Development of a Verified, Efficient Checker for SAT Proofs

Matt Kaufmann
(In collaboration with Marijn Heule and Warren Hunt, Jr.)

ACL2 Seminar
The University of Texas at Austin

February 3, 2017
ABSTRACT

I’ll 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
Towards a Solution
Clauses
Semantics: Assignments and Truth
Proofs
Formalizing Soundness
Efficient Proof-checking

A SEQUENCE OF CHECKERS
[drat]
The LRAT Proof Format
[lrat-1]
[lrat-2]
[lrat-3]
[lrat-4]

CONCLUSION
REFERENCES
OUTLINE

INTRODUCTION

The Problem
Towards a Solution
Clauses
Semantics: Assignments and Truth
Proofs
Formalizing Soundness
Efficient Proof-checking

A SEQUENCE OF CHECKERS

[drat]
The LRAT Proof Format
[lrat-1]
[lrat-2]
[lrat-3]
[lrat-4]

CONCLUSION

REFERENCES
Questions welcome during the talk, feedback afterwards.
Questions welcome during the talk, feedback afterwards.

Feel free to slow me down (will move quickly through early stuff that is probably familiar to all).
Questions welcome during the talk, feedback afterwards.

Feel free to slow me down (will move quickly through early stuff that is probably familiar to all).

Brief summary of talk:
Questions welcome during the talk, feedback afterwards.
Feel free to slow me down (will move quickly through early stuff that is probably familiar to all).

Brief summary of talk:

- Nathan Wetzler wrote and verified an ACL2 program that validates SAT proofs.
Questions welcome during the talk, feedback afterwards.
Feel free to slow me down (will move quickly through early stuff that is probably familiar to all).

Brief summary of talk:

- Nathan Wetzler wrote and verified an ACL2 program that validates SAT proofs.
- This talk discusses development of an efficient such verified checker.
Questions welcome during the talk, feedback afterwards.

Feel free to slow me down (will move quickly through early stuff that is probably familiar to all).

Brief summary of talk:

▶ Nathan Wetzler wrote and verified an ACL2 program that validates SAT proofs.

▶ This talk discusses development of an efficient such verified checker.

Underlining denotes links to the [ACL2+books online manual](https://acl2.org/).
THE PROBLEM

Boolean Satisfiability (SAT) solvers are proliferating and useful.
THE PROBLEM

Boolean Satisfiability (SAT) solvers are proliferating and useful. But how can we *trust* them?
**THE PROBLEM**

Boolean Satisfiability (SAT) solvers are proliferating and useful. But how can we *trust* them? Modern ones [3] admit *proofs*!
THE PROBLEM

Boolean Satisfiability (SAT) solvers are proliferating and useful. But how can we trust them?

Modern ones [3] admit proofs!

But how do we know that these “proofs” are valid?
THE PROBLEM

Boolean Satisfiability (SAT) solvers are proliferating and useful. But how can we *trust* them?

Modern ones [3] admit *proofs*!

But how do we know that these “proofs” are valid?

We check them with software programs called *checkers*!
The Problem

Boolean Satisfiability (SAT) solvers are proliferating and useful. But how can we trust them?

Modern ones [3] admit 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?
The Problem

Boolean Satisfiability (SAT) solvers are proliferating and useful.
But how can we trust them?
Modern ones [3] admit 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...
The Problem

Boolean Satisfiability (SAT) solvers are proliferating and useful.

But how can we *trust* them?

Modern ones [3] admit *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*. 
Towards a Solution

Nathan Wetzler, under the direction of Marijn Heule and Warren Hunt, developed an ACL2-based solution [6, 5, 4].
TOWARDS A SOLUTION

Nathan Wetzler, under the direction of Marijn Heule and Warren Hunt, developed an ACL2-based solution [6, 5, 4].

He wrote a SAT proof-checker in ACL2, then formalized and proved its correctness (soundness):
Towards a Solution

Nathan Wetzler, under the direction of Marijn Heule and Warren Hunt, developed an ACL2-based solution [6, 5, 4]. He wrote a SAT proof-checker in ACL2, then formalized and proved its correctness (soundness):

Suppose the checker takes inputs \( p \) (an alleged proof) and \( F \) (a formula), and checks that \( p \) legally derives a contradiction from \( F \).
Towards a Solution

Nathan Wetzler, under the direction of Marijn Heule and Warren Hunt, developed an ACL2-based solution \([6, 5, 4]\).

He wrote a SAT proof-checker in ACL2, then formalized and proved its correctness (soundness):

Suppose the checker takes inputs \(p\) (an alleged proof) and \(F\) (a formula), and checks that \(p\) legally derives a contradiction from \(F\).

Then \(F\) is always false.
TOWARDS A SOLUTION

Nathan Wetzler, under the direction of Marijn Heule and Warren Hunt, developed an ACL2-based solution [6, 5, 4]. He wrote a SAT proof-checker in ACL2, then formalized and proved its correctness (soundness):

Suppose the checker takes inputs $p$ (an alleged proof) and $F$ (a formula), and checks that $p$ legally derives a contradiction from $F$.

Then $F$ is always false.

Background:
CLauses

A variable is a propositional atom.
**Clauses**

A *variable* is a propositional atom.

- Traditionally, a Boolean formula might be \( P_1 \land \neg P_2 \), where \( P_1 \) and \( P_2 \) are symbols known as *variables*.
- For us, *variables* are *positive integers*. 
**Clauses**

A *variable* is a propositional atom.

- Traditionally, a Boolean formula might be $P_1 \land \neg P_2$, where $P_1$ and $P_2$ are symbols known as *variables*.
- For us, *variables* are *positive integers*.

