COMP4161
Advanced Topics in Software Verification

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

Thomas Sewell, Miki Tanaka, Rob Sison

T3/2024

Content

- ➜ Foundations & Principles
  - Intro, Lambda calculus, natural deduction [1,2]
  - Higher Order Logic, Isar (part 1) [2,3[a]]
  - Term rewriting [3,4]

- ➜ Proof & Specification Techniques
  - Inductively defined sets, rule induction [4,5]
  - Datatype induction, primitive recursion [5,7]
  - General recursive functions, termination proofs [7]
  - Proof automation, Isar (part 2) [8[b]]
  - Hoare logic, proofs about programs, invariants [8,9]
  - C verification [9,10]
  - Practice, questions, exam prep [10[c]]

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

Overview

Part 1: Automatic Proof and Disproof

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

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

Part 2: Structured Proofs

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): CVC3/4/5, 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)*

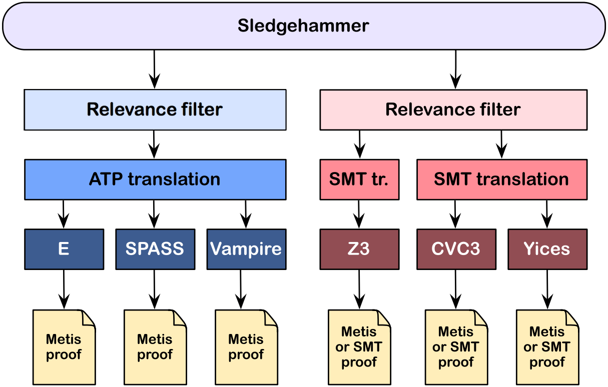
Sledgehammer

Sledgehammer:

- *Connects Isabelle with ATPs and SMT solvers:*
  E, SPASS, Vampire, CVC4, 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

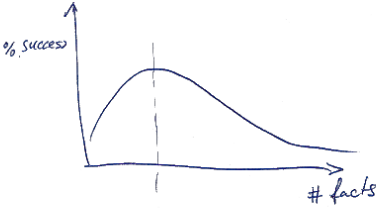
Sledgehammer Architecture



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)

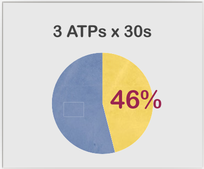
Evaluating Sledgehammer:

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

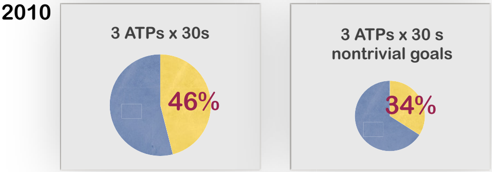
- ➜ ***2010:*** *E, SPASS, Vampire (for 5-120s). 46%*
  $ESV \times 5s \approx V \times 120s$
- ➜ ***2011:*** *Add E-SInE, CVC2, Yices, Z3 (30s).*
  $Z3 > V$
- ➜ ***2012:*** *Better integration with SPASS. 64%*
  SPASS best (small margin)
- ➜ ***2013:*** *Machine learning for fact selection. 69%*
  Improves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

Evaluation

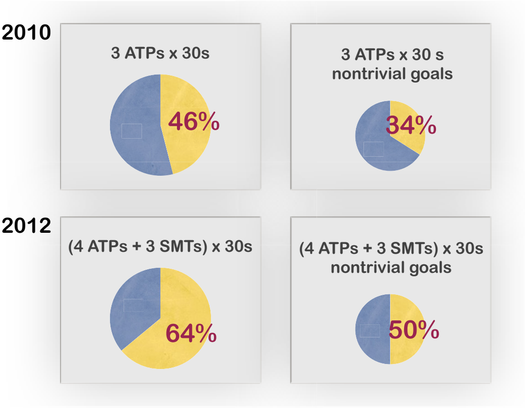
2010



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, . . ., $\approx$ 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.*
  - ➜ ***You have probably seen this already!***

Quickcheck

Covers a number of testing approaches:

- ➜ *Random and exhaustive 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

datatype α list = Nil | Cons α (α list)

Test function:

$\text{test}_{\alpha\ \textit{list}}$ P = P Nil *andalso* test_{α} (λx. $\text{test}_{\alpha\ \textit{list}}$ (λxs. P (Cons x xs)))

Test generators for predicates

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

Problem:
Exhaustive testing creates many useless test cases.

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

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

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

Automation Summary

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

Isar (Part 1)

A Language for Structured Proofs

Motivation

Is this true: $(A \longrightarrow B) = (B \vee \neg A)$?

Motivation

Is this true: $(A \longrightarrow B) = (B \vee \neg A)$?

YES!

```
apply (rule iffI)
 apply (cases A)
  apply (rule disjI1)
  apply (erule impE)
   apply assumption
  apply assumption
 apply (rule disjI2)
 apply assumption
apply (rule impI)
apply (erule disjE)
 apply assumption
apply (erule notE)
apply assumption
done
```

or `by blast`

OK it's true. But WHY? This doesn't look like typical maths proofs.

Isar

apply scripts

- ➜ hard to read
- ➜ hard to maintain

No explicit structure.

What about..

- ➜ Elegance?
- ➜ Explaining deeper insights?

Isar!

