Integration of SMT Solvers with ITPs — There and Back Again

Sascha Böhme and Tjark Weber

TECHNISCHE UNIVERSITÄT MÜNCHEN

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

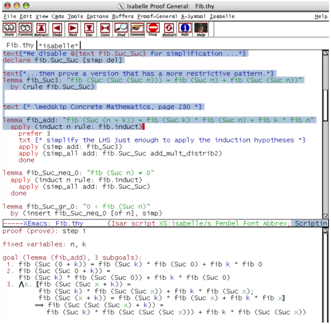
ARG Lunch

2 March 2010

Motivation

HOL4 and Isabelle/HOL are popular interactive theorem provers.

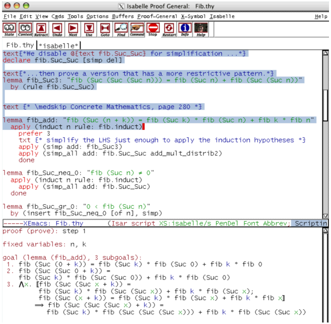
Interactive theorem proving benefits from automation.



Motivation

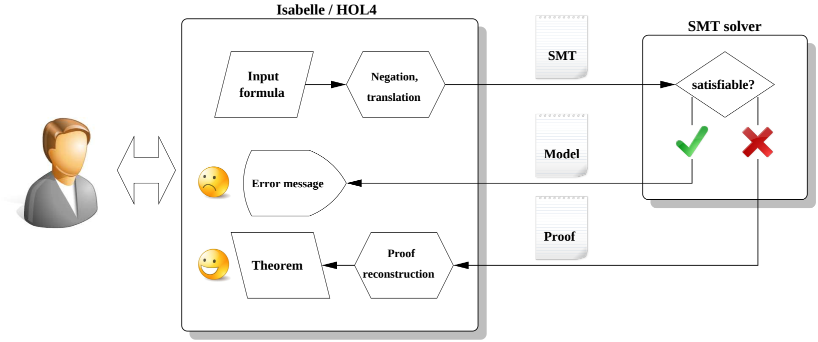
HOL4 and Isabelle/HOL are popular interactive theorem provers.

Interactive theorem proving benefits from automation.

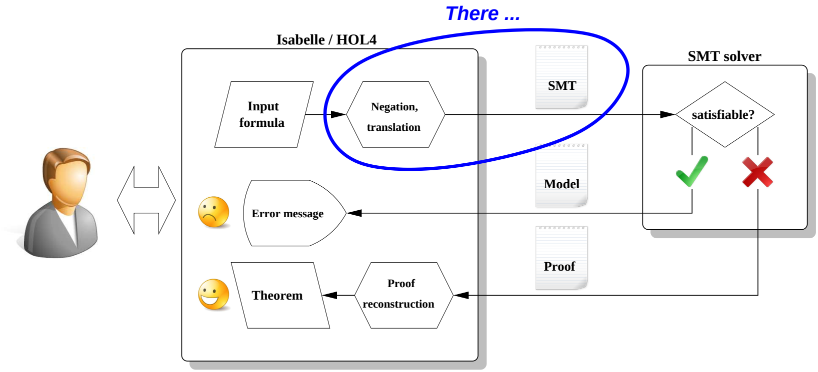


We want to use SMT solvers to decide SMT formulas.

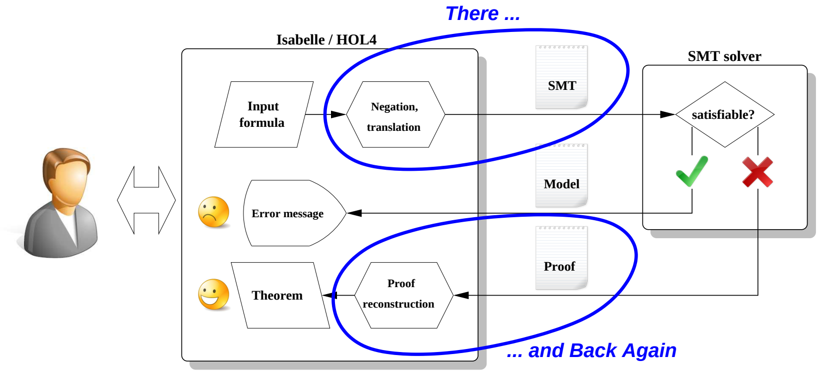
System Overview



System Overview



System Overview



Higher-Order Logic

Polymorphic λ-calculus, based on Church's simple theory of types:

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

Higher-Order Logic

Polymorphic λ-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 (nat, int, real, . . .)
- data types (lists, records, bit vectors, . . .)

