



DATA
61

COMP4161: Advanced Topics in Software Verification

Isar

Gerwin Klein, June Andronick, Ramana Kumar, Miki Tanaka
S2/2017

data61.csiro.au



Content



- Intro & motivation, getting started [1]

- Foundations & Principles
 - Lambda Calculus, natural deduction [1,2]
 - Higher Order Logic [3^a]
 - Term rewriting [4]

- Proof & Specification Techniques
 - Inductively defined sets, rule induction [5]
 - Datatypes, recursion, induction [6, 7]
 - Hoare logic, proofs about programs, C verification [8^b,9]
 - (mid-semester break)
 - Writing Automated Proof Methods [10]
 - Isar, codegen, typeclasses, locales [11^c,12]

^aa1 due; ^ba2 due; ^ca3 due

Isar

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?

Motivation



WHY is this true: $(A \rightarrow B) = (B \vee \neg A)$?

Demo

apply scripts

- unreadable
- hard to maintain
- do not scale

No structure.

What about..

- Elegance?
- Explaining deeper insights?
- Large developments?

Isar!

A typical Isar proof



```
proof
  assume formula0
  have formula1   by simp
  ⋮
  have formulan   by blast
  show formulan+1 by ...
qed
```

proves $\textit{formula}_0 \implies \textit{formula}_{n+1}$

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

Isar core syntax



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

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

statement = **fix** variables (\wedge)
| **assume** proposition (\implies)
| [**from** name⁺] (**have** | **show**) proposition proof
| **next** (separates subgoals)

proposition = [name:] formula

proof and qed



proof [method] statement* **qed**

lemma " $\llbracket A; B \rrbracket \implies A \wedge B$ "

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 " $\llbracket A; B \rrbracket \implies A \wedge B$ "

proof (rule conjI)

- **proof** (rule conjI) changes proof state to
 1. $\llbracket A; B \rrbracket \implies A$
 2. $\llbracket A; B \rrbracket \implies B$
- 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] \implies A \wedge B" **[prove]**

proof (rule conjI) **[state]**

 assume A: "A" **[state]**

 from A **[chain]** show "A" **[prove]** by assumption **[state]**

next **[state]** ...

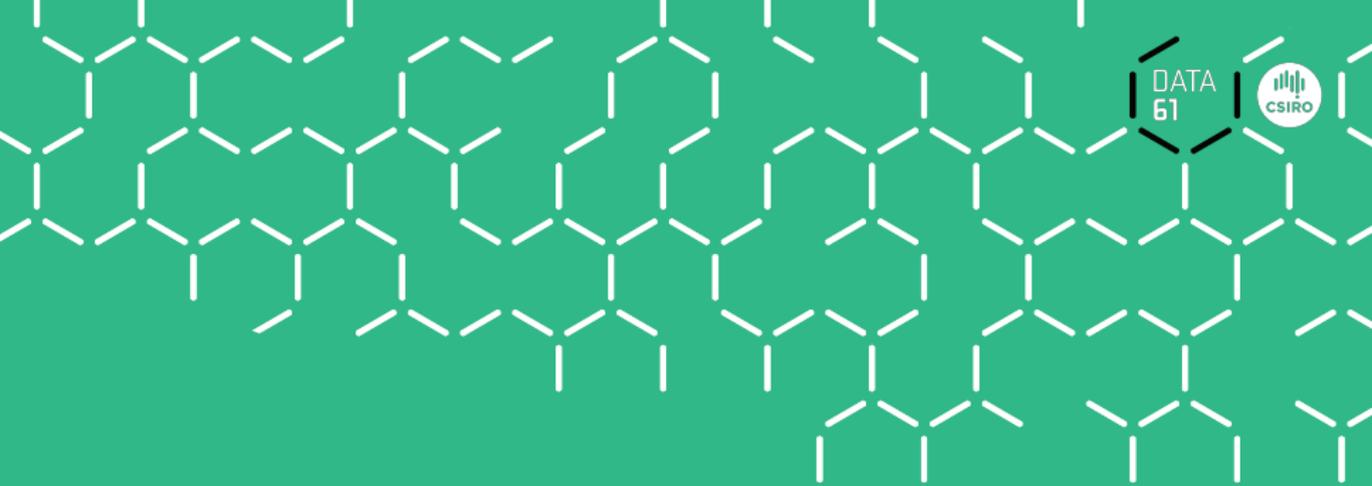
Have



Can be used to make intermediate steps.

