An Introduction to Isabelle/HOL

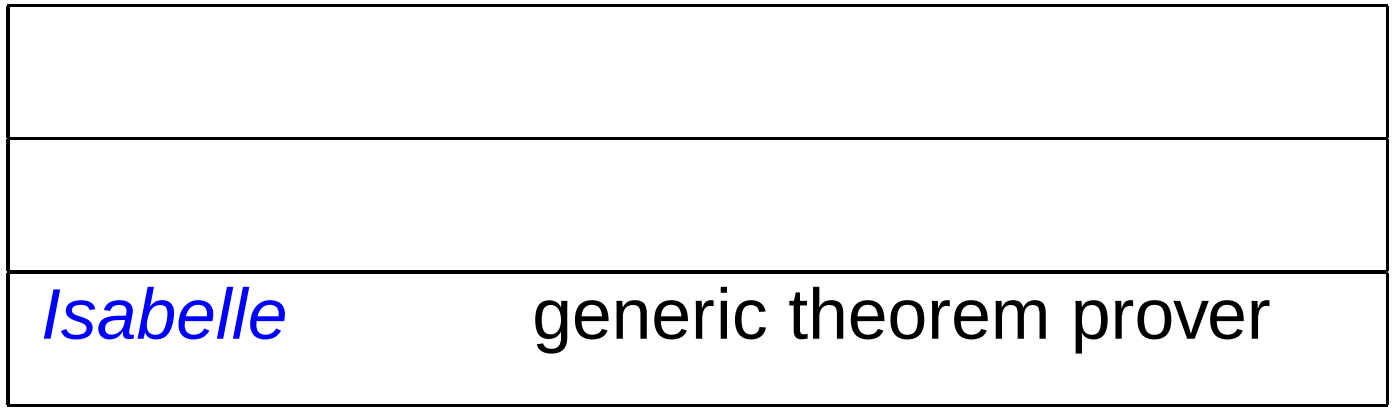
2009-1

Tobias Nipkow

TU München

Overview of Isabelle/HOL

System Architecture



System Architecture

Isabelle/HOL	Isabelle instance for HOL
Isabelle	generic theorem prover

System Architecture

ProofGeneral	(X)Emacs based interface
Isabelle/HOL	Isabelle instance for HOL
Isabelle	generic theorem prover

HOL

HOL = Higher-Order Logic

HOL

HOL = Higher-Order Logic

HOL = Functional programming + Logic

HOL

HOL = Higher-Order Logic

HOL = Functional programming + Logic

HOL has

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

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!

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):

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

Formulae

Syntax (in decreasing priority):

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

Examples

- $\neg A \wedge B \vee C \equiv ((\neg A) \wedge B) \vee C$

Formulae

Syntax (in decreasing priority):

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

Examples

- $\neg A \wedge B \vee C \equiv ((\neg A) \wedge B) \vee C$
- $A = B \wedge C \equiv (A = B) \wedge C$

Formulae

Syntax (in decreasing priority):

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

Examples

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

Formulae

Syntax (in decreasing priority):

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

Examples

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

Scope of quantifiers: as far to the right as possible

Formulae

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

Formulae

Abbreviation: $\forall x\ y.\ P\ x\ y \equiv \forall x.\ \forall y.\ P\ x\ y \qquad (\forall, \exists, \lambda, \ldots)$

Formulae

Abbreviation: $\forall x\, y.\ P\, x\, y \ \equiv\ \forall x.\ \forall y.\ P\, x\, y \qquad (\forall, \exists, \lambda, \ldots)$

Parentheses:

- $\wedge$, $\vee$ and $\longrightarrow$ associate to the right:
  $A \wedge B \wedge C \ \equiv\ A \wedge (B \wedge C)$

Formulae

Abbreviation: $\forall x\, y.\ P\, x\, y \equiv \forall x.\ \forall y.\ P\, x\, y \qquad (\forall, \exists, \lambda, \ldots)$

Parentheses:

- $\wedge$, $\vee$ and $\longrightarrow$ associate to the right:
  $A \wedge B \wedge C \equiv A \wedge (B \wedge C)$
- $A \longrightarrow B \longrightarrow C \equiv A \longrightarrow (B \longrightarrow C) \not\equiv (A \longrightarrow B) \longrightarrow C$ **!**

Warning

Quantifiers have low priority and need to be parenthesized:

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

Types and Terms

Types

Syntax:

$$\begin{array}{rcll} \tau & ::= & (\tau) & \\ & | & \mathit{bool} \mid \mathit{nat} \mid \ldots & \text{base types} \end{array}$$

Types

Syntax:

$$
\begin{array}{lcll}
\tau & ::= & (\tau) & \\
 & | & \mathit{bool} \mid \mathit{nat} \mid \ldots & \text{base types} \\
 & | & \mathit{'a} \mid \mathit{'b} \mid \ldots & \text{type variables}
\end{array}
$$

Types

Syntax:

$$
\begin{array}{lcll}
\tau & ::= & (\tau) & \\
 & | & \mathit{bool} \mid \mathit{nat} \mid \ldots & \text{base types} \\
 & | & \mathit{'a} \mid \mathit{'b} \mid \ldots & \text{type variables} \\
 & | & \tau \Rightarrow \tau & \text{total functions}
\end{array}
$$

Types

Syntax:

$$
\begin{array}{lcll}
\tau & ::= & (\tau) & \\
 & | & \mathit{bool} \mid \mathit{nat} \mid \ldots & \text{base types} \\
 & | & \mathit{'a} \mid \mathit{'b} \mid \ldots & \text{type variables} \\
 & | & \tau \Rightarrow \tau & \text{total functions} \\
 & | & \tau \times \tau & \text{pairs (ascii: } *\text{)}
\end{array}
$$

