74	5.13	2.23	2.24
transform	25.63	6.16	5.81	5.82
Schur_161_5_d43	5341.69	2355.26	840.04	259.82

A SEQUENCE OF CHECKERS (2)

How this work progressed (will elaborate on the next slides).

1. **[rat]** Our ITP 2013 RAT checker: no deletion
2. **[drat]** Added deletion (thus implementing DRAT)
3. **[lrat-1]** Avoid search and delete clauses efficiently, using *fast-alists* (applicative hash tables) and a *linear* proof format, and with soundness proved from scratch
4. **[lrat-2]** Shrink fast-alists to keep the formulas F_i small
5. **[lrat-3]** Minor tweak to formula data-structure
6. **[lrat-4]** Added *stobjs* for assignments
7. **[lrat-5]** Compression, incremental reading, improved soundness theorem

[drat]

Incorporating deletion was straightforward.

- In **[rat]**, a proof is a list of clauses to be added (no deletion).
- A **[drat]** proof is a list of pairs $\langle b, c \rangle$, where b is a Boolean deletion flag and c is a clause.
- We easily modified our ITP 2013 proof.

Deletion improves speed by keeping the formulas F_i small.

But the **[drat]** checker is still slow. **Why?**

- *Unit propagation* (UP) results in many linear searches through F_i.
- Deletion does a linear search and a lot of **cons**ing.

The LRAT Proof Format

Together with others, we developed a *Linear RAT* (LRAT) proof format [1].

Hints direct exactly where unit propagation is done – no search! This addresses the first of the two “Why It’s Slow” problems.

Again:

- *Unit propagation* (UP) results in many linear searches through F_i.
- Deletion does a linear search and a lot of **cons**ing.

Clause indices help solve the second problem.

The remaining checkers implement these efficiencies.

[lrat-1], [lrat-2], AND [lrat-3]

- Proof steps represent the LRAT format.
- We used *fast-alists*, an ACL2 hash-table data structure.
- Unit propagation benefits from fast lookup of clauses.
- How to manage the big change from **[drat]** to **[lrat-1]**?
  - Painful to rework another's proof
  - Decision: Sketch hand proof and carry out a fresh proof
  - Used top-down approach
- Profiling showed 69% of the time inside *hons-get* in **[lrat-1]**.
- The RAT check visits *every* clause in the formula F_i.
- Shrink the formula's fast-alist when heuristics say to do so.

[lrat-4]

A bottleneck in **[lrat-3]**: evaluation of a literal n requires a linear-time search for either n or $-n$ in the assignment.

[lrat-4] solution: use **single-threaded objects** (*stobjs*) to model assignments.

- Lookup is a constant-time array reference.
- Avoids memory allocation (consing) when pushing new literals onto assignment.

Tweaking the **[lrat-3]** proof seemed difficult! Instead....

- We proved *correspondence theorems* relating **[lrat-3]** functions to **[lrat-4]** functions.
- Soundness of **[lrat-4]** follows from soundness of **[lrat-3]**.

[lrat-5]

- Uses the compressed LRAT format, for which size is 25%-35% of uncompressed LRAT
- Supports incremental reading and checking, thereby significantly lowering the memory footprint
- Generalizes the proof checking to partial proofs
- Optionally emits the unsatisfiable formula to deal with parsing trust issues. Uses `diff` to compare with input.

Verified checker used to certify "the largest math proof ever"

- Proof production (solving) time: 13,516 CPU hours
- Proof conversion time (into CLRAT): 22,605 CPU hours
- Proof certification time (using ACL2): 8,651 CPU hours

OUTLINE

INTRODUCTION
The Problem
Propositional Proofs
Efficient Proof-checking

A SEQUENCE OF CHECKERS
The ACL2 Theorem-Proving System
The Input Format
[lrat-1] to **[lrat-5]**

CONCLUSION
Overview
References

CONCLUSION

Verification of unsatisfiability results can now be achieved with reasonable overhead and high confidence in correctness:

- It is easy to emit proofs in a SAT solver;
- The complex checking produces hints for efficient checks;
- A highly trusted checker certifies the result.

All supporting materials for the presented checkers, including proofs, may be found in the `projects/sat/lrat/` directory within the *ACL2 community books*; see its README file.

The technology is now ready for real-world applications:

- This tool chain is already used in industry (at Centaur);
- Huge proofs of mathematical theorems can be certified;
- SAT 2017 Competition used our tools to validate all results.

[1] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified rat verification. In *Automated Deduction – CADE 26*, pages 220–236, Cham, 2017. Springer International Publishing.

[2] Marijn Heule. The DRAT format and DRAT-trim checker. CoRR, abs/1610.06229, 2016. Source code available from:
`https://github.com/marijnheule/drat-trim`.

[3] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *LNCS*, pages 345–359. Springer, 2013.

[4] Peter Lammich. Efficient verified (un)sat certificate checking. In *Automated Deduction – CADE 26*, pages 237–254, Cham, 2017. Springer International Publishing.