# 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

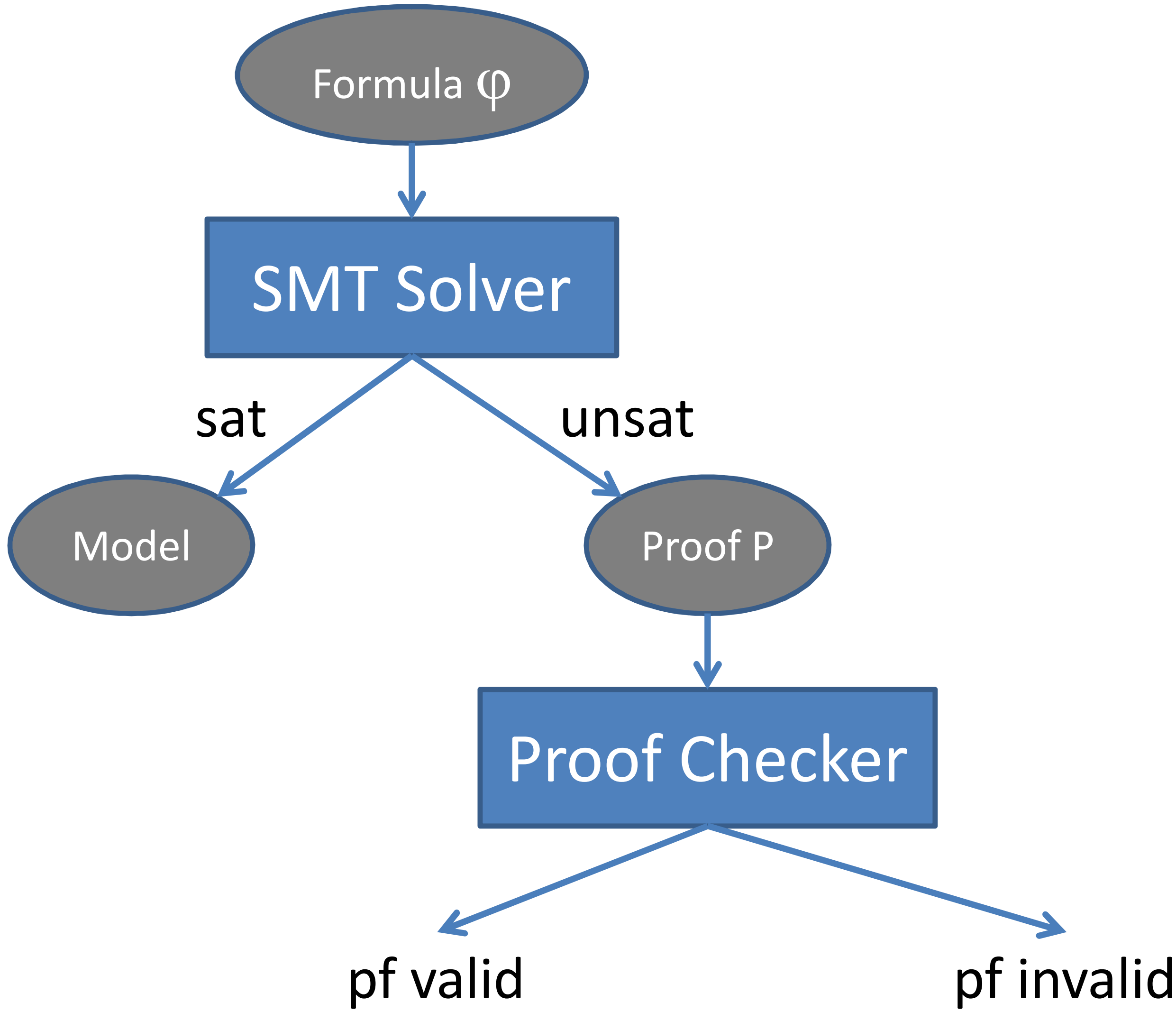
# Acknowledgements


- 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



# 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:

  $( t_1 + \dots t_n ) = ( s_1 + \dots + s_n ),$

  where $s_1 \dots s_n$ is a permutation of $t_1 \dots t_n$
  - 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