Types

Syntax:

$$
\begin{array}{lcll}
\tau & ::= & (\tau) & \\
 & | & \mathit{bool} \mid \mathit{nat} \mid \ldots & \text{base types} \\
 & | & \mathit{'a} \mid \mathit{'b} \mid \ldots & \text{type variables} \\
 & | & \tau \Rightarrow \tau & \text{total functions} \\
 & | & \tau \times \tau & \text{pairs (ascii: } *\text{)} \\
 & | & \tau\ \mathit{list} & \text{lists}
\end{array}
$$

Types

Syntax:

$$
\begin{array}{lcll}
\tau & ::= & (\tau) & \\
 & | & \mathit{bool} \mid \mathit{nat} \mid \ldots & \text{base types} \\
 & | & \mathit{'a} \mid \mathit{'b} \mid \ldots & \text{type variables} \\
 & | & \tau \Rightarrow \tau & \text{total functions} \\
 & | & \tau \times \tau & \text{pairs (ascii: } *\text{)} \\
 & | & \tau\ \mathit{list} & \text{lists} \\
 & | & \ldots & \text{user-defined types}
\end{array}
$$

Types

Syntax:

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

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

Terms: Basic syntax

Syntax:

$$
\begin{array}{lcll}
term & ::= & (term) & \\
 & | & a & \text{constant or variable (identifier)} \\
 & | & term\ term & \text{function application} \\
 & | & \lambda x.\ term & \text{function “abstraction”}
\end{array}
$$

Terms: Basic syntax

Syntax:

$$
\begin{array}{rcll}
term & ::= & (term) & \\
 & | & a & \text{constant or variable (identifier)} \\
 & | & term\ term & \text{function application} \\
 & | & \lambda x.\ term & \text{function “abstraction”} \\
 & | & \ldots & \text{lots of syntactic sugar}
\end{array}
$$

Terms: Basic syntax

Syntax:

$$
\begin{array}{rcll}
term & ::= & (term) & \\
 & | & a & \text{constant or variable (identifier)} \\
 & | & term\ term & \text{function application} \\
 & | & \lambda x.\ term & \text{function “abstraction”} \\
 & | & \ldots & \text{lots of syntactic sugar}
\end{array}
$$

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

Terms: Basic syntax

Syntax:

$$
\begin{array}{rcll}
term & ::= & (term) & \\
 & | & a & \text{constant or variable (identifier)} \\
 & | & term\ term & \text{function application} \\
 & | & \lambda x.\ term & \text{function “abstraction”} \\
 & | & \ldots & \text{lots of syntactic sugar}
\end{array}
$$

Examples: $f\ (g\ x)\ y$ $h\ (\lambda x.\ f\ (g\ x))$

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

λ-calculus on one slide

Informal notation: $t[x]$

λ-calculus on one slide

Informal notation: $t[x]$

- *Function application:*
  $f\ a$ is the call of function f with argument a

λ-calculus on one slide

Informal notation: $t[x]$

- *Function application:*
  $f\ a$ is the call of function f with argument a
- *Function abstraction:*
  $\lambda x.t[x]$ is the function with formal parameter x and body/result $t[x]$, i.e. $x \mapsto t[x]$.

λ-calculus on one slide

Informal notation: $t[x]$

- *Function application:*
  $f\ a$ is the call of function f with argument a
- *Function abstraction:*
  $\lambda x.t[x]$ is the function with formal parameter x and body/result $t[x]$, i.e. $x \mapsto t[x]$.
- *Computation:*
  Replace formal by actual parameter ("β-reduction"):
  $(\lambda x.t[x])\ a \quad \longrightarrow_\beta \quad t[a]$

λ-calculus on one slide

Informal notation: $t[x]$

- *Function application:*
  $f\ a$ is the call of function f with argument a
- *Function abstraction:*
  $\lambda x.t[x]$ is the function with formal parameter x and body/result $t[x]$, i.e. $x \mapsto t[x]$.
- *Computation:*
  Replace formal by actual parameter ("β-reduction"):
  $(\lambda x.t[x])\ a \quad \longrightarrow_\beta \quad t[a]$

Example: $(\lambda\ x.\ x + 5)\ 3 \quad \longrightarrow_\beta \quad (3 + 5)$

$\longrightarrow_\beta$ *in Isabelle: Don't worry, be happy*

Isabelle performs β-reduction automatically

Isabelle considers $(\lambda x.t[x])a$ and $t[a]$ equivalent

Terms and Types

Terms must be well-typed

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

Terms and Types

Terms must be well-typed

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

Notation: $t :: \tau$ means t is a well-typed term of type τ.

Type inference

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

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.

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

Currying

Thou shalt curry your functions

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

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 :: τ_1

Terms: Syntactic sugar

Some predefined syntactic sugar:

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

Terms: Syntactic sugar

Some predefined syntactic sugar:

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

Prefix binds more strongly than infix:

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

Terms: Syntactic sugar

Some predefined syntactic sugar:

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

Prefix binds more strongly than infix:

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

Enclose *if* and *case* in parentheses:

! *(if _ then _ else _)* !

Base types: bool, nat, list

Type bool

Formulae = terms of type *bool*

Type bool

Formulae = terms of type *bool*

True :: bool
False :: bool
$\wedge, \vee, \ldots$ *:: bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool*
$\vdots$

Type bool

Formulae = terms of type *bool*

True :: bool
False :: bool
$\wedge, \vee, \ldots$ *:: bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool*
$\vdots$

if-and-only-if: =

Type nat

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

Type nat

0 :: nat
Suc :: nat ⇒ nat
+, *, ... :: *nat ⇒ nat ⇒ nat*
⋮

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

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

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$_1$,...,x$_n$]*

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*, *length*, *filter*, *set*, *nth*, *take*, *drop*, *distinct*, ...

