Integration of SMT Solvers with ITPs — There and Back Again

Sascha Böhme and Tjark Weber

ARG Lunch
2 March 2010
1 Introduction

2 There . . .
   - Features: SMT-LIB vs. Yices
   - Translation Techniques
   - Caveats

3 . . . and Back Again
   - Z3’s Proofs
   - LCF-style Theorem Proving
   - Reconstruction Techniques
   - Performance

4 Conclusions
HOL4 and Isabelle/HOL are popular interactive theorem provers.

Interactive theorem proving benefits from automation.
HOL4 and Isabelle/HOL are popular interactive theorem provers.

Interactive theorem proving benefits from automation.

We want to use SMT solvers to decide SMT formulas.
Integration of SMT Solvers with Interactive Theorem Provers

System Overview

Isabelle / HOL4

Input formula ➔ Negation, translation

Error message ➔

Theorem ➔ Proof reconstruction

SMT solver

satisfiable?

SMT

Model

Proof

Tjark Weber

Introduction

There . . .

. . . and Back Again

Conclusions

Motivation

System Overview

Higher-Order Logic

Satisfiability Modulo Theories
System Overview

Tjark Weber
Integration of SMT Solvers with Interactive Theorem Provers
System Overview

There ...

Input formula
Negation, translation

Error message
Proof reconstruction

Theorem

Proof
Model

SMT solver

satisfiable?

Isabelle / HOL4

SMT

Tjark Weber
Integration of SMT Solvers with Interactive Theorem Provers
Polymorphic $\lambda$-calculus, based on Church’s simple theory of types:

- $\sigma ::= \alpha \mid (\sigma_1, \ldots, \sigma_n) c$
- $t ::= x_\sigma \mid c_\sigma \mid (t_\sigma \to \tau \mid t_\sigma) \tau \mid (\lambda x_\sigma . t_\tau)_{\sigma \to \tau}$
Polymorphic $\lambda$-calculus, based on Church’s simple theory of types:

- $\sigma ::= \alpha \mid (\sigma_1, \ldots, \sigma_n) c$
- $t ::= x_{\sigma} \mid c_{\sigma} \mid (t_{\sigma \to \tau} t_{\sigma})_{\tau} \mid (\lambda x_{\sigma}. t_{\tau})_{\sigma \to \tau}$

Sufficient for much of mathematics and computer science:

- quantifiers of arbitrary order
- arithmetic ($\text{nat, int, real, } \ldots$)
- data types (lists, records, bit vectors, $\ldots$)

Extensive libraries with thousands of theorems
Goal: To decide the satisfiability of (quantifier-free) first-order formulas with respect to combinations of (decidable) background theories.

\[ \varphi ::= A \mid \neg \varphi \mid \varphi \lor \varphi \mid \varphi \land \varphi \]
Satisfiability Modulo Theories: Example

Theories:
- $\mathcal{I}$: theory of integers
  \[ \Sigma_{\mathcal{I}} = \{\leq, +, -, 0, 1\} \]
- $\mathcal{L}$: theory of lists
  \[ \Sigma_{\mathcal{L}} = \{=, \text{hd}, \text{tl}, \text{nil}, \text{cons}\} \]
- $\mathcal{E}$: theory of equality

Problem: Is
\[ x \leq y \land y \leq x + \text{hd}\,(\text{cons}\,0\,\text{nil}) \land P\,(f\,x - f\,y) \land \neg P\,0 \]
satisfiable in $\mathcal{I} \cup \mathcal{L} \cup \mathcal{E}$?
We must translate HOL formulas into the input language of SMT solvers.

1. SMT-LIB format
2. Yices’s native format
### Features: SMT-LIB vs. Yices

<table>
<thead>
<tr>
<th>Feature</th>
<th>SMT-LIB</th>
<th>Yices</th>
<th>SMT-LIB</th>
<th>Yices</th>
</tr>
</thead>
<tbody>
<tr>
<td>int, real</td>
<td>✓</td>
<td>✓</td>
<td>(✓)</td>
<td>✓</td>
</tr>
<tr>
<td>nat, bool, →</td>
<td>❌</td>
<td>✓</td>
<td>❌</td>
<td>✓</td>
</tr>
<tr>
<td>prop. logic</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>equality</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FOL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>HOL</td>
<td>❌</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>arithmetic</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>let</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>λ-terms</td>
<td>❌</td>
<td>✓</td>
<td>❌</td>
<td>✓</td>
</tr>
<tr>
<td>tuples</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>records</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>data types</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>bit vectors</td>
<td></td>
<td></td>
<td>(✓)</td>
<td>✓</td>
</tr>
</tbody>
</table>
Recursion & Abstraction

