UNSW SYDNEY

COMP4161
Advanced Topics in Software Verification

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

Gerwin Klein, June Andronick, Miki Tanaka, Johannes Åman Pohjola

T3/2022

Content

➜ Foundations & Principles

- Intro, Lambda [illegible] atural deduction [1,2]
- Higher Order [illegible] 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[b]]
- Proof automation, Isar (part 2) [8]
- 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

Au roof and Disproof

➜ Sledgehammer: automatic proofs

Overview

Au roof and Disproof

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

Overview

Au roof and Disproof

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

Overview

Au roof 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 improvemen  automated proofs in the last 2 decades.

Automation

Dramatic improvemen 

automated proofs in the last 2 decades.

➜ First-order logic (ATP): Otter, Vampire, E, SPASS

Automation

Dramatic improvement in automated proofs in the last 2 decades.

- First-order logic (ATP): Otter, Vampire, E, SPASS
- Propositional logic (SAT): MiniSAT, Chaff, RSat

Automation

Dramatic improvemen 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

Automation

Dramatic improvement in 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[illegible]ns, write ML code

Automation in Isabelle

1980s *rule[illegible]ns, write ML code*

1990s *sim[illegible]omatic provers (blast, auto), arithmetic*

Automation in Isabelle

1980s *rule[illegible]ns, write ML code*

1990s *sim[illegible]omatic provers (blast, auto), arithmetic*

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

Sledgehammer

Sledgehammer:

- ➜ *Connects Isab[illegible]TPs and SMT solvers: E, SPASS, Va[illegible]3, Yices, Z3*

Sledgehammer

Sledgehammer:

