Automatic Proof and Disproof in Isabelle/HOL

Jasmin Blanchette, Lukas Bulwahn,
Tobias Nipkow

Fakultät für Informatik
TU München

A tale of two worlds

FOL	*HOL*
$f(s,t)$	$f\ s\ t,\ \ f\ s,\ \ \lambda x.t$
Otter (1987)	*Isabelle* (1986)

They did not talk to each other
because they spoke different languages.

This is the tale of how these two worlds
began to understand and boost each other.

Isabelle

- is an interactive theorem prover
- that has always embraced automation
- but without sacrificing soundness:

All proofs
must ultimately go through the Isabelle kernel

This is the *LCF principle* (Robin Milner).

Two decades of Isabelle development

1990s Basic proof automation
Our own proof search in ML:
simplifier, automatic provers, arithmetic

2000s Embrace external tools
Let them do the proof search,
but don't trust them:
ATPs (FOL provers)
SMT solvers
SAT solvers
Programming languages

Simplifier

N.

- First and higher-order equations (λ)
- Conditional equations
- Contextual simplification
- Special solvers (eg orderings)
- Arithmetic
- Case splitting (triggered by `if` and `case`)
- Large library of default equations

Isabelle's workhorse

The power of Isabelle's internal automated proof methods

- relies on large sets of default rules
- that are user-extensible (`[simp]`)
- and tuned over time.

Tableaux prover

Paulson

- Based on leanT^AP (Beckert & Posegga)
- Generic
- User-extensible by `intro` and `elim` rules
- Proof search in ML,
  proof checking via Isabelle kernel
- Works well for pure logic and set theory
- Does not know anything about equality

Isabelle Demo

Sledgehammer

Paulson et al.

- Connects Isabelle with ATPs and SMT solvers

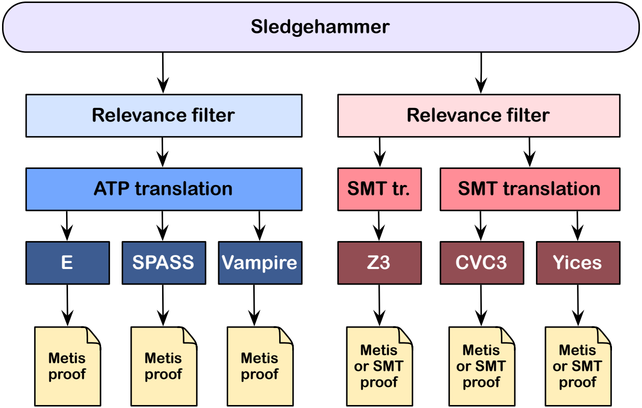
  E, SPASS, Vampire, CVC3, Yices, Z3, ...

- One-click invocation:
  - Users don't need to select facts
  - ... or ensure the problem is first-order

- Exploits local parallelism, remote servers

Sledgehammer: Demo

Sledgehammer: Architecture



Sledgehammer: Fact selection

Meng & Paulson

Provers perform poorly given 1000s of facts

A lightweight, symbol-based filter greatly improves the success rate

Number of facts is optimized for each prover

Sledgehammer: Translation

Meng & Paulson   Bl., Böhme & Smallbone

Source: higher-order, polymorphism + type classes

Target: first-order, untyped/simply-typed

1. Firstorderize
   - SK combinators, λ-lifting
   - Explicit application operator
2. Encode types
   - Monomorphize
   - ... or encode polymorphism

Sledgehammer: Reconstruction

Paulson & Susanto   Böhme & Weber

Four approaches (the 4 Rs):

A. Re-find using Metis
B. Rerun external prover
C. Recheck stored proof
D. Recast into Isar proof

A. Re-find using Metis

lemma length (tl *xs*) $\leq$ length *xs*
by (metis append_Nil2 append_eq_conv_conj
drop_eq_Nil drop_tl tl.simps(1))

Usually fast and reliable

Metis sometimes too slow (5% loss on avg)

B. Rerun external prover

lemma length (tl *xs*) $\leq$ length *xs*
by (smt append_Nil2 append_eq_conv_conj
drop_eq_Nil drop_tl tl.simps(1))

Reinvokes the SMT solver each time!

C. Recheck stored proof

lemma length (tl *xs*) $\leq$ length *xs*
by (smt append_Nil2 append_eq_conv_conj
drop_eq_Nil drop_tl tl.simps(1))

Fast
No need for SMT solver for replay

Fragile

D. Recast into Isar proof

```
lemma length (tl xs) ≤ length xs
proof –
  have tl [] = [] by (metis tl.simps(1))
  hence ∃u. xs @ u = xs ∧ tl u = [] by (metis append_Nil2)
  hence tl (drop (length xs) xs) = [] by (metis append_eq_conv_conj)
  hence drop (length xs) (tl xs) = [] by (metis drop_tl)
  thus length (tl xs) ≤ length xs by (metis drop_eq_Nil)
qed
```

