

# Induction\_Demo.thy

Generalisation

# Generalisation

- Replace constants by variables
- Generalize free variables
  - by *arbitrary* in induction proof
  - (or by universal quantifier in formula)

So far, all proofs were by **structural induction**  
because all functions were **primitive recursive**.

In each induction step, 1 constructor is added.  
In each recursive call, 1 constructor is removed.

Now: induction for complex recursion patterns.

# Computation Induction

## Example

**fun** *div2* :: *nat*  $\Rightarrow$  *nat* **where**  
*div2* 0 = 0 |  
*div2* (*Suc* 0) = 0 |  
*div2* (*Suc*(*Suc* *n*)) = *Suc*(*div2* *n*)

$\rightsquigarrow$  induction rule *div2.induct*:

$$\frac{P(0) \quad P(\text{Suc } 0) \quad \bigwedge n. P(n)}{P(m)}$$

# Computation Induction

If  $f :: \tau \Rightarrow \tau'$  is defined by **fun**, a special induction schema is provided to prove  $P(x)$  for all  $x :: \tau$ :

*for each defining equation*

$$f(e) = \dots f(r_1) \dots f(r_k) \dots$$

*prove  $P(e)$  assuming  $P(r_1), \dots, P(r_k)$ .*

Induction follows course of (terminating!) computation  
Motto: properties of  $f$  are best proved by rule *f.induct*

## How to apply *f.induct*

If  $f :: \tau_1 \Rightarrow \dots \Rightarrow \tau_n \Rightarrow \tau'$ :

(*induction a<sub>1</sub> ... a<sub>n</sub> rule: f.induct*)

Heuristic:

- there should be a call  $f a_1 \dots a_n$  in your goal
- ideally the  $a_i$  should be variables.

# Induction\_Demo.thy

Computation Induction

- ③ Overview of Isabelle/HOL
- ④ Type and function definitions
- ⑤ Induction Heuristics
- ⑥ Simplification**



# Simplification means . . .

Using equations  $l = r$  from left to right

As long as possible

Terminology: equation  $\rightsquigarrow$  *simplification rule*

Simplification = (Term) Rewriting

## An example

*Equations:*

$$\begin{aligned}0 + n &= n && (1) \\(Suc\ m) + n &= Suc\ (m + n) && (2) \\(Suc\ m \leq Suc\ n) &= (m \leq n) && (3) \\(0 \leq m) &= True && (4)\end{aligned}$$

*Rewriting:*

$$\begin{aligned}0 + Suc\ 0 &\leq Suc\ 0 + x && \underline{\underline{(1)}} \\Suc\ 0 &\leq Suc\ 0 + x && \underline{\underline{(2)}} \\Suc\ 0 &\leq Suc\ (0 + x) && \underline{\underline{(3)}} \\0 &\leq 0 + x && \underline{\underline{(4)}} \\&True && \end{aligned}$$

# Conditional rewriting

Simplification rules can be conditional:

$$\llbracket P_1; \dots; P_k \rrbracket \Longrightarrow l = r$$

is applicable only if all  $P_i$  can be proved first, again by simplification.

## Example

$$p(x) \Longrightarrow \begin{array}{l} p(0) = True \\ f(x) = g(x) \end{array}$$

We can simplify  $f(0)$  to  $g(0)$  but we cannot simplify  $f(1)$  because  $p(1)$  is not provable.

# Termination

Simplification may not terminate.

Isabelle uses *simp*-rules (almost) blindly from left to right.