Extensive libraries with thousands of theorems

Satisfiability Modulo Theories

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

$$\varphi ::= \mathcal{A} \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi$$

Satisfiability Modulo Theories: Example

Theories:

- $\mathcal{I}$: theory of integers
  $\Sigma_{\mathcal{I}} = \{\leq, +, -, 0, 1\}$
- $\mathcal{L}$: theory of lists
  $\Sigma_{\mathcal{L}} = \{=, \mathsf{hd}, \mathsf{tl}, \mathsf{nil}, \mathsf{cons}\}$
- $\mathcal{E}$: theory of equality
  Σ: free function and predicate symbols

Problem: Is

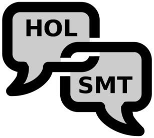
$$x \leq y \wedge y \leq x + \mathsf{hd}\,(\mathsf{cons}\,0\,\mathsf{nil}) \wedge P\,(f\,x - f\,y) \wedge \neg\,P\,0$$

satisfiable in $\mathcal{I} \cup \mathcal{L} \cup \mathcal{E}$?

There . . .

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

	SMT-LIB	Yices		SMT-LIB	Yices
int, real	✓	✓	let	(✓)	✓
nat, bool, $\rightarrow$	✗	✓	λ-terms	✗	✓
prop. logic	✓	✓	tuples	✗	✓
equality	✓	✓	records	✗	✓
FOL	✓	✓	data types	✗	✓
HOL	✗	✓	bit vectors	✓	✓
arithmetic	✓	✓			

Recursion & Abstraction

We translate HOL formulas by recursion over their term structure:

$$[\![P_{\alpha \to \mathsf{bool}}\ x_\alpha]\!] = ([\![P_{\alpha \to \mathsf{bool}}]\!]\ [\![x_\alpha]\!])$$

Abstraction is used to deal with unsupported terms/types.

Recursion & Abstraction

We translate HOL formulas by recursion over their term structure:

$$[\![P_{\alpha\to\mathrm{bool}}\ x_\alpha]\!] = ([\![P_{\alpha\to\mathrm{bool}}]\!]\ [\![x_\alpha]\!])$$

Abstraction is used to deal with unsupported terms/types.

SMT-LIB

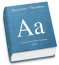
```
:extrasorts (a)
:extrafuns ((x a))
:extrapreds ((P a))
:formula (not (P x))
```

Yices

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

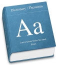
Basic Techniques

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



Basic Techniques

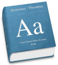
A simple dictionary approach is sufficient for many HOL constants (e.g., propositional logic, arithmetic, bit vectors).



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

Basic Techniques

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

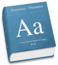


We try to replace HOL constants without SMT counterparts: terms are β-normalized, some constants (e.g., $\in$) are unfolded.

We add (universally quantified) definitions for certain other HOL constants (e.g., min, max).

Basic Techniques

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 β-normalized, some constants (e.g., $\in$) are unfolded.

We add (universally quantified) definitions for certain other HOL constants (e.g., min, 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.

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.

Example: datatype α list $=$ NIL | CONS α α list

```
(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)))
```

Caveats

Uniformly generating fresh identifiers is easier than re-using HOL identifiers.

Caveats

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.

$[\![\bullet]\!]$

Caveats

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.

$[\![\bullet]\!]$

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.

... and Back Again

What if there is a bug in the translation ... or in the SMT solver?