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

Isabelle Theories

Theory = Module

Syntax: `theory` $MyTh$
`imports` $ImpTh_1 \ldots ImpTh_n$
`begin`
(declarations, definitions, theorems, proofs, ...)*
`end`

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

Theory = Module

Syntax: `theory` $MyTh$
`imports` $ImpTh_1 \ldots ImpTh_n$
`begin`
(declarations, definitions, theorems, proofs, ...)*
`end`

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

Usually: `theory` $MyTh$
`imports 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”)

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)*

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 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)*

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]*

Structural induction on lists

P xs holds for all lists *xs* if

Structural induction on lists

P xs holds for all lists *xs* if

- *P Nil*

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)*

A recursive function: append

Definition by *primitive recursion*:

primrec *app :: 'a list ⇒ 'a list ⇒ 'a list* **where**
app Nil ys = ? |
app (Cons x xs) ys = ??

A recursive function: append

Definition by *primitive recursion*:

primrec *app :: 'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list* **where**
app Nil ys = ? |
app (Cons x xs) ys = ??

1 rule per constructor
Recursive calls must drop the constructor $\Longrightarrow$ Termination

Concrete syntax

In `.thy` files:

Types and formulas need to be inclosed in "

Concrete syntax

In `.thy` files:

Types and formulas need to be inclosed in "

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

Concrete syntax

In `.thy` files:

Types and formulas need to be inclosed in "

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

" normally not shown on slides

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.

Top down proofs

Command

sorry

“completes” any proof.

Top down proofs

Command

sorry

“completes” any proof.

Allows top down development:

Assume lemma first, prove it later.

Some useful tools

Disproving tools

Automatic counterexample search by random testing:

quickcheck

Disproving tools

Automatic counterexample search by random testing:

quickcheck

Counterexample search via SAT solver:

nitpick

Finding theorems

1. Click on Find button
2. Input search pattern (e.g. *"_ & True"*)

Demo: Disproving and Finding

Isabelle's meta-logic

Basic constructs

Implication $\Longrightarrow$ (==>)

For separating premises and conclusion of theorems

Basic constructs

Implication $\Longrightarrow$ (==>)

For separating premises and conclusion of theorems

Equality $\equiv$ (==)

For definitions

Basic constructs

Implication $\Longrightarrow$ (==>)

For separating premises and conclusion of theorems

Equality $\equiv$ (==)

For definitions

Universal quantifier $\bigwedge$ (! !)

For binding local variables

Basic constructs

Implication $\Longrightarrow$ (==>)

For separating premises and conclusion of theorems

Equality $\equiv$ (==)

For definitions

Universal quantifier $\bigwedge$ (! !)

For binding local variables

Do not use *inside* HOL formulae

Notation

$$[\![A_1; \ldots ; A_n]\!] \Longrightarrow B$$

abbreviates

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

Notation

$$[\![A_1; \ldots ; A_n]\!] \Longrightarrow B$$

abbreviates

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

$;$ $\approx$ “and”

The proof state

$1.\ \bigwedge x_1 \ldots x_p.\ [\![A_1; \ldots ; A_n]\!] \Longrightarrow B$

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

Type and function definition in Isabelle/HOL

Type definition in Isabelle/HOL

Introducing new types

Keywords:

- **typedecl**: pure declaration
- **types**: abbreviation
- **datatype**: recursive datatype

typedecl

typedecl $name$

Introduces new “opaque” type $name$ without definition

typedecl

typedecl $name$

Introduces new “opaque” type $name$ without definition

Example:

typedecl *addr* — An abstract type of addresses

types

types $name = \tau$

Introduces an *abbreviation* $name$ for type τ

types

types $name = \tau$

Introduces an *abbreviation* $name$ for type τ

Examples:

types
name = string
('a,'b)foo = 'a list $\times$ *'b list*

types

types $name = \tau$

Introduces an *abbreviation* $name$ for type τ

Examples:

types
name = string
('a,'b)foo = 'a list $\times$ *'b list*

Type abbreviations are expanded immediately after parsing
Not present in internal representation and Isabelle output

datatype

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 $(\alpha_1, \ldots, \alpha_n)\tau \quad = \quad C_1\ \tau_{1,1} \ldots \tau_{1,n_1}$

$\qquad\qquad\qquad\qquad\qquad\quad \mid \quad \ldots$

$\qquad\qquad\qquad\qquad\qquad\quad \mid \quad C_k\ \tau_{k,1} \ldots \tau_{k,n_k}$

- *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:*
  $(C_i\ x_1 \ldots x_{n_i} = C_i\ y_1 \ldots y_{n_i}) = (x_1 = y_1 \wedge \ldots \wedge x_{n_i} = y_{n_i})$

The general case

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

- *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

Function definition in Isabelle/HOL

Why nontermination can be harmful

How about $f\ x = f\ x + 1$?

Why nontermination can be harmful

How about $f\ x = f\ x + 1$?

