COMP4161: Advanced Topics in Software Verification

based on slides by J. Blanchette, L. Bulwahn and T. Nipkow

June Andronick, Christine Rizkallah, Miki Tanaka, Johannes Åman Pohjola

T3/2019

data61.csiro.au

Content

- ➜ Intro & motivation, getting started [1]
- ➜ Foundations & Principles
  - Lambda Calculus, natural deduction [1,2]
  - Higher Order Logic, Isar (part 1) [3[a]]
  - Term rewriting [4]
- ➜ Proof & Specification Techniques
  - Inductively defined sets, rule induction [5]
  - Datatypes, recursion, induction, Isar (part 2) [6, 7[b]]
  - Hoare logic, proofs about programs, invariants [8]
  - C verification [9]
  - Practice, questions, exam prep [10[c]]

[a] a1 due; [b] a2 due; [c] a3 due

Overview

Automatic Proof and Disproof

- ➜ Sledgehammer: automatic proofs
- ➜ Quickcheck: counter example by testing
- ➜ Nipick: counter example by SAT

Based on slides by Jasmin Blanchette, Lukas Bulwahn, and Tobias Nipkow (TUM).

Automation

Dramatic improvements in fully automated proofs in the last 2 decades.

- First-order logic (ATP): Otter, Vampire, E, SPASS
- Propositional logic (SAT): MiniSAT, Chaff, RSat
- SAT modulo theory (SMT): CVC3, Yices, Z3

The key:

Efficient reasoning engines, and restricted logics.

Automation in Isabelle

1980s *rule applications, write ML code*

1990s *simplifier, automatic provers (blast, auto), arithmetic*

2000s *embrace external tools, but don't trust them (ATP/SMT/SAT)*

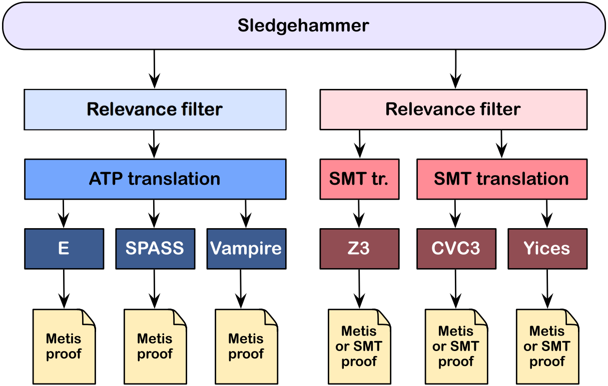
Sledgehammer

Sledgehammer:

- *Connects Isabelle with ATPs and SMT solvers:*
  E, SPASS, Vampire, CVC3, Yices, Z3
- *Simple invocation:*
  - *Users don't need to select or know facts*
  - *or ensure the problem is first-order*
  - *or know anything about the automated prover*
- *Exploits local parallelism and remote servers*

Demo: Sledgehammer

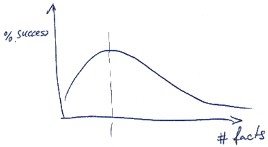
Sledgehammer Architecture



Fact Selection

Provers perform poorly if given 1000s of facts.

- ➜ *Best number of facts depends on the prover*
- ➜ *Need to take care which facts we give them*
- ➜ *Idea: order facts by relevance, give top n to prover ($n = 250, 1000, \ldots$)*
- ➜ *Meng & Paulson method: lightweight, symbol-based filter*
- ➜ *Machine learning method:*
  look at previous proofs to get a probability of relevance



From HOL to FOL

Source: *higher-order, polymorphism, type classes*
Target: *first-order, untyped or simply-typed*

- **First-order:**
  - *SK combinators, λ-lifting*
  - *Explicit function application operator*
- **Encode types:**
  - *Monomorphise (generate multiple instances), or*
  - *Encode polymorphism on term level*

Reconstruction

We don't want to trust the external provers.
Need to check/reconstruct proof.

- ➜ *Re-find using Metis*
  Usually fast and reliable (sometimes too slow)
- ➜ *Rerun external prover for trusted replay*
  Used for SMT. Re-runs prover each time!
- ➜ *Recheck stored explicit external representation of proof*
  Used for SMT, no need to re-run. Fragile.
- ➜ *Recast into structured Isar proof*
  Fast, not always readable.