Fast, self-explanatory

Experimental, bulky

Sledgehammer: Judgment Day

Böhme & N.   Bl., Böhme & Paulson

- 1240 goals arising in 7 older theories

  Arrow, FFT, FTA, Hoare, Jinja, NS, SN

- In 2010: E, SPASS, Vampire (5 to 120 s)

  ESV $\times$ 5 s $\approx$ V $\times$ 120 s

- In 2011: Also E-SInE, CVC3, Yices, Z3 (30 s)

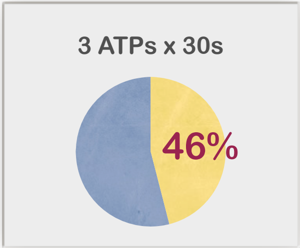
  Z3 $>$ V

- In 2012: Tighter integration with SPASS

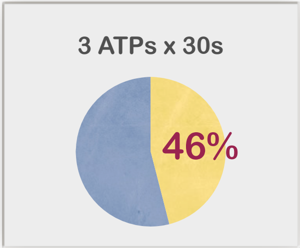
  SPASS most successful backend (by a small margin)

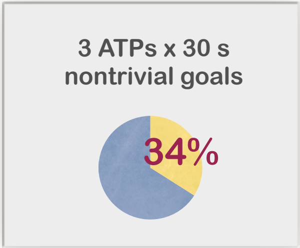
2010

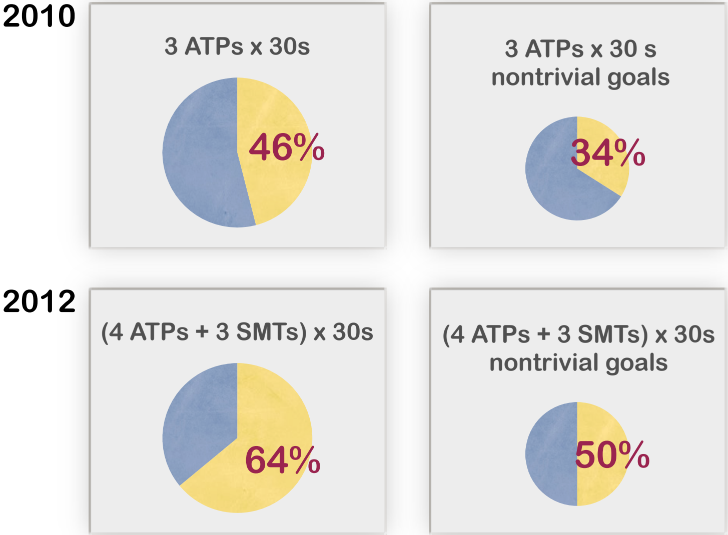
2010



2010






2010
3 ATPs x 30s
46%
3 ATPs x 30 s
nontrivial goals
34%
2012
(4 ATPs + 3 SMTs) x 30s
64%
(4 ATPs + 3 SMTs) x 30s
nontrivial goals
50%

Sledgehammer & Teaching

Paulson

Old way: Low-level tactics + lemma libraries

New way: Isar + Sledgehammer + simp etc.

lemma *blah*
sorry
proof –
  have *blah*$_0$ **sorry by** (metis foo bar)
  hence *blah*$_1$ **sorry by** metis
  hence *blah*$_2$ **sorry by** auto
  thus *blah* **sorry by** (metis baz)
qed

Sledgehammer: Success story

Guttman, Struth & Weber

Developed large Isabelle/HOL repository of algebras for modeling imperative programs
(Kleene Algebra, Hoare logic, . . . , $\approx$ 1000 lemmas)

Intricate refinement and termination theorems

Surprise: Sledgehammer and Z3 automate algebraic proofs at textbook level!

"The integration of ATP, SMT, and Nitpick is for our purposes **very very helpful**." — G. Struth

Theorem proving and testing

Testing can show only the presence of errors, but not their absence. *(Dijkstra)*

Testing cannot prove theorems, but it can refute conjectures!

Two facts of life:

- 95% of all conjectured theorems are wrong.
- Theorem proving is an expensive debugging technique.

Theorem provers need counterexample finders!

Quickcheck

Berghofer & N. Bul.

- Adds lightweight validation by testing
- Motivated by Haskell's QuickCheck
- Employs Isabelle's code generator
- Quick response time
- No-click invocation:
  automatic after parsing a proposition
  (well, at least in ProofGeneral)

Quickcheck: Demo

Quickcheck

Berghofer & N.   Bul.

- Covers different testing approaches
  - Random and exhaustive testing
  - Smart test data generators
  - Narrowing-based testing
- Creates test data generators automatically

Test generators for datatypes

Fast iteration over the large number of tests using continuation-passing-style programming:

For datatype α *list* $=$ Nil $|$ Cons α $(\alpha$ *list*$)$

we create a test function for property P:

$\mathrm{test}_{\alpha\,list}\ P =$
$\quad P$ Nil andalso
$\quad \mathrm{test}_{\alpha}\ (\lambda x.\ \mathrm{test}_{\alpha\,list}\ (\lambda xs. P\ (\mathrm{Cons}\ x\ xs)))$

Test generators for predicates

Testing propositions with preconditions

$\text{distinct } xs \Longrightarrow \text{distinct } (\text{remove1 } x \; xs)$

Problem:

Exhaustive testing creates useless test data

Solution:

Use precondition's definition for smarter generator

Test generators for predicates

From the definition:

$$\begin{aligned}
&\text{distinct Nil} = \text{True}\\
&\text{distinct } (\text{Cons } x\ xs) = (x \notin xs \wedge \text{distinct } xs)
\end{aligned}$$

we create a test function for property P:

$$\begin{aligned}
&\text{test-distinct}_{\alpha list}\ P =\\
&\quad P \text{ Nil andalso}\\
&\quad \text{test}_{\alpha}\ (\lambda x.\ \text{test-distinct}_{\alpha list}\ (\lambda xs.\\
&\qquad \text{if } x \notin xs \text{ then } P\ (\text{Cons } x\ xs) \text{ else True}))
\end{aligned}$$

Non-distinct lists are never generated

Test generators for predicates

Construct generators using data flow analysis:

1. Transform predicates to system of horn clauses
   $x \notin xs \Longrightarrow \text{distinct } xs \Longrightarrow \mathit{distinct}\ (\text{Cons } x\ xs)$
2. Perform data flow analysis:
   which variables can be computed,
   which variables must be generated?
3. Synthesize test data generator

Narrowing-based testing

- Symbolic execution with demand-driven refinement:
  - Test cases can contain variables
  - If execution cannot proceed, variables are instantiated, again by symbolic terms
- Pays off if large search spaces can be discarded

  distinct (Cons 1 (Cons 1 x)) is false for every x

  No further instantiations for x

Implementations of narrowing

- Programming language with native narrowing

  currently still too slow

- Lazy execution with outer refinement loop

  results in many recomputations, but fast

Limitations

Quickcheck only checks *executable* specifications:

- No equality on functions with infinite domain
- No axiomatic specifications

Nitpick

Bl. & N.

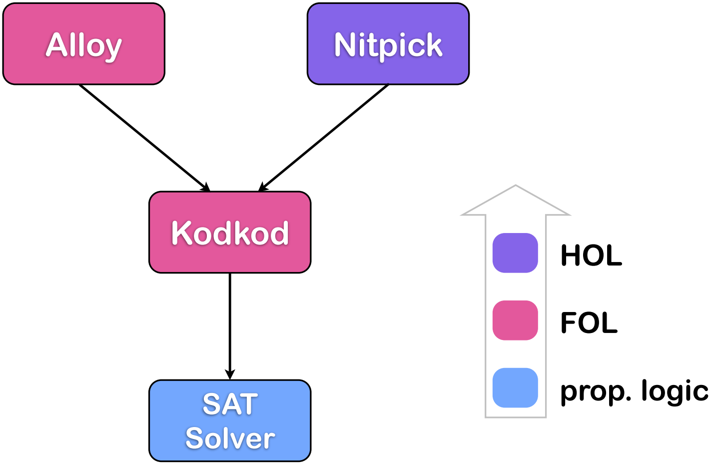
Finite model finder

Based on SAT via Kodkod (Alloy's backend)

Soundly approximates infinite types

Nitpick: Demo

Nitpick: Architecture



Nitpick: Basic translation

For fixed finite cardinalities (1, 2, 3, . . . , 10)

First-order:

$$\tau_1 \to \cdots \to \tau_n \to \mathit{bool} \quad \mapsto \quad A_1 \times \cdots \times A_n$$

$$\tau_1 \to \cdots \to \tau_n \to \tau \quad \mapsto \quad A_1 \times \cdots \times A_n \times A$$

$$+ \text{ constraint}$$

Higher-order args of type $\sigma \to \tau \quad \mapsto$

$$\underbrace{A \times \cdots \times A}_{|\sigma| \text{ times}}$$

Nitpick: Datatypes

Soundly approximated by finite sets (3-valued logic)

Efficient axiomatization:
Subterm-closed substructures (Kuncak & Jackson)

Examples

nat: $\{0, \text{Suc } 0, \text{Suc } (\text{Suc } 0)\}$
α *list*: $\{[], [a_1], [a_2], [a_2, a_1]\}$

Motto: Let the SAT solver spin!
(and trust Kodkod's symmetry breaking)

Nitpick: Inductive predicates

p is the least solution to $p = F(p)$ for some F

Naive idea: Take $p = F(p)$ as p's specification!

Unsound in general, but:

- Sound if recursion $p = F(p)$ is well-founded
- Sound for negative occurrences of p

Otherwise: Unroll! (cf. Biere, Cimatti, Clarke & Zhu)

$p_0 = (\lambda x.\ \text{False}) \qquad p_{i+1} = F(p_i)$

Nitpick: Success stories

Algebraic methods (Guttman, Struth & Weber)

C++ memory model (Bl., Weber, Batty, Owens & Sarkar)

Soundness bugs in TPS and LEO-II

Typical fan mail:

> *"Last night I got stuck on a goal I was sure was a theorem. After 5–10 minutes I gave Nitpick a try, and within a few secs it had found a splendid counterexample—despite the mess of locales and type classes in the context!"*