- ➜ *Connects Isabelle to 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*

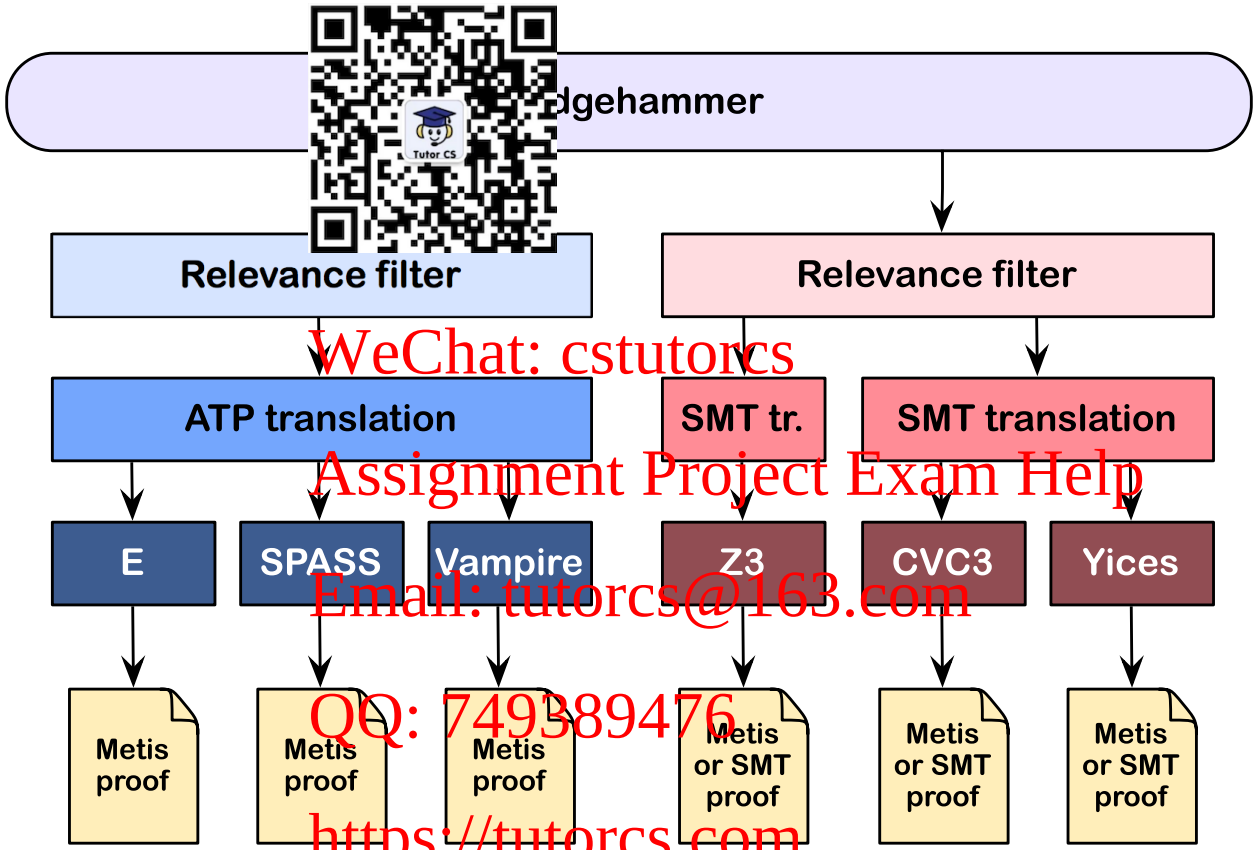
Sledgehammer

Sledgehammer:

- ➜ *Connects Isabelle to 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

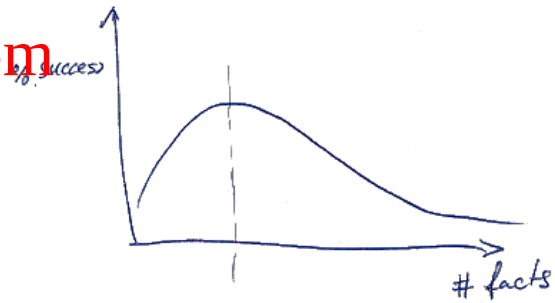
Sledgehammer Architecture



Fact Selection

Provers perform poo[illegible] 1000s of facts.

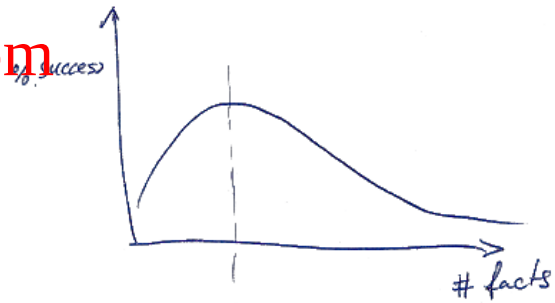
- ➜ *Best number d[illegible]ends on the prover*
- ➜ *Need to take [illegible]acts we give them*
- ➜ *Idea: order f[illegible]evance, give top n to prover* ($n = 250, 1000, \ldots$)



Fact Selection

Provers perform poo[illegible] 1000s of facts.

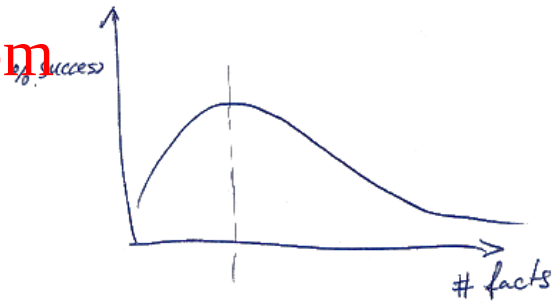
- ➜ *Best number [illegible]ends on the prover*
- ➜ *Need to take [illegible]acts we give them*
- ➜ *Idea: order f[illegible]evance, give top n to prover* ($n = 250, 1000, \ldots$)
- ➜ *Meng & Paulson method: lightweight, symbol-based filter*



Fact Selection

Provers perform poo[illegible] 1000s of facts.

- ➜ *Best number [illegible]ends on the prover*
- ➜ *Need to take [illegible]acts we give them*
- ➜ *Idea: order f[illegible]evance, 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-or*  *orphism, type classes*
Target: *first-orde**or simply-typed*

From HOL to FOL

Source: *higher-or orphism, type classes*
Target: *first-orde or simply-typed*

- ➜ **First-order:**
  - ➜ *SK combinators, λ-lifting*
  - ➜ *Explicit function application operator*

From HOL to FOL

Source: *higher-or* *orphism, type classes*
Target: *first-orde* *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 t  e external provers.

Reconstruction

We don't want t e external provers.
Need to check/rec roof.

Reconstruction

We don't want t[illegible]e external provers.
Need to check/re[illegible]roof.

