Isar — A language for structured proofs

Apply scripts

- unreadable

Apply scripts

- unreadable
- hard to maintain

Apply scripts

- unreadable
- hard to maintain
- do not scale

Apply scripts

- unreadable
- hard to maintain
- do not scale

No structure!

Apply scripts versus Isar proofs

Apply script = assembly language program

Apply scripts versus Isar proofs

Apply script = assembly language program

Isar proof = structured program with comments

Apply scripts versus Isar proofs

Apply script = assembly language program

Isar proof = structured program with comments

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

A typical Isar proof

proof

assume $formula_0$

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

⋮

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

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

qed

A typical Isar proof

proof

  assume $formula_0$

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

  $\vdots$

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

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

qed

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

Overview

- Basic Isar
- Isar by example
- Proof patterns
- Streamlining proofs

Isar core syntax

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

Isar core syntax

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

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

Isar core syntax

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

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

statement = **fix** variables ($\bigwedge$)
| **assume** prop ($\Longrightarrow$)
| [**from** fact$^+$] (**have** | **show**) prop proof

Isar core syntax

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

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

statement = **fix** variables ($\bigwedge$)
| **assume** prop ($\Longrightarrow$)
| [**from** fact$^+$] (**have** | **show**) prop proof
| **next** (separates subgoals)

Isar core syntax

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

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

statement = **fix** variables ($\bigwedge$)
| **assume** prop ($\Longrightarrow$)
| [**from** fact$^+$] (**have** | **show**) prop proof
| **next** (separates subgoals)

prop = [name:] "formula"

Isar core syntax

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

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

statement = **fix** variables ($\bigwedge$)
    | **assume** prop ($\Longrightarrow$)
    | [**from** fact$^+$] (**have** | **show**) prop proof
    | **next** (separates subgoals)

prop = [name:] "formula"

fact = name | name**[**OF fact$^+$**]** | ‘formula‘

Isar by example

Example: Cantor's theorem

lemma *Cantor:* $\neg$ *surj(f :: 'a $\Rightarrow$ 'a set)*

Example: Cantor's theorem

lemma *Cantor:* $\neg$ *surj(f :: 'a ⇒ 'a set)*
proof

Example: Cantor's theorem

lemma *Cantor:* $\neg$ *surj(f :: 'a $\Rightarrow$ 'a set)*
proof assume *surj,* show *False*

Example: Cantor's theorem

lemma *Cantor:* $\neg$ *surj(f :: 'a* $\Rightarrow$ *'a set)*

proof assume *surj,* show *False*

assume *a: surj f*

Example: Cantor's theorem

lemma *Cantor:* $\neg$ *surj(f :: 'a $\Rightarrow$ 'a set)*

proof assume *surj,* show *False*

assume *a: surj f*

from *a* **have** *b:* $\forall$ *A.* $\exists$ *a. A = f a*

Example: Cantor's theorem

lemma *Cantor:* $\neg$ *surj(f :: 'a ⇒ 'a set)*

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)*

Example: Cantor's theorem

lemma *Cantor:* $\neg$ *surj(f :: 'a $\Rightarrow$ 'a set)*
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*

Example: Cantor's theorem

lemma *Cantor:* $\neg$ *surj(f :: 'a ⇒ 'a set)*

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*

Example: Cantor's theorem

lemma *Cantor:* $\neg$ *surj(f :: 'a* $\Rightarrow$ *'a set)*

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*

Example: Cantor's theorem

lemma *Cantor:* $\neg$ *surj(f :: 'a $\Rightarrow$ 'a set)*
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*

Example: Cantor's theorem

```
lemma Cantor: ¬ surj(f :: 'a ⇒ 'a set)
proof   assume surj, show False
  assume a: surj f
  from a have b: ∀ A. ∃ a. A = f a
    by(simp add: surj_def)
  from b have c: ∃ a. {x. x ∉ f x} = f a
    by blast
  from c show False
    by blast
qed
```

Demo: this, then etc

Abbreviations

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

using

First the what, then the how:

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

using

First the what, then the how:

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

=

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

Example: Structured lemma statement

lemma *Cantor':*
fixes *f :: 'a* $\Rightarrow$ *'a set*
assumes *s: surj f*
shows *False*

Example: Structured lemma statement

lemma *Cantor':*
  fixes *f :: 'a* $\Rightarrow$ *'a set*
  assumes *s: surj f*
  shows *False*
proof -

Example: Structured lemma statement

lemma *Cantor':*
  fixes *f :: 'a ⇒ 'a set*
  assumes *s: surj f*
  shows *False*
proof - no automatic proof step

Example: Structured lemma statement

```
lemma Cantor':
  fixes f :: 'a ⇒ 'a set
  assumes s: surj f
  shows False
proof -    no automatic proof step
  have ∃ a. {x. x ∉ f x} = f a using s
    by(auto simp: surj_def)
```

Example: Structured lemma statement

```
lemma Cantor':
  fixes f :: 'a ⇒ 'a set
  assumes s: surj f
  shows False
proof -    no automatic proof step
  have ∃ a. {x. x ∉ f x} = f a using s
    by(auto simp: surj_def)
  thus False by blast
qed
```

Example: Structured lemma statement

lemma *Cantor':*
  fixes *f :: 'a* $\Rightarrow$ *'a set*
  assumes *s: surj f*
  shows *False*
proof - no automatic proof step
  have $\exists$ *a. {x. x* $\notin$ *f x} = f a* **using** *s*
    by*(auto simp: surj_def)*
  thus *False* **by** *blast*
qed

Proves *surj f* $\Longrightarrow$ *False*

Example: Structured lemma statement

```
lemma Cantor':
  fixes f :: 'a ⇒ 'a set
  assumes s: surj f
  shows False
proof -    no automatic proof step
  have ∃ a. {x. x ∉ f x} = f a using s
    by(auto simp: surj_def)
  thus False by blast
qed
```

Proves *surj f* $\Longrightarrow$ *False*

but *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** $y :: \tau_2$ **. . .**
assumes a: *P* **and** b: *Q* . . .
shows *R*

Structured lemma statements

fixes $x :: \tau_1$ **and** $y :: \tau_2$ . . .
assumes a: *P* **and** b: *Q* . . .
shows *R*

- **fixes** and **assumes** sections optional

Structured lemma statements

fixes $x :: \tau_1$ **and** $y :: \tau_2$ ...
assumes a: *P* **and** b: *Q* ...
shows *R*

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

Proof patterns

Propositional proof patterns

show $P \longleftrightarrow Q$
proof
  assume P
  ⋮
  show Q $\ldots$
next
  assume Q
  ⋮
  show P $\ldots$
qed

Propositional proof patterns

show $P \longleftrightarrow Q$
proof
  assume P
  ⋮
  show Q $\ldots$
next
  assume Q
  ⋮
  show P $\ldots$
qed

show $A = B$
proof
  show $A \subseteq B$ $\ldots$
next
  show $B \subseteq A$ $\ldots$
qed

Propositional proof patterns

show $P \longleftrightarrow Q$
proof
  assume P
  $\vdots$
  show Q $\ldots$
next
  assume Q
  $\vdots$
  show P $\ldots$
qed

show $A = B$
proof
  show $A \subseteq B$ $\ldots$
next
  show $B \subseteq A$ $\ldots$
qed

show $A \subseteq B$
proof
  fix x
  assume $x \in A$
  $\vdots$
  show $x \in B$ $\ldots$
qed

Propositional proof patterns

show R
proof *cases*
  assume P
  $\vdots$
  show R $\ldots$
next
  assume $\neg P$
  $\vdots$
  show R $\ldots$
qed

Case distinction

Propositional proof patterns

show R
proof *cases*
  assume P
  ⋮
  show R ...
next
  assume $\neg P$
  ⋮
  show R ...
qed

Case distinction

have $P \vee Q$ ...
then show R
proof
  assume P
  ⋮
  show R ...
next
  assume Q
  ⋮
  show R ...
qed

Case distinction

Propositional proof patterns

show R
proof *cases*
  assume P
  $\vdots$
  show R ...
next
  assume $\neg P$
  $\vdots$
  show R ...
qed