Example:  $f(x) = g(x)$ ,  $g(x) = f(x)$

Principle:

$$\llbracket P_1; \dots; P_k \rrbracket \Longrightarrow l = r$$

is suitable as a *simp*-rule only

if  $l$  is “bigger” than  $r$  and each  $P_i$

$$n < m \Longrightarrow (n < \text{Suc } m) = \text{True} \quad \text{YES}$$

$$\text{Suc } n < m \Longrightarrow (n < m) = \text{True} \quad \text{NO}$$

## Proof method *simp*

Goal: 1.  $\llbracket P_1; \dots; P_m \rrbracket \implies C$

**apply**(*simp add: eq<sub>1</sub> ... eq<sub>n</sub>*)

Simplify  $P_1 \dots P_m$  and  $C$  using

- lemmas with attribute *simp*
- rules from **fun** and **datatype**
- additional lemmas  $eq_1 \dots eq_n$
- assumptions  $P_1 \dots P_m$

Variations:

- (*simp ... del: ...*) removes *simp*-lemmas
- *add* and *del* are optional

## *auto* versus *simp*

- *auto* acts on all subgoals
- *simp* acts only on subgoal 1
- *auto* applies *simp* and more
- *auto* can also be modified:  
(*auto simp add: ... simp del: ...*)

# Rewriting with definitions

Definitions (**definition**) must be used **explicitly**:

*(simp add: f\_def ...)*

*f* is the function whose definition is to be unfolded.

## Case splitting with *simp/*auto

Automatic:

$$\begin{aligned} & P \text{ (if } A \text{ then } s \text{ else } t) \\ & \quad = \\ & (A \longrightarrow P(s)) \wedge (\neg A \longrightarrow P(t)) \end{aligned}$$

By hand:

$$\begin{aligned} & P \text{ (case } e \text{ of } 0 \Rightarrow a \mid \text{Suc } n \Rightarrow b) \\ & \quad = \\ & (e = 0 \longrightarrow P(a)) \wedge (\forall n. e = \text{Suc } n \longrightarrow P(b)) \end{aligned}$$

Proof method: (*simp split: nat.split*)

Or *auto*. Similar for any datatype *t*: *t.split*



Simp\_Demo.thy

⑧ Logical Formulas

⑨ Proof Automation

⑩ Single Step Proofs

⑪ Inductive Definitions

Syntax (in decreasing precedence):

$$\begin{array}{l} \text{form} ::= (\text{form}) \quad | \quad \text{term} = \text{term} \quad | \quad \neg \text{form} \\ \quad \quad | \quad \text{form} \wedge \text{form} \quad | \quad \text{form} \vee \text{form} \quad | \quad \text{form} \longrightarrow \text{form} \\ \quad \quad | \quad \forall x. \text{form} \quad | \quad \exists x. \text{form} \end{array}$$

Examples:

$$\begin{aligned} \neg A \wedge B \vee C &\equiv ((\neg A) \wedge B) \vee C \\ s = t \wedge C &\equiv (s = t) \wedge C \\ A \wedge B = B \wedge A &\equiv A \wedge (B = B) \wedge A \\ \forall x. P x \wedge Q x &\equiv \forall x. (P x \wedge Q x) \end{aligned}$$

Input syntax:  $\longleftrightarrow$  (same precedence as  $\longrightarrow$ )

Variable binding convention:

$$\forall x y. P x y \equiv \forall x. \forall y. P x y$$

Similarly for  $\exists$  and  $\lambda$ .

# Warning

Quantifiers have low precedence  
and need to be parenthesized (if in some context)

$$! \quad P \wedge \forall x. Q x \rightsquigarrow P \wedge (\forall x. Q x) \quad !$$

# Mathematical symbols

... and their ascii representations:

$\forall$	<code>\&lt;forall&gt;</code>	ALL
$\exists$	<code>\&lt;exists&gt;</code>	EX
$\lambda$	<code>\&lt;lambda&gt;</code>	%
$\longrightarrow$	<code>--&gt;</code>	
$\longleftrightarrow$	<code>&lt;-&gt;</code>	
$\wedge$	<code>\&amp;</code>	&
$\vee$	<code>\ </code>	
$\neg$	<code>\&lt;not&gt;</code>	~
$\neq$	<code>\&lt;noteq&gt;</code>	~=

# Sets over type 'a

'a set

- $\{\}, \{e_1, \dots, e_n\}$
- $e \in A, A \subseteq B$
- $A \cup B, A \cap B, A - B, -A$
- ...

