A Compact Introduction to Isabelle/HOL

Tobias Nipkow

TU München

Overview

Overview of Isabelle/HOL

System Architecture

ProofGeneral	(X)Emacs based interface
Isabelle/HOL	Isabelle instance for HOL
Isabelle	generic theorem prover
Standard ML	implementation language

HOL

HOL = Higher-Order Logic

HOL = Functional programming + Logic

HOL has

- datatypes
- recursive functions
- logical operators ($\wedge$, $\longrightarrow$, $\forall$, $\exists$, ...)

HOL is a programming language!

Higher-order = functions are values, too!

Formulae

Syntax (in decreasing priority):

$$\begin{array}{lllll} form & ::= & (form) & \mid\ term = term & \mid\ \neg form \\ & \mid & form \wedge form & \mid\ form \vee form & \mid\ form \longrightarrow form \\ & \mid & \forall x.\ form & \mid\ \exists x.\ form & \end{array}$$

Scope of quantifiers: as far to the right as possible

Examples

- $\neg A \wedge B \vee C \equiv ((\neg A) \wedge B) \vee C$
- $A = B \wedge C \equiv (A = B) \wedge C$
- $\forall x.\ P\ x \wedge Q\ x \equiv \forall x.\ (P\ x \wedge Q\ x)$
- $\forall x.\ \exists y.\ P\ x\ y \wedge Q\ x \equiv \forall x.\ (\exists y.\ (P\ x\ y \wedge Q\ x))$

Types and Terms

Types

Syntax:

τ ::=	(τ)	
$\mid$	*bool* $\mid$ *nat* $\mid \dots$	base types
$\mid$	*'a* $\mid$ *'b* $\mid \dots$	type variables
$\mid$	$\tau \Rightarrow \tau$	total functions
$\mid$	$\tau \times \tau$	pairs (ascii: *)
$\mid$	τ *list*	lists
$\mid$	...	user-defined types

Parentheses: $T1 \Rightarrow T2 \Rightarrow T3 \equiv T1 \Rightarrow (T2 \Rightarrow T3)$

Terms: Basic syntax

Syntax:

$term$	$::=$	$(term)$	
	$\mid$	a	constant or variable (identifier)
	$\mid$	$term\ term$	function application
	$\mid$	$\lambda x.\ term$	function “abstraction”
	$\mid$	$\ldots$	lots of syntactic sugar

Examples: *f (g x) y*   *h* (λ*x. f (g x))*

Parantheses: $f\ a_1\ a_2\ a_3 \equiv ((f\ a_1)\ a_2)\ a_3$

Terms and Types

Terms must be well-typed

(the argument of every function call must be of the right type)

Notation: t **::** τ means t is a well-typed term of type τ.

Type inference

Isabelle automatically computes (“*infers*”) the type of each variable in a term.

In the presence of *overloaded* functions (functions with multiple types) not always possible.

User can help with type annotations inside the term.

Example: *f (x::nat)*

Currying

Thou shalt curry your functions

- Curried: $f :: \tau_1 \Rightarrow \tau_2 \Rightarrow \tau$
- Tupled: $f' :: \tau_1 \times \tau_2 \Rightarrow \tau$

Advantage: *partial application* $f\ a_1$ with $a_1 :: \tau_1$

Terms: Syntactic sugar

Some predefined syntactic sugar:

- *Infix:* +, -, *, #, @, . . .
- *Mixfix:* *if _ then _ else _*, *case _ of*, . . .

Prefix binds more strongly than infix:

! $f\ x + y \equiv (f\ x) + y \not\equiv f\ (x + y)$ **!**

Base types: bool, nat, list

Type bool

Formulae = terms of type *bool*

True :: bool
False :: bool
$\wedge$, $\vee$, . . . :: *bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool*
⋮

if-and-only-if: =

Type nat

0 :: nat
Suc :: nat $\Rightarrow$ *nat*
+, *, ... :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat*
⋮

! Numbers and arithmetic operations are overloaded:
0,1,2,... :: 'a, + :: *'a* $\Rightarrow$ *'a* $\Rightarrow$ *'a*

You need type annotations: *1 :: nat, x + (y::nat)*

. . . unless the context is unambiguous: *Suc z*

Type list

- *[]*: empty list
- *x # xs*: list with first element *x* ("*head*") and rest *xs* ("*tail*")
- Syntactic sugar: $[x_1, \ldots, x_n]$

Large library:
hd, *tl*, *map*, *size*, *filter*, *set*, *nth*, *take*, *drop*, *distinct*, . . .