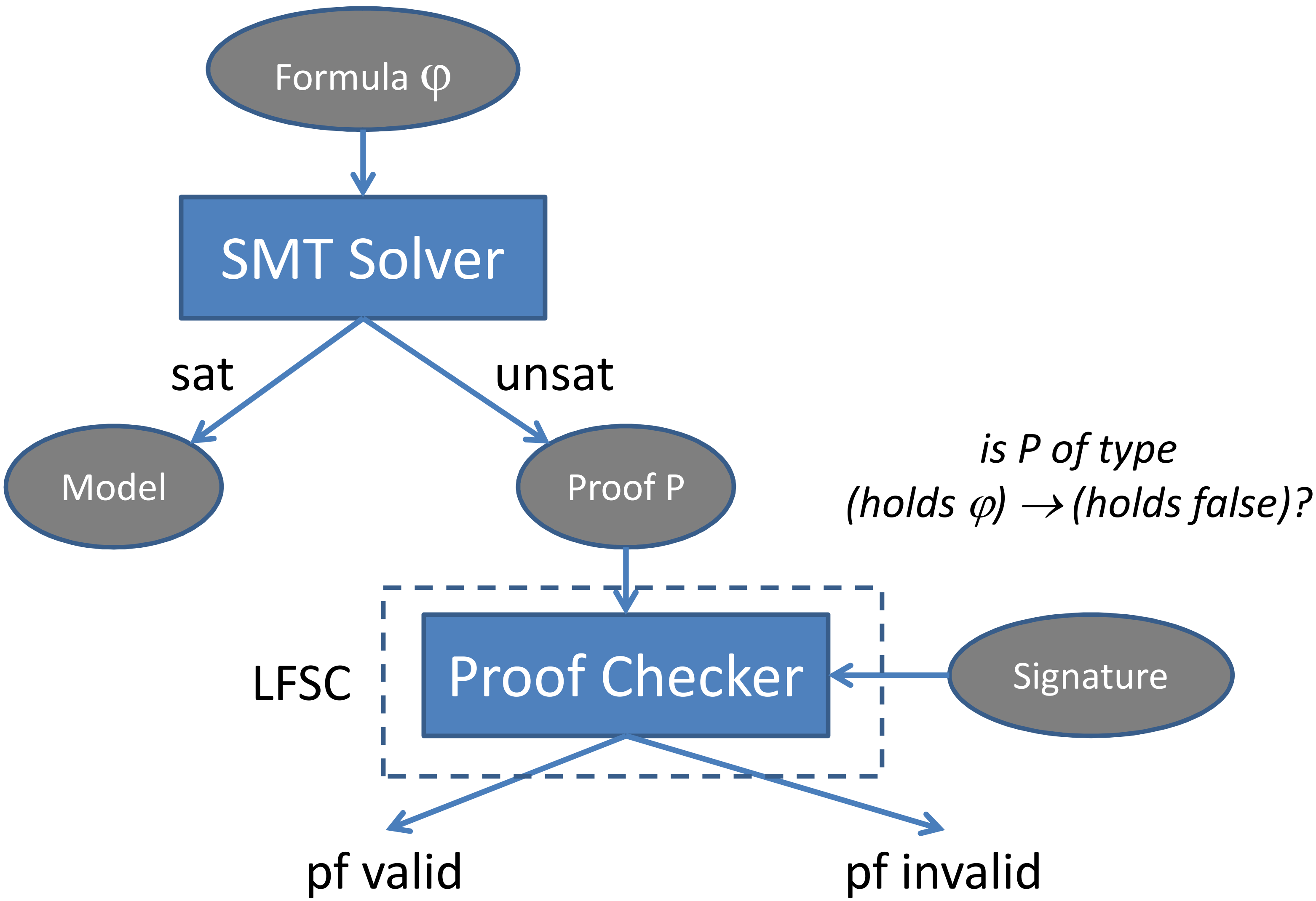
# LFSC: Proof Checker for SMT

- 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



# 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 ( $t_1 - t_2$ )$\downarrow$
  - 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