A *literal* is a variable or its negative (negation), e.g., 3 or -3.
Clauses

A variable is a propositional atom.

- Traditionally, a Boolean formula might be $P_1 \land \neg P_2$, where $P_1$ and $P_2$ are symbols known as variables.
- For us, variables are positive integers.

A literal is a variable or its negative (negation), e.g., 3 or -3.

Complementary literals are negations of each other.
**Clauses**

A *variable* is a propositional atom.

- Traditionally, a Boolean formula might be $P_1 \land \neg P_2$, where $P_1$ and $P_2$ are symbols known as *variables*.
- For us, *variables* are *positive integers*.

A *literal* is a variable or its negative (negation), e.g., 3 or -3.

*Complementary* literals are negations of each other.

A *clause* is a set of literals, implicitly disjoined, containing no complementary literals.
Clauses

A variable is a propositional atom.

- Traditionally, a Boolean formula might be $P_1 \land \neg P_2$, where $P_1$ and $P_2$ are symbols known as variables.
- For us, variables are positive integers.

A literal is a variable or its negative (negation), e.g., 3 or -3.

Complementary literals are negations of each other.

A clause is a set of literals, implicitly disjoined, containing no complementary literals.

- In ACL2: duplicate-free lists of non-zero integers without complementary literals. Example: $(3\ 7\ -2\ 4)$. 
Clauses

A variable is a propositional atom.

- Traditionally, a Boolean formula might be $P_1 \land \neg P_2$, where $P_1$ and $P_2$ are symbols known as variables.
- For us, variables are positive integers.

A literal is a variable or its negative (negation), e.g., 3 or -3.

Complementary literals are negations of each other.

A clause is a set of literals, implicitly disjoined, containing no complementary literals.

- In ACL2: duplicate-free lists of non-zero integers without complementary literals. Example: (3 7 -2 4).

A formula is a set (or list) of clauses, implicitly conjoined. (This is commonly called conjunctive normal form.)
SEMANTICS: ASSIGNMENTS AND TRUTH

An assignment is a finite function mapping variables to Booleans.
Semantics: Assignments and Truth

An assignment is a finite function mapping variables to Booleans.

- In ACL2: same representation as for clauses, e.g.,
  \[(3 \ 7 \ -2 \ 4)\].
**Semantics: Assignments and Truth**

An assignment is a finite function mapping variables to Booleans.

- In ACL2: same representation as for clauses, e.g.,
  \[(3\ 7\ -2\ 4)\].

**Truth value under an assignment:** recursively defined for literals, then clauses, then formulas, to be \(T\), \(NIL\), or \(0\) (unknown).
SEMANTICS: ASSIGNMENTS AND TRUTH

An assignment is a finite function mapping variables to Booleans.

- In ACL2: same representation as for clauses, e.g.,
  \((3\ 7\ -2\ 4)\).

Truth value under an assignment: recursively defined for literals, then clauses, then formulas, to be \(T\), \(NIL\), or \(0\) (unknown).

Example: Is \(F\) true under assignment \(a\)?
\(F: \(\((1\ 7\ -2)\ (-3\ -5\ 6)\ (9\ 2\ 3)\)\)
\(a: \(7\ -3)\)
**Semantics: Assignments and Truth**

An *assignment* is a finite function mapping variables to Booleans.

- In ACL2: same representation as for clauses, e.g.,
  
  \[(3 \; 7 \; -2 \; 4)\].

*Truth value under an assignment*: recursively defined for literals, then clauses, then formulas, to be \(T\), \(NIL\), or \(0\) (unknown).

Example: Is \(F\) true under assignment \(a\)?

\[
F: \;
(\;
(1 \; 7 \; -2) \;
(-3 \; -5 \; 6) \;
(9 \; 2 \; 3) \;
)\]

\[
a: \;
(7 \; -3)\]

Answer: No — the truth value is 0 because of the third clause.
**SEMANTICS: ASSIGNMENTS AND TRUTH**

An *assignment* is a finite function mapping variables to Booleans.

- In ACL2: same representation as for clauses, e.g.,
  
  \[(3 \ 7 \ -2 \ 4)\].

*Truth value under an assignment*: recursively defined for literals, then clauses, then formulas, to be `T`, `NIL`, or `0` (unknown).

Example: Is \(F\) true under assignment \(a\)?

\[F: \ ((1 \ 7 \ -2) \ (-3 \ -5 \ 6) \ (9 \ 2 \ 3))\]

\[a: \ (7 \ -3)\]

Answer: No — the truth value is 0 because of the third clause.

A formula is *satisfiable* if it is *true* under some assignment; otherwise, it is *unsatisfiable.*
A proof (or clausal proof, or refutation) for a formula $F$ is a sequence $p = \langle p_1, p_2, \ldots, p_k \rangle$ such that:
A proof (or clausal proof, or refutation) for a formula $F$ is a sequence $p = \langle p_1, p_2, \ldots, p_k \rangle$ such that:

- Each $p_i$ is $\langle b_i, c_i \rangle$, where $b_i$ is a Boolean and $c_i$ is a clause.
A proof (or clausal proof, or refutation) for a formula $F$ is a sequence $p = \langle p_1, p_2, \ldots, p_k \rangle$ 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
PROOFS

A proof (or clausal proof, or refutation) for a formula $F$ is a sequence $p = \langle p_1, p_2, \ldots, p_k \rangle$ 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.
PROOFS

A proof (or clausal proof, or refutation) for a formula $F$ is a sequence $p = \langle p_1, p_2, \ldots, p_k \rangle$ 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.