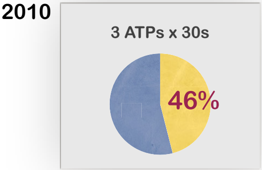
Judgement Day (up to 2013)

Evaluating Sledgehammer:

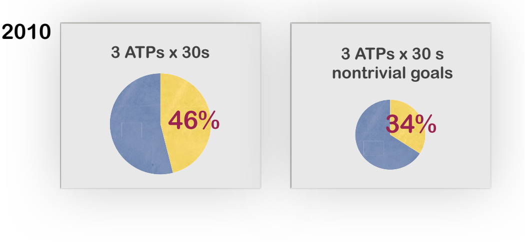
- *1240 goals out of 7 existing theories.*
- *How many can sledgehammer solve?*

- **2010:** *E, SPASS, Vampire (for 5-120s). 46%*
  ESV $\times$ 5*s* $\approx$ *V* $\times$ 120*s*
- **2011:** *Add E-SInE, CVC2, Yices, Z3 (30s).*
  *Z*3 $>$ *V*
- **2012:** *Better integration with SPASS. 64%*
  SPASS best (small margin)
- **2013:** *Machine learning for fact selection. 69%*
  Improves a few percent across provers.

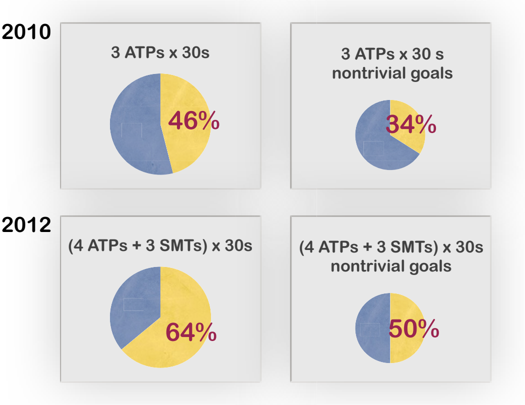
Evaluation



Evaluation



Evaluation



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%

Judgement Day (2016)

Prover	MePo	MaSh	MeSh	Any selector
CVC4 1.5pre	679	749	**783**	830
E 1.8	622	601	**665**	726
SPASS 3.8ds	678	684	**739**	789
Vampire 3.0	703	698	**740**	789
veriT 2014post	543	556	**590**	655
Z3 4.3.2pre	638	668	**703**	788
Any prover	801	885	**919**	943

Fig. 15 Number of successful Sledgehammer invocations per prover on 1230 Judgment Day goals

$$919/1230 = 74\%$$

Sledgehammer rules!

Example application:

- *Large Isabelle/HOL repository of algebras for modelling imperative programs (Kleene Algebra, Hoare logic, . . ., ≈ 1000 lemmas)*
- *Intricate refinement and termination theorems*
- *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

Disproof

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!

Sad facts of life:

- ➜ *Most lemma statements are wrong the first time.*
- ➜ *Theorem proving is expensive as a debugging technique.*

Find counter examples automatically!

Quickcheck

Lightweight validation by testing.

- *Motivated by Haskell's QuickCheck*
- *Uses Isabelle's code generator*
- *Fast*
- *Runs in background, proves you wrong as you type.*

Quickcheck

Covers a number of testing approaches:

- *Random and exhausting testing.*
- *Smart test data generators.*
- *Narrowing-based (symbolic) testing.*

Creates test data generators automatically.

Demo: Quickcheck

Test generators for datatypes

Fast iteration in continuation-passing-style

$$\textbf{datatype}\ \alpha\ \text{list} = \text{Nil} \mid \text{Cons}\ \alpha\ (\alpha\ \text{list})$$

Test function:

$$\text{test}_{\alpha\ \textit{list}}\ \text{P}\ =\ \text{P Nil}\ \textit{andalso}\ \text{test}_{\alpha}\ (\lambda \text{x}.\ \text{test}_{\alpha\ \textit{list}}\ (\lambda \text{xs}.\ \text{P}\ (\text{Cons x xs})))$$

Test generators for predicates

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

Problem:
Exhaustive testing creates many useless test cases.

Solution:
Use definitions in precondition for smarter generator.
Only generate cases where distinct xs is true.

$\textit{test-distinct}_{\alpha\ list}\ P \ = \ P\ Nil\ andalso$
$\textit{test}_{\alpha}\ (\lambda x.\ \textit{test-distinct}_{\alpha\ list}\ (\textit{if}\ x \notin xs\ \textit{then}\ (\lambda xs.\ P\ (Cons\ x\ xs))\ \textit{else}\ True))$

Use data flow analysis to figure out which variables must be computed and which generated.

Narrowing

Symbolic execution with demand-driven refinement

- ➜ *Test cases can contain variables*
- ➜ *If execution cannot proceed: instantiate with further symbolic terms*

Pays off if large search spaces can be discarded:

distinct (Cons 1 (Cons 1 x))

False for any x, no further instantiations for x necessary.

Implementation:

Lazy execution with outer refinement loop.
Many re-computations, but fast.

Quickcheck Limitations

Only executable specifications!