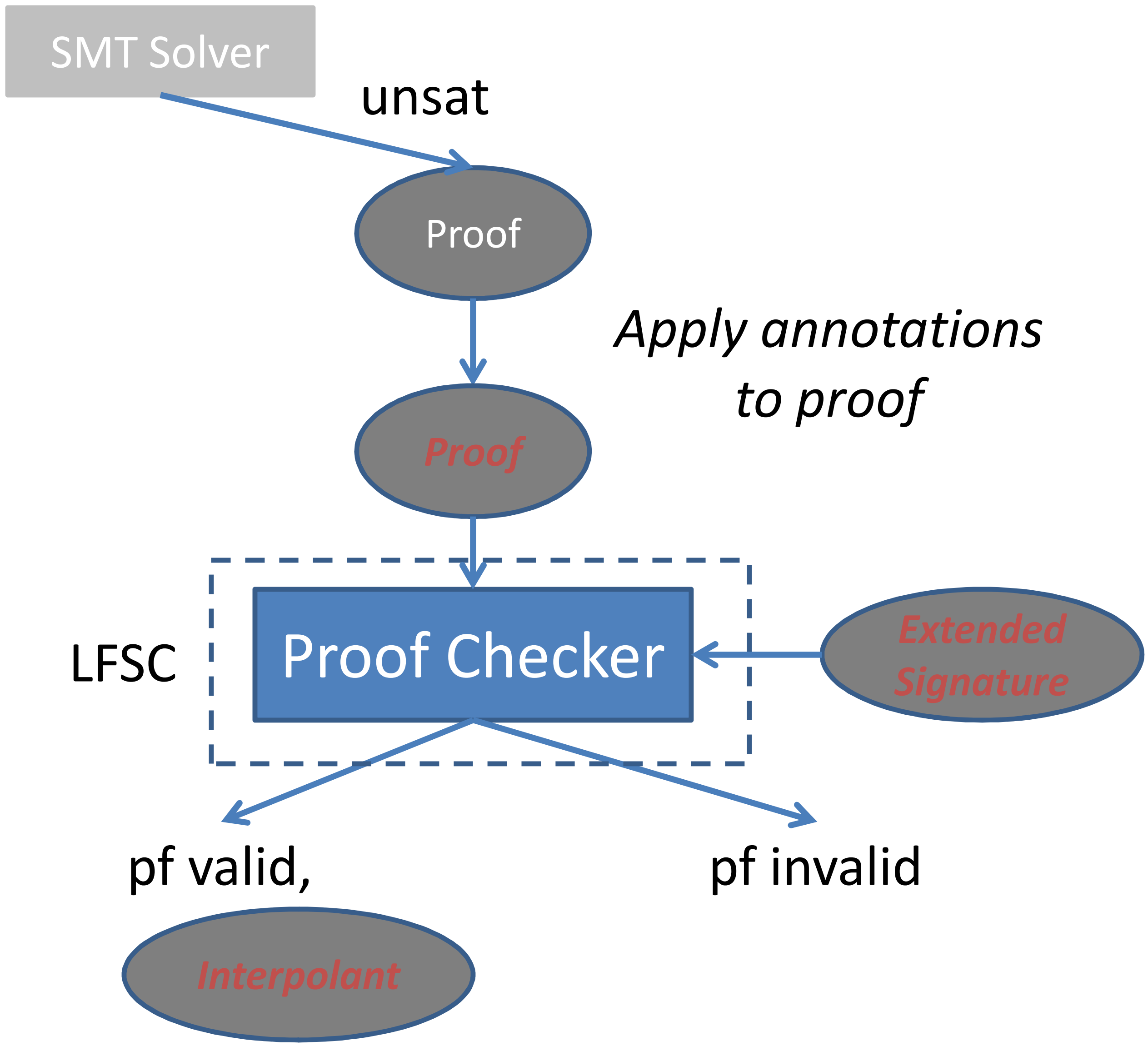
- Experiments on SMT LIB benchmarks
- Used CVC3 for proof generation
- Computational proof system is advantageous
  - For proofs of theory lemmas:
    - 5x reduction in proof size
    - 2.5x reduction in proof checking time
- Proof checking in both systems is fast
  - 10x faster than solving time

# Interpolant Generation [Reynolds et al 11]

- Interpolant for inconsistent formulas (A,B)
  - Summarizes the inconsistency, in language of A ∩ 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:

$$\frac{\varphi_1 \quad \dots \quad \varphi_n}{\varphi}\ \text{rule} \quad \Rightarrow \quad \frac{\varphi_1[I_1] \quad \dots \quad \varphi_n[I_n]}{\varphi[I]}\ \text{rule}'$$

