LFSC for SMT Proofs: Work in Progress

Aaron Stump, Andrew Reynolds, Cesare Tinelli, Austin Laugesen, Harley Eades, Corey Oliver, Ruoyu Zhang

PxTP workshop
June 30th, 2012
Acknowledgements

• Current LFSC team:
  – Aaron Stump, Andrew Reynolds, Cesare Tinelli, Austin Laugesen, Harley Eades, Corey Oliver, Ruoyu Zhang

• Previous work on LFSC:
  – University of Iowa
    • Duckki Oe, Jed McClurg, Cuong Thai
  – New York University
    • Liana Hadarean, Yeting Ge, Clark Barrett
In this talk:

• Previous work:
  – LFSC: meta-format for defining proofs
  – High performance proof checker (C++)
  – Applications to SMT proofs

• New work on LFSC:
  – New implementation (Ocaml), more optimizations
  – Language for defining proof signatures
Proof Checking in SMT

Formula $\varphi$ → SMT Solver

- sat → Model
- unsat → Proof $P$

Proof Checker

- pf valid
- pf invalid
Challenges of Proof Checking in SMT

• Many theories
  – UF, Arrays, Arithmetic, Datatypes, Bitvectors
  – ... Quantifiers

• Solvers have unique implementations
  – Have highly optimized decision procedures
  – Use unique proof inferences

• Proofs can be very large
  – Can be on the order of gigabytes
Challenges of Proof Checking in SMT

• Most SMT solvers:
  – Do propositional reasoning via SAT solver
  – Perform CNF conversion
  – Use theory solvers
  – Apply simplification to input
    • ITE removal, theory-specific rewriting of literals, ...
  – Use theory combination
  – Apply quantifier instantiation/elimination
  – ...

• Proof system must account for all of these
  – In CVC3: 200+ fine/coarse grained proof rules
Challenges of Proof Checking in SMT

• In purely declarative proof format
  – Proof size can be impractical

• Consider arithmetic:
  \[( t_1 + \ldots t_n ) = ( s_1 + \ldots + s_n ),\]
  where \( s_1 \ldots s_n \) is a permutation of \( t_1 \ldots t_n \)
  – Requires \( O( n^2 ) \) applications of declarative rules
    • i.e. associative/commutative properties of addition

➤ Proposed solution:
  – use simple computational checks within proof rules
    • i.e. polynomial normalization
LFSC: Proof Checker for SMT

• Flexible
  – Meta-format for defining proof systems
  – Proof rules in user-defined signature
  – One checker suffices for many signatures

• Fast
  – High performance C++ code
  – Use of side conditions to reduce proof size
  – In most cases, checking time $<<$ solving time
LFSC: LF with Side Conditions

• Edinburgh Logical Framework
  – Curry-Howard Isomorphism
    • Proofs as terms
    • Proof checking becomes type checking

• Extends LF with *side conditions*
  – Written in simple functional programming language
  – Each side condition:
    • (Intended to be) small enough to verify by inspection
Framework for Proof Checking in SMT

- Formula $\phi$
  - SMT Solver
    - sat
      - Model
    - unsat
      - Proof $P$
        - is $P$ of type $(\text{holds } \phi) \rightarrow (\text{holds false})$?
  - LFSC
    - pf valid
    - pf invalid
  - Signature
Previous Work

• LFSC as:
  – Framework defining proof systems
  – Efficient proof checker for SMT
  – Flexible proof checker for linear arithmetic
  – Certified interpolant generator
Optimizations in LFSC [Oe et al 09]

• Optimizations in LFSC
  – Incremental Checking
    • Proofs checked as they are parsed
  – Optimized proof rules for boolean resolution
    • Lazy approach to applying side conditions
  – Side condition compilation
    • Integrated into C++ source, instead of interpreted
• Each leads to order of magnitude speedup
**Linear Real Arithmetic [Reynolds et al 10]**

- **LFSC Signature for Linear Real Arithmetic (LRA)**
  - Conversion of terms to normalized polynomials
    - $t_1 = t_2$ becomes $p = 0$, where $p$ is $(t_1 - t_2)$
  - 60 lines of side condition code
    - Code complexity roughly of merge sort
- **Exploit continuum of possible proof systems**
  - Declarative proof system
    - Rewrite rules of the form $t_1 = t_2 \leftrightarrow t'_1 = t'_2$
  - Computational proof system
    - Side conditions to perform operations on polynomials
Linear Real Arithmetic

