Fast and Flexible Proof-Checking with LFSC

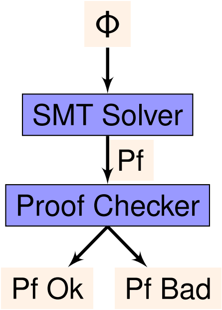
Tianyi Liang Andy Reynolds Aaron Stump Cesare Tinelli

Dept. of Computer Science
The University of Iowa
Iowa City, Iowa, USA

Funding from NSF.

Proofs and SMT Solvers

- SMT solvers large (50-100kloc), complex.
- To increase trust, have solvers emit proofs.
- Check proofs with much simpler checker (2-4kloc).

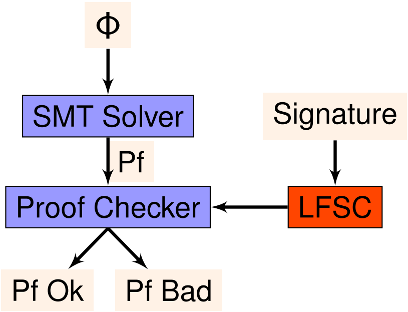


- Large, complex formulas => large proofs.
- Proofs easily 100s MBs or GBs.
- Proof-checking speed important!

The LFSC Proof Format

- "Logical Framework with Side Conditions".
- Goal: a standard proof format for SMT.
- Developed over past 4 years:
  - Comparing Proof Systems for LRA with LFSC. SMT '10
  - Fast and Flexible Proof Checking for SMT. SMT '09
  - Towards an SMT Proof Format. SMT '08
  - Proof Checking Technology for SMT. LFMTP '08
  - A Signature Compiler for the Edinburgh LF. LFMTP '07
- LFSC is a meta-language.
  - Describe abstract syntax, proof rules in a *signature*.
  - LFSC then compiles that signature.
  - Supports many logics (not just SMT).
  - Result: fast custom proof checker.
  - Benefits: speed and flexibility.

LFSC Proofs and SMT Solvers



Benefits of LFSC

- Trustworthiness:
  - Declarative specification of proof checker.
  - Trusted: signature + generic LFSC compiler.
  - More trustworthy than hand-implemented checker.
  - More human-understandable (cf. CVC3's C++ rules).
- Flexibility:
  - SMT solvers have hundreds of rules.
  - No consensus on single "right" proof system.
  - Easily change signature.
  - Auto-generate C++ code for proof production (in progress).
- Performance:
  - Compilation removes overhead of using meta-language.
  - New optimizations implemented once in LFSC.
  - All proof systems can take advantage.

Logical Framework with Side Conditions

- Based on Edinburgh Logical Framework (LF) [Harper et al., '93]
- View proof-checking as type-checking.
- Adds support for computational side conditions [Stump, Oe '09].
- For example, resolution:

$$\frac{\vdash C_1 \quad \vdash C_2}{\vdash C_3} \ \text{resolve}(C_1, C_2, v) = C_3$$

- LFSC supports continuum of proof systems:

Purely Computational	Practical	Purely Declarative

LFSC Signatures by Example

Mathematical version:

$$\text{formula } f \ ::= \ \text{true} \mid \text{false} \mid p \mid (\text{and } f_1 \ f_2) \mid \ldots$$

$$\frac{\vdash f_1 \quad \vdash f_2}{\vdash (\text{and } f_1 \ f_2)} \text{ and-intro}$$

LFSC version:

```
(declare formula type)
(declare true formula)
(declare false formula)
(declare and (! f1 formula (! f2 formula formula)))

(declare holds (! x formula type))
(declare andi (! f1 formula
              (! f2 formula
              (! u1 (holds f1)
              (! u2 (holds f2)
              (holds (and f1 f2)))))))
```

A Sample Proof

Mathematical version:

$$\dfrac{\vdash p \quad \dfrac{\vdash q \quad \vdash q}{\vdash (\text{and } q\ q)}}{\vdash (\text{and } p\ (\text{and } q\ q))}$$

LFSC version:

```
(% p formula
(% q formula
(% u1 (holds p)
(% u2 (holds q)
   (andi _ _ u1 (andi _ _ u2 u2))))))
```

- LFSC assumptions introduce with %.
- _ for the formulas proved by subproofs.

LFSC Proof-Checking Optimizations

1. Compile declarative part of signature [Zeller,Stump,Deters '07].
   - Basic checker: `bool check(sig *s, pf *p)`
   - Partially evaluate this w.r.t. `sig *s`.
   - Custom checker: `bool check-s(pf *p)`
2. Compile side-condition code [Oe,Reynolds,Stump '09].
3. Incremental checking [Stump '08].
   - Traditionally: parse to AST, then check proof.
   - Optimized: parse and check together.
   - Avoid building AST for proof in memory.

5x speedup for SMT benchmarks with each of these.

Next Steps

- Experiment with trade-off between declarative, computational.
  - Comparing Proof Systems for Linear Real Arithmetic with LFSC. Reynolds, Haderean, Tinelli, Ge, Stump, Barrett. SMT '10
- New implementation of LFSC compiler (for fall '10).
  - Currently: 6kloc C++, complex.
  - Currently only implement 2 of the optimizations.
  - Wanted: more trustworthy, more flexible, all optimizations.
  - Reimplement in OCaml.
- New input syntax (Tianyi Liang):
  - BNF for abstract syntax, textual versions of rules:

```
formula f ::= true | false | (and f1 f2)

(holds f1)    (holds f2)
-------------------------
(holds (and f1 f2)
```

- Public release, tool paper.