... and Back Again

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's Proofs

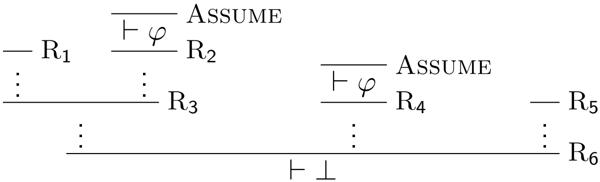
Z3 is a leading SMT solver. It generates natural deduction proofs.

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

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.

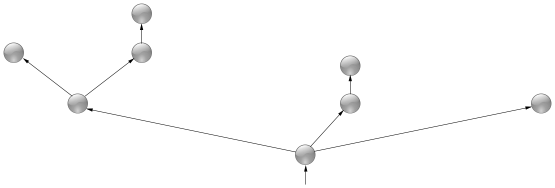


Z3's Proofs

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.

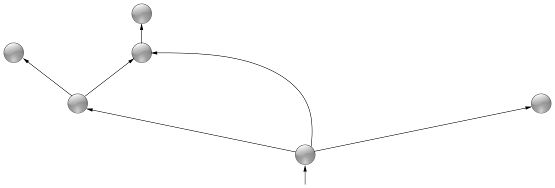


Z3's Proofs

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.

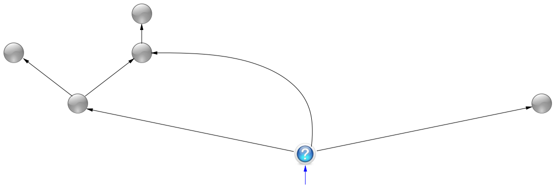


Z3's Proofs

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.



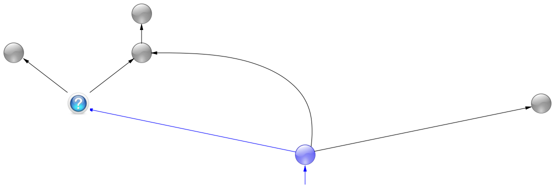
Proofs can be checked by depth-first (postorder) traversal.

Z3'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.



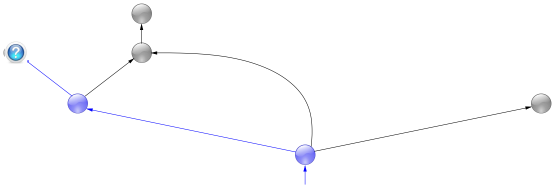
Proofs can be checked by depth-first (postorder) traversal.

Z3'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.



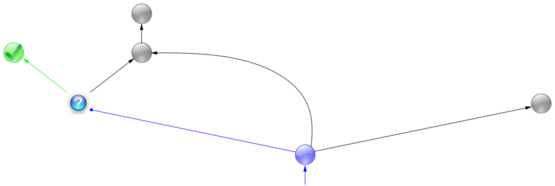
Proofs can be checked by depth-first (postorder) traversal.

Z3'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.



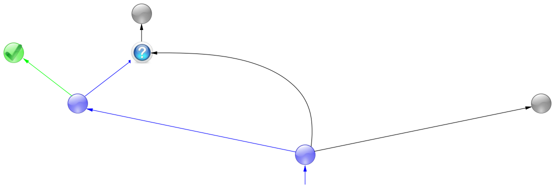
Proofs can be checked by depth-first (postorder) traversal.

Z3'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.



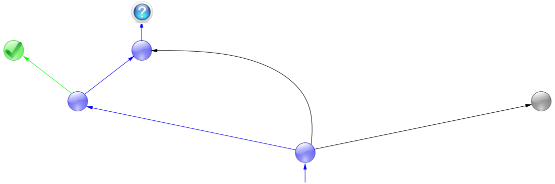
Proofs can be checked by depth-first (postorder) traversal.

Z3'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.



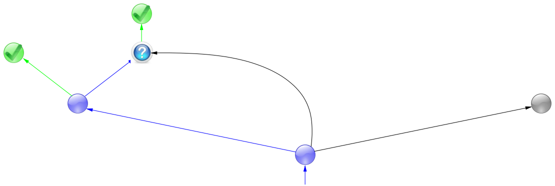
Proofs can be checked by depth-first (postorder) traversal.