- Experiments on SMT LIB benchmarks
- Used CVC3 for proof generation
- Computational proof system is advantageous
  - For proofs of theory lemmas:
    - 5x reduction in proof size
    - 2.5x reduction in proof checking time
- Proof checking in both systems is fast
  - 10x faster than solving time
Interpolant Generation [Reynolds et al 11]

• Interpolant for inconsistent formulas (A,B)
  – Summarizes the inconsistency, in language of A \cap B
• Interpolants are useful in verification
  – Model checking, abstraction refinement, ...
• Correctness of interpolant can be critical
• Often, interpolant can be extracted from proof
  – Use of interpolant generating calculi:

\[
\begin{array}{c}
\varphi_1 & \cdots & \varphi_n \\
\hline
\varphi
\end{array}
\quad \text{rule} \quad \Rightarrow \quad
\begin{array}{c}
\varphi_1[I_1] & \cdots & \varphi_n[I_n] \\
\hline
\varphi[I]
\end{array}
\quad \text{rule'}
\]
Certified Interpolant Generation

SMT Solver

unsat

Proof

Apply annotations to proof

Proof

LFSC

Proof Checker

Extended Signature

pf valid,

pf invalid

Interpolant
Certified Interpolant Generation

• LFSC generates *certified* interpolants
  – Comes as side effect of proof checking

• Approach is practical:
  – 2x slower than checking unannotated proofs
  – Checking is 5x faster than solving
    • 22% overhead
LFSC: Looking Forward

• User-friendly language for defining Pf signatures
  – Surface language
  – Core language
    • Translation from surface to core language

• Highly optimized proof checker
  – Signature compilation
    • Side conditions as well as type checking rules
  – Implicit arguments for proof rules
    • Reduction in proof size
LFSC : Proof Checker

- For optimization, compile signature into proof checker
LFSC : Proof Checker Generator

SMT Solver

sat

unsat

Proof

Signature

Proof Checker

pf valid

pf invalid

⇒ *Generic translation of signature into C++ code for proof checker*
Example Proof System

Formulas $\phi$  ::=  $p$  $|$  $\phi_1 \rightarrow \phi_2$

Contexts $\Gamma$  ::=  $\cdot$  $|$  $\Gamma, \phi$

$\phi \in \Gamma$  $\frac{\Gamma \vdash \phi}{\Gamma \vdash \phi}$  Assump

$\Gamma, \phi_1 \vdash \phi_2$  $\frac{\Gamma \vdash \phi_1 \rightarrow \phi_2}{\Gamma \vdash \phi_2}$  ImpIntro

$\Gamma \vdash \phi_1 \rightarrow \phi_2$  $\frac{\Gamma \vdash \phi_1}{\Gamma \vdash \phi_2}$  ImpElim
Example Proof System in LF

```
formula : Type;
imp : formula -> formula -> formula;

holds : formula -> Type.

imp_intro :
Π f1:formula. Π f2:formula. 
  ((holds f1) -> (holds f2)) -> (holds (imp f1 f2)).

imp_elim :
Π f1:formula. Π f2:formula.
  (holds (imp f1 f2)) -> (holds f1) -> (holds f2).
```

- Can be burdensome to write proof signatures in this format
LFSC : Surface Language Support

SMT Solver

Proof

SMT Solver

Signature

Surface to Core Translation

Core Signature

Proof Checker Generator

Proof Checker

pf valid

pf invalid

sat

unsat
Surface Language

SYNTAX

formula f ::= imp f1 f2.

