COMP4161: Advanced Topics in Software Verification

based on slides by J. Blanchette, L. Bulwahn and T. Nipkow
Gerwin Klein, Johannes Åman Pohjola, Christine Rizkallah, Miki Tanaka
T3/2020
Content

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

→ Proof & Specification Techniques
  • Inductively defined sets, rule induction, datatype induction, primitive recursion [4,5]
  • General recursive functions, termination proofs [7]
  • Proof automation, Hoare logic, proofs about programs, invariants [8]
  • C verification [9,10]
  • Practice, questions, examp prep [10]

\(^a\) a1 due; \(^b\) a2 due; \(^c\) a3 due
Overview

Automatic Proof and Disproof

→ Sledgehammer: automatic proofs
Overview

Automatic Proof and Disproof

- Sledgehammer: automatic proofs
- Quickcheck: counter example by testing
Overview

Automatic Proof and Disproof

- Sledgehammer: automatic proofs
- Quickcheck: counter example by testing
- Nipick: counter example by SAT
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).
Dramatic improvements in fully automated proofs in the last 2 decades.
Dramatic improvements in fully automated proofs in the last 2 decades.

→ First-order logic (ATP): Otter, Vampire, E, SPASS
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
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
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
Automation in Isabelle

1980s  *rule applications, write ML code*

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

Sledgehammer

Relevance filter

ATP translation

E
SPASS
Vampire

Metis proof
Metis proof
Metis proof

Relevance filter

SMT tr.
SMT translation

Z3
CVC3
Yices

Metis or SMT proof
Metis or SMT proof
Metis or SMT proof
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
  \( (n = 250, 1000, \ldots) \)
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
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
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
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.
Reconstruction

We don’t want to trust the external provers. 
Need to check/reconstruct proof.
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)
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!
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.
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?
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%

$ESV \times 5s \approx V \times 120s$
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).
  \[ Z3 > V \]
Judgement Day (up to 2013)

Evaluating Sledgehammer:

1. 1240 goals out of 7 existing theories.
2. How many can sledgehammer solve?

3. 2010: $E$, SPASS, Vampire (for 5-120s). 46%
   
   $ESV \times 5s \approx V \times 120s$

4. 2011: Add E-SInE, CVC2, Yices, Z3 (30s).
   
   $Z3 > V$

5. 2012: Better integration with SPASS. 64%
   
   SPASS best (small margin)
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

3 ATPs x 30s

46%
Evaluation

2010

- 3 ATPs x 30s
  - 46%

- 3 ATPs x 30 s nontrivial goals
  - 34%
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)

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

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

\[
\frac{919}{1230} = 74\% 
\]
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.
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*
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!
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.
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.
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

\[
\text{datatype } \alpha \text{ list } = \text{Nil} \mid \text{Cons } \alpha (\alpha \text{ list})
\]

Test function:

\[
\text{test}_{\alpha \text{ list}} P = P \text{ Nil } \text{andalso } \text{test}_{\alpha}(\lambda x. \text{test}_{\alpha \text{ list}} (\lambda xs. P (\text{Cons } x xs)))
\]
Test generators for predicates

\[ \text{distinct } xs \implies \text{distinct (remove1 } x \times xs) \]

**Problem:**
*Exhaustive testing creates many useless test cases.*
Test generators for predicates

distinct xs \implies \text{distinct (remove1 } x \text{ 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.
Test generators for predicates

\[ \text{distinct } xs \implies \text{distinct } (\text{remove1 } x \times xs) \]

Problem:
Exhaustive testing creates many useless test cases.

Solution:
Use definitions in precondition for smarter generator.
Only generate cases where \( \text{distinct } xs \) is true.

\[
\text{test-distinct}_\alpha \text{ list } P = P \text{ Nil andalso test}_\alpha (\lambda x. \text{test-distinct}_\alpha \text{ list } (\text{if } x \notin xs \text{ then } (\lambda xs. P (\text{Cons } x xs)) \text{ else True}))
\]
Test generators for predicates

\[ \text{distinct } \text{xs} \implies \text{distinct } (\text{remove1 } x \times \text{xs}) \]

**Problem:**
Exhaustive testing creates many useless test cases.

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

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

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
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:

\[ \text{distinct (Cons 1 (Cons 1 x))} \]

False for any \(x\), no further instantiations for \(x\) necessary.
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:

\[\text{distinct (Cons 1 (Cons 1 x))}\]

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
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
We have seen today ...

