Fast and Flexible Proof Checking with LFSC

Andrew Reynolds
University of Iowa

November 11, 2011
• LFSC proof checking technology for SMT
  – University of Iowa
    • Aaron Stump
    • Duckki Oe
    • Andrew Reynolds
    • Cesare Tinelli
  – New York University
    • Liana Hadarean
    • Yeting Ge
    • Clark Barrett
Logical Framework with Side Conditions as:

1. Framework for defining SMT proof systems
2. Optimized Proof Checker
3. Proof System for Linear Real Arithmetic
4. Interpolant Generator via Type Inference
• SMT solvers are difficult to verify
  – Code may be complex (10k+ loc)
  – Code is subject to change

Alternatively....

• Solvers can justify answers with proofs
• There is need for third party certification
  – Must ensure that proof is valid
Proof Checking: Overview

Solver

sat

unsat

Model

Proof of Unsatisfiability

LFSC

Proof Valid

Proof Invalid
- **Speed**
  - Practical for use with solvers
  - Measured time against solving time
- **Flexibility**
  - Different solvers have different needs
  - Solvers can change over time
  - Many different theories
• Edinburgh Logical Framework (LF) [Harper et al 1993]
  – Based on type theory
  – Meta framework for defining logical systems
• LF with side conditions (LFSC) [Stump et al 2008]
  – Meta-logical proof checker
  – Side Conditions
  – Support for Integer, Rational arithmetic
  – If proof term type-checks,
    Then proof is considered valid
Example proof rule

\[
\begin{array}{c}
\psi_1 \\
\hline
\psi_1 \land \psi_2 \\
\end{array}
\]

\[
(declare \ and\_intro

(! \ f1 \ formula
(! \ f2 \ formula
(! \ p1 \ (proof \ f1)
(! \ p2 \ (proof \ f2)
(proof \ (and \ f1 \ f2))))))))
\]
\[ \begin{array}{c}
\frac{p > 0}{\downarrow} \{ p \downarrow c, \ c \not\geq 0 \}
\end{array} \]

(declare ineq_contradiction
  (! p poly
   (! p1 (proof (> p 0))
   (! s (^ (is_positive (simplify p)) ff)
    (proof false))))))
Proof rule with side condition

\[
\begin{aligned}
\frac{p \succ 0}{\{ p \downarrow c, \ c \not\succ 0 \}}
\end{aligned}
\]

• Side conditions
  – Written in simply typed functional language

```scheme
(simplify ((p poly)) real
  (match p
    ((poly c' l')
     (match (is_zero l')
       (tt c')
       (ff fail))))))
```
Why side conditions?

• Mirror high-performance solver inferences
• More Efficient
  – Smaller Proof Size
  – Faster Checking time

• Amount can be fine tuned

Fully Declarative  ❯  Fully Computational
• LFSC for arithmetic [Reynolds et al 10]

• Proofs in Linear Real Arithmetic (LRA)
  – Rules require computational side conditions
    • e.g. \(( t_1 + ( t_2 + t_3)) = ((t_3 + t_1) + t_2)\)
  – Use of side conditions for normalization
    • e.g. \(( t_1 + ( t_2 + t_3)) \downarrow p_1 , ((t_3 + t_1) + t_2) \downarrow p_2\)
    • Verify \( p_1 = p_2 \) using side conditions
• Use SMT solver CVC3 to generate proofs
  – Module to convert proofs to LFSC format
• Flexibility: Multiple Signatures for LRA
  – Declarative
    • Rewrite calculus, native format used by CVC3
    • Rules of form $\Psi_1 \leftrightarrow \Psi_2$
  – Computational
    • Take advantage of LFSC side condition features
    • Rules involving polynomial atoms
• Theory lemmas in QF_LRA
  – Ex: \( \neg (2x > 2y) \lor \neg (y > x + 5) \)
  – Can be done by finding set of coefficients