Case distinction

have $P \vee Q$ ...
then show R
proof
  assume P
  $\vdots$
  show R ...
next
  assume Q
  $\vdots$
  show R ...
qed

Case distinction

show P
proof *(rule ccontr)*
  assume $\neg P$
  $\vdots$
  show *False* ...
qed

Contradiction

Quantifier introduction proof patterns

show $\forall x.\ P(x)$
proof
  fix x *local fixed variable*
  show $P(x)$ . . .
qed

Quantifier introduction proof patterns

show $\forall x.\ P(x)$
proof
  fix x *local fixed variable*
  show $P(x)$ . . .
qed

show $\exists x.\ P(x)$
proof
  ⋮
  show $P(witness)$ . . .
qed

∃ *elimination:* **obtain**

∃ *elimination:* obtain

have $\exists x.\ P(x)$
then obtain x **where** p: $P(x)$ **by** blast
⋮ x local fixed variable

∃ *elimination:* **obtain**

have ∃*x. P(x)*
then **obtain** *x* **where** p: *P(x)* **by** blast
⋮ *x* local fixed variable

Works for one or more *x*

obtain *example*

lemma *Cantor'':* $\neg$ *surj(f :: 'a $\Rightarrow$ 'a set)*
proof
  assume *surj f*
  hence $\exists$ *a. {x. x $\notin$ f x} = f a* **by***(auto simp: surj_def)*

obtain ***example***

lemma *Cantor'':* $\neg$ *surj(f :: 'a $\Rightarrow$ 'a set)*

proof

assume *surj f*

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

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

obtain ***example***

```
lemma Cantor'': ¬ surj(f :: 'a ⇒ 'a set)
proof
  assume surj f
  hence ∃ a. {x. x ∉ f x} = f a  by(auto simp: surj_def)
  then obtain a where {x. x ∉ f x} = f a  by blast
  hence a ∉ f a ⟷ a ∈ f a  by blast
```

obtain *example*

lemma *Cantor'':* $\neg$ *surj(f :: 'a ⇒ 'a set)*
proof
  assume *surj f*
  hence $\exists$ *a. {x. x* $\notin$ *f x} = f a* **by**(*auto simp: surj_def)*
  then obtain *a* **where** *{x. x* $\notin$ *f x} = f a* **by** *blast*
  hence *a* $\notin$ *f a* $\longleftrightarrow$ *a* $\in$ *f a* **by** *blast*
  thus *False* **by** *blast*
qed

proof method

proof method

Applies method and generates subgoal(s):

1. $\bigwedge x_1 \ldots x_n \, [\![\, A_1; \ldots ; A_m \,]\!] \Longrightarrow A$

proof method

Applies method and generates subgoal(s):

1. $\bigwedge x_1 \ldots x_n \; [\![A_1; \ldots ; A_m]\!] \Longrightarrow A$

How to prove each subgoal:

proof method

Applies method and generates subgoal(s):

1. $\bigwedge x_1 \ldots x_n \; [\![A_1; \ldots ; A_m]\!] \Longrightarrow A$

How to prove each subgoal:

fix $x_1 \ldots x_n$
assume $A_1 \ldots A_m$
⋮
show A

proof method

Applies method and generates subgoal(s):

$1.\ \bigwedge x_1 \ldots x_n \ [\![A_1; \ldots ; A_m]\!] \Longrightarrow A$

How to prove each subgoal:

fix $x_1 \ldots x_n$
assume $A_1 \ldots A_m$
$\vdots$
show A

Separated by **next**

Demo: proof

Streamlining proofs: Pattern matching and Quotations

Example: pattern matching

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

Example: pattern matching

show $formula_1 \longleftrightarrow formula_2$ (**is** *?L* $\longleftrightarrow$ *?R*)
proof
  assume *?L*
  ⋮
  show *?R* . . .
next
  assume *?R*
  ⋮
  show *?L* . . .
qed

?thesis

show $formula$
proof -

    ⋮

    show *?thesis* ...

qed

?thesis

show $formula$ (is *?thesis*)
proof -
⋮
show *?thesis* . . .
qed

?thesis

show $formula$ (is *?thesis*)
proof -

⋮

show *?thesis* . . .

qed

Every show implicitly defines *?thesis*

Quoting facts by value

By name:

have x0: "*x* > *0*" . . .

