

COMP4161: Advanced Topics in Software Verification

# Isar

Gerwin Klein, June Andronick, Ramana Kumar S2/2016



### Content



- → Intro & motivation, getting started
- → Foundations & Principles

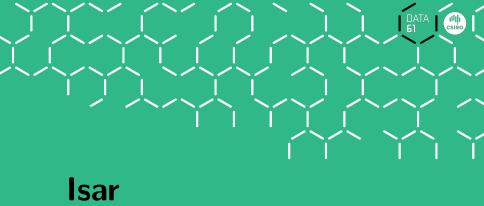
<ul> <li>Lambda Calculus, natural deduction</li> </ul>	[1,2]
Higher Order Logic	[3 <sup>a</sup> ]
<ul> <li>Term rewriting</li> </ul>	[4]

→ Proof & Specification Techniques

<ul> <li>Inductively defined sets, rule induction</li> </ul>	[5]
<ul> <li>Datatypes, recursion, induction</li> </ul>	[6, 7]
<ul> <li>Hoare logic, proofs about programs, C verification</li> </ul>	$[8^{b}, 9]$

- (mid-semester break)
- Writing Automated Proof Methods [10]
- Isar, codegen, typeclasses, locales [11<sup>c</sup>,12]

<sup>&</sup>lt;sup>a</sup>a1 due; <sup>b</sup>a2 due; <sup>c</sup>a3 due



A Language for Structured Proofs

## **Motivation**



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

## **Motivation**



by blast

```
Is this true: (A \longrightarrow 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 (rule disjE)
apply assumption
apply (erule disjE)
apply assumption
apply (erule notE)
apply assumption
done

OK it's true. But WHY?

## **Motivation**



WHY is this true: 
$$(A \longrightarrow B) = (B \lor \neg A)$$
?

Demo

## Isar



#### apply scripts

What about...

- unreadable
- do not scale

- → Elegance?
- hard to maintain 

  Explaining deeper insights?
  - → Large developments?

No structure.

Isar!

# A typical Isar proof



```
proof
                  assume formula<sub>0</sub>
                 have formula<sub>1</sub> by simp
                  have formula<sub>n</sub> by blast
                 show formula<sub>n+1</sub> by . . .
               qed
             proves formula_0 \Longrightarrow formula_{n+1}
(analogous to assumes/shows in lemma statements)
```

# Isar core syntax



# proof and qed

proof (rule conjl)

**lemma** " $[A; B] \Longrightarrow A \wedge B$ "



#### proof [method] statement\* qed

```
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** " $[A; B] \Longrightarrow A \wedge B$ " **proof** (rule conjl)

- → proof (rule conjl) changes proof state to
  - 1.  $[A; B] \Longrightarrow A$
  - 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 "[A; B] \Longrightarrow A \land B" [prove]
proof (rule conjl) [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
```



## **Backward and Forward**



- **Backward reasoning:** ... have " $A \wedge B$ " proof
  - → proof picks an intro rule automatically
  - $\rightarrow$  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

- $\rightarrow$  first *n* assumptions of rule must unify with  $A_1 \ldots A_n$
- → conclusion of rule must unify with *R*

## Fix and Obtain



fix 
$$v_1 \dots v_n$$

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

**obtain** 
$$v_1 \dots v_n$$
 **where**  $<$ prop $>$   $<$ proof $>$ 

Introduces new variables together with property



# **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 ...
have X_2: P_2 ...
have X_n: P_n . . .
from X_1 \dots X_n show . . .
```

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

```
have P_1 ...
moreover have P_2 ...
moreover have P_n ...
ultimately show . . .
```

## **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
ged
      { ...} is a proof block similar to proof ... qed
          { assume P_1 \dots have P <proof> }
                   stands for P_1 \Longrightarrow P
```

# Mixing proof styles



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



# **Datatype case distinction**



```
proof (cases term)
   case Constructor<sub>1</sub>
next
next
   case (Constructor<sub>k</sub> \vec{x})
   \vec{x} ...
ged
       case (Constructor; \vec{x}) \equiv
       fix \vec{x} assume Constructor<sub>i</sub>: "term = Constructor<sub>i</sub> \vec{x}"
```

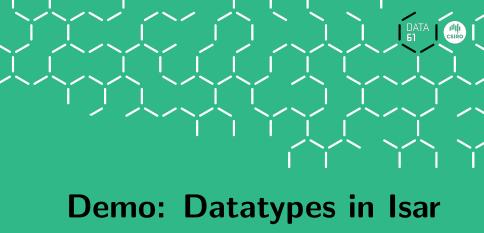
# Structural induction for nat

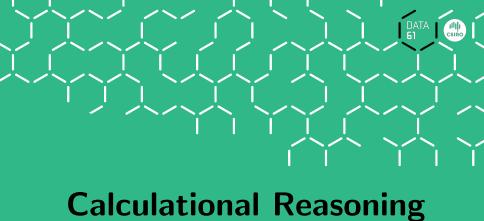


```
show P n
proof (induct n)
                     \equiv let ?case = P 0
  case 0
  show ?case
next
  case (Suc n) \equiv fix n assume Suc: P n
                         let ?case = P (Suc n)
  \cdots n \cdots
  show ?case
qed
```

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







## The Goal



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

assoc:  $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ using:

left\_inv:  $x^{-1} \cdot x = 1$ left\_one:  $1 \cdot x = x$ 

## The Goal



#### Prove:

$$\begin{array}{lll} x \cdot x^{-1} &= 1 \cdot (x \cdot x^{-1}) & \text{assoc:} & (x \cdot y) \cdot z = x \cdot (y \cdot z) \\ \dots &= 1 \cdot x \cdot x^{-1} & \text{left\_inv:} & x^{-1} \cdot x = 1 \\ \dots &= (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1} & \text{left\_one:} & 1 \cdot x = x \\ \dots &= (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1} & \\ \dots &= (x^{-1})^{-1} \cdot 1 \cdot x^{-1} & \\ \dots &= (x^{-1})^{-1} \cdot (1 \cdot x^{-1}) & \\ \dots &= (x^{-1})^{-1} \cdot x^{-1} & \\ \dots &= 1 & \end{array}$$

#### 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

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
have "... = t_2" [proof]
also
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" t_0 = t_n" t_0 = t_{n-1}" t_0 = t_n
```

## More about also



- $\rightarrow$  Works for all combinations of =,  $\leq$  and <.
- → Uses all rules declared as [trans].
- → To view all combinations: print\_trans\_rules

# **Designing [trans] Rules**



have = "
$$I_1 \odot r_1$$
" [proof] also have "...  $\odot r_2$ " [proof] also

#### Anatomy of a [trans] rule:

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

#### Examples:

- $\rightarrow$  pure transitivity:  $[a = b; b = c] \implies a = c$
- $\rightarrow$  mixed:  $\llbracket a \leq b; b < c \rrbracket \Longrightarrow a < c$
- → substitution:  $\llbracket P \ a; a = b \rrbracket \Longrightarrow P \ b$
- $\rightarrow$  antisymmetry:  $[a < b; b < a] \Longrightarrow False$
- → monotonicity:

$$\llbracket a = f \ b; b < c; \bigwedge x \ y. \ x < y \Longrightarrow f \ x < f \ y \rrbracket \Longrightarrow a < f \ c$$