# Certified Interpolant Generation



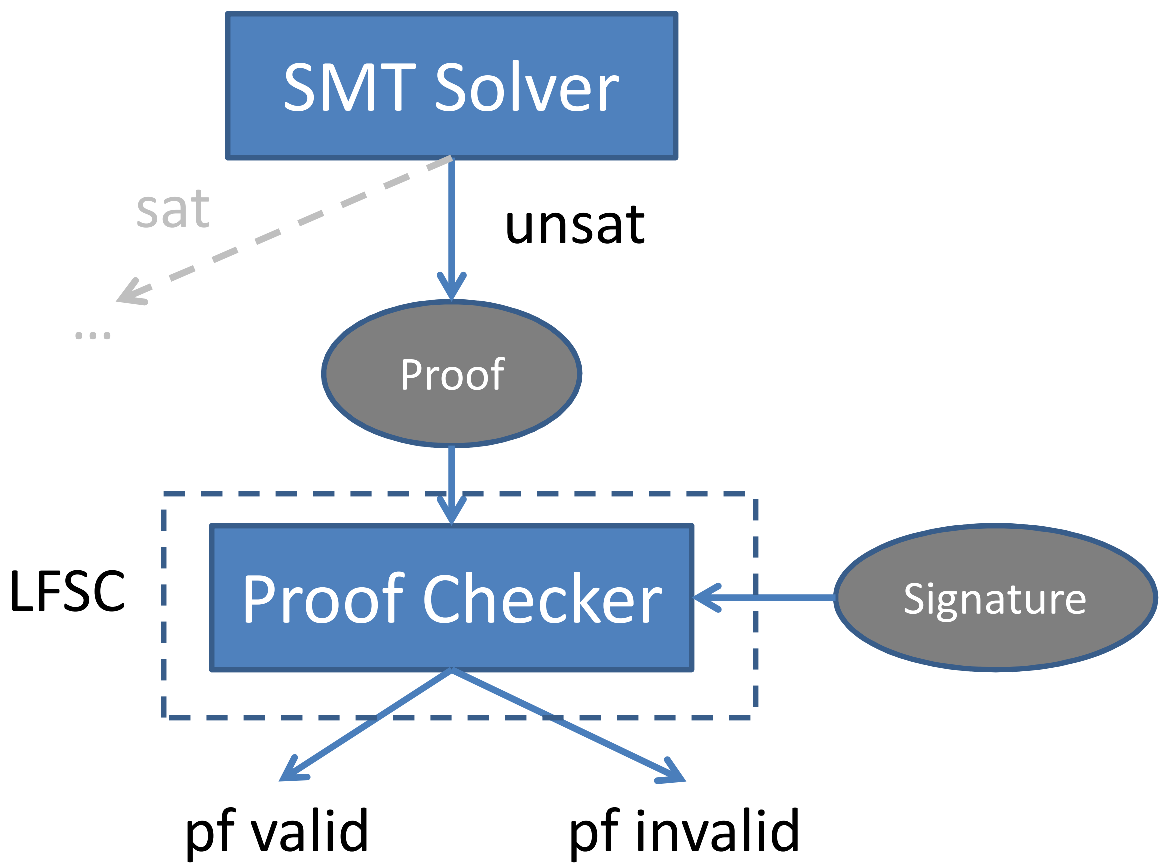
# Certified Interpolant Generation

- 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*

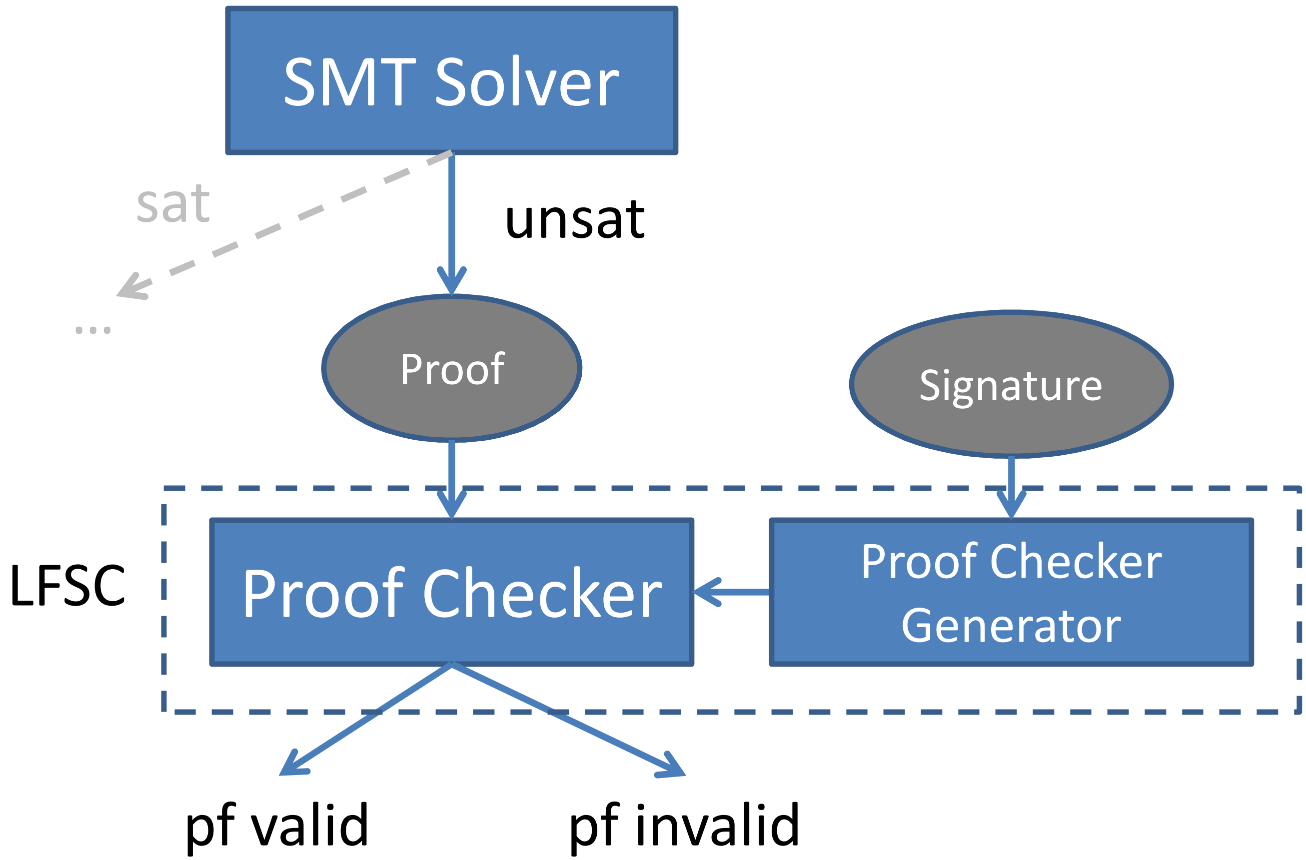
- User-friendly language for defining Pf signatures
  - Surface language
  - Core language
    - Translation from surface to core language
- Highly optimized proof checker
  - Signature compilation
    - Side conditions as well as type checking rules
  - Implicit arguments for proof rules
    - Reduction in proof size

# LFSC : Proof Checker




- For optimization, compile signature into proof checker

# LFSC : Proof Checker Generator




- *Generic translation of signature into C++ code for proof checker*

# Example Proof System

$$\begin{array}{lcl} \textit{formulas}\ \phi & ::= & p \mid \phi_1 \rightarrow \phi_2 \\ \textit{contexts}\ \Gamma & ::= & \cdot \mid \Gamma, \phi \end{array}$$

$$\frac{\phi \in \Gamma}{\Gamma \vdash \phi}\ \textit{Assump} \qquad \frac{\Gamma, \phi_1 \vdash \phi_2}{\Gamma \vdash \phi_1 \rightarrow \phi_2}\ \textit{ImpIntro}$$