Example:

```
lemma "(x :: nat) + 1 = 1 + x"  
proof -  
  have A: "x + 1 = Suc x" by simp  
  have B: "1 + x = Suc x" by simp  
  show "x + 1 = 1 + x" by (simp only: A B)  
qed
```



DATA
61



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 $A_1 \dots A_n$ have R proof

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

Fix and Obtain

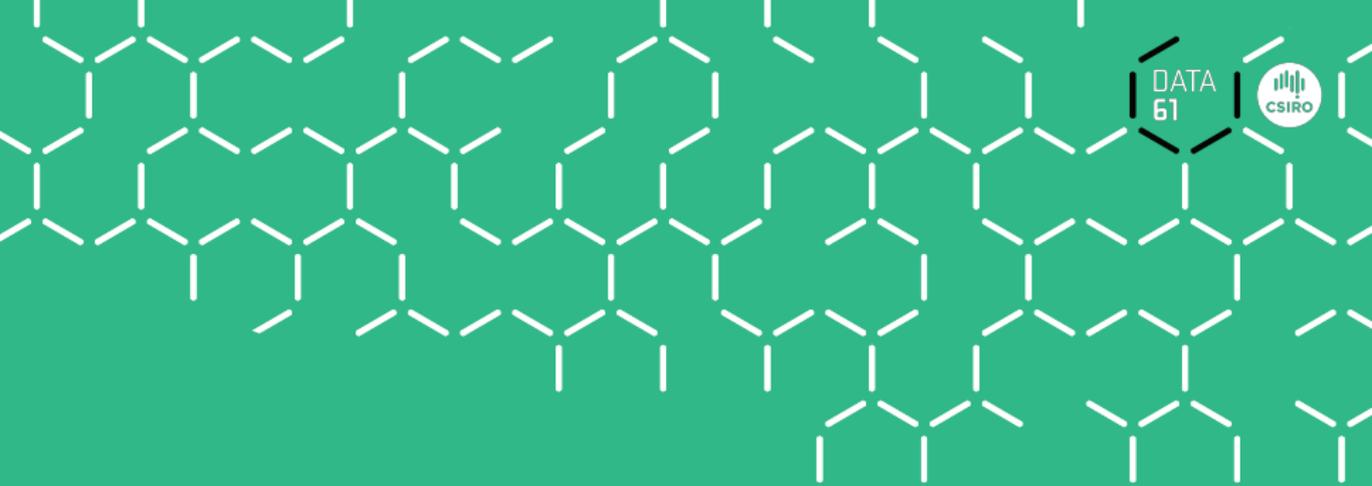


fix $v_1 \dots v_n$

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

obtain $v_1 \dots v_n$ **where** $\langle \text{prop} \rangle \langle \text{proof} \rangle$

Introduces new variables together with property



DATA
61



Demo

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

Moreover and Ultimately



have $X_1: P_1 \dots$
have $X_2: P_2 \dots$
 \vdots
have $X_n: P_n \dots$
from $X_1 \dots X_n$ show \dots

have $P_1 \dots$
moreover have $P_2 \dots$
 \vdots
moreover have $P_n \dots$
ultimately show \dots

wastes lots of brain power
on names $X_1 \dots X_n$

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 $P_1 \implies P$

Mixing proof styles



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

Datatypes in Isar

Datatype case distinction



```
proof (cases term)  
  case Constructor1  
  ⋮  
next  
⋮  
next  
  case (Constructork  $\vec{x}$ )  
    ...  $\vec{x}$  ...  
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            $\equiv$  let ?case =  $P\ 0$ 
  ...
  show ?case
next
  case (Suc  $n$ )     $\equiv$  fix  $n$  assume Suc:  $P\ n$ 
  ...
  ...  $n$  ...
  show ?case
qed
```

Structural induction: \implies and \wedge



show " $\wedge x. A\ n \implies P\ n$ "

proof (induct n)

case 0

 ...

show ?case

next

case (Suc n)

 ...

 ... n ...

 ...