➜ *Re-find using Metis*
Usually fast and reliable (sometimes too slow)

Reconstruction

We don't want t e external provers.
Need to check/re roof.

➜ *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 t[illegible]e external provers.
Need to check/rec[illegible]roof.

- *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 t[illegible]e external provers.
Need to check/rec[illegible]roof.

- *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 Sledgehammering

- ➜ *1240 goals ou[illegible]ng theories.*
- ➜ *How many ca[illegible]mer solve?*

Judgement Day (up to 2013)

Evaluating Sledgehammer

- ➜ *1240 goals ou[illegible]ng theories.*
- ➜ *How many ca[illegible]mer solve?*
- ➜ **2010:** *E, SPASS, Vampire (for 5-120s). 46%*
  ESV $\times$ *5s* $\approx$ *VV* $\times$ *120s*

Judgement Day (up to 2013)

Evaluating Sledgehammering

- ➜ *1240 goals ou[illegible]ng theories.*
- ➜ *How many ca[illegible]mer 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 Sledgehammering

- ➜ *1240 goals ou[illegible]ng theories.*
- ➜ *How many ca[illegible]mer 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)

Judgement Day (up to 2013)

Evaluating Sledgehammering

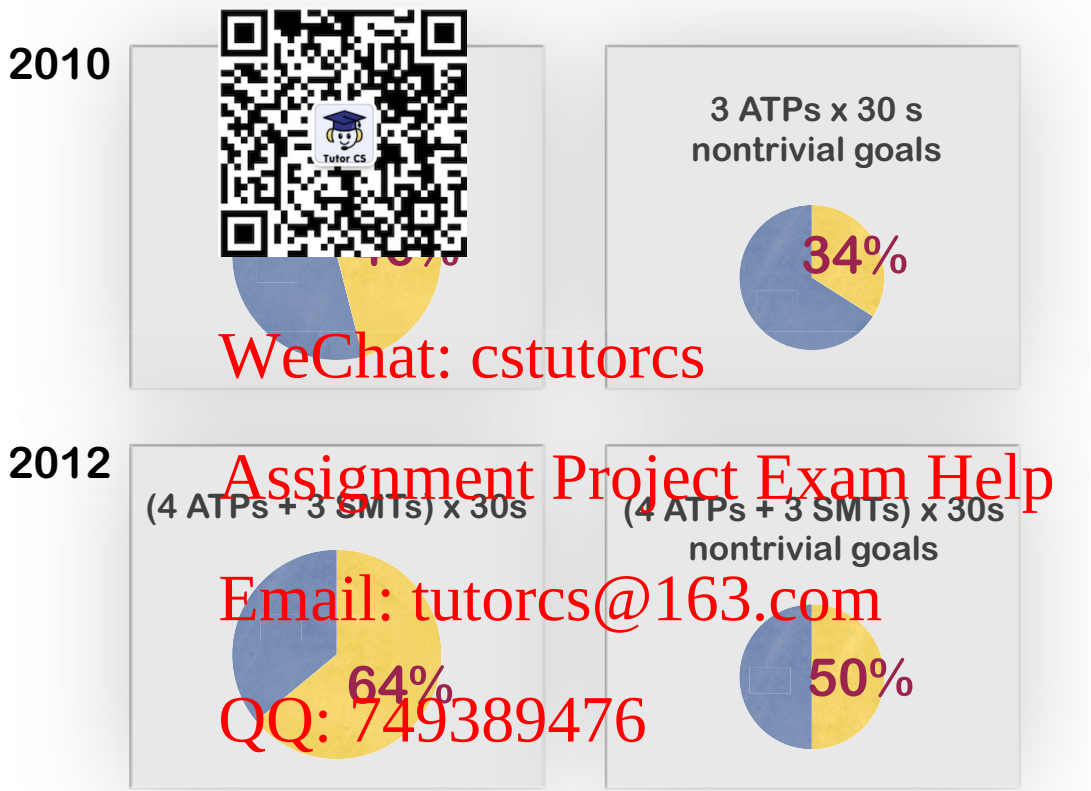
- ➜ *1240 goals ou[illegible]ng theories.*
- ➜ *How many ca[illegible]mer solve?*

- ➜ **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

Evaluation

Evaluation



Judgement Day (2016)

Pr	MePo	MaSh	MeSh	Any selector
CV	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 applica

- ➜ *Large Isabelle/ itory 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 applica

- ➜ *Large Isabelle/ itory 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

Theorem proving and testing

Testing can show [illegible] presence of errors,
but not their abs[illegible] *[illegible]jkstra)*

Testing cannot pr[illegible]ms

Theorem proving and testing

Testing can show ... presence of errors,
but not their abs... *...jkstra)*

Testing cannot pr... ...ms, but it can refute conjectures!

