Towards an SMT Proof Format

Aaron Stump and Duckki Oe

Dept. of Computer Science
The University of Iowa
Iowa City, Iowa, USA

Funding from NSF CRI grant.
Proofs and SMT

- SMT solvers large (50-100kloc), complex.
- To increase trust, have solvers emit proofs.
- Check proofs with much simpler checker (2-4kloc).

\[ \Phi \]

\[
\text{SMT Solver} \quad \text{Pf} \quad \text{Proof Checker}
\]

Pf Ok  Pf Bad
A standard proof format very desirable.
  ▶ Provides common target for solvers.
  ▶ Opens door to exporting to interactive provers.
  ▶ Build on standardization successes of SMT-LIB initiative.

Flexibility also important.
  ▶ A single proof system is useful for standardization.
  ▶ But: different solving algorithms ⇒ different proof systems.
  ▶ Can we let solver implementors modify or develop their own?

Speed required for large proofs (10s to 100s MB).
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 (no de Bruijn indices).
- Challenge: side conditions.
  - Some proof rules have computational side conditions.
  - E.g., resolution, used for clause learning.
  - In pure LF, explicit proofs of side conditions required.
Today’s Talk: LF with Side Conditions (LFSC)

- Extension of LF to support computational side conditions.
- Side conditions written in simple functional language.
- Proofs clearly divided into declarative, computational parts.
- Continuum of proof systems thus supported.
- Example: checking resolution proofs from a SAT solver.
Introduction to LF

- LF is a type theory, used as a meta-logic.
- An object logic is declared via type declarations.
- Proofs in that logic are terms, judgments are types.
- Proof checking is implemented by LF type checking.
- LF is mostly weaker and simpler than theories like Coq.
- Stronger in its built-in support for variable binding.
Encoding Propositional Clauses

(declare var type)
(declare lit type)
(declare pos (! x var lit))
(declare neg (! x var lit))
(declare clause type)
(declare cln clause)
(declare clc (! x lit (! c clause clause)))

\[ P \lor \neg Q \text{ encoded as:} \]

(clc (pos P) (clc (neg Q) cln))
Propositional Resolution

- Consider 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 would dominate the rest of the proof.
- More natural as a program than declaratively.
LF with Side Conditions (LFSC)

- Side conditions associated with proof rules.
- Checked every time rule is applied.
- Simply typed, call-by-value functional code.
  - Pattern matching, recursion, explicit failure.
  - Imperative feature: marking LF variables.
- Syntax for side condition code:

\[
C \ ::= \ x \ || \ c \ || \ N \ || \ (\odot \ C_1 \ \cdots \ C_{n+1}) \ || \ (c \ C_1 \ \cdots \ C_{n+1}) \\
\ || \ (\text{match } C \ (P_1 \ C_1) \ \cdots \ (P_{n+1} \ C_{n+1})) \ || \ (\text{do } C_1 \ \cdots \ C_{n+1}) \\
\ || \ (\text{let } x \ C_1 \ C_2) \ || \ (\text{markvar } C) \ || \ (\text{ifmarked } C_1 \ C_2 \ C_3) \ || \ (\text{fail } T)
\]

\[
P \ ::= \ (c \ x_1 \ \cdots \ x_{n+1}) \ || \ c
\]
Encoding Resolution in LFSC

(declare holds (! c clause type))

(program resolve ((c1 clause) (c2 clause) (v var)) clause
   (let pl (pos v)
      (let nl (neg v)
        (do (in pl c1)
               (in nl c2)
               (let d (append (remove pl c1) (remove nl c2))
                 (dropdups d))))))

(declare R (! c1 clause (! c2 clause (! c3 clause
               (! u1 (holds c1)
               (! u2 (holds c2)
               (! v var
               (! r (^ (resolve c1 c2 v) c3)
                 (holds c3))))))))
An Example Resolution Proof

Variables: $V_1, V_2, V_3$

Clauses: $\neg V_1 \lor V_2, \neg V_2 \lor V_3, \neg V_3 \lor \neg V_2, V_1 \lor V_2$

\[
\begin{array}{cccc}
V_1 \lor V_2 & \neg V_1 \lor V_2 & \neg V_2 \lor V_3 & \neg V_3 \lor \neg V_2 \\
V_2 & \neg V_2 & \neg V_2 & \neg V_2
\end{array}
\]

\textbf{empty}

($v_1\text{ var } (v_2\text{ var } (v_3\text{ var }$

($x_0\text{ (holds (clc (neg } v_1\text{) (clc (pos } v_2\text{) cln)}))}$)

($x_1\text{ (holds (clc (neg } v_2\text{) (clc (pos } v_3\text{) cln)}))}$)

($x_2\text{ (holds (clc (neg } v_3\text{) (clc (neg } v_2\text{) cln)}))}$)

($x_3\text{ (holds (clc (pos } v_1\text{) (clc (pos } v_2\text{) cln)}))}$

(R _ _ _ (R _ _ _ x3 x0 v1) (R _ _ _ x1 x2 v3) v2)))$)

$: $)

(! v1\text{ var } (! v2\text{ var } (! v3\text{ var }$

(! x0\text{ (holds (clc (neg } v_1\text{) (clc (pos } v_2\text{) cln))}$)

\ldots

(! x3\text{ (holds (clc (pos } v_1\text{) (clc (pos } v_2\text{) cln))}$

(holds cln)))$)})

Aaron Stump and Duckki Oe

Towards SMT Proofs

SMT 2008
Prototype LFSC checker.