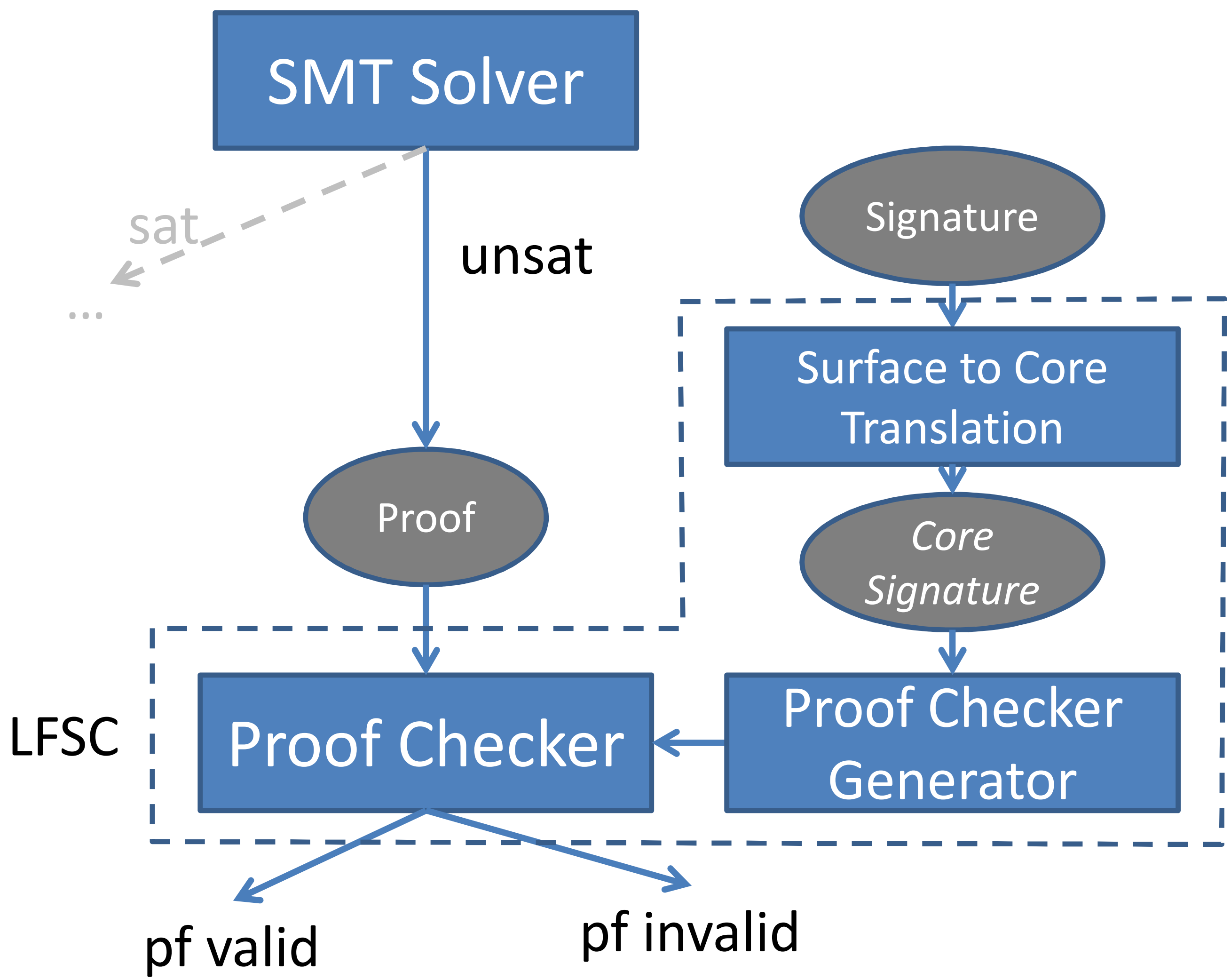
$$\frac{\Gamma \vdash \phi_1 \rightarrow \phi_2 \qquad \Gamma \vdash \phi_1}{\Gamma \vdash \phi_2}\ \textit{ImpElim}$$