$\in$	<code>\&lt;in&gt;</code>	:
$\subseteq$	<code>\&lt;subseteq&gt;</code>	<code>&lt;=</code>
$\cup$	<code>\&lt;union&gt;</code>	<code>Un</code>
$\cap$	<code>\&lt;inter&gt;</code>	<code>Int</code>

# Set comprehension

- $\{x. P\}$  where  $x$  is a variable
- But not  $\{t. P\}$  where  $t$  is a proper term
- Instead:  $\{t \mid x \ y \ z. P\}$   
is short for  $\{v. \exists x \ y \ z. v = t \wedge P\}$   
where  $x, y, z$  are the free variables in  $t$



- ⑧ Logical Formulas
- ⑨ Proof Automation
- ⑩ Single Step Proofs
- ⑪ Inductive Definitions

## *simp* and *auto*

*simp*: rewriting and a bit of arithmetic

*auto*: rewriting and a bit of arithmetic, logic and sets

- Show you where they got stuck
- highly incomplete
- Extensible with new *simp*-rules

Exception: *auto* acts on all subgoals

## *fastforce*

- rewriting, logic, sets, relations and a bit of arithmetic.
- **incomplete** but better than *auto*.
- Succeeds or fails
- Extensible with new *simp*-rules

## *blast*

- A **complete** proof search procedure for FOL ...
- ... but (almost) **without “=”**
- Covers logic, sets and relations
- Succeeds or fails
- Extensible with new deduction rules

# Automating arithmetic

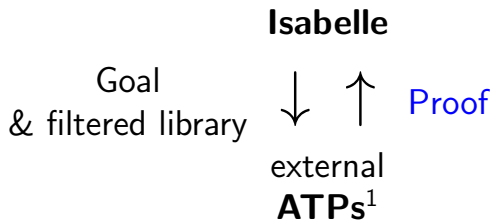
*arith*:

- proves linear formulas (no “\*”)
- complete for quantifier-free *real* arithmetic
- complete for first-order theory of *nat* and *int*  
(Presburger arithmetic)

# Sledgehammer



## Architecture:



## Characteristics:

- Sometimes it works,
- sometimes it doesn't.

Do you feel lucky?

---

<sup>1</sup>Automatic Theorem Provers

**by**(*proof-method*)

≈

**apply**(*proof-method*)  
**done**



Auto\_Proof\_Demo.thy

- ⑧ Logical Formulas
- ⑨ Proof Automation
- ⑩ Single Step Proofs**
- ⑪ Inductive Definitions

Step-by-step proofs can be necessary if automation fails and you have to explore where and why it failed by taking the goal apart.

## What are these *?-variables* ?

After you have finished a proof, Isabelle turns all free variables  $V$  in the theorem into  $?V$ .

Example: theorem conjI:  $\llbracket ?P; ?Q \rrbracket \Longrightarrow ?P \wedge ?Q$

These *?-variables* can later be instantiated:

- By hand:

`conjI[of "a=b" "False"]`  $\rightsquigarrow$   
 $\llbracket a = b; False \rrbracket \Longrightarrow a = b \wedge False$

- By **unification**:

unifying  $?P \wedge ?Q$  with  $a=b \wedge False$   
sets  $?P$  to  $a=b$  and  $?Q$  to  $False$ .