- All addition steps preserve satisfiability (see next slide).
For \( p = \langle p_1, p_2, ..., p_k \rangle \) as above, recursively define formulas \( \langle F_0, F_1, ..., F_k \rangle \) by executing the \( p_i \):
Proofs (2)

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

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

- \( F_0 = F \).
- For \( b_i \) true, delete \( c_{i+1} \) from \( F_i \) to get \( F_{i+1} \).
Proofs (2)

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

- $F_0 = F$.
- For $b_i$ true, delete $c_{i+1}$ from $F_i$ to get $F_{i+1}$.
- For $b_i$ false, add $c_{i+1}$ to $F_i$ to get $F_{i+1}$.
PROOFS (2)

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

- $F_0 = F$.
- For $b_i$ true, delete $c_{i+1}$ from $F_i$ to get $F_{i+1}$.
- For $b_i$ false, add $c_{i+1}$ to $F_i$ to get $F_{i+1}$.

Then $p$ preserves satisfiability when for each addition step $p_i$, if $F_{i-1}$ is satisfiable then $F_i$ is satisfiable.
NOTE: The definition above of clausal proof is very general. A checker may impose more specific syntactic requirements that guarantee the property.
NOTE: The definition above of clausal proof is very general. A checker may impose more specific syntactic requirements that guarantee the property.

The next slide shows Nathan’s formalization based on the RAT (Reduced Asymmetric Tautology) check. Details on RAT are not the subject of today’s talk.
NOTE: The definition above of clausal proof is very general. A checker may impose more specific syntactic requirements that guarantee the property.

The next slide shows Nathan’s formalization based on the RAT (Reduced Asymmetric Tautology) check. Details on RAT are not the subject of today’s talk.

All checkers discussed today use a formalization like the one on the next slide, based on RAT.
FORMALIZING SOUNDNESS

Below, \texttt{proofp} is a recognizer for proofs, and \texttt{solutionp} checks that a formula is true under a given assignment,

\begin{verbatim}
(defun refutationp (proof formula)
  (declare (xargs :guard (formulap formula)))
  (and (proofp proof formula)
       (member *empty-clause* proof)))

(defun-sk exists-solution (formula)
  (exists assignment
      (solutionp assignment formula)))

(defun main-theorem
  (implies (and (formulap formula)
                 (refutationp clause-list formula))
           (not (exists-solution formula))))
\end{verbatim}
**FORMALIZING SOUNDNESS (2)**

The following is easily proved by induction.

**Lemma.** Suppose that $p = \langle p_1, p_2, ..., p_k \rangle$ is a proof and $F_0$ is satisfiable. Then each $F_i$ is satisfiable.
FORMALIZING SOUNDNESS (2)

The following is easily proved by induction.

**Lemma.** Suppose that \( p = \langle p_1, p_2, ..., p_k \rangle \) is a proof and \( F_0 \) is satisfiable. Then each \( F_i \) is satisfiable.

Soundness argument:
The following is easily proved by induction.

**Lemma.** Suppose that \( p = \langle p_1, p_2, \ldots, p_k \rangle \) is a proof and \( F_0 \) is satisfiable. Then each \( F_i \) is satisfiable.

Soundness argument:

1. Deletion steps clearly preserve satisfiability.
2. **Addition steps preserve satisfiability.** [Must be proved!]
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:** Nathan’s checker was intended to be a proof of concept, not an efficient tool. On one example:
Efficient Proof-checking

HOWEVER: Nathan’s checker was intended to be a proof of concept, not an efficient tool. On one example:

- Marijn’s checker: 1.5 seconds
Efficient Proof-checking

HOWEVER: Nathan’s checker was intended to be a proof of concept, not an efficient tool. On one example:

- Marijns’s checker: 1.5 seconds
- Nathan’s checker: 1 week
Efficient Proof-checking

However: Nathan’s checker was intended to be a proof of concept, not an efficient tool. On one example:

- Marijn’s checker: 1.5 seconds
- Nathan’s checker: 1 week

Marijn’s request: a formally verified checker for SAT competitions
EFFICIENT PROOF-CHECKING

HOWEVER: Nathan’s checker was intended to be a proof of concept, not an efficient tool. On one example:

- Marijn’s checker: 1.5 seconds
- Nathan’s checker: 1 week

Marijn’s request: a formally verified checker for SAT competitions

This talk tells the (true) story of the development of such a checker.
Efficient Proof-checking

HOWEVER: Nathan’s checker was intended to be a proof of concept, not an efficient tool. On one example:

- Marijn’s checker: 1.5 seconds
- Nathan’s checker: 1 week

Marijn’s request: a formally verified checker for SAT competitions

This talk tells the (true) story of the development of such a checker.

- Its efficiency benefits in part from some techniques not yet invented at the time of Nathan’s work.
The flow for efficient, verified SAT proof-checking:
Efficient Proof-checking (2)

The flow for efficient, verified SAT proof-checking:

1. SAT solver verifies unsatisfiability of formula $F$; generates alleged proof, $p_0$. 
Efficient Proof-checking (2)

The flow for efficient, verified SAT proof-checking:

1. SAT solver verifies unsatisfiability of formula $F$; generates alleged proof, $p_0$.

2. DRAT-Trim [2] consumes $p_0$, outputs alleged proof $p_1$ for checker, in a format amenable to efficient checking.
Efficient Proof-checking (2)

The flow for efficient, verified SAT proof-checking:

1. SAT solver verifies unsatisfiability of formula $F$; generates alleged proof, $p_0$.

2. *DRAT-Trim* [2] consumes $p_0$, outputs alleged proof $p_1$ for checker, in a format amenable to efficient checking.