⋮

from x0 . . .

Quoting facts by value

By name:

have x0: "*x > 0*" . . .

⋮

from x0 . . .

By value:

have "*x > 0*" . . .

⋮

from ‘*x>0*‘ . . .

Quoting facts by value

By name:

have x0: "$x > 0$" . . .

⋮

from x0 . . .

By value:

have "$x > 0$" . . .

⋮

from `$x>0$` . . .

back quotes

Demo: pattern matching and quotations

Advanced Isar

Overview

- Case distinction
- Induction
- Chains of (in)equations

Case distinction

Demo: case distinction

Datatype case distinction

datatype $t \;=\; C_1\ \vec{\tau} \;\mid\; \ldots$

Datatype case distinction

datatype $t = C_1\ \vec{\tau}\ \mid\ \ldots$

proof (*cases* $term$)

case ($C_1\ \vec{x}$)

$\ldots\ \vec{x}\ \ldots$

next

⋮

qed

Datatype case distinction

datatype t = $C_1\ \vec{\tau}\ \mid\ \ldots$

proof (cases $term$)
    case ($C_1\ \vec{x}$)
    $\cdots\ \vec{x}\ \cdots$
next
$\vdots$
qed

where **case** ($C_i\ \vec{x}$) $\equiv$

fix $\vec{x}$
assume $\underbrace{C_i\text{:}}_{\text{label}}\ \underbrace{term = (C_i\ \vec{x})}_{\text{formula}}$

Induction

Overview

- Structural induction
- Rule induction
- Induction with fun

Structural induction for type nat

```
show P(n)
proof (induct n)
   case 0
   . . .
   show ?case
next
   case (Suc n)
   . . .
   . . . n . . .
   show ?case
qed
```

Structural induction for type nat

show *P(n)*
proof *(induct n)*
  case *0* $\equiv$ **let** *?case* = $P(0)$
  . . .
  show *?case*
next
  case *(Suc n)*
  . . .
  $\cdots$ *n* $\cdots$
  show *?case*
qed

Structural induction for type 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(Suc n)*
  ... *n* ...
  show *?case*
qed

Demo: structural induction

Structural induction with $\Longrightarrow$

show $A(n) \Longrightarrow P(n)$

proof *(induct n)*

  case *0*

  ...

  show *?case*

next

  case *(Suc n)*

  ...

  $\cdots$ n $\cdots$

  ...

  show *?case*

qed

Structural induction with $\Longrightarrow$

show *A(n)* $\Longrightarrow$ *P(n)*

proof *(induct n)*

case *0* $\equiv$ **fix** *x* **assume** *0: A(0)*

. . . **let** *?case = P(0)*

show *?case*

next

case *(Suc n)*

. . .

$\cdots$ *n* $\cdots$

. . .

show *?case*

qed

Structural induction with $\Longrightarrow$

show $A(n) \Longrightarrow P(n)$		
proof *(induct n)*		
case *0*	$\equiv$	**fix** *x* **assume** *0:* $A(0)$
. . .		**let** *?case* = $P(0)$
show *?case*		
next		
case *(Suc n)*	$\equiv$	**fix** *n*
. . .		**assume** *Suc:* $A(n) \Longrightarrow P(n)$
$\cdots$ *n* $\cdots$		$A(Suc\ n)$
. . .		**let** *?case* = $P(Suc\ n)$
show *?case*		
qed		

A remark on style

- **case** *(Suc n)* . . . **show** *?case*
  is easy to write and maintain

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*)

Demo: structural induction with $\Longrightarrow$

Rule induction

Inductive definition

inductive_set *S*

intros

rule$_1$: $[\![$ *s* $\in$ *S; A* $]\!] \Longrightarrow$ *s'* $\in$ *S*

$\vdots$

rule$_n$: $\ldots$

Rule induction

show $x \in S \Longrightarrow P(x)$

proof *(induct rule: S.induct)*

  case $rule_1$

  . . .

  show *?case*

next

⋮

next

  case $rule_n$

  . . .