## Rule application

Example: rule:  $[[?P; ?Q]] \implies ?P \wedge ?Q$

subgoal: 1.  $\dots \implies A \wedge B$

Result: 1.  $\dots \implies A$

2.  $\dots \implies B$

The general case: applying rule  $[[ A_1; \dots ; A_n ]] \implies A$   
to subgoal  $\dots \implies C$ :

- Unify  $A$  and  $C$
- Replace  $C$  with  $n$  new subgoals  $A_1 \dots A_n$

**apply**(*rule xyz*)

“Backchaining”

## Typical backwards rules

$$\frac{?P \quad ?Q}{?P \wedge ?Q} \text{conjI}$$

$$\frac{?P \implies ?Q}{?P \longrightarrow ?Q} \text{impI} \quad \frac{\bigwedge x. ?P x}{\forall x. ?P x} \text{allI}$$

$$\frac{?P \implies ?Q \quad ?Q \implies ?P}{?P = ?Q} \text{iffI}$$

They are known as **introduction rules** because they *introduce* a particular connective.

## Automating intro rules

If  $r$  is a theorem  $\llbracket A_1; \dots; A_n \rrbracket \implies A$  then

$(blast\ intro: r)$

allows *blast* to backchain on  $r$  during proof search.

Example:

**theorem** *le\_trans*:  $\llbracket ?x \leq ?y; ?y \leq ?z \rrbracket \implies ?x \leq ?z$

**goal** 1.  $\llbracket a \leq b; b \leq c; c \leq d \rrbracket \implies a \leq d$

**proof** **apply**(*blast intro: le\_trans*)

Also works for *auto* and *fastforce*

Can greatly increase the search space!

## Forward proof: OF

If  $r$  is a theorem  $A \implies B$

and  $s$  is a theorem that unifies with  $A$  then

$$r[OF\ s]$$

is the theorem obtained by proving  $A$  with  $s$ .

Example: theorem `refl`:  $?t = ?t$

`conjI [OF refl [of "a"]]`

$$?Q \implies \overset{\rightsquigarrow}{a = a} \wedge ?Q$$



The general case:

If  $r$  is a theorem  $\llbracket A_1; \dots; A_n \rrbracket \implies A$   
and  $r_1, \dots, r_m$  ( $m \leq n$ ) are theorems then

$$r[OF\ r_1\ \dots\ r_m]$$

is the theorem obtained

by proving  $A_1 \dots A_m$  with  $r_1 \dots r_m$ .

Example: theorem `refl`:  $?t = ?t$

`conjI[OF refl[of "a"] refl[of "b"]]`

$\rightsquigarrow$

$$a = a \wedge b = b$$

From now on: ? mostly suppressed on slides

Single\_Step\_Demo.thy

$\implies$  versus  $\longrightarrow$

$\implies$  is part of the Isabelle framework. It structures theorems and proof states:  $\llbracket A_1; \dots; A_n \rrbracket \implies A$

$\longrightarrow$  is part of HOL and can occur inside the logical formulas  $A_i$  and  $A$ .

Phrase theorems like this  $\llbracket A_1; \dots; A_n \rrbracket \implies A$   
not like this  $A_1 \wedge \dots \wedge A_n \longrightarrow A$

- ⑧ Logical Formulas
- ⑨ Proof Automation
- ⑩ Single Step Proofs
- ⑪ Inductive Definitions

## Example: even numbers

Informally:

- 0 is even
- If  $n$  is even, so is  $n + 2$
- These are the only even numbers

In Isabelle/HOL:

**inductive**  $ev :: nat \Rightarrow bool$

**where**

$ev\ 0 \quad |$

$ev\ n \Longrightarrow ev\ (n + 2)$

An easy proof: *ev 4*

$$ev\ 0 \implies ev\ 2 \implies ev\ 4$$

Consider

```
fun evn :: nat  $\Rightarrow$  bool where  
  evn 0 = True |  
  evn (Suc 0) = False |  
  evn (Suc (Suc n)) = evn n
```

A trickier proof:  $ev\ m \Longrightarrow evn\ m$

By induction on the *structure* of the derivation of  $ev\ m$

Two cases:  $ev\ m$  is proved by

- rule  $ev\ 0$   
 $\Longrightarrow m = 0 \Longrightarrow evn\ m = True$
- rule  $ev\ n \Longrightarrow ev\ (n+2)$   
 $\Longrightarrow m = n+2$  and  $evn\ n$  (IH)  
 $\Longrightarrow evn\ m = evn\ (n+2) = evn\ n = True$