3. Verified ACL2 checker validates that $p_1$ is a proof for $F$. 
A SEQUENCE OF CHECKERS

This table shows times (in seconds) for some checker runs, on examples provided by Marijn.

<table>
<thead>
<tr>
<th>test</th>
<th>[rat] (Wetzler)</th>
<th>[drat] (deletion)</th>
<th>[lrat-1] (fast-alist)</th>
<th>[lrat-2] (shrink)</th>
<th>[lrat-3] (clean up)</th>
<th>[lrat-4] (stobjs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>uuf-100-3</td>
<td>20.64</td>
<td>8.59</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.00</td>
</tr>
<tr>
<td>tph6[-dd]</td>
<td>-</td>
<td>-</td>
<td>6.18</td>
<td>0.56</td>
<td>0.54</td>
<td>0.46</td>
</tr>
<tr>
<td>R_4_4_18</td>
<td>∼1 week</td>
<td>-</td>
<td>217.91</td>
<td>9.62</td>
<td>3.21</td>
<td>2.56</td>
</tr>
<tr>
<td>transform</td>
<td>-</td>
<td>-</td>
<td>47.80</td>
<td>9.59</td>
<td>8.82</td>
<td>8.77</td>
</tr>
<tr>
<td>schur</td>
<td>-</td>
<td>-</td>
<td>4674.18</td>
<td>1872.07</td>
<td>1884.23</td>
<td>246.94</td>
</tr>
</tbody>
</table>

Times do not include parsing. Warren Hunt has sped up our original parser, and there are plans to speed it up further by using a binary proof format (not discussed further here).
A Sequence of Checkers

This table shows times (in seconds) for some checker runs, on examples provided by Marijn.

<table>
<thead>
<tr>
<th>test</th>
<th>[rat] (Wetzler)</th>
<th>[drat] (deletion)</th>
<th>[lrat-1] (fast-alist)</th>
<th>[lrat-2] (shrink)</th>
<th>[lrat-3] (clean up)</th>
<th>[lrat-4] (stobjs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>uuf-100-3</td>
<td>20.64</td>
<td>8.59</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.00</td>
</tr>
<tr>
<td>tph6[-dd]</td>
<td></td>
<td>-</td>
<td>6.18</td>
<td>0.56</td>
<td>0.54</td>
<td>0.46</td>
</tr>
<tr>
<td>R_4_4_18</td>
<td>~1 week</td>
<td>-</td>
<td>217.91</td>
<td>9.62</td>
<td>3.21</td>
<td>2.56</td>
</tr>
<tr>
<td>transform</td>
<td>-</td>
<td>-</td>
<td>47.80</td>
<td>9.59</td>
<td>8.82</td>
<td>8.77</td>
</tr>
<tr>
<td>schur</td>
<td>-</td>
<td>-</td>
<td>4674.18</td>
<td>1872.07</td>
<td>1884.23</td>
<td>246.94</td>
</tr>
</tbody>
</table>

Times do not include parsing. Warren Hunt has sped up our original parser, and there are plans to speed it up further by using a *binary proof format* (not discussed further here).
A SEQUENCE OF CHECKERS (2)

How this work progressed (will elaborate on the next slides).
A Sequence of Checkers (2)

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

1. [rat] Nathan’s RAT checker: no deletion
A SEQUENCE OF CHECKERS (2)

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

1. [rat] Nathan’s RAT checker: no deletion
2. [drat] Added deletion (thus implementing DRAT)
How this work progressed (will elaborate on the next slides).

1. **[rat]** Nathan’s 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
A Sequence of Checkers (2)

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

1. [rat] Nathan’s 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
A SEQUENCE OF CHECKERS (2)

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

1. [rat] Nathan’s 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
A SEQUENCE OF CHECKERS (2)

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

1. [rat] Nathan’s RAT checker: no deletion
2. [drat] Added deletion (thus implementing DRAT)
3. [lrat-1] Avoid search and delete clauses efficiently, using *fast-alist* (applicative hash tables) and a *linear* proof format, and with soundness proved from scratch
4. [lrat-2] Shrink *fast-alist* to keep the formulas $F_i$ small
5. [lrat-3] Minor tweak to formula data-structure
6. [lrat-4] Added *stobjs* for assignments
A Sequence of Checkers (3)

Acknowledgments:
A Sequence of Checkers (3)

Acknowledgments:

▶ Marijn helped a lot with getting us up to speed on SAT proof-checking based on RAT, and by supplying examples.
A SEQUENCE OF CHECKERS (3)

Acknowledgments:

- Marijn helped a lot with getting us up to speed on SAT proof-checking based on RAT, and by supplying examples.
- Warren worked with me in the initial stages.
A SEQUENCE OF CHECKERS (3)

Acknowledgments:

- Marijn helped a lot with getting us up to speed on SAT proof-checking based on RAT, and by supplying examples.
- Warren worked with me in the initial stages.

Profiling (Marijn’s suggestion) helped with discovering bottlenecks:

```
(include-book "centaur/memoize/old/profile"
 :dir :system)
(profile-acl2)
<evaluate forms>
(memsum)
```
A Sequence of Checkers (4)

This project illustrates the interplay between ACL2 as a programming language and as a theorem prover:
A Sequence of Checkers (4)

This project illustrates the interplay between ACL2 as a programming language and as a theorem prover:

- Optimize the program for efficiency.
This project illustrates the interplay between ACL2 as a programming language and as a theorem prover:

- Optimize the program for efficiency.
- Deal with proving correctness for the optimizations.
Incorporating deletion was straightforward.
Incorporating deletion was straightforward.

