Proof Checking Technology for Satisfiability Modulo Theories

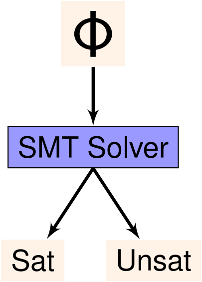
Aaron Stump

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.

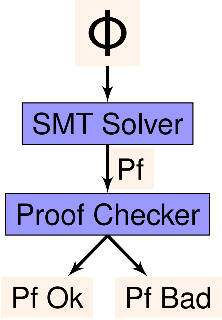


Confirming Solver Results

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

One solution:

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



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.,

```
(refl x+y) => x+y == x+y
```

  - AST must be built for `x+y`.
  - But not `(refl x+y)`.
- Note: orthogonal to signature compilation [Zeller+ 07].

Formalization: Judgments

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 λ-abstractions:

$$\frac{\Gamma, x : T_1 \mid I \;\Leftarrow^c\; t : T_2 \mid I'}{\Gamma \mid \lambda, x, I \;\Leftarrow^c\; \lambda x.\, t : \Pi x : T_1.\, T_2 \mid I'}$$

- Synthesizing rule for applications:

$$\frac{\Gamma \mid I \;\Rightarrow^c\; t_1 : \Pi x : T_1.\, T_2 \mid I' \qquad \Gamma \mid I' \;\Leftarrow^{c \,\vee\, x \in FV(T_2)}\; t_2 : T_1 \mid I''}{\Gamma \mid @, I \;\Rightarrow^c\; (t_1\ t_2) : [t_2/x]T_2 \mid I''}$$

  - Here we update the flag 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:

$$\begin{array}{lcl} \Gamma \mid I \Leftarrow^c t : T \mid I' & \mapsto & \Gamma \vdash t \Leftarrow T \\ \Gamma \mid I \Rightarrow^c t : T \mid I' & \mapsto & \Gamma \vdash t \Rightarrow T \end{array}$$

- 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.

benchmark	size	incr	custom	Twelf
cnt01e	179 KB	0.25	0.28	4.0
tree-exa2-10	381 KB	0.35	0.50	6.1
cnt01re	267 KB	0.23	0.39	7.4
toilet_02_01.2	1.1 MB	0.92	1.3	150
1qbf-160cl.0	1.5 MB	0.98	1.1	750
tree-exa2-15	4.3 MB	3.7	5.8	timeout
toilet_02_01.3	8.2 MB	7.1	11.5	timeout

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.

LF with Side Conditions (LFSC)

- 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

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

- 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.