Subtract $f\ x$ on both sides.

$$\Longrightarrow 0 = 1$$

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 **definition**
  No problem

Function definition schemas in Isabelle/HOL

- Non-recursive with **definition**
  No problem
- Primitive-recursive with **primrec**
  Terminating by construction

Function definition schemas in Isabelle/HOL

- Non-recursive with **definition**
  No problem
- Primitive-recursive with **primrec**
  Terminating by construction
- Well-founded recursion with **fun**
  Automatic termination proof

Function definition schemas in Isabelle/HOL

- Non-recursive with **definition**
  No problem
- Primitive-recursive with **primrec**
  Terminating by construction
- Well-founded recursion with **fun**
  Automatic termination proof
- Well-founded recursion with **function**
  User-supplied termination proof

definition

Definition (non-recursive) by example

definition *sq :: nat ⇒ nat* **where** *sq n = n * n*

Definitions: pitfalls

definition *prime :: nat* $\Rightarrow$ *bool* **where**
prime p = (1 < p $\wedge$ *(m dvd p* $\longrightarrow$ *m = 1* $\vee$ *m = p))*

Definitions: pitfalls

definition *prime :: nat* $\Rightarrow$ *bool* **where**
prime p = (1 < p $\wedge$ *(m dvd p* $\longrightarrow$ *m = 1* $\vee$ *m = p))*

Not a definition: free *m* not on left-hand side

Definitions: pitfalls

definition *prime :: nat* $\Rightarrow$ *bool* **where**
prime p = (1 < p $\wedge$ *(m dvd p* $\longrightarrow$ *m = 1* $\vee$ *m = p))*

Not a definition: free *m* not on left-hand side

! Every free variable on the rhs must occur on the lhs **!**

Definitions: pitfalls

definition *prime :: nat* $\Rightarrow$ *bool* **where**
prime p = (1 < p $\wedge$ *(m dvd p* $\longrightarrow$ *m = 1* $\vee$ *m = p))*

Not a definition: free *m* not on left-hand side

! Every free variable on the rhs must occur on the lhs **!**

prime p = (1 < p $\wedge$ *(*$\forall$*m. m dvd p* $\longrightarrow$ *m = 1* $\vee$ *m = p))*

Using definitions

Definitions are not used automatically

Using definitions

Definitions are not used automatically

Unfolding the definition of *sq*:

apply*(unfold sq_def)*

primrec

The example

primrec *app :: 'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list* **where**

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*:

$$
\begin{array}{lcl}
f\ x_1 \ldots (C_1\ y_{1,1} \ldots y_{1,n_1}) \ldots x_p & = & r_1 \mid \\
\vdots & & \\
f\ x_1 \ldots (C_k\ y_{k,1} \ldots y_{k,n_k}) \ldots x_p & = & r_k
\end{array}
$$

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*:

$$
\begin{array}{lcl}
f\ x_1 \ldots (C_1\ y_{1,1} \ldots y_{1,n_1}) \ldots x_p & = & r_1 \mid \\
\vdots & & \\
f\ x_1 \ldots (C_k\ y_{k,1} \ldots y_{k,n_k}) \ldots x_p & = & r_k
\end{array}
$$

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*

nat is a datatype

datatype *nat = 0 | Suc nat*

Functions on *nat* definable by primrec!

primrec *f :: nat ⇒ ...*
f 0 = ...
f(Suc n) = ... f n ...

More predefined types and functions

Type option

datatype *'a option = None | Some 'a*

Type option

datatype *'a option = None | Some 'a*

Important application:

$$\ldots \Rightarrow \textit{'a option} \approx \text{partial function:}$$

$$\textit{None} \approx \text{no result}$$

$$\textit{Some a} \approx \text{result } a$$

Type option

datatype *'a option = None | Some 'a*

Important application:

$$\ldots \Rightarrow \textit{'a option} \quad \approx \quad \text{partial function:}$$

$$\textit{None} \quad \approx \quad \text{no result}$$

$$\textit{Some a} \quad \approx \quad \text{result } a$$

Example:

primrec *lookup :: 'k* $\Rightarrow$ *('k* $\times$ *'v) list* $\Rightarrow$ *'v option* **where**

Type option

datatype *'a option = None | Some 'a*

Important application:

$$\ldots \Rightarrow \textit{'a option} \approx \text{partial function:}$$

$$\textit{None} \approx \text{no result}$$

$$\textit{Some a} \approx \text{result } a$$

Example:

primrec *lookup :: 'k* $\Rightarrow$ *('k* $\times$ *'v) list* $\Rightarrow$ *'v option* **where**

lookup k [] = None

Type option

datatype *'a option = None | Some 'a*

Important application:

$$\ldots \Rightarrow \textit{'a option} \approx \text{partial function:}$$

$$\textit{None} \approx \text{no result}$$

$$\textit{Some a} \approx \text{result } a$$

Example:

primrec *lookup :: 'k ⇒ ('k × 'v) list ⇒ 'v option* **where**

lookup k [] = None |

lookup k (x#xs) =

(if fst x = k then Some(snd x) else lookup k xs)

case

Datatype values can be taken apart with *case* expressions:

$$(case\ xs\ of\ []\ \Rightarrow\ \ldots\ |\ y\#ys\ \Rightarrow\ ...\ y\ ...\ ys\ ...)$$

case

Datatype values can be taken apart with *case* expressions:

$$(case\ xs\ of\ []\ \Rightarrow\ \ldots\ |\ y\#ys\ \Rightarrow\ ...\ y\ ...\ ys\ ...)$$

Wildcards:

$$(case\ xs\ of\ []\ \Rightarrow\ []\ |\ y\#\_\ \Rightarrow\ [y])$$

case

Datatype values can be taken apart with *case* expressions:

$$(case\ xs\ of\ []\ \Rightarrow\ \ldots\ |\ y\#ys\ \Rightarrow\ ...\ y\ ...\ ys\ ...)$$

Wildcards:

$$(case\ xs\ of\ []\ \Rightarrow\ []\ |\ y\#\_\ \Rightarrow\ [y])$$

Nested patterns:

$$(case\ xs\ of\ [0]\ \Rightarrow\ 0\ |\ [Suc\ n]\ \Rightarrow\ n\ |\ \_\ \Rightarrow\ 2)$$

case

Datatype values can be taken apart with *case* expressions:

$$(case\ xs\ of\ []\ \Rightarrow\ \ldots\ |\ y\#ys\ \Rightarrow\ ...\ y\ ...\ ys\ ...)$$