- In [rat], a proof is a list of clauses to be added (no deletion).
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 \( \langle b, c \rangle \) — in ACL2, \( (b . c) \), where \( b \) is a Boolean deletion flag and \( c \) is a clause.
Incorporating deletion was straightforward.

- In \textbf{[rat]}, a proof is a list of clauses to be added (no deletion).
- A \textbf{[drat]} proof is a list of pairs $\langle b, c \rangle$ — in ACL2, $(b . c)$, where $b$ is a Boolean deletion flag and $c$ is a clause.
- Warren and I easily modified Nathan’s proof.
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 $\langle b, c \rangle$ — in ACL2, $(b \ . \ c)$, where $b$ is a Boolean deletion flag and $c$ is a clause.
- Warren and I easily modified Nathan’s proof.

Deletion should help with speed by keeping the formulas $F_i$ small.
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 \( \langle b, c \rangle \) — in ACL2, \((b . ~ c)\), where \(b\) is a Boolean deletion flag and \(c\) is a clause.
- Warren and I easily modified Nathan’s proof.

Deletion should help with speed by keeping the formulas \(F_i\) small.

But the [drat] checker is still slow. **Why?**
[drat]: *Why It’s Slow*
[drat]: Why It’s Slow

- *Unit propagation* (UP) results in many linear searches through $F_i$. 
[drat]: Why It’s Slow

- Unit propagation (UP) results in many linear searches through $F_i$.

- Deletion does a linear search and much consing.
The LRAT Proof Format

Marijn, with others, has developed a *Linear RAT* (LRAT) proof format.[1]
The LRAT Proof Format

Marijn, with others, has developed a Linear RAT (LRAT) proof format.[1]

▶ “Others” includes 2 Coq users who have also developed a verified SAT proof-checker.
THE LRAT PROOF FORMAT

Marijn, with others, has developed a Linear RAT (LRAT) proof format.[1]

- “Others” includes 2 Coq users who have also developed a verified SAT proof-checker.
- Theirs takes 10 minutes on one example compared to our 9 seconds.
Marijn, with others, has developed a *Linear RAT* (LRAT) proof format.[1]

- “Others” includes 2 Coq users who have also developed a verified SAT proof-checker.
- Theirs takes 10 minutes on one example compared to our 9 seconds.

Example LRAT proof step $p_i$: 

```plaintext
820 -59 -17 -58 0 807 246 423 40 -87 308 117 819 809 404 310 -163 -313 0
```
THE LRAT PROOF FORMAT

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

- “Others” includes 2 Coq users who have also developed a verified SAT proof-checker.
- Theirs takes 10 minutes on one example compared to our 9 seconds.

Example LRAT proof step $p_i$:

```
820  -59  -17  -58  0  807  246  423  40  -87  308  117  819  809  404  310  -163  -313  0
```
THE LRAT PROOF FORMAT

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

- “Others” includes 2 Coq users who have also developed a verified SAT proof-checker.
- Theirs takes 10 minutes on one example compared to our 9 seconds.

Example LRAT proof step $p_i$:

```
820 -59 -17 -58 0 807 246 423 40 -87 308 117 819 809 404 310 -163 -313 0
```

The next slide breaks this line apart.
The clause to be added has index 820:

820
The clause to be added has index 820:
820

It is the set of literals, \{-59, -17, -58\}:
-59, -17, -58
The clause to be added has index 820:

820

It is the set of literals, \{-59 \ -17 \ -58\}:

-59 \ -17 \ -58

Separator:

0
The clause to be added has index 820:

820

It is the set of literals, \{-59 \ -17 \ -58\}:

-59  -17  -58

Separator:

0

Apply unit propagation (UP) to these four clauses, in order:

807  246  423  40
The clause to be added has index 820:
820

It is the set of literals, \{ -59, -17, -58 \}:
-59  -17  -58

Separator:
0

Apply unit propagation (UP) to these four clauses, in order:
807  246  423  40

For the RAT check on clause 87, restrict UP to the clauses 308, 117, ..., and 310, in order.
For the RAT check on clause 163, no UP is performed.
For the RAT check on clause 313, no UP is performed.
-87  308  117  819  809  404  310  -163  -313
The clause to be added has index 820:

820

It is the set of literals, \{-59 \ -17 \ -58\}:

-59 \ -17 \ -58

Separator:

0

Apply unit propagation (UP) to these four clauses, in order:

807 246 423 40

For the RAT check on clause 87, restrict UP to the clauses 308, 117, ..., and 310, in order.

For the RAT check on clause 163, no UP is performed.

For the RAT check on clause 313, no UP is performed.

-87 308 117 819 809 404 310 -163 -313

End of proof step:

0
**The LRAT Proof Format (The Big Take-away)**

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

Again:
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 much consing.
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 much consing.

Clause indices help solve the second problem.
THE LRAT PROOF FORMAT (THE BIG TAKE-AWAY)

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 much consing.

Clause indices help solve the second problem.
The next checker implements these efficiencies.
[lrat-1]
[lrat-1]

- Proof steps represent the LRAT format.
Proof steps represent the **LRAT format**.

- A formula represents a list of pairs \((i \quad c)\) where \(i\) is a natural number, the *index* of clause \(c\).
[lrat-1]

- Proof steps represent the LRAT format.

- A formula represents a list of pairs \( (i . \ c) \) where \( i \) is a natural number, the index of clause \( c \).

  - This list is a fast-alist: ACL2 uses a hash-table to find \( c \) from \( i \) in essentially constant time.
Proof steps represent the **LRAT format**.