- Proof: Sledgehammer
- Counter examples: Quickcheck
We have seen today ...

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

(Part 2)
Datatypes in Isar
Datatype case distinction

\begin{proof}
  (cases term)
  \begin{align*}
    \text{case } \text{Constructor}_1 \\
    \vdots \\
    \text{next} \\
    \vdots \\
    \text{next} \\
    \begin{align*}
      \text{case } (\text{Constructor}_k \, \vec{x}) \\
      \ldots \vec{x} \ldots
    \end{align*}
  \end{align*}
\end{proof}

qed
Datatype case distinction

proof (cases term)
  case Constructor\(_1\)
  :
  next
  :
  next
  case (Constructor\(_k\) \vec{x})
  \ldots \vec{x} \ldots
qed

\begin{align*}
\text{case } (\text{Constructor}_i \vec{x}) & \equiv \\
\text{fix } \vec{x} & \text{ assume Constructor}_i : " \text{term} = \text{Constructor}_i \vec{x}" 
\end{align*}
Structural induction for nat

\[\text{show } P \ n\]
\[\text{proof } (\text{induct } n)\]
\[\text{case } 0 \quad \equiv \quad \text{let } ?\text{case} = P \ 0\]
\[\ldots\]
\[\text{show } ?\text{case}\]
\[\text{next}\]
\[\text{case } (\text{Suc } n) \quad \equiv \quad \text{fix } n \ \text{assume } \text{Suc}: P \ n\]
\[\text{let } ?\text{case} = P \ (\text{Suc } n)\]
\[\ldots \ n \ \ldots\]
\[\text{show } ?\text{case}\]
\text{qed}
Structural induction: $\Rightarrow$ and $\land$

```isar
show "$\land x. A n \Rightarrow P n$"
proof (induct n)
  case 0
    
  show ?case
next
  case (Suc n)
    
  assume Suc: "$\land x. A n \Rightarrow P n"
    
  let ?case = "P (Suc n)"
qed
```

\[\equiv\]

```isar
fix x assume 0: "A 0"
let ?case = "P 0"

fix n and x
assume Suc: "$\land x. A n \Rightarrow P n"
    
  "A (Suc n)"
let ?case = "P (Suc n)"
```

\[\equiv\]
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:
\[ x \cdot x^{-1} = 1 \cdot (x \cdot x^{-1}) \]
\[ \ldots = 1 \cdot x \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot 1 \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot (1 \cdot x^{-1}) \]
\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \]
\[ \ldots = 1 \]

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:

\[ x \cdot x^{-1} = 1 \cdot (x \cdot x^{-1}) \]
\[ \ldots = 1 \cdot x \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot 1 \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot (1 \cdot x^{-1}) \]
\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \]
\[ \ldots = 1 \]

Can we do this in Isabelle?

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:
\[ x \cdot x^{-1} = 1 \cdot (x \cdot x^{-1}) \]
\[ \ldots = 1 \cdot x \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot 1 \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot (1 \cdot x^{-1}) \]
\[ \ldots = 1 \]

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
The Goal

Prove:

\[ x \cdot x^{-1} = 1 \cdot (x \cdot x^{-1}) \]
\[ \ldots = 1 \cdot x \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot 1 \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot (1 \cdot x^{-1}) \]
\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \]
\[ \ldots = 1 \]

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
The Goal

Prove:
\[ x \cdot x^{-1} = 1 \cdot (x \cdot x^{-1}) \]
\[ \ldots = 1 \cdot x \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot 1 \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot (1 \cdot x^{-1}) \]
\[ \ldots = 1 \]

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

\[ a = b \]
\[ \ldots = c \]
\[ \ldots = d \]

shows \( a = d \) by transitivity of \( = \)
Chains of equations

The Problem

\[ a = b \]
\[ \ldots = c \]
\[ \ldots = d \]

shows \( a = d \) by transitivity of \( = \)