Wildcards:

$$(case\ xs\ of\ []\ \Rightarrow\ []\ |\ y\#\_\ \Rightarrow\ [y])$$

Nested patterns:

$$(case\ xs\ of\ [0]\ \Rightarrow\ 0\ |\ [Suc\ n]\ \Rightarrow\ n\ |\ \_\ \Rightarrow\ 2)$$

Complicated patterns mean complicated proofs!

case

Datatype values can be taken apart with *case* expressions:

$$(case\ xs\ of\ []\ \Rightarrow\ \ldots\ |\ y\#ys\ \Rightarrow\ ...\ y\ ...\ ys\ ...)$$

Wildcards:

$$(case\ xs\ of\ []\ \Rightarrow\ []\ |\ y\#\_\ \Rightarrow\ [y])$$

Nested patterns:

$$(case\ xs\ of\ [0]\ \Rightarrow\ 0\ |\ [Suc\ n]\ \Rightarrow\ n\ |\ \_\ \Rightarrow\ 2)$$

Complicated patterns mean complicated proofs!

Needs *( )* in context

Proof by case distinction

If $t :: \tau$ and τ is a datatype

apply(*case_tac* t)

Proof by case distinction

If $t :: \tau$ and τ is a datatype

apply(*case_tac* t)

creates k subgoals

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

one for each constructor C_i of type τ.

Demo: trees

fun

From primitive recursion to arbitrary pattern matching

Example: Fibonacchi

fun *fib :: nat* $\Rightarrow$ *nat* **where**

fib 0 = 0 |
fib (Suc 0) = 1 |
fib (Suc(Suc n)) = fib (n+1) + fib n

Example: Separation

fun *sep :: 'a ⇒ 'a list ⇒ 'a list* **where**

sep a [] = [] |
sep a [x] = [x] |
sep a (x#y#zs) = x # a # sep a (y#zs)

Example: Ackermann

fun *ack :: nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* **where**

```
ack 0         n         = Suc n  |
ack (Suc m) 0           = ack m (Suc 0)  |
ack (Suc m) (Suc n) = ack m (ack (Suc m) n)
```

Key features of fun

- Arbitrary pattern matching

Key features of fun

- Arbitrary pattern matching
- Order of equations matters

Key features of fun

- Arbitrary pattern matching
- Order of equations matters
- Termination must be provable
  by lexicographic combination of size measures

Size

- *size(n::nat) = n*

Size

- *size(n::nat) = n*
- *size(xs) = length xs*

Size

- *size(n::nat) = n*
- *size(xs) = length xs*
- *size* counts number of (non-nullary) constructors

Lexicographic ordering

Either the first component decreases, or it stays unchanged and the second component decreases:

Lexicographic ordering

Either the first component decreases, or it stays unchanged and the second component decreases:

$$(5,3) > (4,7) > (4,6) > (4,0) > (3,42) > \cdots$$

Lexicographic ordering

Either the first component decreases, or it stays unchanged and the second component decreases:

$$(5,3) > (4,7) > (4,6) > (4,0) > (3,42) > \cdots$$

Similar for tuples:

$$(5,6,3) > (4,12,5) > (4,11,9) > (4,11,8) > \cdots$$

Lexicographic ordering

Either the first component decreases, or it stays unchanged and the second component decreases:

$$(5,3) > (4,7) > (4,6) > (4,0) > (3,42) > \cdots$$

Similar for tuples:

$$(5,6,3) > (4,12,5) > (4,11,9) > (4,11,8) > \cdots$$

Theorem If each component ordering terminates, then their *lexicographic product* terminates, too.

Ackermann terminates

ack 0 n = Suc n

ack (Suc m) 0 = ack m (Suc 0)

ack (Suc m) (Suc n) = ack m (ack (Suc m) n)

Ackermann terminates

ack 0 n = Suc n

ack (Suc m) 0 = ack m (Suc 0)

ack (Suc m) (Suc n) = ack m (ack (Suc m) n)

because the arguments of each recursive call are lexicographically smaller than the arguments on the lhs.

Ackermann terminates

ack 0 n = Suc n

ack (Suc m) 0 = ack m (Suc 0)

ack (Suc m) (Suc n) = ack m (ack (Suc m) n)

because the arguments of each recursive call are lexicographically smaller than the arguments on the lhs.

Note: order of arguments not important for Isabelle!

Computation Induction

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

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 equation $f(e) = t$,
> prove $P(e)$ assuming $P(r)$ for all recursive calls $f(r)$ in t.

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 equation $f(e) = t$,
> prove $P(e)$ assuming $P(r)$ for all recursive calls $f(r)$ in t.

Induction follows course of (terminating!) computation

Computation Induction: Example

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

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`:

$$\frac{P(0) \quad P(Suc\ 0) \quad P(n) \Longrightarrow P(Suc(Suc\ n))}{P(m)}$$

Demo: fun

Proof by Simplification

Overview

- Term rewriting foundations
- Term rewriting in Isabelle/HOL
  - Basic simplification
  - Extensions

Term rewriting foundations

Term rewriting means . . .

Using equations $l = r$ from left to right

Term rewriting means . . .

Using equations $l = r$ from left to right

As long as possible

Term rewriting means . . .