Theorem proving and testing

Testing can show [illegible] presence of errors,
but not their abs[illegible] *[illegible]jkstra)*

Testing cannot pr[illegible]ms, *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 [illegible] presence of errors,
but not their abs[illegible][illegible]*jkstra)*

Testing cannot pr[illegible]ms, 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

Ligh alidation 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 numbe g approaches:

- *Random and e esting.*
- *Smart test dat s.*
- *Narrowing-based (symbolic) testing.*

Creates test data generators automatically.

Demo: Quickcheck

Test generators for datatypes

Fast iter  **ontinuation-passing-style**

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

Test function:

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

Test generators for predicates

distinct (remove1 x xs)

Problem:

Exhaustive testing ... many useless test cases.

Test generators for predicates

distinct (remove1 x xs)

Problem:

Exhaustive testing ... many useless test cases.

Solution:

Use definitions in precondition for smarter generator.

Only generate cases where distinct xs is true.

Test generators for predicates

distinct (remove1 x xs)

Problem:

Exhaustive testing ... many useless test cases.

Solution:

Use definitions in precondition for smarter generator.

Only generate cases where distinct xs is true.

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

Test generators for predicates

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.

test-distinct$_{\alpha\ list}$ *P* = *P Nil andalso*
test$_{\alpha}$ (λ*x*. *test-distinct*$_{\alpha\ list}$ (*if x* ∉ *xs then* (λ*xs*. *P* (*Cons x xs*)) *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 [illegible] riables*
- ➜ *If execution c[illegible]ed: instantiate with further symbolic terms*

Narrowing

Symbolic execution with demand-driven refinement

- ➜ *Test cases can [illegible] riables*
- ➜ *If execution c[illegible]ed: 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.

Narrowing

Symbolic execution with demand-driven refinement

- ➜ *Test cases can ... riables*
- ➜ *If execution c... ed: 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

Or able specifications!

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

Nitpick

Nitpick

model finder

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

Nitpick Successes

- *Algebraic meth*
- *C++ memory*
- *Found soundn* *TPS and LEO-II*

Nitpick Successes

- ➜ *Algebraic meti*
- ➜ *C++ memory*
- ➜ *Found soundn* *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: Sledgehamm

We have seen today ...

- Proof: Sledgehamm
- Counter examples:

We have seen today ...

- ➜ Proof: Sledgehamm
- ➜ Counter examples:
- ➜ Counter examples:

Isar

(Part 2)

Datatypes in Isar

Datatype case distinction

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

Datatype case distinction

proof (cases *term*)
  case Constructor$_1$
  ⋮
next
⋮
next
  case (Constructor$_k$ $\vec{x}$)
  ⋯ $\vec{x}$ ⋯
qed

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

Structural induction for nat

show $P\ n$
proof (induct n)
  case 0
  ...
  show ?*case*
next
  case (Suc n)
  ...
  ... n ...
  show ?*case*
qed

$\equiv$ **fix** n **assume** Suc: $P\ n$
  let ?*case* $= P$ (Suc n)

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

show "$\bigwedge x.\ A\ n \Longrightarrow P\ n$"
proof (induct n)
  case 0
  ...
  show ?*case*
next
  case (Suc n)
  ...
  ... n ...
  ...
  show ?*case*
qed

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

$\equiv$ **fix** n and x
assume Suc: "$\bigwedge x.\ A\ n \Longrightarrow P\ n$"
"A (Suc n)"
let ?*case* = "P (Suc n)"

Demo: Datatypes in Isar

Calculational Reasoning

The Goal

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

g:

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:

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

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

The Goal

Prove:

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

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?