Don't reinvent, reuse!
$\leadsto$ `HOL/List.thy`

Isabelle Theories

Theory = Module

Syntax:

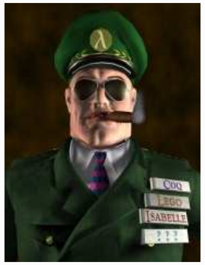
theory $MyTh$ = $ImpTh_1$ + . . . + $ImpTh_n$:
(declarations, definitions, theorems, proofs, ...)*
end

- $MyTh$: name of theory. Must live in file $MyTh$`.thy`
- $ImpTh_i$: name of *imported* theories. Import transitive.

Unless you need something special:

theory $MyTh$ = Main:

Proof General



An Isabelle Interface

by David Aspinall

Proof General

Customized version of (x)emacs:

- all of emacs (info: C-h i)
- Isabelle aware (when editing .thy files)
- mathematical symbols (“x-symbols”)

Interaction:

- via mouse
- or keyboard (key bindings see C-h m)

X-Symbols

Input of funny symbols in Proof General

- via menu (“X-Symbol”)
- via ascii encoding (similar to LaTeX): \<and>, \<or>, ...
- via abbreviation: /\, \/, -->, ...

x-symbol	∀	∃	λ	¬	∧	∨	⟶	⇒
ascii (1)	\<forall>	\<exists>	\<lambda>	\<not>	/\	\/	-->	=>
ascii (2)	ALL	EX	%	~	&	|		

(1) is converted to x-symbol, (2) stays ascii.

Demo: terms and types

An introduction to recursion and induction

A recursive datatype: toy lists

datatype *'a list = Nil | Cons 'a "'a list"*

***Nil*:** empty list

***Cons x xs*:** head *x :: 'a*, tail *xs :: 'a list*

A toy list: *Cons False (Cons True Nil)*

Predefined lists: *[False, True]*

Concrete syntax

In `.thy` files:
Types and formulae need to be inclosed in *"..."*

Except for single identifiers, e.g. *'a*

"..." normally not shown on slides

Structural induction on lists

P xs holds for all lists *xs* if

- *P Nil*
- and for arbitrary *x* and *xs*, *P xs* implies *P (Cons x xs)*

Demo: append and reverse

Proofs

General schema:

lemma $name$: `"..."`
apply `(...)`
apply `(...)`
⋮
done

If the lemma is suitable as a simplification rule:

lemma $name$`[simp]: "..."`

Proof methods

- Structural induction
  - Format: *(induct x)*
    x must be a free variable in the first subgoal.
    The type of x must be a datatype.
  - Effect: generates 1 new subgoal per constructor
- Simplification and a bit of logic
  - Format: *auto*
  - Effect: tries to solve as many subgoals as possible using simplification and basic logical reasoning.

The proof state

1. $\bigwedge x_1 \ldots x_p. \llbracket A_1; \ldots ; A_n \rrbracket \Longrightarrow B$

$x_1 \ldots x_p$	Local constants
$A_1 \ldots A_n$	Local assumptions
B	Actual (sub)goal

Notation

$$\llbracket A_1; \ldots ; A_n \rrbracket \Longrightarrow B$$

abbreviates

$$A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$$

$;$ $\approx$ “and”

Type and function definition in Isabelle/HOL

Datatype definition in Isabelle/HOL

The example

datatype *'a list = Nil | Cons 'a "'a list"*

Properties:

- Types: *Nil :: 'a list*
  Cons :: 'a ⇒ 'a list ⇒ 'a list
- Distinctness: *Nil ≠ Cons x xs*
- Injectivity: *(Cons x xs = Cons y ys) = (x = y ∧ xs = ys)*

The general case

datatype $(\alpha_1, \ldots, \alpha_n)\tau \;=\; C_1\ \tau_{1,1} \ldots \tau_{1,n_1}$
$\mid\ \ldots$
$\mid\ C_k\ \tau_{k,1} \ldots \tau_{k,n_k}$