Using equations $l = r$ from left to right

As long as possible

Terminology: equation $\leadsto$ *rewrite rule*

An example

Equations:

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

An example

Equations:

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

$$0 + Suc\ 0 \ \leq \ Suc\ 0 + x$$

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:

$$
\begin{aligned}
0 + Suc\ 0 &\leq Suc\ 0 + x \quad \overset{(1)}{=}\\
Suc\ 0 &\leq Suc\ 0 + x
\end{aligned}
$$

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:

$$
\begin{aligned}
0 + Suc\ 0 &\leq Suc\ 0 + x && \overset{(1)}{=}\\
Suc\ 0 &\leq Suc\ 0 + x && \overset{(2)}{=}\\
Suc\ 0 &\leq Suc\ (0 + x)
\end{aligned}
$$

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:

$$
\begin{aligned}
0 + Suc\ 0 &\leq Suc\ 0 + x && \overset{(1)}{=}\\
Suc\ 0 &\leq Suc\ 0 + x && \overset{(2)}{=}\\
Suc\ 0 &\leq Suc\ (0 + x) && \overset{(3)}{=}\\
0 &\leq 0 + x
\end{aligned}
$$

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:

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

More formally

substitution = mapping from variables to terms

More formally

substitution = mapping from variables to terms

- $l = r$ is *applicable* to term $t[s]$
  if there is a substitution σ such that $\sigma(l) = s$

More formally

substitution = mapping from variables to terms

- $l = r$ is *applicable* to term $t[s]$
  if there is a substitution σ such that $\sigma(l) = s$
- Result: $t[\sigma(r)]$

More formally

substitution = mapping from variables to terms

- $l = r$ is *applicable* to term $t[s]$
  if there is a substitution σ such that $\sigma(l) = s$
- Result: $t[\sigma(r)]$
- Note: $t[s] = t[\sigma(r)]$

More formally

substitution = mapping from variables to terms

- $l = r$ is *applicable* to term $t[s]$
  if there is a substitution σ such that $\sigma(l) = s$
- Result: $t[\sigma(r)]$
- Note: $t[s] = t[\sigma(r)]$

Example:

Equation: $0 + n = n$

Term: $a + (0 + (b + c))$

More formally

substitution = mapping from variables to terms

- $l = r$ is *applicable* to term $t[s]$
  if there is a substitution σ such that $\sigma(l) = s$
- Result: $t[\sigma(r)]$
- Note: $t[s] = t[\sigma(r)]$

Example:

Equation: $0 + n = n$

Term: $a + (0 + (b + c))$

$\sigma = \{n \mapsto b + c\}$

More formally

substitution = mapping from variables to terms

- $l = r$ is *applicable* to term $t[s]$
  if there is a substitution σ such that $\sigma(l) = s$
- Result: $t[\sigma(r)]$
- Note: $t[s] = t[\sigma(r)]$

Example:

Equation: $0 + n = n$

Term: $a + (0 + (b + c))$

$\sigma = \{n \mapsto b + c\}$

Result: $a + (b + c)$

Extension: conditional rewriting

Rewrite rules can be conditional:

$$[\![P_1 \dots P_n]\!] \Longrightarrow l = r$$

Extension: conditional rewriting

Rewrite rules can be conditional:

$$[\![P_1 \dots P_n]\!] \Longrightarrow l = r$$

is *applicable* to term $t[s]$ with σ if

- $\sigma(l) = s$ and
- $\sigma(P_1)$, . . . , $\sigma(P_n)$ are provable (again by rewriting).

Interlude: Variables in Isabelle

Schematic variables

Three kinds of variables:

- bound: $\forall x.\ x = x$
- free: $x = x$

Schematic variables

Three kinds of variables:

- bound: $\forall x.\ x = x$
- free: $x = x$
- schematic: $?x = ?x$ (“unknown”)

Schematic variables

Three kinds of variables:

- bound: $\forall x.\ x = x$
- free: $x = x$
- schematic: $?x = ?x$ (“unknown”)

Schematic variables:

Schematic variables

Three kinds of variables:

- bound: $\forall x.\ x = x$
- free: $x = x$
- schematic: $?x = ?x$ (“unknown”)

Schematic variables:

- Logically: free = schematic

Schematic variables

Three kinds of variables:

- bound: $\forall x.\ x = x$
- free: $x = x$
- schematic: $?x = ?x$ (“unknown”)

Schematic variables:

- Logically: free = schematic
- Operationally:
  - free variables are fixed
  - schematic variables are instantiated by substitutions

From x to ?x

State lemmas with free variables:

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

From x to ?x

State lemmas with free variables:

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

⋮

done

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*

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*

Example: rewriting

rev(a @ []) = rev a

using *app_Nil2* with $\sigma = \{?xs \mapsto a\}$

Term rewriting in Isabelle

Basic simplification

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

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

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*

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**, **fun** and **datatype**

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**, **fun** and **datatype**
- additional lemmas $eq_1 \ldots eq_n$

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**, **fun** and **datatype**
- additional lemmas $eq_1 \ldots eq_n$
- assumptions $P_1 \ldots P_m$

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**, **fun** and **datatype**
- additional lemmas $eq_1 \ldots eq_n$
- assumptions $P_1 \ldots P_m$

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

Termination

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

Termination

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

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

Termination

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

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

$$[\![P_1 \ldots P_n]\!] \Longrightarrow l = r$$

is suitable as a *simp*-rule only
if l is "bigger" than r and each P_i

Termination

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

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

$$[\![P_1 \ldots P_n]\!] \Longrightarrow l = r$$

is suitable as a *simp*-rule only
if l is "bigger" than r and each P_i

n < *m* $\Longrightarrow$ (*n* < *Suc m*) = *True*
Suc n < *m* $\Longrightarrow$ (*n* < *m*) = *True*

Termination

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

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

$$[\![P_1 \ldots P_n]\!] \Longrightarrow l = r$$

is suitable as a *simp*-rule only
if l is "bigger" than r and each P_i

n < *m* $\Longrightarrow$ (*n* < *Suc m*) = *True* YES
Suc n < *m* $\Longrightarrow$ (*n* < *m*) = *True* NO

Rewriting with definitions

Definitions do not have the *simp* attribute.