We translate HOL formulas by recursion over their term structure:

$$[[P_{\alpha \rightarrow \text{bool}} \ x_{\alpha}]] = ([[P_{\alpha \rightarrow \text{bool}}]] \ [[x_{\alpha}]]$$

Abstraction is used to deal with unsupported terms/types.
Recursion & Abstraction

We translate HOL formulas by recursion over their term structure:

\[
\left[ P_{\alpha \rightarrow \text{bool}} \ x_\alpha \right] = \left[ P_{\alpha \rightarrow \text{bool}} \right] \left[ x_\alpha \right]
\]

**Abstraction** is used to deal with unsupported terms/types.

**SMT-LIB**

- :extrasorts (a)
- :extrafuns ((x a))
- :extrapreds ((P a))
- :formula (not (P x))

**Yices**

- (define-type a)
- (define P::(-> a bool))
- (define x::a)
- (assert (not (P x)))
Basic Techniques

A simple dictionary approach is sufficient for many HOL constants (e.g., propositional logic, arithmetic, bit vectors).
A simple dictionary approach is sufficient for many HOL constants (e.g., propositional logic, arithmetic, bit vectors).

We try to replace HOL constants without SMT counterparts: terms are $\beta$-normalized, some constants (e.g., $\in$) are unfolded.
A simple **dictionary** approach is sufficient for many HOL constants (e.g., propositional logic, arithmetic, bit vectors).

We try to **replace** HOL constants without SMT counterparts: terms are $\beta$-normalized, some constants (e.g., $\in$) are unfolded.

We add (universally quantified) **definitions** for certain other HOL constants (e.g., $\text{min}$, $\text{max}$).
A simple dictionary approach is sufficient for many HOL constants (e.g., propositional logic, arithmetic, bit vectors).

We try to replace HOL constants without SMT counterparts: terms are $\beta$-normalized, some constants (e.g., $\in$) are unfolded.

We add (universally quantified) definitions for certain other HOL constants (e.g., $\text{min}$, $\text{max}$).

Some terms require special code (e.g., numerals, quantifiers).
Monomorphisation

In HOL, types can depend on type parameters. Since Yices only supports monomorphic types, we may need to create multiple copies of a polymorphic data type.
In HOL, types can depend on type parameters. Since Yices only supports monomorphic types, we may need to create multiple copies of a polymorphic data type.

Example: \texttt{datatype } \alpha \texttt{ list = NIL \mid CONS } \alpha \alpha \texttt{ list}

\begin{verbatim}
(define-type a)
(define-type a-list (datatype
   a-NIL (a-CONS a-hd::a a-tl::a-list)))

(define-type b)
(define-type b-list (datatype
   b-NIL (b-CONS b-hd::b b-tl::b-list)))
\end{verbatim}
Uniformly generating *fresh identifiers* is easier than re-using HOL identifiers.
Uniformly generating fresh identifiers is easier than re-using HOL identifiers.

There are subtle semantic differences between certain HOL and (allegedly corresponding) SMT-LIB/Yices functions.
Uniformly generating **fresh identifiers** is easier than re-using HOL identifiers.

There are subtle **semantic differences** between certain HOL and (allegedly corresponding) SMT-LIB/Yices functions.

Yices “does no checking and can behave unpredictably if given bad input.” The burden to produce **correct input** for the SMT solver is on our translation.
What if there is a **bug** in the translation . . . or in the SMT solver?
What if there is a **bug** in the translation . . . or in the SMT solver?

We require the SMT solver to produce a **proof** of unsatisfiability.

The proof is then **checked** (automatically) in the interactive prover.
Z3 is a leading SMT solver. It generates natural deduction proofs.

Z3’s proof calculus consists of 34 axiom schemata and inference rules—some simple, some very powerful.
Z3 is a leading SMT solver. It generates natural deduction proofs.