# Example Proof System in LF

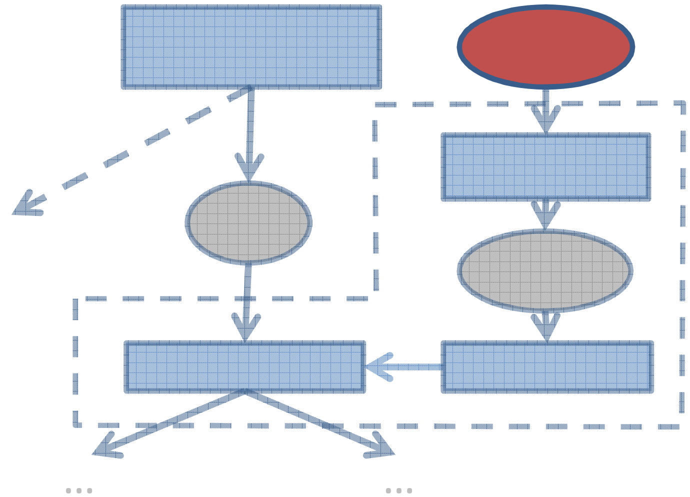
```
formula : Type;
imp : formula -> formula -> formula;

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

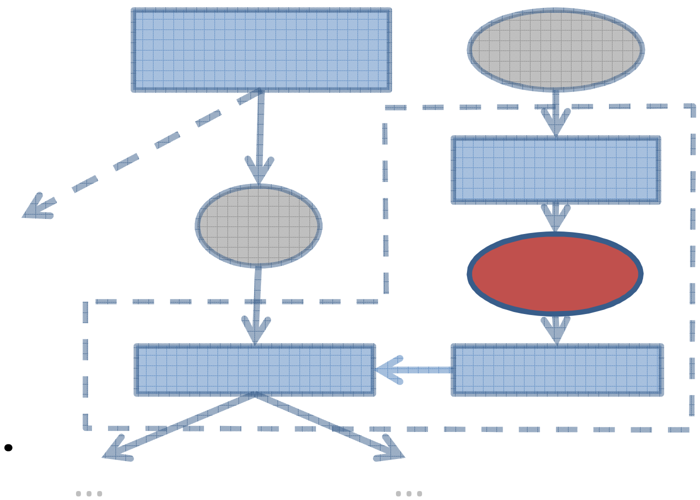


# Surface Language



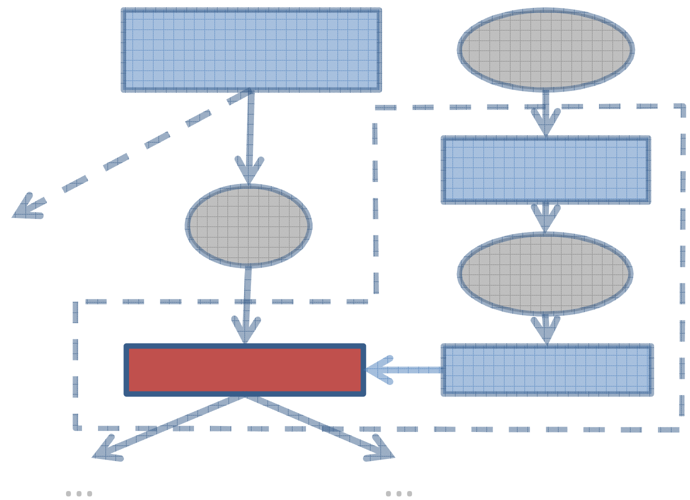
```
SYNTAX
   formula f ::= imp f1 f2.
JUDGMENTS
   (holds f)
RULES
   [ holds f1 ] |- holds f2
   -----------------------------  imp_intro
   holds (imp f1 f2) .

   holds (imp f1 f2) ,  holds f1
   -----------------------------  imp_elim
   holds f2 .
```

# Core Language



```
tctor formula : Type .

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

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

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

ctor imp_elim :
  Pi-(f1:formula, f2:formula).
  Pi+(p1:(holds (imp f1 f2)), p2:(holds f1)).
            (holds f2).
```

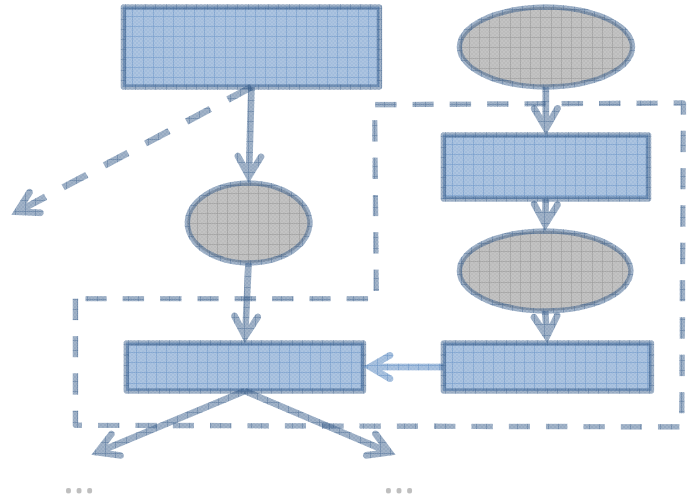
# Compiled C++



