Proof Checking Technology for Satisfiability Modulo Theories

Aaron Stump

Computer Science and Engineering
Washington University
St. Louis, Missouri, USA

Funding from NSF CRI grant.
Satisfiability Modulo Theories (SMT) Solvers

- Support large formulas, expressive theories.
- Used for discharging verification conditions.
- Examples include Z3, YICES, CVC3, many others.
- SMT-LIB, SMT-COMP, SMT-EXEC.

\[ \Phi \]

\[ \text{SMT Solver} \]

\[ \text{Sat} \quad \text{Unsat} \]
Confirming Solver Results

- SMT solvers large (50-100kloc), complex.
- Hard to justify trusting.

One solution:

- Have solvers emit proofs.
- Check proofs with much simpler checker (2-4kloc).

```
Φ
```

```
SMT Solver
```

```
Proof Checker
```

```
Pf Ok  Pf Bad
```

Aaron Stump  Proof Checking for SMT  LFMTP 2008
Fast, Flexible, Standardized

- Speed required for large proofs (100MB to 100GB?).
- Flexibility also critical.
  - Different solving algorithms => different proof systems.
  - At least premature to pick a single proof system.
- A standard proof format very desirable.
  - Provides common target for solvers.
  - Opens door to exporting to interactive provers.
  - Standardization important to the SMT-LIB initiative.
Proposal: Standardize with a Logical Framework

- Start with Edinburgh Logical Framework (LF) [Harper+ 93].
- LF provides flexibility.
  - Logics described by a *signature*.
  - One proof checker suffices for all logics.
  - Relatively simple to check proofs.
  - Good built-in support for binding constructs.
- Challenge: efficient proof checking for large LF proofs.
Two Problems for SMT Proof Checking

1. Proofs may be too large for main memory.
   - Traditionally: parse proof to AST, then check.
   - Bad, because of large proofs.

2. Side conditions on inference rules.
   - Some proof rules have computational side conditions.
   - E.g., resolution, for clause learning in modern SAT/SMT solvers.
   - In pure LF, explicit proofs of side conditions required.
**Solutions**

1. **Incremental checking** for large proofs.
   - Intertwine parsing and checking.
   - Avoid building ASTs whenever possible.
   - Consume proof as it is produced.

2. **LF with Side Conditions (LFSC).**
   - Allow declared signature constants to state side conditions.
   - Side conditions written in simple functional programming language.
   - Side conditions checked each time the constant is used.
Incremental Checking

- Basic idea: intertwine parsing and checking.
- Combine with bidirectional type checking.
  - Synthesizing: $\Gamma \vdash t \Rightarrow T$.
  - Checking: $\Gamma \vdash t \Leftarrow T$.
- ASTs built for subterms iff they will appear in the type $T$.
  E.g.,

\[(\text{refl } x+y) \Rightarrow x+y == x+y\]

  - AST must be built for $x+y$.
  - But not $(\text{refl } x+y)$.

- Note: orthogonal to signature compilation [Zeller+ 07].
Judgments extended to include input:

\[ \Gamma \mid I \Rightarrow^c t : T \mid I' \]

- \( I \) is initial list of input tokens.
- \( I' \) is rest of list, after synthesizing \( T \) for \( t \).
- \( c \) tells whether or not to create AST for \( t \).
- Similarly \( \Gamma \mid I \Leftarrow^c t : T \mid I' \).
Formalization: Example Rules

- Checking rule for \( \lambda \)-abstractions:

\[
\begin{align*}
\Gamma, x : T_1 | I & \iff^c t : T_2 | I' \\
\Gamma | \lambda, x, I & \iff^c \lambda x . t : \Pi x : T_1 . T_2 | I'
\end{align*}
\]

- Synthesizing rule for applications:

\[
\begin{align*}
\Gamma | I & \Rightarrow^c t_1 : \Pi x : T_1 . T_2 | I' \\
\Gamma | I' & \iff^c \lor x \in FV(T_2) t_2 : T_1 | I'' \\
\Gamma | @, I & \Rightarrow^c (t_1 \ t_2) : [t_2/x] T_2 | I''
\end{align*}
\]

- Here we update the flag \( \text{c} \).
- This flag initially false for top-level checking of a term.
- For simply typed \( t_1 \), avoid creating \( t_2 \) (unless already needed).
Correctness and Implementation

- Correctness established by erasure:

  \[
  \Gamma \vdash l \leftarrow c t : T \mid l' \quad \iff \quad \Gamma \vdash t \leftarrow T \\
  \Gamma \vdash l \Rightarrow c t : T \mid l' \quad \iff \quad \Gamma \vdash t \Rightarrow T
  \]

- Implemented in C++.
  - Around 2300 lines.
  - Manual reference counting used for managing memory.
  - Memory errors including leaks debugged with VALGRIND.
  - Allows holes if determined by types of subsequent arguments.
Empirical Results for Incremental Checking

- Same benchmarks as in [Zeller+ 07].
- Here, proofs from a simple proof-producing QBF solver.
- Compare with custom checker emitted by signature compilation.
- Also compare with Twelf, for third party tool.
- All times in seconds, timeout 30 minutes.