\[
\begin{align*}
\frac{1}{2} * & \quad 2x > 2y \\
1 * & \quad y > x + 5 \\
\hline
x + y > y + x + 5 \\
\Rightarrow \quad 0 > 5
\end{align*}
\]
\[
\begin{align*}
2x > 2y & \quad 2x > 2y \iff x > y \\
\quad x > y & \quad y > x + 5 \\
\quad x > x + 5 & \quad \vdots \\
\quad x > x + 5 & \iff \bot
\end{align*}
\]
Proof transformation
• Configurations
  – Literal translation (Lit)
    • Faithful encoding of CVC3’s native format
  – Liberal translation (Lib)
    • Capitalize on side conditions
• For theory lemmas: 3x faster proof checking time
  – Theory lemma proofs 5x smaller in size on average
• Proof checking $\sim 10x$ faster than solving
• In addition to proofs of unsatisfiability, use LFSC for richer proof calculi

lifting

Proof \[\rightarrow\] Extended Proof

• Interpolant generating proofs [Reynolds et al 11]
  – Theory of uninterpreted functions with equality (EUF)
For theory $T$, a $T$-interpolant $I$ for $(A,B)$

\begin{align*}
(1) & \quad A \models_T I \\
(2) & \quad B, I \models_T \bot \\
(3) & \quad L(I) \subseteq L(A) \cap L(B)
\end{align*}

- In some cases, may be efficiently generated from proofs
- Applications
  - Model Checking, Predicate Abstraction, ...
- Use LFSC to generate \textit{certified} interpolants
• Since LFSC is meta-framework, we can extend signature to type-check proofs about interpolants
Proof Checking: Interpolants

\[ A \land B \]

- Solver
  - sat
  - Model
  - \ldots
  - \ldots
- unsat, interpolant \( I \)
  - Proof
    - \ldots
    - LFSC Pf Checker
      - pf valid
      - pf invalid

Extended Signature

\text{pf invalid}
(check
  (% ... 
  (% a (proof A)
  (% b (proof B)
  (: (interpolant A B I) P

  )))...)

• Check if P is of type (interpolant A B I),
  for formulas A, B, I

• If so, then I is a certified interpolant for (A, B)
• SMT solver produces interpolant + proof

• LFSC verifies that proof:
  (1) Successfully type checks, and
  (2) Shows claimed interpolant is an interpolant.

• Solver + Checker must agree on the interpolant
• Alternatively:

*Use proof checker as the interpolant generator*

• Solver writes proof in same signature
  – Constructs proof of type \((\text{interpolant } A \, B \, I)\),
    • for some value of \(I\), unknown a priori
  – Value of \(I\) computed by type inference
• LFSC proofs may contain hole symbols "_"

• For example:

\[
\text{(trans } _ _ _ \text{ (} = t_1 \ t_2 \text{ ) } ( = t_2 \ t_3 \text{ ) })
\]

• Allow proof checker fill in value for interpolant
  – Certified correct by construction
(check
  (% ... 
  (% a (proof A))
  (% b (proof B))
  (: (interpolant A B _)
      \( P \)) ...)

- The interpolant field left unspecified "_"
- If \( P \) is of type \((\text{interpolant} \ A \ B \ I)\),
  - Value of \( I \) is given to user
  - \( I \) is a certified interpolant for \((A, B)\)
Interpolant Generation via Proof Checking

Formula: \( A \land B \)

Solver

Model

Proof + Extended Signature

LFSC Pf Checker

sat

unsat

pf valid,

\[ I \]

pf invalid
• Tested configurations
  – **euf**: proof checking
  – **eufi**: proof checking with interpolant generation

• Proof checking fast w.r.t to solving
  – **euf** 11x faster than solving
  – **eufi** 5x faster than solving

• Interpolants come at small overhead
  – **eufi** 22% overhead with respect to solving + pf generation
• Optimizations for LFSC
  – Incremental Checking
    • Proof is checked while it is parsed
      – Instead of being read into memory
  – Optimized boolean resolution checking
    • Resolvent clauses produced lazily
  – Signature Compilation [Oe et al 09]
    • Side conditions run directly in compiled C++
      – Instead of using an interpreter
proof.plf

LFSC

LFSC signature.plf proof.plf

Proof
Valid/Invalid
**Signature Compilation**

- **proof.plf**
- **LFSC**
- **LFSC –gen-scc signature.plf**
- **LFSC code base**
  - **scc_code.h/.cpp**
- **[compile]**
- **LFSC –run-scc proof.plf**
- **LFSC signature.plf**

Proof Valid/Invalid
• Integration into CVC4
  – Extensions to other theories
    • Datatypes, Bit Vectors, Arrays, etc.

• New release of LFSC
  – Usability of user-defined signatures
  – Improved performance
  – ...

Future Work