- Supports incremental checking (combine parsing and checking).
- Not yet signature compilation (compile sig. to customized checker).

Signature for propositional resolution

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.
Empirical Results for LFSC

<table>
<thead>
<tr>
<th>benchmark</th>
<th>pf (s)</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>4.56</td>
<td>35</td>
<td>14.3</td>
<td>14.75</td>
<td>11.54</td>
</tr>
<tr>
<td>E-sr06-tc6b</td>
<td>0.96</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>6.62</td>
<td>43</td>
<td>4.6</td>
<td>10.90</td>
<td>2.55</td>
</tr>
<tr>
<td>M-c6nid_s</td>
<td>15.58</td>
<td>33</td>
<td>72.9</td>
<td>48.35</td>
<td>3.63</td>
</tr>
<tr>
<td>M-f6b</td>
<td>20.76</td>
<td>30</td>
<td>1018.6</td>
<td>3237.22</td>
<td>202.24</td>
</tr>
<tr>
<td>M-f6n</td>
<td>16.59</td>
<td>26</td>
<td>847.6</td>
<td>2848.03</td>
<td>233.42</td>
</tr>
<tr>
<td>M-g6bid</td>
<td>20.05</td>
<td>27</td>
<td>797.5</td>
<td>1165.57</td>
<td>75.05</td>
</tr>
<tr>
<td>M-g7n</td>
<td>16.12</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>25.04</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>4.18</td>
<td>9.8</td>
<td>416.2</td>
<td>553.30</td>
<td>193.92</td>
</tr>
</tbody>
</table>

- **pf**: time to solve and produce proof (seconds).
- **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.
Discussion

- 90% checking time used for interpreting side conditions.
- So compile side condition code.
- Enabled by separating declarative, computational parts.
  - Not separated in Moskal’s proposal (reduction under $\lambda$).
  - Despite his good performance, may limit speed.
- CNF conversion, theory reasoning must be implemented.
  - Introduction of new variables supported directly by LF.
  - Ad hoc solution required in Moskal’s approach.
  - LFSC checker already includes support for arithmetic.
  - Can check rules like

```plaintext
(declare not<=<=
  (! x (term Int) (! y (term Int) (! c mpz (! d mpz
     (! u (th_holds (not (<= (- x y) (an_int c)))))
     (! r (^ (mpz_add ( mpz_neg c) (~ 1)) d)
       (th_holds (<= (- y x) (an_int d))))))))))
```
Towards an SMT Standard?

- SMT-LIB could provide:
  - Fast LFSC checker (with signature compilation).
  - Example signature(s) and proofs.

- Solver implementors have several options:
  - Use the example signatures directly.
  - Modify or extend these.
  - Write their own.

- Proof checking enthusiasts can implement own checkers.
- LFSC provides basis for exporting (to Coq, Isabelle, et al.).
- Exported (example) signatures => exported proofs.
Other Future Work.

1. Improve speed with compilation.
2. Extend CLSAT proofs from SAT to SMT.
3. Implement verified version.
   - Developing dependently typed PL called GURU.
   - Like Coq but supports general recursion, mutable state.
   - Case study: incremental LF checker (“GOLFSOCK”).
   - Statically verify character input parsed to type-correct LF.
## Comparing clsat

<table>
<thead>
<tr>
<th>benchmark</th>
<th>size (MB)</th>
<th>CLSAT</th>
<th>MINISAT</th>
<th>TINISAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>E-sr06-par1</td>
<td>8.4</td>
<td>1.54</td>
<td>1.46</td>
<td>1.43</td>
</tr>
<tr>
<td>E-sr06-tc6b</td>
<td>1.9</td>
<td>0.38</td>
<td>0.22</td>
<td>0.34</td>
</tr>
<tr>
<td>M-c10ni_s</td>
<td>10</td>
<td>4.94</td>
<td>43.42</td>
<td>7.14</td>
</tr>
<tr>
<td>M-c6nid_s</td>
<td>7.4</td>
<td>13.81</td>
<td>162.01</td>
<td>93.56</td>
</tr>
<tr>
<td>M-f6b</td>
<td>1.7</td>
<td>16.03</td>
<td>4.02</td>
<td>5.41</td>
</tr>
<tr>
<td>M-f6n</td>
<td>1.7</td>
<td>12.22</td>
<td>4.57</td>
<td>6.58</td>
</tr>
<tr>
<td>M-g6bid</td>
<td>1.8</td>
<td>15.59</td>
<td>3.60</td>
<td>3.99</td>
</tr>
<tr>
<td>M-g7n</td>
<td>1.1</td>
<td>11.27</td>
<td>2.75</td>
<td>6.46</td>
</tr>
<tr>
<td>V-uns-1.0-04</td>
<td>1.0</td>
<td>19.37</td>
<td>5.19</td>
<td>5.63</td>
</tr>
<tr>
<td>V-1.0-cl</td>
<td>0.18</td>
<td>2.86</td>
<td>0.41</td>
<td>0.21</td>
</tr>
</tbody>
</table>