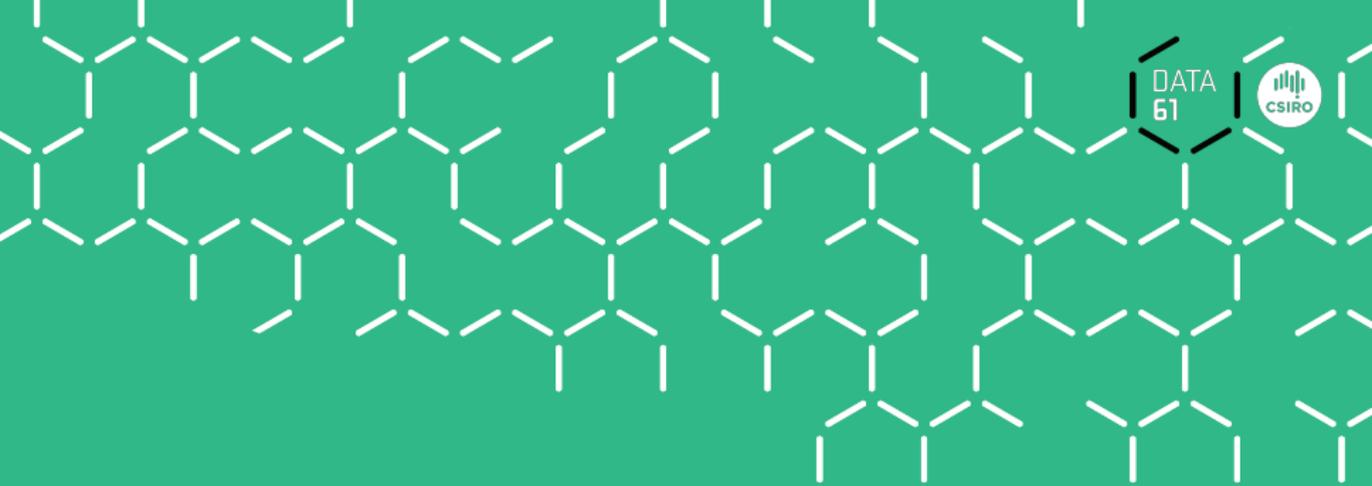
show ?case

qed

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

\equiv **fix** n and x
assume Suc: " $\wedge x. A\ n \implies P\ n$ "
 " $A\ (\text{Suc } n)$ "
let ?case = " $P\ (\text{Suc } n)$ "

Demo: Datatypes in Isar



DATA
61



Computational 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:

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

$$\begin{aligned}\text{assoc:} & (x \cdot y) \cdot z = x \cdot (y \cdot z) \\ \text{left_inv:} & x^{-1} \cdot x = 1 \\ \text{left_one:} & 1 \cdot x = x\end{aligned}$$

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

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

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

Each step usually nontrivial (requires own subproof)

Solution in Isar:

- Keywords **also** and **finally** to delimit steps
- \dots : 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 " $\dots = t_2$ " [proof]

also

\vdots

also

have " $\dots = t_n$ " [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



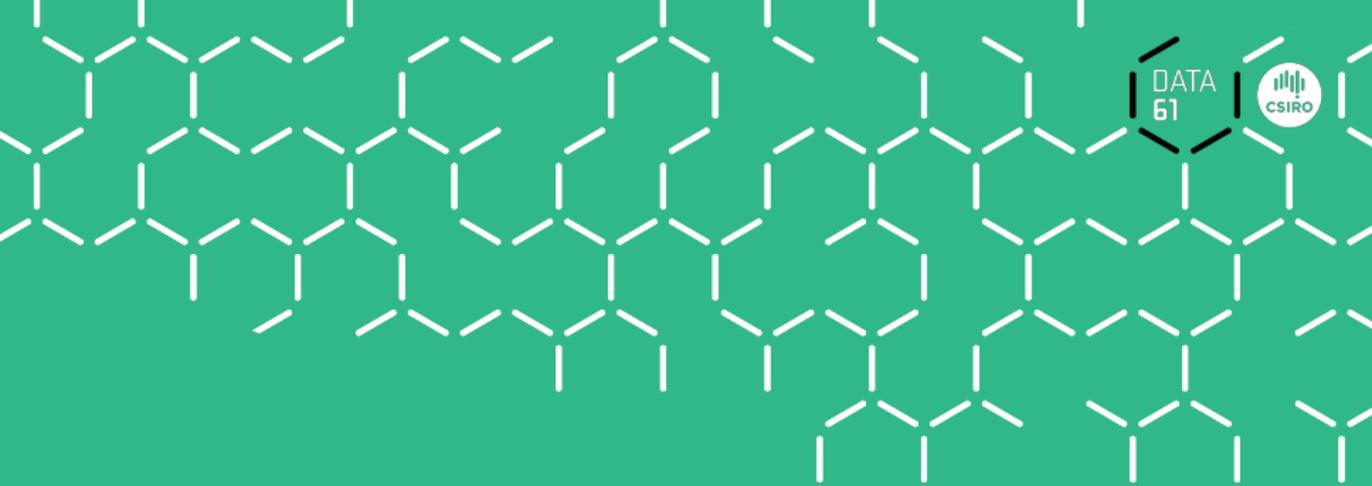
have = " $l_1 \odot r_1$ " [proof]
also
have " $\dots \odot r_2$ " [proof]
also

Anatomy of a [trans] rule:

- Usual form: plain transitivity $\llbracket l_1 \odot r_1; r_1 \odot r_2 \rrbracket \implies l_1 \odot r_2$
- More general form: $\llbracket P \ l_1 \ r_1; Q \ r_1 \ r_2; A \rrbracket \implies C \ l_1 \ r_2$

Examples:

- pure transitivity: $\llbracket a = b; b = c \rrbracket \implies a = c$
- mixed: $\llbracket a \leq b; b < c \rrbracket \implies a < c$
- substitution: $\llbracket P \ a; a = b \rrbracket \implies P \ b$
- antisymmetry: $\llbracket a < b; b < a \rrbracket \implies \text{False}$
- monotonicity:
 $\llbracket a = f \ b; b < c; \bigwedge x \ y. x < y \implies f \ x < f \ y \rrbracket \implies a < f \ c$



DATA
61



Demo