<table>
<thead>
<tr>
<th>benchmark</th>
<th>size</th>
<th>incr</th>
<th>custom</th>
<th>Twelf</th>
</tr>
</thead>
<tbody>
<tr>
<td>cnt01e</td>
<td>179 KB</td>
<td>0.25</td>
<td>0.28</td>
<td>4.0</td>
</tr>
<tr>
<td>tree-exa2-10</td>
<td>381 KB</td>
<td>0.35</td>
<td>0.50</td>
<td>6.1</td>
</tr>
<tr>
<td>cnt01re</td>
<td>267 KB</td>
<td>0.23</td>
<td>0.39</td>
<td>7.4</td>
</tr>
<tr>
<td>toilet_02_01.2</td>
<td>1.1 MB</td>
<td>0.92</td>
<td>1.3</td>
<td>150</td>
</tr>
<tr>
<td>1qbf-160cl.0</td>
<td>1.5 MB</td>
<td>0.98</td>
<td>1.1</td>
<td>750</td>
</tr>
<tr>
<td>tree-exa2-15</td>
<td>4.3 MB</td>
<td>3.7</td>
<td>5.8</td>
<td>timeout</td>
</tr>
<tr>
<td>toilet_02_01.3</td>
<td>8.2 MB</td>
<td>7.1</td>
<td>11.5</td>
<td>timeout</td>
</tr>
</tbody>
</table>
Resolution and its Side Conditions

- Our proof format must support rules like resolution.
- Simple e.g.: binary propositional resolution with factoring.
- Resolve clauses $C$ and $D$ on variable $v$ to $E$ iff
  1. $C$ contains $v$ positively.
  2. $D$ contains $v$ negatively.
  3. Removing all positive $v$ from $C$ yields $C'$.
  4. Removing all negative $v$ from $D$ yields $D'$.
  5.Appending $C'$ and $D'$ yields $E$.
  6. May also drop duplicate literals from $E$.
- Explicit proof seems to be of size $\Theta(|C| + |D|)$.
- Side condition proofs will dominate the rest of the proof.
Extend LF to allow computational side conditions.
Declared signature constants can state these.
Side conditions written with simply typed functional code.
  ▶ Pattern matching, general recursion, finite failure allowed.
  ▶ Call-by-value reduction.
  ▶ Limited mutable state: marking LF variables.
Code for computing resolvent in linear time easily implemented.
Checking Proofs from a Modern SAT Solver

- Incremental checker supports LFSC.
- LFSC signature for binary propositional resolution with factoring.
- Test with the CLSAT SAT solver.
  - Implemented mostly by Duckki Oe.
  - Competitive with MINISAT, TINISAT.
  - Produces resolution proofs in LFSC format.
  - Lemmas emitted for all learned clauses.
  - Run on benchmarks from SAT Race 2008 Test Set 1.
- Care needed to allow tail recursion when checking these proofs.
## Empirical Results for LFSC

<table>
<thead>
<tr>
<th>benchmark</th>
<th>size (MB)</th>
<th>num R (k)</th>
<th>check (s)</th>
<th>overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>E-sr06-par1</td>
<td>35</td>
<td>14.3</td>
<td>14.75</td>
<td>11.54</td>
</tr>
<tr>
<td>E-sr06-tc6b</td>
<td>8.4</td>
<td>8.7</td>
<td>11.68</td>
<td>32.26</td>
</tr>
<tr>
<td>M-c10ni_s</td>
<td>43</td>
<td>4.6</td>
<td>10.90</td>
<td>2.55</td>
</tr>
<tr>
<td>M-c6nid_s</td>
<td>33</td>
<td>72.9</td>
<td>48.35</td>
<td>3.63</td>
</tr>
<tr>
<td>M-f6b</td>
<td>30</td>
<td>1018.6</td>
<td>3237.22</td>
<td>202.24</td>
</tr>
<tr>
<td>M-f6n</td>
<td>26</td>
<td>847.6</td>
<td>2848.03</td>
<td>233.42</td>
</tr>
<tr>
<td>M-g6bid</td>
<td>27</td>
<td>797.5</td>
<td>1165.57</td>
<td>75.05</td>
</tr>
<tr>
<td>M-g7n</td>
<td>28</td>
<td>1006.8</td>
<td>1707.43</td>
<td>151.93</td>
</tr>
<tr>
<td>V-eng-uns-1.0-04</td>
<td>41</td>
<td>1692.7</td>
<td>5913.22</td>
<td>305.57</td>
</tr>
<tr>
<td>V-sss-1.0-cl</td>
<td>9.8</td>
<td>416.2</td>
<td>553.30</td>
<td>193.92</td>
</tr>
</tbody>
</table>

- **size**: size of proof (megabytes).
- **num R**: number of resolutions (thousands).
- **check**: time to check the proof (seconds).
- **overhead**: ratio of proof production + checking time to solving time.
Future Work.

1. Improve speed with compilation.
   ▶ 90% of runtime going to interpreting side conditions.
   ▶ So combine with signature compilation.
   ▶ Close gap with fast but ad hoc solution by M. Moskal.

2. Extend CLSAT proofs from SAT to SMT.
   ▶ CLSAT solves integer difference logic (QF_IDL).
   ▶ CNF conversion a little tricky due to formula renaming.

3. Implement verified version.
   ▶ Developing dependently typed PL called GURU.
   ▶ Supports input/output and mutable state using uniqueness types.
   ▶ Case study: incremental LF checker (“GOLFSOCK”).
   ▶ Statically verify character input parsed to type-correct LF.
   ▶ Mapping from symbol table trie to typing context almost verified.