Each step usually nontrivial (requires own subproof)
Chains of equations

The Problem

\begin{align*}
    a & = b \\
    \ldots & = c \\
    \ldots & = d
\end{align*}

shows \( a = d \) by transitivity of \( = \)

Each step usually nontrivial (requires own subproof)

**Solution in Isar:**

- Keywords **also** and **finally** to delimit steps
Chains of equations

The Problem

\[ a = b \]
\[ \ldots = c \]
\[ \ldots = d \]

shows \( a = d \) by transitivity of =

Each step usually nontrivial (requires own subproof)

Solution in Isar:

\[ \rightarrow \] Keywords also and finally to delimit steps

\[ \rightarrow \ldots : \] predefined schematic term variable,
refers to right hand side of last expression
Chains of equations

The Problem

\[\begin{align*}
a &= b \\
\ldots &= c \\
\ldots &= d
\end{align*}\]

shows \(a = d\) by transitivity of =

Each step usually nontrivial (requires own subproof)

Solution in Isar:

- Keywords also and finally to delimit steps
- \ldots: 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
also/finally

\[ t_0 = t_1 \]

also have \[ t_0 = t_1 \] [proof]

calculation register \[ t_0 = t_1 \]

finally
also/finally

have "\( t_0 = t_1 \)" [proof]
also
have "\( \ldots = t_2 \)" [proof]
calculation register
"\( t_0 = t_1 \)"
also/finally

have "t₀ = t₁" [proof]
also
have ". . . = t₂" [proof]
also

calculation register
"t₀ = t₁"
"t₀ = t₂"
also/finally

have "\( t_0 = t_1 \)" [proof]
also have "\( \ldots = t_2 \)" [proof]
also
::
also

"\( t_0 = t_n \)" [proof]
calculation register
"\( t_0 = t_1 \)"
"\( t_0 = t_2 \)"
::
"\( t_0 = t_{n-1} \)"
also/finally

have "\( t_0 = t_1 \)" [proof]
also
have "\( \ldots = t_2 \)" [proof]
also
:
also
have "\( \ldots = t_n \)" [proof]

... [proof]

... [proof]

... [proof]

... [proof]

... [proof]

"\( t_0 = t_1 \)"

"\( t_0 = t_2 \)"

"\( t_0 = t_{n-1} \)"

calculation register

finally

"\( t_0 = t_n \)"

'finally' pipes fact "\( t_0 = t_n \)" into the proof
also/finally

have "\( t_0 = t_1 \)" [proof]
also
have "\( \ldots = t_2 \)" [proof]
also
: ;
also
have "\( \ldots = t_n \)" [proof]
finally
calculation register
"\( t_0 = t_1 \)"
"\( t_0 = t_2 \)"
: ;
"\( t_0 = t_{n-1} \)"
\( t_0 = t_n \)
also/finally

have "t₀ = t₁" [proof]
also have "... = t₂" [proof]
also have "... = tₙ" [proof]
finally show P
— 'finally' pipes fact "t₀ = tₙ" into the proof

calculation register
"t₀ = t₁"

"t₀ = t₂"

... [proof]
"t₀ = tₙ−₁"

Finally, t₀ = tₙ
More about also

→ Works for all combinations of $=$, $\leq$ and $<$. 
More about also

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

\[ \text{have} = "l_1 \odot r_1" \quad \text{[proof]} \]
also
\[ \text{have} \; "\ldots \odot r_2" \quad \text{[proof]} \]
also
Designing [trans] Rules

have = "l₁ ⊙ r₁" [proof]
also
have "... ⊙ r₂" [proof]
also

Anatomy of a [trans] rule:
→ Usual form: plain transitivity [[l₁ ⊙ r₁; r₁ ⊙ r₂] → l₁ ⊙ r₂]
Designing [trans] Rules

have = "l₁ ⊙ r₁" [proof]
also
have "... ⊙ r₂" [proof]
also

Anatomy of a [trans] rule:

→ Usual form: plain transitivity \[
[l₁ ⊙ r₁; r₁ ⊙ r₂] \rightarrow l₁ ⊙ r₂
\]

→ More general form: \[
[P l₁ r₁; Q r₁ r₂; A] \rightarrow C l₁ r₂
\]