Z3'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.



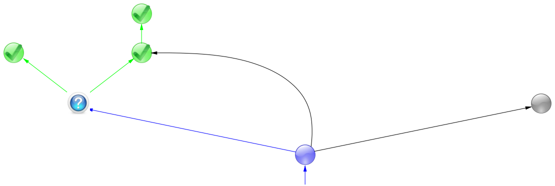
Proofs can be checked by depth-first (postorder) traversal.

Z3'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.



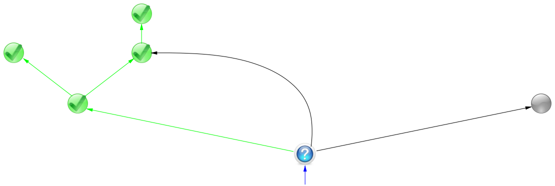
Proofs can be checked by depth-first (postorder) traversal.

Z3'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.



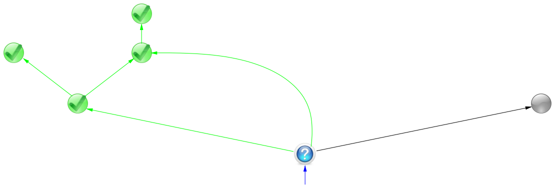
Proofs can be checked by depth-first (postorder) traversal.

Z3'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.



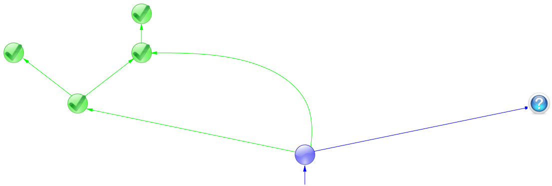
Proofs can be checked by depth-first (postorder) traversal.

Z3'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.



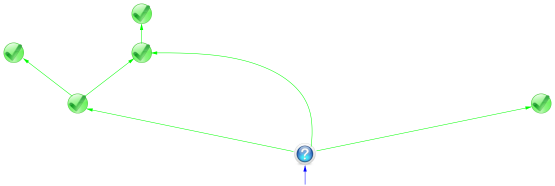
Proofs can be checked by depth-first (postorder) traversal.

Z3'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.



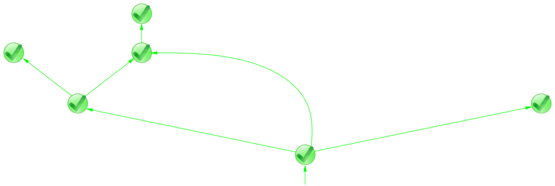
Proofs can be checked by depth-first (postorder) traversal.

Z3'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.

LCF-style Theorem Proving

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.

LCF-style Theorem Proving

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.

LCF-style Theorem Proving — Disadvantages

- Proof procedures are more difficult to implement. 
- Proof procedures are less efficient. 

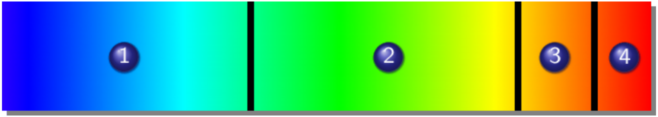
Reconstruction Techniques

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

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

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:

$$\frac{\vdash \bigwedge_{i=1}^{n} \varphi_i}{\vdash \bigwedge_{i=1}^{n} \varphi_{\pi(i)}}\ \textsc{Rewrite}$$

Schematic Theorems

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

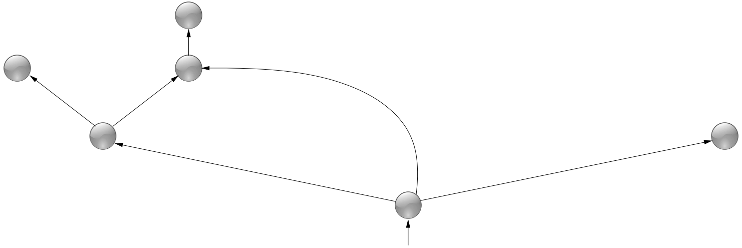
Examples:

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

Over 230 theorems allow about 76% of all REWRITE goals to be proved by instantiation.