Z3’s proof calculus consists of 34 axiom schemata and inference rules—some simple, some very powerful.

\[
\begin{align*}
\neg R_1 & \quad \vdash \varphi \quad \text{ASSUME} \\
\vdash \varphi \quad \text{R}_2 \\
\therefore & \quad \text{R}_3 \\
\vdash \varphi \quad \text{R}_4 \\
\therefore & \quad \text{R}_5 \\
\vdash \bot \quad \text{R}_6
\end{align*}
\]
Z3 is a leading SMT solver. It generates natural deduction proofs.

Z3’s proof calculus consists of 34 axiom schemata and inference rules—some simple, some very powerful.

Proofs are directed acyclic graphs. Nodes are inference steps.
Z3 is a leading SMT solver. It generates natural deduction proofs.

Z3’s proof calculus consists of 34 axiom schemata and inference rules—some simple, some very powerful.

Proofs are directed acyclic graphs. Nodes are inference steps.
Z3 is a leading SMT solver. It generates natural deduction proofs.

Z3’s proof calculus consists of 34 axiom schemata and inference rules—some simple, some very powerful.

Proofs are directed acyclic graphs. Nodes are inference steps.

Proofs can be checked by depth-first (postorder) traversal.
Z3 is a leading SMT solver. It generates natural deduction proofs.

Z3’s proof calculus consists of 34 axiom schemata and inference rules—some simple, some very powerful.

Proofs are directed acyclic graphs. Nodes are inference steps.

Proofs can be checked by depth-first (postorder) traversal.
Z3’s Proofs

Z3 is a leading SMT solver. It generates natural deduction proofs.

Z3’s proof calculus consists of 34 axiom schemata and inference rules—some simple, some very powerful.

Proofs are directed acyclic graphs. Nodes are inference steps.

Proofs can be checked by depth-first (postorder) traversal.
Z3 is a leading SMT solver. It generates natural deduction proofs.

Z3’s proof calculus consists of 34 axiom schemata and inference rules—some simple, some very powerful.

Proofs are directed acyclic graphs. Nodes are inference steps.

Proofs can be checked by depth-first (postorder) traversal.
Z3 is a leading SMT solver. It generates natural deduction proofs.

Z3’s proof calculus consists of 34 axiom schemata and inference rules—some simple, some very powerful.

Proofs are directed acyclic graphs. Nodes are inference steps.

Proofs can be checked by depth-first (postorder) traversal.
Z3 is a leading SMT solver. It generates *natural deduction* proofs.

Z3’s proof calculus consists of 34 axiom schemata and inference rules—some simple, some very *powerful*.

Proofs are *directed acyclic graphs*. Nodes are inference steps.

Proofs can be checked by *depth-first* (postorder) traversal.
Z3 is a leading SMT solver. It generates natural deduction proofs.

Z3’s proof calculus consists of 34 axiom schemata and inference rules—some simple, some very powerful.

Proofs are directed acyclic graphs. Nodes are inference steps.

Proofs can be checked by depth-first (postorder) traversal.
Z3's Proofs

Z3 is a leading SMT solver. It generates natural deduction proofs.

Z3’s proof calculus consists of 34 axiom schemata and inference rules—some simple, some very powerful.

Proofs are directed acyclic graphs. Nodes are inference steps.

Proofs can be checked by depth-first (postorder) traversal.
Z3 is a leading SMT solver. It generates natural deduction proofs.

Z3’s proof calculus consists of 34 axiom schemata and inference rules—some simple, some very powerful.

Proofs are directed acyclic graphs. Nodes are inference steps.

Proofs can be checked by depth-first (postorder) traversal.
Z3 is a leading SMT solver. It generates natural deduction proofs.

Z3’s proof calculus consists of 34 axiom schemata and inference rules—some simple, some very powerful.

Proofs are directed acyclic graphs. Nodes are inference steps.

Proofs can be checked by depth-first (postorder) traversal.
Z3 is a leading SMT solver. It generates natural deduction proofs.

Z3’s proof calculus consists of 34 axiom schemata and inference rules—some simple, some very powerful.

Proofs are directed acyclic graphs. Nodes are inference steps.

Proofs can be checked by depth-first (postorder) traversal.
Z3 is a leading SMT solver. It generates natural deduction proofs.

Z3’s proof calculus consists of 34 axiom schemata and inference rules—some simple, some very powerful.

Proofs are directed acyclic graphs. Nodes are inference steps.