A typical Isar proof

proof
  assume $\mathit{formula}_0$
  have $\mathit{formula}_1$ **by** simp
  ⋮
  have $\mathit{formula}_n$ **by** blast
  show $\mathit{formula}_{n+1}$ **by** ...
qed

proves $\mathit{formula}_0 \Longrightarrow \mathit{formula}_{n+1}$

(analogous to **assumes**/**shows** in lemma statements)

Isar core syntax

proof = **proof** [method] statement* **qed**
  | **by** method

method = (simp . . .) | (blast . . .) | (rule . . .) | . . .

statement = **fix** variables $(\bigwedge)$
  | **assume** proposition $(\Longrightarrow)$
  | [**from** name$^+$] (**have** | **show**) proposition proof
  | **next** (separates subgoals)

proposition = [name:] formula

proof and qed

proof [method] statement* **qed**

lemma "$[\![A; B]\!] \Longrightarrow A \wedge B$"
proof (rule conjI)
    assume A: "A"
    from A **show** "A" **by** assumption
next
    assume B: "B"
    from B **show** "B" **by** assumption
qed

➜ **proof** (<method>)	applies method to the stated goal
➜ **proof**	applies a single rule that fits
➜ **proof -**	does nothing to the goal

How do I know what to Assume and Show?

Look at the proof state!

lemma "$\llbracket A; B \rrbracket \Longrightarrow A \wedge B$"
proof (rule conjI)

- **proof** (rule conjI) changes proof state to
  1. $\llbracket A; B \rrbracket \Longrightarrow A$
  2. $\llbracket A; B \rrbracket \Longrightarrow B$
- so we need 2 shows: **show** "A" and **show** "B"
- We are allowed to **assume** A,
  because A is in the assumptions of the proof state.

The Three Modes of Isar

- **[prove]**:
  goal has been stated, proof needs to follow.
- **[state]**:
  proof block has opened or subgoal has been proved,
  new *from* statement, goal statement or assumptions can follow.
- **[chain]**:
  from statement has been made, goal statement needs to follow.

lemma "$[\![A; B]\!] \Longrightarrow A \wedge B$" **[prove]**
proof (rule conjI) **[state]**
  assume A: "A" **[state]**
  from A **[chain]** **show** "A" **[prove]** **by** assumption **[state]**
next **[state]** **. . .**

Have

Can be used to make intermediate steps.

Example:

lemma "$(x :: \text{nat}) + 1 = 1 + x$"
proof -
  have A: "$x + 1 = \text{Suc } x$" **by** simp
  have B: "$1 + x = \text{Suc } x$" **by** simp
  show "$x + 1 = 1 + x$" **by** (simp only: A B)
qed

Demo

Backward and Forward

Backward reasoning: … **have** "$A \wedge B$" **proof**

- **proof** picks an **intro** rule automatically
- conclusion of rule must unify with $A \wedge B$

Forward reasoning: …

assume AB: "$A \wedge B$"
from AB **have** "…" **proof**

- now **proof** picks an **elim** rule automatically
- triggered by **from**
- first assumption of rule must unify with AB

General case: from $A_1 \ldots A_n$ **have** R **proof**

- first n assumptions of rule must unify with $A_1 \ldots A_n$
- conclusion of rule must unify with R

Fix and Obtain

fix $v_1 \ldots v_n$

Introduces new arbitrary but fixed variables
($\sim$ parameters, $\bigwedge$)

obtain $v_1 \ldots v_n$ **where** <prop> <proof>

Introduces new variables together with property

Fancy Abbreviations

this	=	the previous fact proved or assumed
then	=	**from** this
thus	=	**then show**
hence	=	**then have**
with $A_1 \dots A_n$	=	**from** $A_1 \dots A_n$ this
?thesis	=	the last enclosing goal statement

Demo

Moreover and Utimately

have X_1: P_1 . . .
have X_2: P_2 . . .
⋮
have X_n: P_n . . .
from $X_1 \dots X_n$ **show** . . .

wastes brain power
on names $X_1 \dots X_n$

have P_1 . . .
moreover have P_2 . . .
⋮
moreover have P_n . . .
ultimately show . . .

General Case Distinctions

show *formula*
proof -
  have $P_1 \vee P_2 \vee P_3$ <proof>
  moreover { **assume** P_1 ... **have** ?thesis <proof> }
  moreover { **assume** P_2 ... **have** ?thesis <proof> }
  moreover { **assume** P_3 ... **have** ?thesis <proof> }
  ultimately show ?thesis **by** blast
qed

{ ... } is a proof block similar to **proof** ... **qed**

{ **assume** P_1 ... **have** P <proof> }
stands for $P_1 \Longrightarrow P$

Mixing proof styles

from . . .
have . . .
  apply -   make incoming facts assumptions
  apply (. . .)
  ⋮
  apply (. . .)
  done

ISAR

(PART 2)

Datatypes in Isar

Datatype case distinction

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

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

Structural induction for nat

show $P\ n$		
proof (induct n)		
case 0	$\equiv$	**let** $?case = P\ 0$
...		
show $?case$		
next		
case (Suc n)	$\equiv$	**fix** n **assume** Suc: $P\ n$
...		**let** $?case = P\ (\mathrm{Suc}\ n)$
$\cdots\ n\ \cdots$		
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)"
  . . .
  $\cdots\ n\ \cdots$
  . . .
  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]
also
have "$\ldots = t_2$" [proof]
also
$\vdots$
also
have "$\cdots = t_n$" [proof]
finally
show P
— 'finally' pipes fact "$t_0 = t_n$" into the proof

calculation register
"$t_0 = t_1$"

"$t_0 = t_2$"
$\vdots$
"$t_0 = t_{n-1}$"

$t_0 = t_n$

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: $[\![a < b; b < a]\!] \Longrightarrow \mathit{False}$
- monotonicity: $[\![a = f\ b; b < c; \bigwedge x\ y.\ x < y \Longrightarrow f\ x < f\ y]\!] \Longrightarrow a < f\ c$

DEMO