- *Types:* $C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\tau$
- *Distinctness:* $C_i\ \ldots \neq C_j\ \ldots$ if $i \neq j$
- *Injectivity:*
  $(C_i\ x_1 \ldots x_{n_i} = C_i\ y_1 \ldots y_{n_i}) = (x_1 = y_1 \wedge \ldots \wedge x_{n_i} = y_{n_i})$

Distinctness and Injectivity are applied automatically
Induction must be applied explicitly

case

Every datatype introduces a *case* construct, e.g.

(case xs of [] ⇒ . . . | y#ys ⇒ ... y ... ys ...)

In general: one case per constructor

Same order of cases as in datatype

No nested patterns (e.g. *x#y#zs*)
But nested cases

Needs *( )* in context

Case distinctions

apply(*case_tac* t)

creates k subgoals

$$t = C_i\ x_1 \dots x_p \Longrightarrow \dots$$

one for each constructor C_i.

Function definition in Isabelle/HOL

Why nontermination can be harmful

How about *f x = f x + 1* ?

Subtract *f x* on both sides.

$\Longrightarrow$ *0 = 1*

! All functions in HOL must be total **!**

Function definition schemas in Isabelle/HOL

- Non-recursive with **defs/constdefs**
  No problem
- Primitive-recursive with **primrec**
  Terminating by construction
- Well-founded recursion with **recdef**
  User must (help to) prove termination
  ($\leadsto$ later)

primrec

The example

primrec
"app Nil ys = ys"
"app (Cons x xs) ys = Cons x (app xs ys)"

The general case

If τ is a datatype (with constructors $C_1, \ldots, C_k$) then
$f :: \cdots \Rightarrow \tau \Rightarrow \cdots \Rightarrow \tau'$ can be defined by *primitive recursion*:

$$f\ x_1 \ldots (C_1\ y_{1,1} \ldots y_{1,n_1}) \ldots x_p \;=\; r_1$$
$$\vdots$$
$$f\ x_1 \ldots (C_k\ y_{k,1} \ldots y_{k,n_k}) \ldots x_p \;=\; r_k$$

The recursive calls in r_i must be *structurally smaller*,
i.e. of the form $f\ a_1 \ldots y_{i,j} \ldots a_p$

nat is a datatype

datatype *nat = 0 | Suc nat*

Functions on *nat* definable by primrec!

primrec
f 0 = ...
f(Suc n) = ... f n ...

Demo: trees

Proof by Simplification

Term rewriting foundations

Term rewriting means . . .

Using equations $l = r$ from left to right

As long as possible

Terminology: equation $\rightsquigarrow$ *rewrite rule*

An example

Equations:

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

Rewriting:

$$\begin{array}{rcll} 0 + Suc\ 0 & \leq & Suc\ 0 + x & \stackrel{(1)}{=} \\ Suc\ 0 & \leq & Suc\ 0 + x & \stackrel{(2)}{=} \\ Suc\ 0 & \leq & Suc\ (0+x) & \stackrel{(3)}{=} \\ 0 & \leq & 0 + x & \stackrel{(4)}{=} \\ & True & & \end{array}$$

Interlude: Variables in Isabelle

Schematic variables

Three kinds of variables:

- bound: ∀*x. x* = *x*
- free: *x* = *x*
- schematic: *?x* = *?x* (“unknown”)

Can be mixed: ∀*b. f ?a y* = *b*

- Logically: free = schematic
- Operationally:
  - free variables are fixed
  - schematic variables are instantiated by substitutions (e.g. during rewriting)

From x to ?x

State lemmas with free variables:

lemma *app_Nil2[simp]: "xs @ [] = xs"*

⋮

done

After the proof: Isabelle changes *xs* to *?xs* (internally):

?xs @ [] = ?xs

Now usable with arbitrary values for *?xs*

Term rewriting in Isabelle

Basic simplification

Goal: $1.\ [\![P_1; \ldots ; P_m]\!] \Longrightarrow C$

apply(*simp add:* $eq_1 \ldots eq_n$)

Simplify $P_1 \ldots P_m$ and C using

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

auto versus simp

- *auto* acts on all subgoals
- *simp* acts only on subgoal 1
- *auto* applies *simp* and more

Termination

Simplification may not terminate.
Isabelle uses *simp*-rules (almost) blindly from left to right.