Proofs can be checked by depth-first (postorder) traversal.
Z3 is a leading SMT solver. It generates natural deduction proofs.

Z3’s proof calculus consists of 34 axiom schemata and inference rules—some simple, some very powerful.

Proofs are directed acyclic graphs. Nodes are inference steps.

Proofs can be checked by depth-first (postorder) traversal.
Theorems are implemented as an abstract data type. There is a fixed number of constructor functions—one for each axiom schema/inference rule of HOL.

More complicated proof procedures must be implemented by composing these functions.
Theorems are implemented as an **abstract data type**. There is a **fixed number** of constructor functions—one for each axiom schema/inference rule of HOL. More complicated proof procedures must be implemented by **composing** these functions.

**The trusted code base** consists only of the theorem ADT.
Proof procedures are more difficult to implement.

Proof procedures are less efficient.
Reconstruction Techniques

1. A single primitive inference rule or theorem instantiation
2. Combinations of primitive inferences/instantiations
3. Automated proof procedures
4. Combinations of the above
Reconstruction Techniques

1. A single primitive inference rule or theorem instantiation
2. Combinations of primitive inferences/instantiations
3. Automated proof procedures
4. Combinations of the above

Implementation of Z3’s inference rules:
Performance Optimizations

Profiling is essential!

- Avoiding automated proof procedures
- Schematic theorems
- Theorem memoization
- Generalization

Speed-ups of up to 3 orders of magnitude
About two thirds of Z3’s proof rules perform propositional or simple first-order reasoning. They *could be* implemented by a single call to an automated proof procedure.

😊 Rapid prototyping 💡

😭 Bad performance 😟
Avoiding Automated Proof Procedures

About two thirds of Z3’s proof rules perform propositional or simple first-order reasoning. They could be implemented by a single call to an automated proof procedure.

😊 Rapid prototyping 🧠
😊 Bad performance 🕒

Instead, we use derived rules: combinations of primitive inferences of manageable size that perform specific reasoning tasks.

Example:

\[
\frac{\vdash \bigwedge_{i=1}^{n} \varphi_i}{\vdash \bigwedge_{i=1}^{n} \varphi_{\pi(i)}} \quad \text{REWRITE}
\]
Schematic Theorems

Instantiating a generic theorem is typically much faster than proving the specific instance using primitive inferences alone.

Examples:

- $\vdash (p \implies q) \iff (\neg p \lor q)$
- $\vdash (x = y) \iff (y = x)$
- $\vdash x + 0 = x$

😊 Over 230 theorems allow about 76% of all \texttt{Rewrite} goals to be proved by instantiation.
Theorem Memoization

Theorems derived by **Rewrite** and **Th-Lemma** are indexed by a term net and **re-used** rather than re-proved when possible.
Theorem Memoization

Theorems derived by \texttt{Rewrite} and \texttt{Th-Lemma} are indexed by a term net and \textit{re-used} rather than re-proved when possible.
Theorems derived by \texttt{Rewrite} and \texttt{Th-Lemma} are indexed by a term net and \texttt{re-used} rather than re-proved when possible.
Theorems derived by \texttt{Rewrite} and \texttt{Th-Lemma} are indexed by a term net and \textit{re-used} rather than re-proved when possible.
Theorems derived by \texttt{Rewrite} and \texttt{Th-Lemma} are indexed by a term net and \textit{re-used} rather than re-proved when possible.
Theorems derived by \texttt{Rewrite} and \texttt{Th-Lemma} are indexed by a term net and \textit{re-used} rather than re-proved when possible.
Theorem Memoization

Theorems derived by \texttt{Rewrite} and \texttt{Th-Lemma} are indexed by a term net and \texttt{re-used} rather than re-proved when possible.
Theorem Memoization

Theorems derived by **Rewrite** and **Th-Lemma** are indexed by a term net and **re-used** rather than re-proved when possible.
Theorem Memoization

Theorems derived by Rewrite and Th-Lemma are indexed by a term net and re-used rather than re-proved when possible.
Theorem Memoization

Theorems derived by \texttt{Rewrite} and \texttt{Th-Lemma} are indexed by a term net and \textit{re-used} rather than re-proved when possible.
Theorem Memoization

Theorems derived by \texttt{REWRITE} and \texttt{TH-LEMMA} are indexed by a term net and \textit{re-used} rather than re-proved when possible.
Theorems derived by **Rewrite** and **Th-Lemma** are indexed by a term net and re-used rather than re-proved when possible.
Theorem Memoization

Theorems derived by **REWRITE** and **TH-LEMMA** are indexed by a term net and **re-used** rather than re-proved when possible.
Theorems derived by \texttt{Rewrite} and \texttt{Th-Lemma} are indexed by a term net and \texttt{re-used} rather than re-proved when possible.
Generalization

Goals proved by Th-Lemma are generalized before being passed to a theory-specific decision procedure.