Rewriting with definitions

Definitions do not have the *simp* attribute.

They must be used explicitly: *(simp add: f_def . . .)*

Extensions of rewriting

Local assumptions

Simplification of $A \longrightarrow B$:

1. Simplify A to A'
2. Simplify B using A'

Case splitting with simp

$$P(\text{if } A \text{ then } s \text{ else } t)$$
$$=$$
$$(A \longrightarrow P(s)) \wedge (\neg A \longrightarrow P(t))$$

Case splitting with simp

$$P(\textit{if } A \textit{ then } s \textit{ else } t)$$
$$=$$
$$(A \longrightarrow P(s)) \wedge (\neg A \longrightarrow P(t))$$

Automatic

Case splitting with simp

$$P(\text{if } A \text{ then } s \text{ else } t)$$
$$=$$
$$(A \longrightarrow P(s)) \wedge (\neg A \longrightarrow P(t))$$

Automatic

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

Case splitting with simp

$$P(\text{if } A \text{ then } s \text{ else } t)$$
$$=$$
$$(A \longrightarrow P(s)) \wedge (\neg A \longrightarrow P(t))$$

Automatic

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

By hand: *(simp split: nat.split)*

Case splitting with simp

$$P(\mathit{if}\ A\ \mathit{then}\ s\ \mathit{else}\ t)$$
$$=$$
$$(A \longrightarrow P(s)) \wedge (\neg A \longrightarrow P(t))$$

Automatic

$$P(\mathit{case}\ e\ \mathit{of}\ 0 \Rightarrow a \mid \mathit{Suc}\ n \Rightarrow b)$$
$$=$$
$$(e = 0 \longrightarrow P(a)) \wedge (\forall n.\ e = \mathit{Suc}\ n \longrightarrow P(b))$$

By hand: *(simp split: nat.split)*

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

Ordered rewriting

Problem: $?x + ?y = ?y + ?x$ does not terminate

Ordered rewriting

Problem: $?x + ?y = ?y + ?x$ does not terminate

Solution: permutative *simp*-rules are used only if the term becomes lexicographically smaller.

Ordered rewriting

Problem: $?x + ?y = ?y + ?x$ does not terminate

Solution: permutative *simp*-rules are used only if the term becomes lexicographically smaller.

Example: $b + a \rightsquigarrow a + b$ but not $a + b \rightsquigarrow b + a$.

Ordered rewriting

Problem: $?x + ?y = ?y + ?x$ does not terminate

Solution: permutative *simp*-rules are used only if the term becomes lexicographically smaller.

Example: $b + a \rightsquigarrow a + b$ but not $a + b \rightsquigarrow b + a$.

For types *nat*, *int* etc:

- lemmas *add_ac* sort any sum ($+$)
- lemmas *times_ac* sort any product ($*$)

Ordered rewriting

Problem: $?x + ?y = ?y + ?x$ does not terminate

Solution: permutative *simp*-rules are used only if the term becomes lexicographically smaller.

Example: $b + a \rightsquigarrow a + b$ but not $a + b \rightsquigarrow b + a$.

For types *nat*, *int* etc:

- lemmas *add_ac* sort any sum ($+$)
- lemmas *times_ac* sort any product ($*$)

Example: *(simp add: add_ac)* yields

$$(b + c) + a \rightsquigarrow \cdots \rightsquigarrow a + (b + c)$$

Preprocessing

simp-rules are preprocessed (recursively) for maximal simplification power:

$$\begin{aligned} \neg A &\mapsto A = False \\ A \longrightarrow B &\mapsto A \Longrightarrow B \\ A \wedge B &\mapsto A,\ B \\ \forall x.A(x) &\mapsto A(?x) \\ A &\mapsto A = True \end{aligned}$$

Preprocessing

simp-rules are preprocessed (recursively) for maximal simplification power:

$$\begin{aligned} \neg A &\mapsto A = False \\ A \longrightarrow B &\mapsto A \Longrightarrow B \\ A \wedge B &\mapsto A,\ B \\ \forall x.A(x) &\mapsto A(?x) \\ A &\mapsto A = True \end{aligned}$$

Example:

$(p \longrightarrow q \wedge \neg r) \wedge s \quad \mapsto$

Preprocessing

simp-rules are preprocessed (recursively) for maximal simplification power:

$$\begin{aligned} \neg A &\mapsto A = False \\ A \longrightarrow B &\mapsto A \Longrightarrow B \\ A \wedge B &\mapsto A,\ B \\ \forall x.A(x) &\mapsto A(?x) \\ A &\mapsto A = True \end{aligned}$$

Example:

$$(p \longrightarrow q \wedge \neg\, r) \wedge s \quad \mapsto \quad \left\{ \begin{array}{r} p \Longrightarrow q = True \\ p \Longrightarrow r = False \\ s = True \end{array} \right\}$$

When everything else fails: Tracing

Set trace mode on/off in Proof General:

Isabelle $\rightarrow$ Settings $\rightarrow$ Trace simplifier

Output in separate `trace` buffer

Demo: simp

Induction heuristics

Basic heuristics

Theorems about recursive functions are proved by induction

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

primrec *itrev :: 'a list ⇒ 'a list ⇒ 'a list*

A tail recursive reverse

primrec *itrev :: 'a list ⇒ 'a list ⇒ 'a list* **where**

itrev [] ys = ys |

itrev (x#xs) ys =

A tail recursive reverse

primrec *itrev :: 'a list ⇒ 'a list ⇒ 'a list* **where**

itrev [] ys = ys |

itrev (x#xs) ys = itrev xs (x#ys)

A tail recursive reverse

primrec *itrev :: 'a list ⇒ 'a list ⇒ 'a list* **where**

itrev [] ys = ys |

itrev (x#xs) ys = itrev xs (x#ys)

lemma *itrev xs [] = rev xs*

A tail recursive reverse

primrec *itrev :: 'a list ⇒ 'a list ⇒ 'a list* **where**

itrev [] ys = ys |

itrev (x#xs) ys = itrev xs (x#ys)

lemma *itrev xs [] = rev xs*

Why in this direction?