Conditional *simp*-rules are only applied
if conditions are provable.

Demo: simp

Induction heuristics

Basic heuristics

Theorems about recursive functions are proved by induction

Induction on argument number i of f
if f is defined by recursion on argument number i

A tail recursive reverse

consts *itrev* :: *'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list*
primrec
itrev [] *ys = ys*
itrev (x#xs) *ys = itrev xs (x#ys)*

lemma *itrev xs [] = rev xs*

Why in this direction?

Because the lhs is “more complex” than the rhs.

Demo: first proof attempt

Generalisation (1)

Replace constants by variables

lemma *itrev xs ys = rev xs @ ys*

Demo: second proof attempt

Generalisation (2)

Quantify free variables by $\forall$
(except the induction variable)

lemma $\forall$ *ys. itrev xs ys = rev xs @ ys*

HOL: Propositional Logic

Overview

- Natural deduction
- Rule application in Isabelle/HOL

Rule notation

$\frac{A_1 \ldots A_n}{A}$ instead of $[\![A_1 \ldots A_n]\!] \Longrightarrow A$

Natural Deduction

Natural deduction

Two kinds of rules for each logical operator $\oplus$:

Introduction: how can I prove $A \oplus B$?

Elimination: what can I prove from $A \oplus B$?

Natural deduction for propositional logic

$$\frac{A \quad B}{A \wedge B}\ \texttt{conjI} \qquad \frac{A \wedge B \quad [\![A;B]\!] \Longrightarrow C}{C}\ \texttt{conjE}$$

$$\frac{A}{A \vee B}\ \frac{B}{A \vee B}\ \texttt{disjI1/2} \qquad \frac{A \vee B \quad A \Longrightarrow C \quad B \Longrightarrow C}{C}\ \texttt{disjE}$$

$$\frac{A \Longrightarrow B}{A \longrightarrow B}\ \texttt{impI} \qquad \frac{A \longrightarrow B \quad A \quad B \Longrightarrow C}{C}\ \texttt{impE}$$

$$\frac{A \Longrightarrow B \quad B \Longrightarrow A}{A = B}\ \texttt{iffI} \qquad \frac{A{=}B}{A \Longrightarrow B}\ \texttt{iffD1} \quad \frac{A{=}B}{B \Longrightarrow A}\ \texttt{iffD2}$$

$$\frac{A \Longrightarrow False}{\neg A}\ \texttt{notI} \qquad \frac{\neg A \quad A}{C}\ \texttt{notE}$$

Operational reading

$$\frac{A_1 \ldots A_n}{A}$$

Introduction rule:

To prove A it suffices to prove $A_1 \ldots A_n$.

Elimination rule

If I know A_1 and want to prove A
it suffices to prove $A_2 \ldots A_n$.

Classical contradiction rules

$$\frac{\neg A \Longrightarrow False}{A}\ \texttt{ccontr} \qquad \frac{\neg A \Longrightarrow A}{A}\ \texttt{classical}$$

Proof by assumption

$$\frac{A_1 \quad \ldots \quad A_n}{A_i}\ \texttt{assumption}$$

Rule application: the rough idea

Applying rule $[\![A_1; \ldots ; A_n]\!] \Longrightarrow A$ to subgoal C:

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

Working backwards, like in Prolog!

Example: rule: $[\![?P; ?Q]\!] \Longrightarrow ?P \wedge ?Q$
subgoal: 1. $A \wedge B$

Result: 1. A
2. B

Rule application: the details

Rule:	$[\![A_1; \ldots ; A_n]\!] \Longrightarrow A$
Subgoal:	1. $[\![B_1; \ldots ; B_m]\!] \Longrightarrow C$
Substitution:	$\sigma(A) \equiv \sigma(C)$
New subgoals:	1. $\sigma([\![B_1; \ldots ; B_m]\!] \Longrightarrow A_1)$
	$\vdots$
	n. $\sigma([\![B_1; \ldots ; B_m]\!] \Longrightarrow A_n)$

Command:

apply(*rule <rulename>*)

Proof by assumption

apply *assumption*

proves

1. $[\![B_1; \ldots ; B_m]\!] \Longrightarrow C$

by unifying *C* with one of the B_i (backtracking!)