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

Nitpick

Nitpick

Finite model finder

- ➜ *Based on SAT via Kodkod (backend of Alloy prover)*
- ➜ *Soundly approximates infinite types*

Nitpick Successes

- *Algebraic methods*
- *C++ memory model*
- *Found soundness bugs in TPS and LEO-II*

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!"*

Demo: Nitpick

We have seen today ...

- ➜ Proof: Sledgehammer
- ➜ Counter examples: Quickcheck
- ➜ Counter examples: Nitpick

Isar

(Part 2)

Datatypes in Isar

Datatype case distinction

proof (cases *term*)
  case Constructor_1
  ⋮
next
⋮
next
  case $(\text{Constructor}_k\ \vec{x})$
  $\cdots\ \vec{x}\ \cdots$
qed

case $(\text{Constructor}_i\ \vec{x})$ $\equiv$
fix $\vec{x}$ **assume** Constructor_i : " $\mathit{term} = \text{Constructor}_i\ \vec{x}$"

Structural induction for nat

```
show P n
proof (induct n)
  case 0              ≡  let ?case = P 0
  ...
  show ?case
next
  case (Suc n)        ≡  fix n assume Suc: P n
  ...                    let ?case = P (Suc n)
  ... n ...
  show ?case
qed
```

Structural induction: $\Longrightarrow$ and $\bigwedge$

show "$\bigwedge x.\ A\ n \Longrightarrow P\ n$"
proof (induct n)
  case 0 $\equiv$ **fix** x **assume** 0: "A 0" **let** ?*case* = "P 0"
  ...
  show ?*case*
next
  case (Suc n) $\equiv$ **fix** n and x **assume** Suc: "$\bigwedge x.\ A\ n \Longrightarrow P\ n$" "$A$ (Suc n)" **let** ?*case* = "P (Suc n)"
  ...
  ... n ...
  ...
  show ?*case*
qed

Demo: Datatypes in Isar

Calculational Reasoning

The Goal

Prove:
$x \cdot x^{-1} = 1$

using:

assoc:	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$
left_inv:	$x^{-1} \cdot x = 1$
left_one:	$1 \cdot x = x$

The Goal

Prove:

$$
\begin{aligned}
x \cdot x^{-1} &= 1 \cdot (x \cdot x^{-1}) \\
\ldots &= 1 \cdot x \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot 1 \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot (1 \cdot x^{-1}) \\
\ldots &= (x^{-1})^{-1} \cdot x^{-1} \\
\ldots &= 1
\end{aligned}
$$

assoc: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

left_inv: $x^{-1} \cdot x = 1$

left_one: $1 \cdot x = x$

Can we do this in Isabelle?

- ➜ Simplifier: too eager
- ➜ Manual: difficult in apply style
- ➜ Isar: with the methods we know, too verbose

Chains of equations

The Problem

$$\begin{array}{rcl} a & = & b \\ \ldots & = & c \\ \ldots & = & d \end{array}$$

shows $a = d$ by transitivity of $=$

Each step usually nontrivial (requires own subproof)

Solution in Isar:

- Keywords **also** and **finally** to delimit steps
- **. . .**: predefined schematic term variable, refers to right hand side of last expression
- Automatic use of transitivity rules to connect steps

also/finally

have "$t_0 = t_1$" [proof]	calculation register
also	"$t_0 = t_1$"
have "$\ldots = t_2$" [proof]	
also	"$t_0 = t_2$"
$\vdots$	$\vdots$
also	"$t_0 = t_{n-1}$"
have "$\cdots = t_n$" [proof]	
finally	$t_0 = t_n$
show P	

— 'finally' pipes fact "$t_0 = t_n$" into the proof

More about also

- Works for all combinations of $=$, $\leq$ and $<$.
- Uses all rules declared as `[trans]`.
- To view all combinations: `print_trans_rules`

Designing [trans] Rules

have $=$ " $l_1 \odot r_1$ " [proof]
also
have " $\ldots \odot r_2$ " [proof]
also

Anatomy of a [trans] rule:

- Usual form: plain transitivity $[\![l_1 \odot r_1; r_1 \odot r_2]\!] \Longrightarrow l_1 \odot r_2$
- More general form: $[\![P\ l_1\ r_1; Q\ r_1\ r_2; A]\!] \Longrightarrow C\ l_1\ r_2$

Examples:

- pure transitivity: $[\![a = b; b = c]\!] \Longrightarrow a = c$
- mixed: $[\![a \leq b; b < c]\!] \Longrightarrow a < c$
- substitution: $[\![P\ a; a = b]\!] \Longrightarrow P\ b$
- antisymmetry: $[\![a < b; b < a]\!] \Longrightarrow False$
- monotonicity:
  $[\![a = f\ b; b < c; \bigwedge x\ y.\ x < y \Longrightarrow f\ x < f\ y]\!] \Longrightarrow a < f\ c$

Demo