  show *?case*

qed

Implicit selection of induction rule

assume *A*: $x \in S$

⋮

show *P(x)*

using *A* **proof** *induct*

⋮

qed

Implicit selection of induction rule

assume *A*: $x \in S$

⋮

show *P(x)*

using *A* **proof** *induct*

⋮

qed

lemma assumes *A*: $x \in S$ **shows** *P(x)*

using *A* **proof** *induct*

⋮

qed

Renaming free variables in rule

case *(rule*$_i$ *x*$_1$ $\ldots$ *x*$_k$*)*

Renames the (alphabetically!) first k variables in *rule*$_i$ to *x*$_1$ $\ldots$ *x*$_k$.

Demo: rule induction

Induction with fun

Definition:

fun *f*

⋮

Induction with fun

Definition:

fun *f*

⋮

Proof:

show . . . *f(. . .)* . . .

proof *(induct* x_1 . . . x_k *rule: f.induct)*

Induction with fun

Definition:

fun *f*

⋮

Proof:

show . . . *f*(. . .) . . .

proof *(induct* x_1 . . . x_k *rule: f.induct)*

    case *1*

    ⋮

Induction with fun

Definition:

fun *f*

⋮

Proof:

show $\ldots$ *f(*$\ldots$*)* $\ldots$

proof *(induct x*$_1$ $\ldots$ *x*$_k$ *rule: f.induct)*

  case *1*

  ⋮

Case i refers to equation i in the definition of *f*

Induction with fun

Definition:

fun *f*

⋮

Proof:

show $\ldots$ *f(*$\ldots$*)* $\ldots$

proof *(induct x*$_1$ $\ldots$ *x*$_k$ *rule: f.induct)*

  case *1*

  ⋮

Case i refers to equation i in the definition of *f*
More precisely: to equation i in *f.simps*

Demo: induction with fun

Chains of (in)equations

also

have "t_0 = t_1" . . .

also

have "t_0 = t_1" $\ldots$

also

have "$\ldots$ = t_2" $\ldots$

also

have "t_0 = t_1" $\ldots$

also

have "$\ldots$ = t_2" $\ldots$ $\qquad\qquad$ $\ldots \equiv t_1$

also

have "$t_0 = t_1$" ...

also

have "$\ldots = t_2$" ... $\ldots \equiv t_1$

also

⋮

also

have "$\ldots = t_n$" ...

also

have "$t_0 = t_1$" ...

also

have "... $= t_2$" ... $\quad\quad$ ... $\equiv t_1$

also

$\vdots$

also

have "... $= t_n$" ... $\quad\quad$ ... $\equiv t_{n-1}$

also

have "t_0 = t_1" ...

also

have "... = t_2" ... $\quad\quad$ ... $\equiv t_1$

also

⋮

also

have "... = t_n" ... $\quad\quad$ ... $\equiv t_{n-1}$

finally show ...

also

have "$t_0 = t_1$" ...

also

have "... $= t_2$" ... $\qquad$ ... $\equiv t_1$

also

$\vdots$

also

have "... $= t_n$" ... $\qquad$ ... $\equiv t_{n-1}$

finally show ...

— like **from** ‘$t_0 = t_n$‘ **show**

also

- "..." is merely an abbreviation

also

- "$\ldots$" is merely an abbreviation
- **also** works for other transitive relations ($<$, $\leq$, $\ldots$)

Demo: also

Accumulating facts

moreover

have $formula_1$ . . .

moreover

have $formula_1$ . . .
moreover
have $formula_2$ . . .

moreover

have $formula_1$ ...
moreover
have $formula_2$ ...
moreover
⋮
moreover
have $formula_n$ ...

moreover

have $formula_1$ . . .
moreover
have $formula_2$ . . .
moreover
⋮
moreover
have $formula_n$ . . .
ultimately show . . .

moreover

have $formula_1$ **. . .**
moreover
have $formula_2$ **. . .**
moreover
$\vdots$
moreover
have $formula_n$ **. . .**
ultimately show . . .
— like **from** $f_1 \ldots f_n$ **show** but needs no labels

Demo: moreover