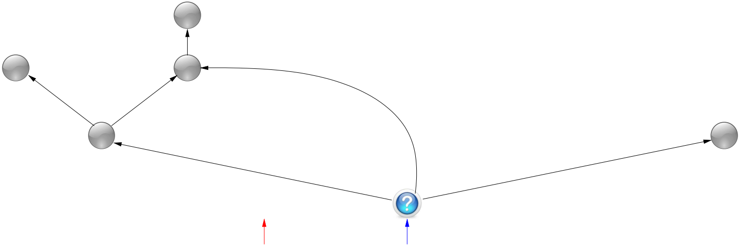
Theorem Memoization

Theorems derived by REWRITE and TH-LEMMA are indexed by a term net and re-used rather than re-proved when possible.



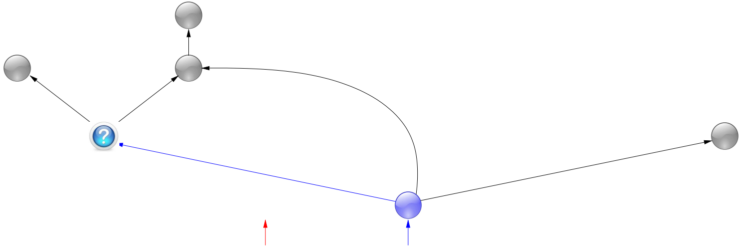
Theorem Memoization

Theorems derived by REWRITE and TH-LEMMA are indexed by a term net and re-used rather than re-proved when possible.



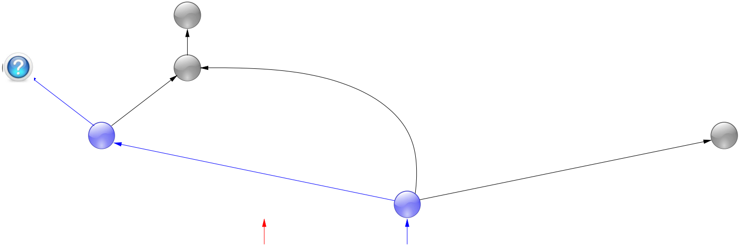
Theorem Memoization

Theorems derived by REWRITE and TH-LEMMA are indexed by a term net and re-used rather than re-proved when possible.



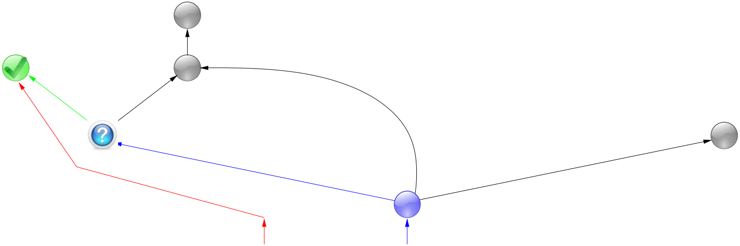
Theorem Memoization

Theorems derived by REWRITE and TH-LEMMA are indexed by a term net and re-used rather than re-proved when possible.



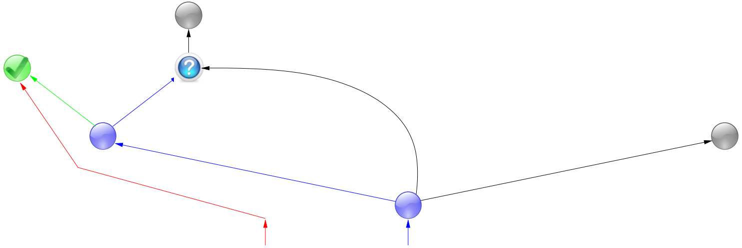
Theorem Memoization

Theorems derived by REWRITE and TH-LEMMA are indexed by a term net and re-used rather than re-proved when possible.