JUDGMENTS

(holds f)

RULES

[ holds f1 ] |- holds f2

------------------------------------------ imp_intro

holds (imp f1 f2) .

holds (imp f1 f2), holds f1

------------------------------------------ imp_elim

holds f2 .
Core Language

tctor formula : Type .

tctor imp :
   Pi+(f1: formula, f2:formula) .

tctor holds : Pi(f:formula).Type .

tctor imp_intro :
   Pi-(f2:formula).
   Pi+(f1:formula, p:Pi+(p:(holds f1)).(holds f2)).
      (holds (imp f1 f2)).

tctor imp_elim :
   Pi-(f1:formula, f2:formula).
   Pi+(p1:(holds (imp f1 f2)), p2:(holds f1)).
      (holds f2).
Compiled C++

... string s = parse_string();
if( s=="imp_intro" ){
    ...
}
else if( s=="imp_elim" ){
    Expr* e1 = parse_expr();
    Expr* e2 = parse_expr();
    if( e1->kind==k_holds &&
        e2->kind==k_holds &&
        e1->child[0]==e2->child[0] ){
        return e1->child[1];
    }else{
        Error("proof checking failed");
    }
}  

- Actual generated C++ code is highly optimized
Example Proof

\[
\frac{p, (p \rightarrow q)}{\vdash (p \rightarrow q)} \quad \frac{p, (p \rightarrow q) \vdash p}{p \vdash (p \rightarrow q) \rightarrow q} \quad \frac{p \vdash ((p \rightarrow q) \rightarrow q)}{\vdash p \rightarrow ((p \rightarrow q) \rightarrow q)}
\]
Example Proof : LFSC

\[\begin{align*}
    & p, (p \rightarrow q) \vdash (p \rightarrow q) \\
    & p, (p \rightarrow q) \vdash p \\
    & p \vdash (p \rightarrow q) \rightarrow q \\
    & \therefore p \vdash ((p \rightarrow q) \rightarrow q)
\end{align*}\]

\[
\text{imp_intro} \ (\text{imp} \ p \ (\text{imp} \ (\text{imp} \ p \ q) \ q)) \ p \\
\ u \ . \ \text{imp_intro} \ (\text{imp} \ (\text{imp} \ p \ q) \ q) \ (\text{imp} \ p \ q) \\
\ v \ . \ \text{imp_elim} \ (\text{imp} \ p \ q) \ q \ u \ v
\]
Example Proof: LFSC

- Proof size may be reduced via use of implicit arguments:

```latex
imp_intro p
  u . imp_intro (imp p q)
  v . imp_elim u v
```

- Automatically determine which arguments made implicit
Surface Language Example : SMT

SYNTAX

sort $s ::= \text{arrow } s_1 \; s_2 \mid \text{bool}$.

term$<\text{sort}> \; t ::=$

true$<\text{bool}>$

| false$<\text{bool}>$

| (not $t_1><\text{bool}>)<\text{bool}>$

| (and $t_1><\text{bool}> \; t_2><\text{bool}>)<\text{bool}>$

| (or $t_1><\text{bool}> \; t_2><\text{bool}>)<\text{bool}>$

| (ite $t_1><\text{bool}> \; t_2><\text{s}> \; t_3><\text{s}>)<\text{s}>$

| (forall $t><\text{s}> \; ^ t><\text{bool}>)<\text{bool}>$

| (apply $t_1><\text{arrow } s_1 \; s_2 \; t_2><s_1><s_2>$)

| (eq $t_1><\text{s}> \; t_2><\text{s}>)<\text{bool}>$.

formula $f ::= t><\text{bool}>$. 
Surface Language Example: SMT

... JUDGMENTS (th_holds f)

RULES
--------------- refl
th_holds (eq t1<s> t2<s>) .

th_holds (eq t1<s> t2<s>)
--------------- symm
th_holds (eq t2<s> t1<s>) .

th_holds (eq t1<s1> t2<s1>)
--------------- cong
th_holds (eq (apply t3<arrow s1 s2> t1<s1>)
   (apply t3<arrow s1 s2> t2<s1>) ) .

th_holds (eq t1<s> t2<s>) th_holds (eq t2<s> t3<s>)
--------------- trans
th_holds (eq t1<s> t3<s>) .
Current Work on LFSC

• Design of core language
  – Side conditions
  – Implicit/Explicit arguments
• Conversion of core language to proof checker
• Optimizations for proof checking
• Develop signatures for various SMT theories
  – Arithmetic, parametric datatypes, quantifiers
• Integration of LFSC into SMT solver CVC4
Summary

• Previous work on LFSC:
  – Fast and flexible approach for SMT proofs

• New version of LFSC:
  – Generates proof checker from user signature
  – Surface language for defining proof signatures
  – Plans for highly optimized proof checker

• Currently in Development
Questions?