Fast and Flexible Proof Checking for SMT

Duckki Oe
Andrew Reynolds
Aaron Stump

Computer Science, The University of Iowa, USA
An SMT Proof System

• Solver generates formal proofs of unsatisfiability
  – Solver is not trusted
  – Answers can be trusted by verifying the proofs

• Verifier checks the proofs against the axioms and the formulas
  – Trusted component
  – Challenge 1: flexible to accommodate variety of solvers
  – Challenge 2: fast enough for practical usage
CLSAT

• CLSAT 1.0
  – SAT solver w/ proof generation
  – SMT solver (QF_IDL)
  – New: proof generation for SMT

• Proof Formalism
  – Based on Edinburgh Logical Framework (LF)
  – LF is a simple meta logic
  – SMT syntax and axioms defined in LF
LF and LFSC

LF
• LF is based on type theory
• Looks like a functional programming language
• Type computation and checking

LFSC (LF with Side Conditions)
• LF lacks looping and recursion
  – No complicated pattern matching and term building
• LFSC extends LF for
  – Computational side conditions
  – Built-in integer type and arithmetic
A Theorem of Unsatisfiability

\[ \Gamma, f : \Phi \vdash t : false \]

- \( \Gamma \) : SMT syntax and axioms
  - 61 rules (32 for CNF conversion, 17 for IDL)
  - 897 lines in LFSC
- \( f : \Phi \) : assumption of the input formula
- \( t \) : the proof statement from the solver
  - Mostly, \( \leq 200 \) MB for benchmarks solved \( \leq 900 \)s
  - But, a few of them are greater than 1GB
  - Overhead of proof production was less than 10\%
Proof Encoding in LF

\[
\Gamma \quad \Gamma, f_1 : \Phi_1, f_2 : \Phi_2
\]

\[\vdash P_1 \quad \vdash P_2\]

\[\Phi_1 \land \Phi_2 \quad false\]

\[false\] \quad \land_e

\[\Rightarrow (\text{and}_e P_1 (\lambda f_1 : \Phi_1. (\lambda f_2 : \Phi_2. P_2)))\]

LF variables are used
- to name derived formulas and clauses (as assumptions)
- to introduce new variables of the logic
- to store contextual information
CNF Conversion w/ Partial Clauses

- Tseitin rules will apply elimination and renaming at the same time (no choice of one)
- Partial Clauses represent intermediate steps

\[
\left\lbrack (\phi_1, \cdots, \phi_n; l_1, \cdots, l_n) \right\rbrack = \phi_1 \lor \cdots \lor \phi_n \lor l_1 \lor \cdots \lor l_n
\]

- Starts with a single partial clause \((\phi; \cdot)\)

\[
(\phi_1 \land \phi_2, \phi; C), \Pi \implies (\phi_1, \phi; C), (\phi_1, \phi; C), \Pi
\]
\[
(\phi_1 \lor \phi_2, \phi; C), \Pi \implies (\phi_1, \phi_2, \phi; C), \Pi
\]
\[
(\phi, \phi; C), \Pi \implies (\phi; v, C), \Pi \quad (v \leftarrow \phi)
\]
\[
(\cdot; C), \Pi \implies C, \Pi
\]
LFSC

- Based on Edinburgh Logical Framework (LF)
- Meta-logical proof checker
- Logic declared in user signature
  - Clause, Literal, True/False, Lists ...
- If a proof type checks, then it is considered valid
- Optimizations
  - Side Condition Compilation
  - Deferred Resolution
LFSC Side Conditions

• Proofs need computational side conditions
• Example: “resolve” rule for SMT proofs
• Written in simple functional language

```
... (program append ((c1 clause) (c2 clause)) clause
    (match c1 (cln c2) ((clc l c1') (clc l (append c1' c2))))))
```

• Side conditions executed with interpreter
• Idea: Convert to C++ and execute directly
Approach

LFSC

"lfsc sig.plf --compile-scc"

LFSC code

scccodel.cpp
scccodel.h

C++ Compiler

"lfsc sig.plf proof.plf" [old]

Proof valid?

"lfsc sig.plf proof.plf --run-scc"
Checking Resolution

• Resolution rule: clauses C and D on variable \( v \)
  – C contains \( v \)
  – D contains \( \neg v \)
  – Removing occurrences of \( v \) from C yields \( C' \)
  – Removing occurrences of \( \neg v \) from D yields \( D' \)
  – Appending \( C' \) to \( D' \) yields clause \( E \)
  – Duplicate literals eliminated from \( E \)

• Naively checked on every resolution step

• Idea: Calculate resolvent clause lazily
Approach

• Extended definition of clauses:
  – cln: empty clause
  – clc L C: clause C with literal L concatenated
  – clr L C: clause C with literal L removed
  – concat $C_1 C_2$: append clauses $C_1$ and $C_2$ in standard form

• Resolution rule becomes:
  – C contains $v$
  – D contains $\neg v$
  – Return (concat (clr $v$ C) (clr $\neg v$ D))

• Resolution deferred until final step
  – Calculate extended clause
  – Convert extended clause to standard clause
Conversion to Standard Clause

$[G]^{\sigma} = C$

- Extended clause $G$, standard clause $C$, set of literals $\sigma$.

...  

$$\begin{align*}
\[(\text{clc } L \; C)\]^{\sigma} &= \text{if } (L \in \sigma) \; ([C]^{\sigma}) \; \text{else} \; (\text{clc } L \; [C]^{\sigma} + L) \\
\[(\text{clr } L \; C)\]^{\sigma} &= [C]^{\sigma} + L
\end{align*}$$

- Literals to remove stored in $\sigma$
  - Literals marked for deletion eliminated
  - Duplicate literals eliminated
Results

- Benchmarks QF_IDL difficulty 0-3
- Timeout of 1800s
- Public job on SMT EXEC

<table>
<thead>
<tr>
<th>Solver</th>
<th>Score</th>
<th>Unknown</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>clsat (w/o proof)</td>
<td>542/622</td>
<td>50</td>
<td>30</td>
</tr>
<tr>
<td>clsat+lfsc (optimized)</td>
<td>538/622</td>
<td>51</td>
<td>33</td>
</tr>
<tr>
<td>clsat+lfsc (unoptimized)</td>
<td>485/622</td>
<td>58</td>
<td>79</td>
</tr>
</tbody>
</table>
## Results

<table>
<thead>
<tr>
<th>Solver</th>
<th>Score</th>
<th>Time1</th>
<th>Time2</th>
</tr>
</thead>
<tbody>
<tr>
<td>clsat (w/o proof)</td>
<td>542/622</td>
<td>20168.7s</td>
<td>31843.6s</td>
</tr>
<tr>
<td>clsat+lfsc (optimized)</td>
<td>538/622</td>
<td>23741.4s</td>
<td>41420.8s</td>
</tr>
<tr>
<td>clsat+lfsc (unoptimized)</td>
<td>485/622</td>
<td>52373.8s</td>
<td>n/a</td>
</tr>
</tbody>
</table>

- **Time1**: Total time to solve 485 benchmarks solved by all three configurations
- **Time2**: Total time to solve 538 benchmarks solved by first two configurations
Results

[Graph showing a scatter plot with a linear trend line. The x-axis is labeled 'clsat+lfsc' and the y-axis is labeled 'clsat'. The values range from 0 to 100,000 on both axes.]

Download PDF

Conclusion

• Provides fast and flexible proof checking
• Proof production overhead is less than 10%
• Lowest reported proof checking time
• Proof checking overhead converging to 2x
  • 30.1% on average