Demo: application of introduction rule

Applying elimination rules

apply(*erule <elim-rule>*)

Like *rule* but also

- unifies first premise of rule with an assumption
- eliminates that assumption

Example:

Rule:	$[\![?P \wedge ?Q; [\![?P; ?Q]\!] \Longrightarrow ?R]\!] \Longrightarrow ?R$
Subgoal:	*1.* $[\![X; A \wedge B; Y]\!] \Longrightarrow Z$
Unification:	$?P \wedge ?Q \equiv A \wedge B$ and $?R \equiv Z$
New subgoal:	*1.* $[\![X; Y]\!] \Longrightarrow [\![A; B]\!] \Longrightarrow Z$
same as:	*1.* $[\![X; Y; A; B]\!] \Longrightarrow Z$

How to prove it by natural deduction

- Intro rules decompose formulae to the right of $\Longrightarrow$.

  apply(*rule <intro-rule>*)

- Elim rules decompose formulae on the left of $\Longrightarrow$.

  apply(*erule <elim-rule>*)

Demo: examples

$\Longrightarrow$ vs $\longrightarrow$

- Write theorems as $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$
  not as $A_1 \wedge \ldots \wedge A_n \longrightarrow A$ (to ease application)
- *Exception* (in **apply**-style): induction variable must not occur in the premises.

  Example: $[\![A; B(x)]\!] \Longrightarrow C(x) \leadsto A \Longrightarrow B(x) \longrightarrow C(x)$

  Reverse transformation (after proof):
  lemma *abc[rule_format]: A* $\Longrightarrow B(x) \longrightarrow C(x)$

Demo: further techniques

HOL: Predicate Logic

Parameters

Subgoal:

1. $\bigwedge x_1 \dots x_n$. *Formula*

The x_i are called parameters of the subgoal.
Intuition: local constants, i.e. arbitrary but fixed values.

Rules are automatically lifted over $\bigwedge x_1 \dots x_n$ and applied directly to *Formula*.

Scope

- Scope of parameters: whole subgoal
- Scope of $\forall$, $\exists$, . . .: ends with *;* or $\Longrightarrow$

$$\bigwedge x\ y.\ [\![\forall y.\ P\ y \longrightarrow Q\ z\ y;\ \ Q\ x\ y]\!] \Longrightarrow \exists x.\ Q\ x\ y$$

means

$$\bigwedge x\ y.\ [\![(\forall y_1.\ P\ y_1 \longrightarrow Q\ z\ y_1);\ \ Q\ x\ y]\!] \Longrightarrow \exists x_1.\ Q\ x_1\ y$$

α-Conversion

Bound variables are renamed automatically to avoid name clashes with other variables.

Natural deduction for quantifiers

$$\frac{\bigwedge x.\ P(x)}{\forall x.\ P(x)}\ \texttt{allI} \qquad \frac{\forall x.\ P(x) \quad P(?x) \Longrightarrow R}{R}\ \texttt{allE}$$

$$\frac{P(?x)}{\exists x.\ P(x)}\ \texttt{exI} \qquad \frac{\exists x.\ P(x) \quad \bigwedge x.\ P(x) \Longrightarrow R}{R}\ \texttt{exE}$$

- `allI` and `exE` introduce new parameters ($\bigwedge x$).
- `allE` and `exI` introduce new unknowns (*?x*).

Instantiating rules

apply(*rule_tac x = "$term$" in $rule$*)

Like *rule*, but *?x* in $rule$ is instantiated by $term$ before application.

