Towards Reliable and Maintainable Constraint Solvers
Andres Nötzli, Stanford University
noetzli@stanford.edu

Background
- Use cases for Satisfiability modulo theories (SMT) include program verification:
  
  ```
  if (x != -1) {
    foo = bar / (x + 1);
  }
  ```

  Can division by zero occur?
  
  - Query SMT solver:
    
    \[(x \neq -1) \land (x + 1 = 0)\]

  - Need to be able to trust output...
    
    ... but constraint solvers are complex
    
    Susceptible to bugs
  - Increase confidence in answer by providing an independently checkable proof:
    
    Proof describes reasoning, proof checker makes sure that reasoning is consistent with proof rules

SMT query  \rightarrow \text{Proof}

Proof rules  \rightarrow \text{Proof checker}

Motivation
- Preprocessing simplifies formulas
- All SMT solvers rely on preprocessing for good performance (and sometimes correctness)
- SMT solvers produce proofs for core procedures but not preprocessing steps
- Manual implementation is tedious and error-prone:
  - Hundreds of rules
  - Solver has to produce proof for each rule
  - Proof checker has to be able to check all rules

Example

**Rewrite rule**

Name: writeOverRead
(stores #a #i (select #a #i)) \Rightarrow #a

```cpp
if (node[0] == node[2][0] &&
    node[1] == node[2][1] &&
    node.getKind() == kind::STORE &&
    node[2].getKind() == kind::SELECT) {
  return RewriteResponse(REWRITE_DONE, node[0]);
}
```

**Logical Framework with Side Conditions (LFSC) proof rule**

```latex
\text{(declare wor}
\text{
  (! s1 sort)
  (! s2 sort)
  (! i (term s1))
  (! oa (term (Array s1 s2)))
  (! a (term (Array s1 s2)))
  (! u (th_holds (= _ oa)
      (apply _ _ (apply _ _ (write s1 s2) a) i))
    (apply _ _ (apply _ _ (read s1 s2) a) i)))
  (th_holds (= _ oa a))))
```

**C++ code performing the rewrite**

```cpp
if (node[0] == node[2][0] &&
    node[1] == node[2][1] &&
    node.getKind() == kind::STORE &&
    node[2].getKind() == kind::SELECT) {
  return RewriteResponse(REWRITE_DONE, node[0]);
}
```

**Condition**

**Target expression**

Idea
- Most of the preprocessing module of a solver can be expressed as a set of rewrite rules:

- Use a domain-specific language for rewrite rules
- Implement a compiler that:
  - Generates code to perform the rewrite including code to produce a proof
  - Generates proof rule for the proof checker
  - Supports reasoning about rewrite rules

The Domain-Specific Language
- Design goals: intuitive but expressive enough for most rewrite rules
- Rules consist of a source template, a target template and a condition (optional)
- Source template: pattern that SMT solver is searching for
- Condition: evaluated at runtime by SMT solver
- Expression is replaced to match target template if source template matches and condition is fulfilled

Implementation
- Currently targeting CVC4, which uses the LFSC meta-logic for proofs and proof rules
- Challenges:
  - Generate efficient code \rightarrow optimize across multiple rewrite rules
  - Proofs that are simple to produce and efficient to check