
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** ...

qed

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 $formula_0 \implies formula_{n+1}$

Overview

- Basic Isar
- Propositional logic
- Predicate logic

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 (\wedge)
| **assume** proposition (\implies)
| [**from** thm⁺] (**have** | **show**) proposition **proof**

Isar core syntax

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

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

statement = **fix** variables (\wedge)
| **assume** proposition (\implies)
| [**from** thm⁺] (**have** | **show**) proposition proof
| **next** (separates subgoals)

Isar core syntax

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

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

statement = **fix** variables (\wedge)
| **assume** proposition (\implies)
| [**from** thm⁺] (**have** | **show**) proposition proof
| **next** (separates subgoals)

proposition = [name:] "formula"

Isar core syntax

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

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

statement = **fix** variables (\wedge)
| **assume** proposition (\implies)
| [**from** thm⁺] (**have** | **show**) proposition proof
| **next** (separates subgoals)

proposition = [name:] "formula"

thm = name | name[**OF** thm⁺] | 'formula'

Demo: propositional logic, introduction rules

Basic atomic proofs

by *method*

prove proposition by applying *method*

Basic atomic proofs

by *method*

prove proposition by applying *method*

Basic proof method:

(rule r)

Basic atomic proofs

by *method*

prove proposition by applying *method*

Basic proof method:

(rule r)

If no r is given:

rule = *(rule r)*

where r is the first matching standard elim/intro rule.

Demo: propositional logic, forward reasoning

Forward reasoning with from

from *thms* **have** "*formula*" *proof*

pipes/feeds *thms* **into** *proof*

Forward reasoning with from

from *thms* **have** "*formula*" *proof*
pipes/feeds *thms* into *proof*

Piping *thms* into method *rule*:

from *thms* **have** "*formula*" **by**(*rule* *r*)
= have "*formula*" **by**(*rule* *r*[OF *thms*])

Forward reasoning with from

from *thms* **have** "*formula*" *proof*

pipes/feeds *thms* into *proof*

Piping *thms* into method *rule*:

from *thms* **have** "*formula*" **by**(*rule* *r*)

= **have** "*formula*" **by**(*rule* *r*[OF *thms*])

Most other methods (*simp/auto/blast/...*) treat *thms* as additional assumptions.

Forward reasoning with `from`

`from thms have "formula" proof`
pipes/feeds *thms* into *proof*

Piping *thms* into method *rule*:

`from thms have "formula" by(rule r)`
`= have "formula" by(rule r[OF thms])`

Most other methods (*simp/auto/blast/...*) treat *thms* as additional assumptions.

Abbreviation:

`..` = `by rule`

Demo: propositional logic, elimination rules

proof . . . qed

Explicit rule application:

- **have** *formula* **proof(rule r) . . . qed**

proof ... qed

Explicit rule application:

- **have** *formula* **proof(rule *r*) ... qed**
applies *r* to *formula*; leaves premises of *r* as subgoals;

proof ... qed

Explicit rule application:

- **have** *formula* **proof(rule *r*) ... qed**
applies *r* to *formula*; leaves premises of *r* as subgoals;
their proofs (...) are separated by **next**.

proof ... qed

Explicit rule application:

- **have** *formula* **proof(rule *r*) ... qed**
applies *r* to *formula*; leaves premises of *r* as subgoals;
their proofs (...) are separated by **next**.
- **from** *thms* **have** *formula* **proof(rule *r*)**
= **have** *formula* **proof(rule *r*[OF *thms*])**

proof ... qed

Explicit rule application:

- **have** *formula* **proof(rule** *r*) ... **qed**
applies *r* to *formula*; leaves premises of *r* as subgoals;
their proofs (...) are separated by **next**.
- **from** *thms* **have** *formula* **proof(rule** *r*)
= **have** *formula* **proof(rule** *r*[OF *thms*])

Implicit rule selection:

- Elim rules are triggered by *thms* fed into a proof:
from *thms* **have** *formula* **proof**

proof ... qed

Explicit rule application:

- **have** *formula* **proof**(*rule* *r*) ... **qed**
applies *r* to *formula*; leaves premises of *r* as subgoals;
their proofs (...) are separated by **next**.
- **from** *thms* **have** *formula* **proof**(*rule* *r*)
= **have** *formula* **proof**(*rule* *r*[OF *thms*])

Implicit rule selection:

- Elim rules are triggered by *thms* fed into a proof:
from *thms* **have** *formula* **proof**
- **proof** alone abbreviates **proof** *rule*

proof ... qed

Explicit rule application:

- **have** *formula* **proof**(*rule* *r*) ... **qed**
applies *r* to *formula*; leaves premises of *r* as subgoals;
their proofs (...) are separated by **next**.
- **from** *thms* **have** *formula* **proof**(*rule* *r*)
= **have** *formula* **proof**(*rule* *r*[OF *thms*])

Implicit rule selection:

- Elim rules are triggered by *thms* fed into a proof:
from *thms* **have** *formula* **proof**
- **proof** alone abbreviates **proof** *rule*
- *rule*: automatic selection of elim/intro rule based on
thms and *formula*

Abbreviations

| | | |
|----------------|---|--|
| <i>this</i> | = | the previous proposition proved or assumed |
| then | = | from <i>this</i> |
| thus | = | then show |
| hence | = | then have |
| with \vec{a} | = | from \vec{a} <i>this</i> |

using

First the what, then the how:

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

using

First the what, then the how:

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

using

First the what, then the how:

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

Can be mixed:

from major-facts (have|show) proposition **using** minor-facts

using

First the what, then the how:

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

Can be mixed:

from major-facts (have|show) proposition **using** minor-facts
=
from major-facts minor-facts (have|show) proposition

Demo: avoiding duplication

Schematic term variables

?A

Schematic term variables

$?A$

- Defined by pattern matching:

$$x = 0 \wedge y = 1 \text{ (is } ?A \wedge _)$$

Schematic term variables

$?A$

- Defined by pattern matching:

$$x = 0 \wedge y = 1 \text{ (is } ?A \wedge _)$$

- Predefined: *?thesis*
The last enclosing **show** formula

Demo: predicate calculus

obtain

Syntax:

obtain variables where proposition proof

Mixing proof styles

from . . .

have . . .

apply - make incoming facts assumptions

apply(...)

⋮

apply(...)

done

Advanced Isar

Overview

- Case distinction
- Induction
- Computational reasoning

Case distinction

Boolean case distinction

proof cases

assume *formula*

⋮

next

assume \neg *formula*

⋮

qed

Boolean case distinction

proof cases

assume *formula*

⋮

next

assume \neg *formula*

⋮

qed

proof (*cases formula*)

case *True*

⋮

next

case *False*

⋮

qed

Boolean case distinction

proof cases

assume *formula*

⋮

next

assume \neg *formula*

⋮

qed

proof (*cases formula*)

case *True*

⋮

next

case *False*

⋮

qed

case *True* \equiv

assume *True*: *formula*

Demo: case distinction

Datatype case distinction

```
proof (cases term)  
  case Constructor1  
  ∴  
next  
∴  
next  
  case (Constructork  $\vec{x}$ )  
  ...  $\vec{x}$  ...  
qed
```

Datatype case distinction

proof (*cases term*)
 case *Constructor*₁
 ⋮

next

⋮

next

case (*Constructor*_{*k*} \vec{x})

 ... \vec{x} ...

qed

case (*Constructor*_{*i*} \vec{x}) ≡

fix \vec{x} **assume** *Constructor*_{*i*}: *term* = (*Constructor*_{*i*} \vec{x})

Induction

Overview

- Structural induction
- Rule induction
- Induction with fun

Structural induction for type *nat*

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

Structural induction for type *nat*

show $P(n)$

proof (*induction n*)

case 0 \equiv **let** $?case = P(0)$

...

show $?case$

next

case (*Suc n*)

...

... n ...

show $?case$

qed

Structural induction for type *nat*

show $P(n)$

proof (*induction n*)

case 0 \equiv **let** $?case = P(0)$

...

show $?case$

next

case ($Suc\ n$) \equiv **fix** n **assume** $Suc: P(n)$

...

... n ...

show $?case$

qed

Demo: structural induction

Structural induction with \implies and \wedge

show $\wedge x. A(n) \implies P(n)$

proof (*induction n*)

case 0

...

show ?case

next

case (*Suc n*)

...

... *n* ...

...

show ?case

qed

Structural induction with \implies and \wedge

show $\wedge x. A(n) \implies P(n)$

proof (*induction n*)

case 0

...

show ?case

next

case (*Suc n*)

...

... *n* ...