## Rule induction for $ev$

To prove

$$ev\ n \Longrightarrow P\ n$$

by *rule induction* on  $ev\ n$  we must prove

- $P\ 0$
- $P\ n \Longrightarrow P(n+2)$

Rule  $ev.induct$ :

$$\frac{ev\ n \quad P\ 0 \quad \bigwedge n. \llbracket ev\ n; P\ n \rrbracket \Longrightarrow P(n+2)}{P\ n}$$

## Format of inductive definitions

**inductive**  $I :: \tau \Rightarrow bool$  **where**  
    [[  $I a_1; \dots ; I a_n$  ]]  $\Longrightarrow I a$  |  
    :  
    :

Note:

- $I$  may have multiple arguments.
- Each rule may also contain *side conditions* not involving  $I$ .

## Rule induction in general

To prove

$$I x \implies P x$$

by *rule induction* on  $I x$

we must prove for every rule

$$\llbracket I a_1; \dots ; I a_n \rrbracket \implies I a$$

that  $P$  is preserved:

$$\llbracket I a_1; P a_1; \dots ; I a_n; P a_n \rrbracket \implies P a$$

! Rule induction is absolutely central  
to (operational) semantics  
and the rest of this lecture course !

Inductive\_Demo.thy

# Inductively defined sets

**inductive\_set**  $I :: \tau$  set **where**

$\llbracket a_1 \in I; \dots ; a_n \in I \rrbracket \implies a \in I \mid$

$\vdots$

Difference to **inductive**:

- arguments of  $I$  are tupled, not curried
- $I$  can later be used with set theoretic operators, eg  $I \cup \dots$

# Chapter 5

## Isar: A Language for Structured Proofs

12 Isar by example

13 Proof patterns

14 Streamlining Proofs

15 Proof by Cases and Induction



# Apply scripts

- unreadable
- hard to maintain
- do not scale

No structure!

# Apply scripts versus Isar proofs

Apply script = assembly language program

Isar proof = structured program with assertions

But: **apply** still useful for proof exploration

## A typical Isar proof

**proof**

**assume**  $formula_0$

**have**  $formula_1$  **by** *simp*

$\vdots$

**have**  $formula_n$  **by** *blast*

**show**  $formula_{n+1}$  **by**  $\dots$

**qed**

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

## Isar core syntax

proof = **proof** [method] step\* **qed**  
| **by** method

method = (*simp* ...) | (*blast* ...) | (*induction* ...) | ...

step = **fix** variables ( $\wedge$ )  
| **assume** prop ( $\implies$ )  
| [**from** fact<sup>+</sup>] (**have** | **show**) prop proof

prop = [name:] "formula"

fact = name | ...

12 Isar by example

13 Proof patterns

14 Streamlining Proofs

15 Proof by Cases and Induction

## Example: Cantor's theorem

```
lemma  $\neg$  surj(f :: 'a  $\Rightarrow$  'a set)
proof default proof: assume surj, show False
  assume a: surj f
  from a have b:  $\forall$  A.  $\exists$  a. A = f a
    by(simp add: surj_def)
  from b have c:  $\exists$  a.  $\{x. x \notin f\ x\} = f\ a$ 
    by blast
  from c show False
    by blast
qed
```

Isar\_Demo.thy

Cantor and abbreviations

# Abbreviations

*this* = the previous proposition proved or assumed  
**then** = **from** *this*  
**thus** = **then show**  
**hence** = **then have**



## using and with

(**have|show**) prop **using** facts  
=  
**from** facts (**have|show**) prop

**with** facts  
=  
**from** facts *this*

# Structured lemma statement

**lemma**

**fixes**  $f :: 'a \Rightarrow 'a \text{ set}$

**assumes**  $s: \text{surj } f$

**shows**  $\text{False}$

**proof** – no automatic proof step

**have**  $\exists a. \{x. x \notin f x\} = f a$  **using**  $s$

**by** (*auto simp: surj-def*)

**thus**  $\text{False}$  **by** *blast*

**qed**

*Proves*  $\text{surj } f \implies \text{False}$

*but*  $\text{surj } f$  becomes local fact  $s$  in proof.