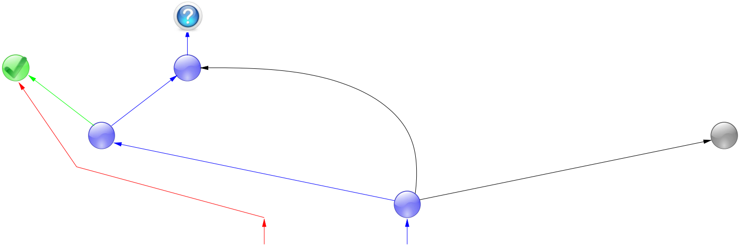
Theorem Memoization

Theorems derived by REWRITE and TH-LEMMA are indexed by a term net and re-used rather than re-proved when possible.



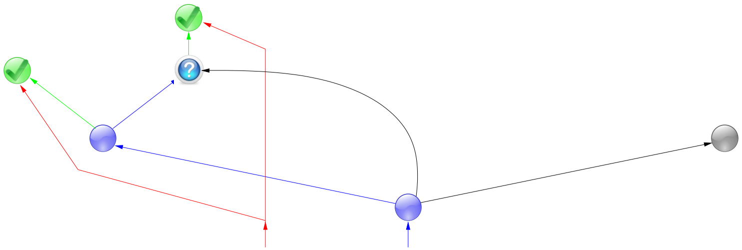
Theorem Memoization

Theorems derived by REWRITE and TH-LEMMA are indexed by a term net and re-used rather than re-proved when possible.



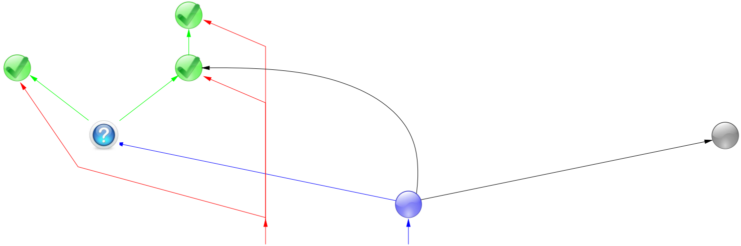
Theorem Memoization

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



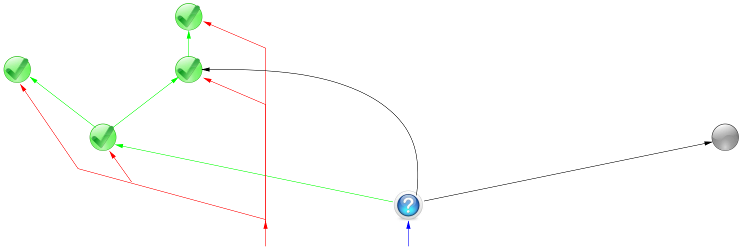
Theorem Memoization

Theorems derived by REWRITE and TH-LEMMA are indexed by a term net and re-used rather than re-proved when possible.



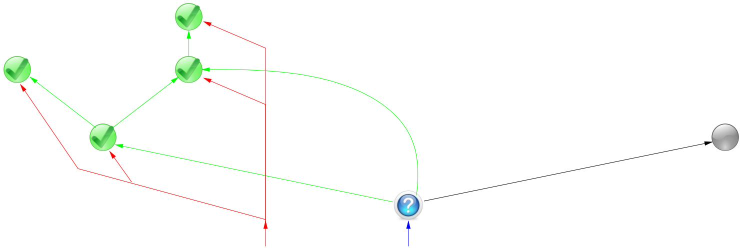
Theorem Memoization

Theorems derived by REWRITE and TH-LEMMA are indexed by a term net and re-used rather than re-proved when possible.



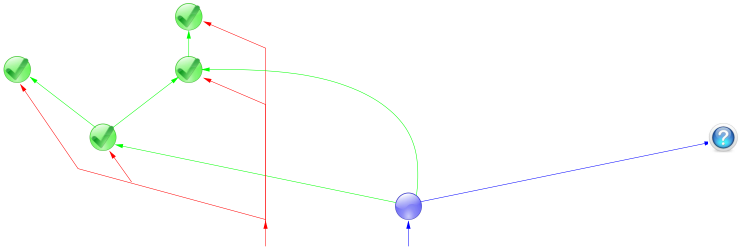
Theorem Memoization

Theorems derived by REWRITE and TH-LEMMA are indexed by a term net and re-used rather than re-proved when possible.