...

show ?case

qed

\equiv **fix X assume 0: A(0)**
let ?case = P(0)

Structural induction with \implies and \wedge

show $\wedge x. A(n) \implies P(n)$

proof (*induction 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 x*

assume *Suc*: $\wedge x. A(n) \implies P(n)$

A(*Suc n*)

let ?case = *P*(*Suc n*)

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 \implies and \wedge

Rule induction

Inductive definition

inductive_set S

intros

$rule_1: \llbracket s \in S; A \rrbracket \implies s' \in S$

\vdots

$rule_n: \dots$

Rule induction

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

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 $(rule_i x_1 \dots x_k)$

Renames the (alphabetically!) first k variables in $rule_i$ to $X_1 \dots X_k$.

Demo: rule induction

Induction with fun

Definition:

fun f

⋮

Induction with fun

Definition:

fun f

⋮

Proof:

show ... $f(\dots)$...

proof (*induction* $x_1 \dots x_k$ *rule: f.induct*)

Induction with fun

Definition:

fun f

⋮

Proof:

show ... $f(\dots)$...

proof (*induction* $x_1 \dots x_k$ *rule: f.induct*)

case *1*

⋮

Induction with fun

Definition:

fun f

⋮

Proof:

show ... $f(\dots)$...

proof (*induction* $x_1 \dots x_k$ *rule: f.induct*)

case 1

⋮

Case i refers to equation i in the definition of f

Induction with fun

Definition:

fun f

⋮

Proof:

show ... $f(\dots)$...

proof (*induction* $x_1 \dots x_k$ *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

Computational Reasoning

Overview

- Accumulating facts
- Chains of equations and inequations

moreover

have *formula*₁ . . .

moreover

have *formula*₂ . . .

moreover

⋮

moreover

have *formula*_{*n*} . . .

ultimately show . . .

— pipes facts *formula*₁ . . . *formula*_{*n*} into the proof

proof

⋮

also

have $t_0 = t_1 \dots$

also

have $\dots = t_2 \dots$

also

⋮

also

have $\dots = t_n \dots$

...

“...” is merely an abbreviation

Demo: moreover and also

Variations on also

Transitivity:

have $t_0 = t_1 \dots$

also have $\dots = t_2 \dots$

also/finally \rightsquigarrow

Variations on also

Transitivity:

have $t_0 = t_1 \dots$

also have $\dots = t_2 \dots$

also/finally $\rightsquigarrow t_0 = t_2$

Variations on also

Transitivity:

have $t_0 = t_1 \dots$

also have $\dots = t_2 \dots$

also/finally $\rightsquigarrow t_0 = t_2$

Substitution:

have $P(s) \dots$

also have $s = t \dots$

also/finally \rightsquigarrow

Variations on also

Transitivity:

have $t_0 = t_1 \dots$

also have $\dots = t_2 \dots$

also/finally $\rightsquigarrow t_0 = t_2$

Substitution:

have $P(s) \dots$

also have $s = t \dots$

also/finally $\rightsquigarrow P(t)$

From = to \leq and $<$

Transitivity:

have $t_0 \leq t_1 \dots$

also have $\dots \leq t_2 \dots$

also/finally \rightsquigarrow

From = to \leq and $<$

Transitivity:

have $t_0 \leq t_1 \dots$

also have $\dots \leq t_2 \dots$

also/finally $\rightsquigarrow t_0 \leq t_2$

From = to \leq and $<$

Transitivity:

have $t_0 \leq t_1$

also have $\dots \leq t_2$

also/finally $\rightsquigarrow t_0 \leq t_2$

Substitution:

have $r \leq f(s)$

also have $s < t$

also/finally \rightsquigarrow

From = to \leq and $<$

Transitivity:

have $t_0 \leq t_1$

also have $\dots \leq t_2$

also/finally $\rightsquigarrow t_0 \leq t_2$

Substitution:

have $r \leq f(s)$

also have $s < t$

also/finally $\rightsquigarrow (\bigwedge x. x < y \implies f(x) < f(y)) \implies r < f(t)$

From = to \leq and $<$

Transitivity:

have $t_0 \leq t_1 \dots$

also have $\dots \leq t_2 \dots$

also/finally $\rightsquigarrow t_0 \leq t_2$

Substitution:

have $r \leq f(s) \dots$

also have $s < t \dots$

also/finally $\rightsquigarrow (\bigwedge x. x < y \implies f(x) < f(y)) \implies r < f(t)$

Similar for all other combinations of =, \leq and $<$.

All about also

To view all combinations in Proof General:

Isabelle/Isar → Show me → Transitivity rules

Demo: monotonicity reasoning