- A formula represents a list of pairs \((i \ . \ c)\) where \(i\) is a natural number, the **index** of clause \(c\).
  - This list is a **fast-alist**: ACL2 uses a hash-table to find \(c\) from \(i\) in essentially constant time.
  - A formula is a pair \((\text{max} \ . \ \text{fal})\), where \(\text{fal}\) is its fast-alist and \(\text{max}\) is an upper bound on its indices.
How do fast-alists help with efficiency?
[lrat-1] (2)

How do fast-alists help with efficiency?

- Unit propagation benefits from fast lookup to obtain a clause from its index; and
How do fast-alists help with efficiency?

- Unit propagation benefits from fast lookup to obtain a clause from its index; and

- Deletion of clause $i$ simply extends the fast-alist with a pair $(i, *\text{deleted-clause}*)$. 

\[\text{lrat-1} \ (2)\]
How do fast-alists help with efficiency?

- Unit propagation benefits from fast lookup to obtain a clause from its index; and

- Deletion of clause $i$ simply extends the fast-alist with a pair $(i . *\text{deleted-clause}*).$
  - The value of $*\text{deleted-clause}* \text{ is a non-nil atom,} \text{ hence not a clause.}$
[lrat-1]: PROOF

Proof Problem: How to manage the substantial change from [drat] to [lrat-1].
[lrat-1]: PROOF

Proof Problem: How to manage the substantial change from [drat] to [lrat-1].

- Painful to rework another’s proof
Proof Problem: How to manage the substantial change from [drat] to [lrat-1].

- Painful to rework another’s proof
- Decision: Sketch hand proof and manage a fresh proof
[lrat-1]: PROOF

Proof Problem: How to manage the substantial change from [drat] to [lrat-1].

▶ Painful to rework another’s proof
▶ Decision: Sketch hand proof and manage a fresh proof
▶ Used top-down approach (see my 1999 ACL2 Workshop paper)
satisfiable-add-proof-clause.lisp

<hand proof in comment>
(in-package "ACL2")
(include-book "lrat-checker")

(local (encapsulate ()
  (local (include-book "satisfiable-add-proof-clause-rup"))
  (local (include-book "satisfiable-add-proof-clause-drat"))
  (set-enforce-redundancy t)
  (defthm satisfiable-add-proof-clause-rup
    ...
  )
  (defthm satisfiable-add-proof-clause-drat
    ...
 ))

(defthm satisfiable-add-proof-clause
  ...
  :hints
  ("Goal" :use (satisfiable-add-proof-clause-rup
                 satisfiable-add-proof-clause-drat)
           :in-theory (union-theories '(verify-clause)
                                      (theory 'minimal-theory)))))
Profiling showed 69% of the time inside `hons-get` (looking up clause indices).
Profiling showed 69% of the time inside `hons-get` (looking up clause indices).

The RAT check visits every clause in the formula $F_i$. 
[lrat-2]

Profiling showed 69% of the time inside `hons-get` (looking up clause indices).

The RAT check visits *every* clause in the formula $F_i$.

The [lrat-2] checker improves on [lrat-1] in two ways:
[lrat-2]

Profiling showed 69% of the time inside \texttt{hons-get} (looking up clause indices).

The RAT check visits every clause in the formula $F_i$.

The [lrat-2] checker improves on [lrat-1] in two ways:

- Shrink the formula’s fast-alist when heuristics say to do so.
Profiling showed 69% of the time inside \texttt{hons-get} (looking up clause indices).

The RAT check visits \textit{every} clause in the formula $F_i$.

The \texttt{lrat-2} checker improves on \texttt{lrat-1} in two ways:

- Shrink the formula’s fast-alist when heuristics say to do so.
- RAT check recurs through the fast-alist instead of recurring down from the max index.
[lrat-2]: SHRINKING

Two counts maintained on the formula:
[lrat-2]: SHRINKING

Two counts maintained on the formula:

- $ndel$: number of pairs (i.e., ${\text*{deleted-clause}}$)
[lrat-2]: SHRINKING

Two counts maintained on the formula:

- \( n_{del} \): number of pairs \((i \cdot \text{deleted-clause})\)
- \( n_{cls} \): the number of pairs \((i \cdot \ c)\) representing clauses that have not been deleted
**[lrat-2]: SHRINKING**

Two counts maintained on the formula:

- $ndel$: number of pairs $i \cdot \text{deleted-clause}$
- $ncls$: the number of pairs $i \cdot c$ representing clauses that have not been deleted

Heuristically shrink the fast-alist at an addition proof step, based on experimentation:
[lrat-2]: SHRINKING

Two counts maintained on the formula:

- $ndel$: number of pairs $(i \cdot \text{*deleted-clause*})$
- $ncls$: the number of pairs $(i \cdot c)$ representing clauses that have not been deleted

Heuristically shrink the fast-alist at an addition proof step, based on experimentation:

- whenever $ndel > 10 \times ncls$;
[lrat-2]: SHRINKING

Two counts maintained on the formula:

- \( n_{del} \): number of pairs \( (i \ . \ *\text{deleted-clause}* ) \)
- \( n_{cls} \): the number of pairs \( (i \ . \ c) \) representing clauses that have not been deleted

Heuristically shrink the fast-alist at an addition proof step, based on experimentation:

- whenever \( n_{del} > 10 \times n_{cls} \);
- when RAT check is necessary, shrink first if \( n_{del} > 1/3 \times n_{cls} \).
To shrink a fast-alist (will discuss only if time):

```
(defun remove-deleted-clauses (fal acc)
  (declare (xargs :guard (alistp fal)))
  (cond ((endp fal) (make-fast-alist acc))
        (t (remove-deleted-clauses
            (cdr fal)
            (if (deleted-clause-p (cdar fal))
                acc
                (cons (car fal) acc))))))
```

```
(defund shrink-formula-fal (fal)
  (declare (xargs :guard (formula-fal-p fal)))
  (let ((fal2 (fast-alist-clean fal)))
    (fast-alist-free-on-exit
     fal2
     (remove-deleted-clauses fal2 nil))))
```
[lrat-2]: PROOF

Proved soundness by tweaking the [lrat-1] proof:
[lrat-2]: PROOF

Proved soundness by tweaking the [lrat-1] proof:

▶ Disabled the top-level “maybe shrink” function
[lrat-2]: PROOF

Proved soundness by tweaking the [lrat-1] proof:

- Disabled the top-level “maybe shrink” function
- Re-ran the [lrat-1] proof on [lrat-2]
[lrat-2]: PROOF

Proved soundness by tweaking the [lrat-1] proof:

- Disabled the top-level “maybe shrink” function
- Re-ran the [lrat-1] proof on [lrat-2]
- Looked at key checkpoints on failure to determine lemmas to prove (about shrinking).
[lrat-3]

Changed formula from \((\max \ . \ fal)\) to simply \(fal\).
Changed formula from (max . fal) to simply fal.

- Max was only used for RAT check recursion, but [lrat-2] recurs through fal.
Changed formula from $(\max . \ fal)$ to simply $fal$.

- $\max$ was only used for RAT check recursion, but [lrat-2] recurs through $fal$.

- This simplification seemed useful before starting the next checker, and it saves consing.
Changed formula from \((\max . \text{fal})\) to simply \(\text{fal}\).

- \(\text{Max}\) was only used for RAT check recursion, but \([\text{lrat-2}]\) recurs through \(\text{fal}\).

- This simplification seemed useful before starting the next checker, and it saves consing.

- Soundness proof for \([\text{lrat-2}]\) was easy to modify for \([\text{lrat-3}]\).
A bottleneck in [lrat-3]: evaluation of a literal $n$ requires a linear-time search for either $n$ or $-n$ in the assignment.
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 ($\text{stobjs}$) to model assignments.
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.
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.
[lrat-4]: ASSIGNMENTS

(defstobj a$
  (a$ptr :type (integer 0 *) ; stack pointer
   :initially 0)
  (a$stk :type (array t (1)) ; stack of a$arr indices
     :resizable t)
  (a$arr :type (array t (1)) ; array of 0, t, nil
     :initially 0
     :resizable t)
  :renaming ((a$arrp a$arrp-weak)
     (a$p a$p-weak)))
[lrat-4]: ASSIGNMENTS (2)

Operations on assignments:
[lrat-4]: ASSIGNMENTS (2)

Operations on assignments:

- (push-literal lit a$) extends assignment a$ with literal lit (writes to a$stk, increments a$ptr).
[lrat-4]: ASSIGNMENTS (2)

Operations on assignments:

- \((\text{push-literal } \text{lit } \text{a$})\) extends assignment \text{a$} with literal \text{lit} (writes to \text{a$stk}, increments \text{a$ptr}).

- \((\text{pop-literals } \text{ptr } \text{a$})\) updates \text{a$ptr} to \text{ptr}.
[lrat-4]: ASSIGNMENTS (2)

Operations on assignments:

- \((\text{push-literal lit a$})\) extends assignment a$ with literal lit (writes to a$stk, increments a$ptr).

- \((\text{pop-literals ptr a$})\) updates a$ptr to ptr.

**KEY OBSERVATION:** These operations generate calls to \(\text{nth}\) and \(\text{update-nth}\), but for [lrat-3], they are implemented with \(\text{cons}\) and \(\text{cdr}\).
[lrat-4]: ASSIGNMENTS (2)

Operations on assignments:

- (push-literal lit a$) extends assignment a$ with literal lit (writes to a$stk, increments a$ptr).

- (pop-literals ptr a$) updates a$ptr to ptr.

**KEY OBSERVATION:** These operations generate calls to nth and update-nth, but for [lrat-3], they are implemented with cons and cdr.

Tweaking the [lrat-3] proof seemed difficult! Instead....
[lrat-4]: PROOF

[lrat-4]: PROOF


- Then I derived the soundness of [lrat-4] directly from those correspondence theorems and the soundness of [lrat-3].
(defthm main-theorem-list-based
  (implies (and (formula-p formula)
                (refutation-p proof formula))
    (not (satisfiable formula)))
  :hints ...)

(defthm refutation-p-equiv
  (implies (and (formula-p formula)
                (refutation-p$ proof formula))
    (refutation-p proof formula)))

(defthm main-theorem-stobj-based
  (implies (and (formula-p formula)
                (refutation-p$ proof formula))
    (not (satisfiable formula)))
  :hints ("Goal"
    :in-theory '(refutation-p-equiv
                 :use main-theorem-list-based))
  :hints ...
All of these checkers are guard-verified, for runtime efficiency.
All of these checkers are \textit{guard}-verified, for runtime efficiency.

For that, needed \textit{invariant} on the stobj that is preserved when a function returns a modified stobj.
[lrat-4]: PROOF (3)

All of these checkers are **guard**-verified, for runtime efficiency.

For that, needed **invariant** on the stobj that is preserved when a function returns a modified stobj.

1. Developed that invariant, \((a \\triangleright p \ a)\)
All of these checkers are **guard**-verified, for runtime efficiency.

For that, needed **invariant** on the stobj that is preserved when a function returns a modified stobj.

