Intro & motivation, getting started

Foundations & Principles
- Lambda Calculus, natural deduction
- Higher Order Logic
- Term rewriting

Proof & Specification Techniques
- Inductively defined sets, rule induction
- Datatypes, recursion, induction
- Hoare logic, proofs about programs, C verification
- (mid-semester break)
- Writing Automated Proof Methods
- Isar, codegen, typeclasses, locales

---

\(a_1\) due; \(b_2\) due; \(c_3\) due
Isar

A Language for Structured Proofs
Is this true: \((A \rightarrow B) = (B \lor \neg A)\)?
Motivation

Is this true: \((A \rightarrow B) = (B \lor \neg A)\) ?

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: \((A \rightarrow B) = (B \lor \neg A)\) ?

Demo
Isar

apply scripts

→ unreadable
→ hard to maintain
→ do not scale

What about..

→ Elegance?
→ Explaining deeper insights?
→ Large developments?

No structure.

Isar!
A typical Isar proof

proof
  assume \( \text{formula}_0 \)
  have \( \text{formula}_1 \) by simp
  
  have \( \text{formula}_n \) by blast
  show \( \text{formula}_{n+1} \) by \ldots

qed

proves \( \text{formula}_0 \implies \text{formula}_{n+1} \)

(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

lemma "[A; B] \implies A \land 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 "\[A; B\] \implies A \land B"
proof (rule conjI)

\[
\begin{align*}
\text{\textbf{\rightarrow}} \quad \text{proof (rule conjI) changes proof state to} \\
1. \quad & \ [A; B] \implies A \\
2. \quad & \ [A; B] \implies B \\
\text{\textbf{\rightarrow}} \quad \text{so we need 2 shows: show "}A\text{" and show "}B\text{"} \\
\text{\textbf{\rightarrow}} \quad \text{We are allowed to assume }A, \\
& \text{because }A \text{ is in the assumptions of the proof state.}
\end{align*}
\]
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** "\([A; B] \implies A \land B\)" **[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 """"$A \land B$"""" proof

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

**Forward reasoning:** . . .

   assume AB: """"$A \land B$"
   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

\textbf{fix } v_1 \ldots v_n

Introduces new arbitrary but fixed variables
\((\sim \text{ parameters, } \land)\)

\textbf{obtain } v_1 \ldots v_n \textbf{ where } \langle \text{prop} \rangle \langle \text{proof} \rangle

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 $X_2$: $P_2$ 
: 
have $X_n$: $P_n$ 
from $X_1 \ldots X_n$ show 

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

have $P_1$ 
moreover have $P_2$ 
: 
moreover have $P_n$ 
ultimately show 

---

19 | COMP4161 | © Data61, CSIRO: provided under Creative Commons Attribution License
General Case Distinctions

show formula

proof -

have $P_1 \lor P_2 \lor P_3$ \textless proof\textgreater

moreover \{ assume $P_1$ \ldots have ?thesis \textless proof\textgreater \}

moreover \{ assume $P_2$ \ldots have ?thesis \textless proof\textgreater \}

moreover \{ assume $P_3$ \ldots have ?thesis \textless proof\textgreater \}

ultimately show ?thesis by blast

qed

\{ \ldots \} is a proof block similar to proof ... qed

\{ assume $P_1$ \ldots have $P$ \textless proof\textgreater \}

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

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

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

\[
\text{show } " \land x. A \ n \Rightarrow P \ n"
\]

\[
\text{proof (induct } n\text{)}
\]

\[
\text{case 0}
\]

\[
\ldots
\]

\[
\text{show } ?\text{case}
\]

\[
\text{next}
\]

\[
\text{case } (\text{Suc } n)
\]

\[
\ldots
\]

\[
\ldots \ n \ldots
\]

\[
\ldots
\]

\[
\text{show } ?\text{case}
\]

\[
\text{qed}
\]

\[
\equiv \ \text{fix } x \ \text{assume } 0: " A \ 0"
\]

\[
\text{let } ?\text{case} = " P \ 0"
\]

\[
\equiv \ \text{fix } n \ \text{and } x
\]

\[
\text{assume } \text{Suc}: " \land x. A \ n \Rightarrow P \ n"
\]

\[
" A \ (\text{Suc } n)"
\]

\[
\text{let } ?\text{case} = " P \ (\text{Suc } n)"
\]
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:
\[ x \cdot x^{-1} = 1 \cdot (x \cdot x^{-1}) \]
\[ \ldots = 1 \cdot x \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot 1 \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot (1 \cdot x^{-1}) \]
\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \]
\[ \ldots = 1 \]

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
→ 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 =

Each step usually nontrivial (requires own subproof)

Solution in Isar:

\[\rightarrow\] Keywords also and finally to delimit steps
\[\rightarrow\] . . . : predefined schematic term variable, refers to right hand side of last expression
\[\rightarrow\] Automatic use of transitivity rules to connect steps
also/finally

have "t_0 = t_1" [proof]
also
have "\ldots = t_2" [proof]
also
:
also
have "\ldots = 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"
:
"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 = "l₁ ⊙ r₁" [proof]
also
have "... ⊙ r₂" [proof]
also

Anatomy of a [trans] rule:

- Usual form: plain transitivity \([l₁ ⊙ r₁; r₁ ⊙ r₂] \implies l₁ ⊙ r₂\)
- More general form: \([P l₁ r₁; Q r₁ r₂; A] \implies C l₁ r₂\)

Examples:

- pure transitivity: \([a = b; b = c] \implies a = c\)
- mixed: \([a \leq b; b < c] \implies a < c\)
- substitution: \([P a; a = b] \implies P b\)
- antisymmetry: \([a < b; b < a] \implies False\)
- monotonicity:
  \([a = f b; b < c; \land x y. x < y \implies f x < f y] \implies a < f c\)
Demo