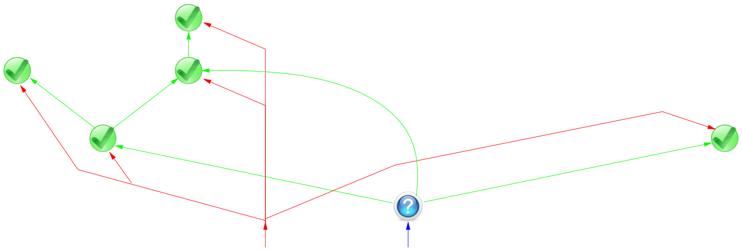
Theorem Memoization

Theorems derived by REWRITE and TH-LEMMA are indexed by a term net and re-used rather than re-proved when possible.



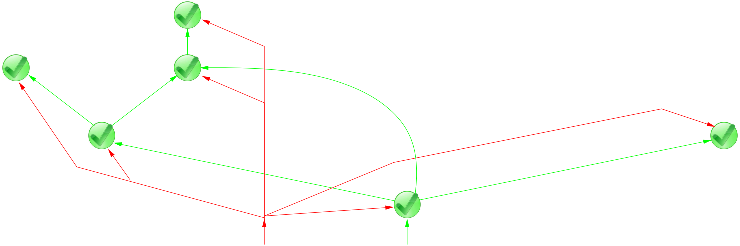
Theorem Memoization

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 REWRITE and TH-LEMMA are indexed by a term net and 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:
$\vdash \text{some lengthy expression} < \text{some lengthy expression} + 1$ is a theorem of linear arithmetic—instead we prove $\vdash x < x + 1$.

- Avoids expensive preprocessing in the decision procedure
- More potential for theorem re-use

Evaluation

Logic	Solved (Z3)			Reconstructed		Ratios		
	#	Time	Size	#	Time	Success	Timeout	Time
AUFLIA+p	187	0.095 s	64 KB	187	0.413 s	100%	0%	4.34
AUFLIA−p	192	0.117 s	81 KB	190	1.962 s	98%	0%	16.72
AUFLIRA	189	0.292 s	366 KB	144	0.794 s	76%	0%	2.72
QF_AUFLIA	92	0.158 s	694 KB	49	136.498 s	53%	42%	863.85
QF_IDL	40	2.322 s	12 MB	19	173.875 s	47%	52%	74.89
QF_LIA	100	17.154 s	77 MB	26	208.713 s	26%	65%	12.17
QF_LRA	88	4.849 s	10 MB	55	142.351 s	62%	36%	29.36
QF_RDL	52	9.773 s	16 MB	26	173.953 s	50%	50%	17.80
QF_UF	87	16.131 s	62 MB	73	73.242 s	83%	16%	4.54
QF_UFIDL	55	4.511 s	12 MB	8	260.351 s	14%	85%	57.72
QF_UFLIA	91	1.543 s	4 MB	85	29.086 s	93%	6%	18.85
QF_UFLRA	100	0.086 s	914 KB	100	3.916 s	100%	0%	45.68
Total	1273	3.656 s	13 MB	962	67.785 s	75%	19%	18.54

Evaluation

Logic	Solved (Z3)			Reconstructed		Ratios		
	#	Time	Size	#	Time	Success	Timeout	Time
Total	1273	3.656 s	13 MB	962	67.785 s	75%	19%	18.54

- 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.

Conclusions

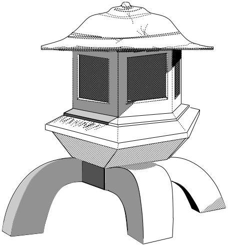
Integration of SMT solvers with HOL4 and Isabelle/HOL

- 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!