The Goal

Prove:

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

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

The Goal

Prove:

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

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

The Goal

Prove:

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

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

$$\begin{aligned} &= b \\ &= c \\ &= d \end{aligned}$$

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

Chains of equations

The Problem

$$\begin{aligned} &= b \\ &= c \\ &= d \end{aligned}$$

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

Each step usually nontrivial (requires own subproof)

Chains of equations

The Problem

$$\begin{aligned} &= b \\ &= c \\ &= d \end{aligned}$$

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

$$\begin{aligned} &= b \\ &= c \\ &= d \end{aligned}$$

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

Chains of equations

The Problem

$$\begin{aligned} &= b \\ &= c \\ &= d \end{aligned}$$

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

also/finally

have "$t_0 = t_1$" [proof]
also

calculation register
"$t_0 = t_1$"

also/finally

have "$t_0 = t_1$" [proof]
also
have "$\ldots = t_2$" [proo

calculation register
"$t_0 = t_1$"

also/finally

have "$t_0 = t_1$" [proof]
also
have "$\ldots = t_2$" [proo
also

calculation register
"$t_0 = t_1$"

"$t_0 = t_2$"

also/finally

have "$t_0 = t_1$" [proof]
also
have "$\ldots = t_2$" [proo
also
$\vdots$
also

calculation register
"$t_0 = t_1$"
"$t_0 = t_2$"
$\vdots$
"$t_0 = t_{n-1}$"

also/finally

have "$t_0 = t_1$" [proof]
also
have "$\ldots = t_2$" [proo
also
$\vdots$
also
have "$\cdots = t_n$" [proof]

calculation register
"$t_0 = t_1$"
"$t_0 = t_2$"
$\vdots$
"$t_0 = t_{n-1}$"

also/finally

have "$t_0 = t_1$" [proof]
also
have "$\ldots = t_2$" [proo
also
$\vdots$
also
have "$\cdots = t_n$" [proof]
finally

calculation register
"$t_0 = t_1$"
"$t_0 = t_2$"
$\vdots$
"$t_0 = t_{n-1}$"
$t_0 = t_n$

also/finally

have "$t_0 = t_1$" [proof]
also
have "$\ldots = t_2$" [proo
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 combi  , $\leq$ and $<$.

More about also

- Works for all combi=, $\leq$ and $<$.
- Uses all rules declar...s].

More about also

- Works for all combi=, $\leq$ and $<$.
- Uses all rules declar`ns]`.
- To view all combina`t_trans_rules`

Designing [trans] Rules

have $=$ "$l_1 \odot r_1$" [proof]

. $\odot r_2$" [proof]

Designing [trans] Rules

have $= "l_1 \odot r_1"$ [proof]

$. \odot r_2"$ [proof]

Anatomy of a [trans]

➜ Usual form: plain transitivity $[\![l_1 \odot r_1; r_1 \odot r_2]\!] \Longrightarrow l_1 \odot r_2$

Designing [trans] Rules

have $= "l_1 \odot r_1"$ [proof]

$. \odot r_2"$ [proof]

Anatomy of a [trans]

- ➜ 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:

Designing [trans] Rules

have $= "l_1 \odot r_1"$ [proof]

$. \odot r_2"$ [proof]

Anatomy of a [trans]

- ➜ 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: $[\![a = b; b = c]\!] \Longrightarrow a = c$

Designing [trans] Rules

have $= "l_1 \odot r_1"$ [proof]

$. \odot r_2"$ [proof]

Anatomy of a [trans]

- ➜ 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: $[\![a \leq b; b < c]\!] \Longrightarrow a < c$

Designing [trans] Rules

have $= "l_1 \odot r_1"$ [proof]

$. \odot r_2"$ [proof]

Anatomy of a [trans]

- ➜ 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: $[\![P\ a; a = b]\!] \Longrightarrow P\ b$

Designing [trans] Rules

have $= "l_1 \odot r_1"$ [proof]

$. \odot r_2"$ [proof]

Anatomy of a [trans]

- 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: $[\![a < b; b < a]\!] \Longrightarrow False$

Designing [trans] Rules

have $= "l_1 \odot r_1"$ [proof]

$. \odot r_2"$ [proof]

Anatomy of a [trans]

- 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