# The essence of structured proofs

Assumptions and intermediate facts  
can be named and referred to explicitly and selectively

# Structured lemma statements

**fixes**  $x :: \tau_1$  **and**  $y :: \tau_2 \dots$

**assumes**  $a: P$  **and**  $b: Q \dots$

**shows**  $R$

- **fixes** and **assumes** sections optional
- **shows** optional if no **fixes** and **assumes**

12 Isar by example

13 Proof patterns

14 Streamlining Proofs

15 Proof by Cases and Induction

## Case distinction

**show**  $R$   
**proof** *cases*  
  **assume**  $P$   
  :  
  **show**  $R$   $\langle proof \rangle$   
**next**  
  **assume**  $\neg P$   
  :  
  **show**  $R$   $\langle proof \rangle$   
**qed**

**have**  $P \vee Q$   $\langle proof \rangle$   
**then show**  $R$   
**proof**  
  **assume**  $P$   
  :  
  **show**  $R$   $\langle proof \rangle$   
**next**  
  **assume**  $Q$   
  :  
  **show**  $R$   $\langle proof \rangle$   
**qed**

## Contradiction

```
show  $\neg P$   
proof  
  assume  $P$   
   $\vdots$   
  show False  $\langle proof \rangle$   
qed
```

```
show  $P$   
proof (rule ccontr)  
  assume  $\neg P$   
   $\vdots$   
  show False  $\langle proof \rangle$   
qed
```



```
show  $P \longleftrightarrow Q$   
proof  
  assume  $P$   
   $\vdots$   
  show  $Q$   $\langle proof \rangle$   
next  
  assume  $Q$   
   $\vdots$   
  show  $P$   $\langle proof \rangle$   
qed
```