Examples:
Designing [trans] Rules

\[ \text{have} = "l_1 \circ r_1" \quad \text{[proof]} \]
also
\[ \text{have} "\ldots \circ r_2" \quad \text{[proof]} \]
also

Anatomy of a [trans] rule:

\[ \rightarrow \text{Usual form: plain transitivity } [l_1 \circ r_1; r_1 \circ r_2] \implies l_1 \circ r_2 \]
\[ \rightarrow \text{More general form: } [P \ l_1 \ r_1; Q \ r_1 \ r_2; A] \implies C \ l_1 \ r_2 \]

Examples:

\[ \rightarrow \text{pure transitivity: } [a = b; b = c] \implies a = c \]
Designing [trans] Rules

\[ \text{have} = "l_1 \odot r_1" \quad [\text{proof}] \]
\[ \text{also} \]
\[ \text{have} "\ldots \odot r_2" \quad [\text{proof}] \]
\[ \text{also} \]

Anatomy of a [trans] rule:

- Usual form: plain transitivity \( [l_1 \odot r_1; r_1 \odot r_2] \Rightarrow l_1 \odot r_2 \)
- More general form: \( [P l_1 r_1; Q r_1 r_2; A] \Rightarrow C l_1 r_2 \)

Examples:

- pure transitivity: \( [a = b; b = c] \Rightarrow a = c \)
- mixed: \( [a \leq b; b < c] \Rightarrow a < c \)
Designing [trans] Rules

\[
\text{have} = "l_1 \odot r_1" \quad \text{[proof]}
\]
also
\[
\text{have } "\ldots \odot r_2" \quad \text{[proof]}
\]
also

Anatomy of a [trans] rule:

→ Usual form: plain transitivity \([l_1 \odot r_1; r_1 \odot r_2] \implies l_1 \odot r_2\)
→ More general form: \([P \ l_1 \ r_1; Q \ r_1 \ r_2; A] \implies C \ l_1 \ r_2\)

Examples:

→ pure transitivity: \([a = b; b = c] \implies a = c\)
→ mixed: \([a \leq b; b < c] \implies a < c\)
→ substitution: \([P \ a; a = b] \implies P \ b\)
Designing [trans] Rules

\[
\text{have} = "l_1 \odot r_1" \quad \text{[proof]}
\]
also

\[
\text{have} = "\ldots \odot r_2" \quad \text{[proof]}
\]
also

Anatomy of a [trans] rule:

- Usual form: plain transitivity \[\lbrack l_1 \odot r_1; r_1 \odot r_2 \rbrack \implies l_1 \odot r_2\]
- More general form: \[\lbrack P \ l_1 \ r_1; Q \ r_1 \ r_2; A \rbrack \implies C \ l_1 \ r_2\]

Examples:

- pure transitivity: \[\lbrack a = b; b = c \rbrack \implies a = c\]
- mixed: \[\lbrack a \leq b; b < c \rbrack \implies a < c\]
- substitution: \[\lbrack P \ a; a = b \rbrack \implies P \ b\]
- antisymmetry: \[\lbrack a < b; b < a \rbrack \implies False\]
Designing [trans] Rules

\[ \text{have} = "l_1 \odot r_1" \text{ [proof]} \]

Also
\[ \text{also have } "\ldots \odot r_2" \text{ [proof]} \]

Anatomy of a [trans] rule:

→ Usual form: plain transitivity \[ [l_1 \odot r_1; r_1 \odot r_2] \Rightarrow l_1 \odot r_2 \]

→ More general form: \[ [P l_1 r_1; Q r_1 r_2; A] \Rightarrow C l_1 r_2 \]

Examples:

→ pure transitivity: \[ [a = b; b = c] \Rightarrow a = c \]

→ mixed: \[ [a \leq b; b < c] \Rightarrow a < c \]

→ substitution: \[ [P a; a = b] \Rightarrow P b \]

→ antisymmetry: \[ [a < b; b < a] \Rightarrow \text{False} \]

→ monotonicity: \[ [a = f b; b < c; \wedge x y. x < y \Rightarrow f x < f y] \Rightarrow a < f c \]
Demo