```
...
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");
  }
}
```

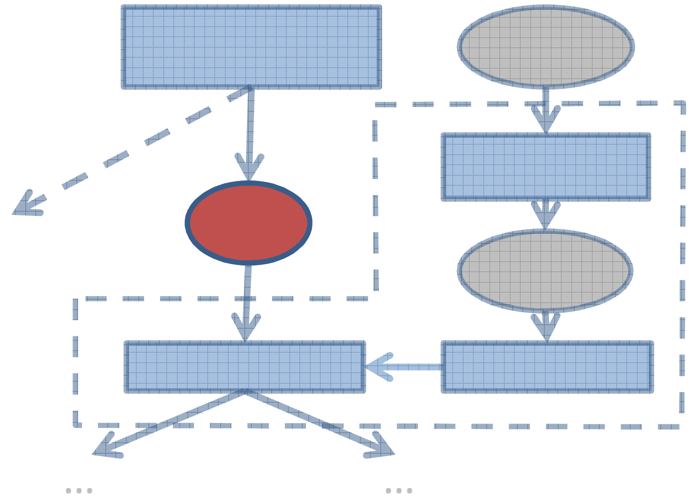
- *Actual generated C++ code is highly optimized*

# Example Proof



$$\dfrac{\dfrac{\dfrac{\dfrac{}{p, (p \to q) \vdash (p \to q)} \qquad \dfrac{}{p, (p \to q) \vdash p}}{p, (p \to q) \vdash q}}{p \vdash (p \to q) \to q}}{\cdot \vdash p \to ((p \to q) \to q)}$$

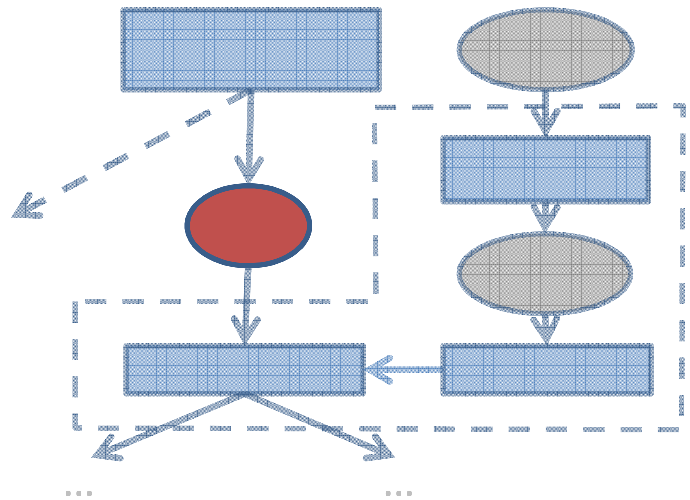
# Example Proof : LFSC



$$\dfrac{\dfrac{\dfrac{\overline{p, (p \to q) \vdash (p \to q)} \qquad \overline{p, (p \to q) \vdash p}}{p, (p \to q) \vdash q}}{p \vdash (p \to q) \to q}}{\cdot \vdash p \to ((p \to q) \to q)}$$

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

# Example Proof : LFSC



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

```
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 ::= arrow s1 s2 | bool .

term<sort> t ::=
              true<bool>
            | false<bool>
            | (not t1<bool>)<bool>
            | (and t1<bool> t2<bool>)<bool>
            | (or t1<bool> t2<bool>)<bool>
            ...
            | (ite t1<bool> t2<s> t3<s>)<s>
            | (forall t<s> ^ t<bool>)<bool>
            | (apply t1<arrow s1 s2> t2<s1>)<s2>
            | (eq t1<s> t2<s>)<bool>.

formula f ::= t<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?