## $\forall$ and $\exists$ introduction

**show**  $\forall x. P(x)$

**proof**

**fix**  $x$  local fixed variable

**show**  $P(x)$   $\langle proof \rangle$

**qed**

**show**  $\exists x. P(x)$

**proof**

$\vdots$

**show**  $P(witness)$   $\langle proof \rangle$

**qed**

## $\exists$ elimination: **obtain**

**have**  $\exists x. P(x)$

**then obtain**  $x$  **where**  $p: P(x)$  **by** *blast*

$\vdots$   $x$  fixed local variable

Works for one or more  $x$

## obtain example

**lemma**  $\neg \text{surj}(f :: 'a \Rightarrow 'a \text{ set})$

**proof**

**assume**  $\text{surj } f$

**hence**  $\exists a. \{x. x \notin f x\} = f a$  **by**  $(\text{auto simp: surj\_def})$

**then obtain**  $a$  **where**  $\{x. x \notin f x\} = f a$  **by**  $\text{blast}$

**hence**  $a \notin f a \iff a \in f a$  **by**  $\text{blast}$

**thus**  $\text{False}$  **by**  $\text{blast}$

**qed**

## Set equality and subset

**show**  $A = B$

**proof**

**show**  $A \subseteq B$   $\langle proof \rangle$

**next**

**show**  $B \subseteq A$   $\langle proof \rangle$

**qed**

**show**  $A \subseteq B$

**proof**

**fix**  $x$

**assume**  $x \in A$

$\vdots$

**show**  $x \in B$   $\langle proof \rangle$

**qed**

# Isar\_Demo.thy

Exercise

12 Isar by example

13 Proof patterns

14 Streamlining Proofs

15 Proof by Cases and Induction

## 14 Streamlining Proofs

Pattern Matching and Quotations

Top down proof development

**moreover**

Local lemmas

## Example: pattern matching

**show**  $formula_1 \longleftrightarrow formula_2$  (**is**  $?L \longleftrightarrow ?R$ )

**proof**

**assume**  $?L$

$\vdots$

**show**  $?R$   $\langle proof \rangle$

**next**

**assume**  $?R$

$\vdots$

**show**  $?L$   $\langle proof \rangle$

**qed**



*?thesis*

```
show formula (is ?thesis)  
proof -  
  ⋮  
  show ?thesis  $\langle$ proof $\rangle$   
qed
```

Every **show** implicitly defines *?thesis*

# let

Introducing local abbreviations in proofs:

```
let ?t = "some-big-term"
```

```
⋮
```

```
have "... ?t ..."
```

## Quoting facts by value

By name:

```
have x0: "x > 0" ...  
:  
from x0 ...
```

By value:

```
have "x > 0" ...  
:  
from 'x>0' ...  
      ↑   ↑  
      back quotes
```

# Isar\_Demo.thy

Pattern matching and quotations

## 14 Streamlining Proofs

Pattern Matching and Quotations

Top down proof development

**moreover**

Local lemmas

# Example

**lemma**

$\exists ys\ zs. xs = ys @ zs \wedge$

$(length\ ys = length\ zs \vee length\ ys = length\ zs + 1)$

**proof ???**

# Isar\_Demo.thy

Top down proof development

## When automation fails

Split proof up into smaller steps.

Or explore by **apply**:

**have** ... **using** ...

**apply** -

to make incoming facts  
part of proof state

**apply** *auto*

or whatever

**apply** ...

At the end:

- **done**
- Better: **convert to structured proof**



## 14 Streamlining Proofs

Pattern Matching and Quotations

Top down proof development

**moreover**

Local lemmas

## moreover—ultimately

**have**  $P_1 \dots$

**moreover**

**have**  $P_2 \dots$

**moreover**

$\vdots$

**moreover**

**have**  $P_n \dots$

**ultimately**

**have**  $P \dots$

$\approx$

**have**  $lab_1: P_1 \dots$

**have**  $lab_2: P_2 \dots$

$\vdots$

**have**  $lab_n: P_n \dots$

**from**  $lab_1 lab_2 \dots$

**have**  $P \dots$

With names

## 14 Streamlining Proofs

Pattern Matching and Quotations

Top down proof development

**moreover**

Local lemmas

## Local lemmas

**have**  $B$  **if** *name*:  $A_1 \dots A_m$  **for**  $x_1 \dots x_n$   
*<proof>*

proves  $\llbracket A_1; \dots ; A_m \rrbracket \implies B$

where all  $x_i$  have been replaced by  $?x_i$ .

## Proof state and Isar text

In general:     **proof** *method*

Applies *method* and generates subgoal(s):

$$\bigwedge x_1 \dots x_n. \llbracket A_1; \dots ; A_m \rrbracket \implies B$$

How to prove each subgoal:

```
fix  $x_1 \dots x_n$   
assume  $A_1 \dots A_m$   
:  
show  $B$ 
```

Separated by **next**

12 Isar by example

13 Proof patterns

14 Streamlining Proofs

15 Proof by Cases and Induction

# Isar\_Induction\_Demo.thy

Proof by cases

# Datatype case analysis

**datatype**  $t = C_1 \vec{\tau} \mid \dots$

```
proof (cases "term")  
  case ( $C_1 x_1 \dots x_k$ )  
    ...  $x_j$  ...  
next  
   $\vdots$   
qed
```

where **case** ( $C_i x_1 \dots x_k$ )  $\equiv$

```
fix  $x_1 \dots x_k$   
assume  $\underbrace{C_i}_{\text{label}} : \underbrace{\text{term} = (C_i x_1 \dots x_k)}_{\text{formula}}$ 
```



