Towards an SMT Proof Format

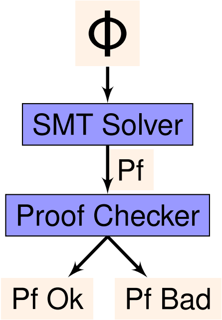
Aaron Stump and Duckki Oe

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).



Φ

SMT Solver

Pf

Proof Checker

Pf Ok

Pf Bad

Standardized, Flexible, Fast

- 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)))
```

$P \vee \neg Q$ 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:

$$
\begin{array}{rcl}
C & ::= & x \mid\mid c \mid\mid N \mid\mid (\odot\ C_1\ \cdots\ C_{n+1}) \mid\mid (c\ C_1\ \cdots\ C_{n+1}) \\
 & & \mid\mid (\mathit{match}\ C\ (P_1\ C_1)\ \cdots\ (P_{n+1}\ C_{n+1})) \mid\mid (\mathit{do}\ C_1\ \cdots\ C_{n+1}) \\
 & & \mid\mid (\mathit{let}\ x\ C_1\ C_2) \mid\mid (\mathit{markvar}\ C) \mid\mid (\mathit{ifmarked}\ C_1\ C_2\ C_3) \mid\mid (\mathit{fail}\ T)
\end{array}
$$

$$
P \quad ::= \quad (c\ x_1\ \cdots\ x_{n+1}) \mid\mid c
$$

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

$$\dfrac{\dfrac{V_1 \lor V_2 \qquad \neg V_1 \lor V_2}{V_2} \qquad \dfrac{\neg V_2 \lor V_3 \qquad \neg V_3 \lor \neg V_2}{\neg V_2}}{\mathit{empty}}$$

```
($ v1 var ($ v2 var ($ v3 var
($ x0 (holds (clc (neg v1) (clc (pos v2) cln)))
($ x1 (holds (clc (neg v2) (clc (pos v3) cln)))
($ x2 (holds (clc (neg v3) (clc (neg v2) cln)))
($ x3 (holds (clc (pos v1) (clc (pos v2) cln)))
  (R _ _ _ (R _ _ _ x3 x0 v1) (R _ _ _ x1 x2 v3) v2)))))))) :

(! v1 var (! v2 var (! v3 var
(! x0 (holds (clc (neg v1) (clc (pos v2) cln)))
...
(! x3 (holds (clc (pos v1) (clc (pos v2) cln)))
  (holds cln)))))))
```

Checking Proofs from a Modern SAT Solver

- 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

benchmark	pf (s)	size (MB)	num R (k)	check (s)	overhead
E-sr06-par1	4.56	35	14.3	14.75	11.54
E-sr06-tc6b	0.96	8.4	8.7	11.68	32.26
M-c10ni_s	6.62	43	4.6	10.90	2.55
M-c6nid_s	15.58	33	72.9	48.35	3.63
M-f6b	20.76	30	1018.6	3237.22	202.24
M-f6n	16.59	26	847.6	2848.03	233.42
M-g6bid	20.05	27	797.5	1165.57	75.05
M-g7n	16.12	28	1006.8	1707.43	151.93
V-eng-uns-1.0-04	25.04	41	1692.7	5913.22	305.57
V-sss-1.0-cl	4.18	9.8	416.2	553.30	193.92

- 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 λ).
    - 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

```
(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

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