1. Developed that invariant, \((a$\ p\ a$)\)

2. Verified guards (perhaps easier than correspondence theorems), which required invariance proofs
All of these checkers are guard-verified, for runtime efficiency.

For that, needed invariant on the stobj that is preserved when a function returns a modified stobj.

1. Developed that invariant, \((a\leq p \ a\geq)\)

2. Verified guards (perhaps easier than correspondence theorems), which required invariance proofs

3. Proved correspondence theorems
I’ll very briefly discuss the invariant:

(defun a$p (a$)
  (declare (xargs :stobjs a$))
  (and (a$p-weak a$)
       (<= (a$ptr a$) (a$stk-length a$))
       (equal (a$arr-length a$) (1+ (a$stk-length a$)))
       (good-stk-p (a$ptr a$) a$)
       (a$arrp a$)
       (arr-matches-stk (a$arr-length a$) a$)))
[lrat-4]: PROOF (5)

A challenge: The correspondence proofs broke down!
[lrat-4]: PROOF (5)

A challenge: The correspondence proofs broke down!

**[lrat-4]: PROOF (5)**

A challenge: The correspondence proofs broke down!

- Two [lrat-3] functions, \texttt{unit-propagation} and \texttt{rat-assignment}, match up nicely with corresponding [lrat-4] functions.

- One [lrat-3] function, \texttt{negate-clause-or-assignment}, did \textbf{not} match up with its corresponding [lrat-4] function.

\begin{verbatim}
(defun negate-clause-or-assignment (clause)
  (declare (xargs :guard (clause-or-assignment-p clause)))
  (if (atom clause)
      nil
    (cons (negate (car clause))
          (negate-clause-or-assignment (cdr clause))))
\end{verbatim}
[lrat-4]: PROOF (5)

A challenge: The correspondence proofs broke down!


- One [lrat-3] function, negate-clause-or-assignment, did not match up with its corresponding [lrat-4] function.

The [lrat-2] function (originally used in [lrat-3]):

```lisp
(defun negate-clause-or-assignment (clause)
  (declare (xargs :guard (clause-or-assignment-p clause)))
  (if (atom clause)
      nil
      (cons (negate (car clause))
            (negate-clause-or-assignment (cdr clause))))
)```
[lrat-4]: PROOF (6)

What to do? Status when problem was discovered:
[lrat-4]: PROOF (6)

What to do? Status when problem was discovered:

- Soundness for [lrat-3] was already established
- Guards for [lrat-4] were already verified.
- Some equivalence proofs were complete.
[lrat-4]: PROOF (6)

What to do? Status when problem was discovered:

- Soundness for [lrat-3] was already established
- Guards for [lrat-4] were already verified.
- Some equivalence proofs were complete.

Solution: Modified [lrat-3] by changing the definition of function \texttt{negate-clause-or-assignment} and fixing failed proofs.
What to do? Status when problem was discovered:

- Soundness for [lrat-3] was already established
- Guards for [lrat-4] were already verified.
- Some equivalence proofs were complete.

Solution: Modified [lrat-3] by changing the definition of function `negate-clause-or-assignment` and fixing failed proofs.

Then completed correspondence theorems, which yielded soundness for [lrat-4].
OUTLINE

INTRODUCTION
The Problem
Towards a Solution
Clauses
Semantics: Assignments and Truth
Proofs
Formalizing Soundness
Efficient Proof-checking

A SEQUENCE OF CHECKERS
[drat]
The LRAT Proof Format
[lrat-1]
[lrat-2]
[lrat-3]
[lrat-4]

CONCLUSION

REFERENCES
CONCLUSION

There is now an **efficient formally verified SAT checker**!
CONCLUSION

There is now an efficient formally verified SAT checker!

- On a large example, its time of 4.1 minutes (without parsing) compares very favorably with DRAT-trim time of 20 minutes (with very fast C parsing).
CONCLUSION

There is now an efficient formally verified SAT checker!

- On a large example, its time of 4.1 minutes (without parsing) compares very favorably with DRAT-trim time of 20 minutes (with very fast C parsing).

- Warren is working on a faster parser (it takes about 20 minutes with mine, which is based on read-object).
CONCLUSION

There is now an efficient formally verified SAT checker!

▶ On a large example, its time of 4.1 minutes (without parsing) compares very favorably with DRAT-trim time of 20 minutes (with very fast C parsing).

▶ Warren is working on a faster parser (it takes about 20 minutes with mine, which is based on read-object).

Checkers [lrat-3] and [lrat-4] are in the community books in these directories, respectively.

projects/sat/lrat/list-based/
projects/sat/lrat/stobj-based/
CONCLUSION

There is now an efficient formally verified SAT checker!

- On a large example, its time of 4.1 minutes (without parsing) compares very favorably with DRAT-trim time of 20 minutes (with very fast C parsing).

- Warren is working on a faster parser (it takes about 20 minutes with mine, which is based on read-object).

Checkers [lrat-3] and [lrat-4] are in the community books in these directories, respectively.

projects/sat/lrat/list-based/
projects/sat/lrat/stobj-based/

Other checkers are available via links from the seminar page.
This work can be found in the community books, with the latest version on github:

books/projects/sat/lrat/
This work can be found in the community books, with the latest version on github:

books/projects/sat/lrat/

Nathan Wetzler’s checker can also be found in the community books:

books/projects/sat/proof-checker-itp13/
REFERENCES

This work can be found in the community books, with the latest version on github:

books/projects/sat/lrat/

Nathan Wetzler’s checker can also be found in the community books:

books/projects/sat/proof-checker-itp13/

The next slide has references for citations in this talk.