# Isar\_Induction\_Demo.thy

Structural induction for *nat*

## Structural induction for $nat$

```
show  $P(n)$   
proof (induction n)  
  case 0 ≡ let  $?case = P(0)$   
   $\vdots$   
  show  $?case$   
next  
  case (Suc n) ≡ fix  $n$  assume  $Suc: P(n)$   
   $\vdots$  let  $?case = P(Suc\ n)$   
  show  $?case$   
qed
```

# Structural induction with $\implies$

```
show  $A(n) \implies P(n)$   
proof (induction n)  
  case 0  $\equiv$  assume 0:  $A(0)$   
   $\vdots$  let ?case =  $P(0)$   
  show ?case  
next  
  case (Suc n)  $\equiv$  fix n  
   $\vdots$  assume Suc:  $A(n) \implies P(n)$   
   $A(\text{Suc } n)$   
   $\vdots$  let ?case =  $P(\text{Suc } n)$   
  show ?case  
qed
```

## Named assumptions

In a proof of

$$A_1 \implies \dots \implies A_n \implies B$$

by structural induction:

In the context of

**case**  $C$

we have

$C.IH$  the induction hypotheses

$C.prem_s$  the premises  $A_i$

$C$   $C.IH + C.prem_s$

## A remark on style

- **case** (*Suc n*) ... **show** *?case*  
is easy to write and maintain
- **fix** *n* **assume** *formula* ... **show** *formula'*  
is easier to read:
  - all information is shown locally
  - no contextual references (e.g. *?case*)

## 15 Proof by Cases and Induction

Rule Induction

Rule Inversion

# Isar\_Induction\_Demo.thy

Rule induction

# Rule induction

```
inductive  $I :: \tau \Rightarrow \sigma \Rightarrow \text{bool}$   
where  
   $rule_1: \dots$   
   $\vdots$   
   $rule_n: \dots$ 
```

```
show  $I x y \Longrightarrow P x y$   
proof (induction rule: I.induct)  
  case  $rule_1$   
     $\dots$   
    show  $?case$   
next  
   $\vdots$   
next  
  case  $rule_n$   
     $\dots$   
    show  $?case$   
qed
```



## Fixing your own variable names

**case** (*rule<sub>i</sub>*  $x_1 \dots x_k$ )

Renames the first  $k$  variables in *rule<sub>i</sub>* (from left to right) to  $x_1 \dots x_k$ .

## Named assumptions

In a proof of

$$I \dots \implies A_1 \implies \dots \implies A_n \implies B$$

by

rule induction on  $I \dots$ :

In the context of

**case**  $R$

we

have

$R.IH$  the induction hypotheses

$R.hyps$  the assumptions of rule  $R$

$R.prem$ s the premises  $A_i$

$R$   $R.IH + R.hyps + R.prem$ s

## 15 Proof by Cases and Induction

Rule Induction

Rule Inversion

## Rule inversion

**inductive**  $ev :: nat \Rightarrow bool$  **where**

$ev0$ :  $ev\ 0 \mid$

$evSS$ :  $ev\ n \Longrightarrow ev(Suc(Suc\ n))$

What can we deduce from  $ev\ n$  ?

That it was proved by either  $ev0$  or  $evSS$  !

$ev\ n \Longrightarrow n = 0 \vee (\exists k. n = Suc\ (Suc\ k) \wedge ev\ k)$

Rule inversion = case distinction over rules

# Isar\_Induction\_Demo.thy

Rule inversion

## Rule inversion template

**from**  $\text{'ev } n\text{'}$  **have**  $P$

**proof** *cases*

**case**  $ev0$

$n = 0$

$\vdots$

**show**  $?thesis \dots$

**next**

**case**  $(evSS\ k)$

$n = Suc\ (Suc\ k),\ ev\ k$

$\vdots$

**show**  $?thesis \dots$

**qed**

Impossible cases disappear automatically