Similar: *erule_tac*

! *x* is in $rule$, not in the goal **!**

Two successful proofs

1. $\forall x.\ \exists y.\ x = y$

apply(*rule allI*)

1. $\bigwedge x.\ \exists y.\ x = y$

best practice	exploration
apply(*rule_tac x = "x" in exI*)	**apply**(*rule exI*)
1. $\bigwedge x.\ x = x$	*1.* $\bigwedge x.\ x = ?y\ x$
apply(*rule refl*)	**apply**(*rule refl*)
	$?y \mapsto \lambda u.\ u$
simpler & clearer	shorter & trickier

Demo: quantifier proofs

Safe and unsafe rules

Safe `allI, exE`

Unsafe `allE, exI`

Create parameters first, unknowns later

Demo: proof methods

Sets

Overview

- Set notation
- Inductively defined sets

Set notation

Sets

Type *'a set*: sets over type *'a*

- $\{\}$, $\{e_1,\ldots,e_n\}$, $\{x.\ P\ x\}$
- $e \in A$, $A \subseteq B$
- $A \cup B$, $A \cap B$, $A - B$, $- A$
- $\bigcup_{x \in A} B\ x$, $\bigcap_{x \in A} B\ x$
- $\{i..j\}$
- *insert* :: $'a \Rightarrow 'a\ set \Rightarrow 'a\ set$
- ...

Proofs about sets

Natural deduction proofs:

- `equalityI`: $[\![A \subseteq B; B \subseteq A]\!] \Longrightarrow A = B$
- `subsetI`: $(\bigwedge x.\ x \in A \Longrightarrow x \in B) \Longrightarrow A \subseteq B$
- ... (see Tutorial)

Demo: proofs about sets

Inductively defined sets

Example: finite sets

Informally:

- The empty set is finite
- Adding an element to a finite set yields a finite set
- These are the only finite sets

In Isabelle/HOL:

consts *Fin :: 'a set set* — The set of all finite set
inductive *Fin*
intros
{} $\in$ *Fin*
A $\in$ *Fin* $\Longrightarrow$ *insert a A* $\in$ *Fin*

Example: even numbers

Informally:

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

In Isabelle/HOL:

consts *Ev :: nat set* — The set of all even numbers
inductive *Ev*
intros
0 $\in$ *Ev*
n $\in$ *Ev* $\Longrightarrow$ *n + 2* $\in$ *Ev*

Format of inductive definitions

consts *S* :: τ *set*
inductive *S*
intros
$[\![a_1 \in S; \ldots ; a_n \in S; A_1; \ldots; A_k]\!] \Longrightarrow a \in S$
$\vdots$

where $A_1; \ldots; A_k$ are side conditions not involving *S*.

Proving properties of even numbers

Easy: *4* $\in$ *Ev*

$$0 \in Ev \Longrightarrow 2 \in Ev \Longrightarrow 4 \in Ev$$

Trickier: *m* $\in$ *Ev* $\Longrightarrow$ *m+m* $\in$ *Ev*
Idea: induction on the length of the derivation of *m* $\in$ *Ev*
Better: induction on the *structure* of the derivation
Two cases: *m* $\in$ *Ev* is proved by

- rule *0* $\in$ *Ev*
  $\Longrightarrow$ *m = 0* $\Longrightarrow$ *0+0* $\in$ *Ev*
- rule *n* $\in$ *Ev* $\Longrightarrow$ *n+2* $\in$ *Ev*
  $\Longrightarrow$ *m = n+2* and *n+n* $\in$ *Ev* (ind. hyp.!)
  $\Longrightarrow$ *m+m = (n+2)+(n+2) = ((n+n)+2)+2* $\in$ *Ev*

Rule induction for Ev

To prove

$$n \in Ev \Longrightarrow P\ n$$

by *rule induction* on $n \in Ev$ we must prove

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

Rule `Ev.induct`:

$$[\![\ n \in Ev;\ P\ 0;\ \bigwedge n.\ P\ n \Longrightarrow P(n+2)\ ]\!] \Longrightarrow P\ n$$

An elimination rule

Rule induction in general

Set S is defined inductively.
To prove

$$x \in S \Longrightarrow P\ x$$

by *rule induction* on $x \in S$
we must prove for every rule

$$[\![\ a_1 \in S;\ \dots\ ;\ a_n \in S\ ]\!] \Longrightarrow a \in S$$

that P is preserved:

$$[\![\ P\ a_1;\ \dots\ ;\ P\ a_n\ ]\!] \Longrightarrow P\ a$$

In Isabelle/HOL:

apply(*erule S.induct*)

Demo: inductively defined sets