DATA 61

COMP4161: Advanced Topics in Software Verification

Isar

Gerwin Klein, June Andronick, Ramana Kumar
S2/2016

data61.csiro.au

CSIRO

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]

---

[a] a1 due; [b] a2 due; [c] a3 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: $(A \longrightarrow B) = (B \vee \neg A)$?

Demo

Isar

apply scripts	**What about..**
➜ unreadable	➜ Elegance?
➜ hard to maintain	➜ Explaining deeper insights?
➜ do not scale	➜ Large developments?
No structure.	**Isar!**

A typical Isar proof

```
proof
  assume $formula_0$
  have $formula_1$     by simp
  ⋮
  have $formula_n$     by blast
  show $formula_{n+1}$ by ...
qed
```

proves $formula_0 \Longrightarrow formula_{n+1}$

(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⟧ ⟹ A ∧ 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 openend 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 "$\llbracket A; B \rrbracket \Longrightarrow A \wedge B$" **[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 :: 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
```

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

Demo

Fancy Abbreviations

this = the previous fact proved or assumed

then = **from** this
thus = **then show**
hence = **then have**
with $A_1 \ldots A_n$ = **from** $A_1 \ldots A_n$ this

?thesis = the last enclosing goal statement

Moreover and Ultimately

have X_1: P_1 . . .	**have** P_1 . . .
have X_2: P_2 . . .	**moreover** **have** P_2 . . .
⋮	⋮
have X_n: P_n . . .	**moreover** **have** P_n . . .
from $X_1 \ldots X_n$ **show** . . .	**ultimately** **show** . . .

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

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 $P_1 \Longrightarrow P$

Mixing proof styles

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

Datatypes in Isar

Datatype case distinction

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

case (Constructor$_i$ $\vec{x}$) $\quad\equiv$
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: $P\ n$
  let $?case = P\ (\text{Suc}\ n)$
  ...
  ... n ...
  show $?case$
qed

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

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

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

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

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

$$\text{shows } a = d \text{ by transitivity of } =$$

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