Example:
\[ \vdash \text{some lengthy expression} < \text{some lengthy expression} + 1 \text{ is a theorem of linear arithmetic} \text{—instead we prove } \vdash x < x + 1. \]

Avoids expensive preprocessing in the decision procedure

More potential for theorem re-use
## Evaluation

<table>
<thead>
<tr>
<th>Logic</th>
<th>Solved (Z3)</th>
<th>Reconstructed</th>
<th>Ratios</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#</td>
<td>Time</td>
<td>Size</td>
</tr>
<tr>
<td>AUFLIA⁺⁺</td>
<td>187</td>
<td>0.095 s</td>
<td>64 KB</td>
</tr>
<tr>
<td>AUFLIA⁻⁻</td>
<td>192</td>
<td>0.117 s</td>
<td>81 KB</td>
</tr>
<tr>
<td>AUFLIRA</td>
<td>189</td>
<td>0.292 s</td>
<td>366 KB</td>
</tr>
<tr>
<td>QF_AUFLIA</td>
<td>92</td>
<td>0.158 s</td>
<td>694 KB</td>
</tr>
<tr>
<td>QF_IDL</td>
<td>40</td>
<td>2.322 s</td>
<td>12 MB</td>
</tr>
<tr>
<td>QF_LIA</td>
<td>100</td>
<td>17.154 s</td>
<td>77 MB</td>
</tr>
<tr>
<td>QF_LRA</td>
<td>88</td>
<td>4.849 s</td>
<td>10 MB</td>
</tr>
<tr>
<td>QF_RDL</td>
<td>52</td>
<td>9.773 s</td>
<td>16 MB</td>
</tr>
<tr>
<td>QF_UF</td>
<td>87</td>
<td>16.131 s</td>
<td>62 MB</td>
</tr>
<tr>
<td>QF_UFIDL</td>
<td>55</td>
<td>4.511 s</td>
<td>12 MB</td>
</tr>
<tr>
<td>QF_UFLIA</td>
<td>91</td>
<td>1.543 s</td>
<td>4 MB</td>
</tr>
<tr>
<td>QF_UFLRA</td>
<td>100</td>
<td>0.086 s</td>
<td>914 KB</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1273</td>
<td>3.656 s</td>
<td>13 MB</td>
</tr>
</tbody>
</table>
### Evaluation

<table>
<thead>
<tr>
<th>Logic</th>
<th>Solved (Z3)</th>
<th>Reconstructed</th>
<th>Ratios</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#</td>
<td>Time</td>
<td>Size</td>
</tr>
<tr>
<td>Total</td>
<td>1273</td>
<td>3.656 s</td>
<td>13 MB</td>
</tr>
</tbody>
</table>

😊 We can check sizeable proofs with millions of inferences.

😢 Proof search in Z3 is almost 20 times faster (on average) than LCF-style proof reconstruction.

- Not enough proof information for theory-specific reasoning.
Integration of SMT solvers with HOL4 and Isabelle/HOL

SMT-LIB is restrictive—custom translations seem more worthwhile than sophisticated SMT-LIB encodings.

Z3’s proofs could be easier to check.

LCF-style proof checking for SMT is feasible.

Isabelle: http://isabelle.in.tum.de/
HOL4: http://hol.sourceforge.net/

Related papers at http://www.cl.cam.ac.uk/~tw333/
Future Work

- A **more expressive** SMT-LIB format (Version 2.0?!) 
- A better SMT **proof format** (a standard?!) 
- Proof reconstruction for **bit vectors**
- Case studies, applications
Future Work

- A more expressive SMT-LIB format (Version 2.0?!) 
- A better SMT proof format (a standard?!) 
- Proof reconstruction for bit vectors 
- Case studies, applications 

Thank You!