A tail recursive reverse

primrec *itrev :: 'a list ⇒ 'a list ⇒ 'a list* **where**

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

Generalisation

- Replace constants by variables

Generalisation

- Replace constants by variables
- Generalize free variables
  - by $\forall$ in formula
  - by *arbitrary* in induction proof

HOL: Propositional Logic

Overview

- Natural deduction
- Rule application in Isabelle/HOL

Rule notation

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

Natural Deduction

Natural deduction

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

Natural deduction

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

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

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

$$\frac{}{A \wedge B}\ \texttt{conjI} \qquad \frac{A \wedge B}{\quad}\ \texttt{conjE}$$

$$\frac{A}{A \vee B}\ \frac{B}{A \vee B}\ \texttt{disjI1/2} \qquad \frac{A \vee B \quad \begin{array}{c}[A]\\ C\end{array} \quad \begin{array}{c}[B]\\ C\end{array}}{C}\ \texttt{disjE}$$

$$\frac{\begin{array}{c}[A]\\ B\end{array}}{A \rightarrow B}\ \texttt{impI} \qquad \frac{A \rightarrow B \quad A}{B}\ \texttt{impE}$$

$$\frac{A \rightarrow B \quad B \rightarrow A}{A \leftrightarrow B}\ \texttt{iffI} \qquad \frac{A \leftrightarrow B}{A \rightarrow B}\ \texttt{iffD1} \qquad \frac{A \leftrightarrow B}{B \rightarrow A}\ \texttt{iffD2}$$

$$\frac{\begin{array}{c}[A]\\ \bot\end{array}}{\neg A}\ \texttt{notI} \qquad \frac{\neg A \quad A}{\bot}\ \texttt{notE}$$

Natural deduction for propositional logic

$$\frac{A \quad B}{A \wedge B}\ \texttt{conjI}$$

$$\frac{A \wedge B}{\quad}\ \texttt{conjE}$$

$$\frac{A}{A \vee B}\ \texttt{disjI1/2} \qquad \frac{B}{A \vee B}\ \texttt{disjI1/2}$$

$$\frac{A \vee B \quad A \Rightarrow C \quad B \Rightarrow C}{C}\ \texttt{disjE}$$

$$\frac{\begin{array}{c}[A] \\ B\end{array}}{A \Rightarrow B}\ \texttt{impI}$$

$$\frac{A \Rightarrow B \quad A}{B}\ \texttt{impE}$$

$$\frac{A \Rightarrow B \quad B \Rightarrow A}{A \Leftrightarrow B}\ \texttt{iffI}$$

$$\frac{A \Leftrightarrow B \quad A}{B}\ \texttt{iffD1} \qquad \frac{A \Leftrightarrow B \quad B}{A}\ \texttt{iffD2}$$

$$\frac{\begin{array}{c}[A] \\ \bot\end{array}}{\neg A}\ \texttt{notI}$$

$$\frac{\neg A \quad A}{\bot}\ \texttt{notE}$$

Natural deduction for propositional logic

$$\frac{A \quad B}{A \wedge B}\ \texttt{conjI} \qquad \frac{A \wedge B}{A}\ \texttt{conjE}$$

$$\frac{A}{A \vee B}\ \texttt{disjI1} \quad \frac{B}{A \vee B}\ \texttt{disjI2} \qquad \frac{A \vee B \quad A \Rightarrow C \quad B \Rightarrow C}{C}\ \texttt{disjE}$$

$$\frac{B}{A \Rightarrow B}\ \texttt{impI} \qquad \frac{A \Rightarrow B \quad A}{B}\ \texttt{impE}$$

$$\frac{A \Rightarrow B \quad B \Rightarrow A}{A \Leftrightarrow B}\ \texttt{iffI} \qquad \frac{A \Leftrightarrow B}{A \Rightarrow B}\ \texttt{iffD1} \qquad \frac{A \Leftrightarrow B}{B \Rightarrow A}\ \texttt{iffD2}$$

$$\frac{A \Rightarrow \bot}{\neg A}\ \texttt{notI} \qquad \frac{\neg A \quad A}{\bot}\ \texttt{notE}$$

Natural deduction for propositional logic

$$\frac{A \quad B}{A \wedge B}\ \texttt{conjI} \qquad \frac{A \wedge B}{A}\ \texttt{conjE}$$

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

$$\frac{}{A \Rightarrow A}\ \texttt{impI} \qquad \frac{A \Rightarrow B \quad A}{B}\ \texttt{impE}$$

$$\frac{A \Rightarrow B \quad B \Rightarrow A}{A \Leftrightarrow B}\ \texttt{iffI} \qquad \frac{A \Leftrightarrow B}{A \Rightarrow B}\ \texttt{iffD1} \qquad \frac{A \Leftrightarrow B}{B \Rightarrow A}\ \texttt{iffD2}$$

$$\frac{A \Rightarrow \bot}{\neg A}\ \texttt{notI} \qquad \frac{\neg A \quad A}{\bot}\ \texttt{notE}$$

Natural deduction for propositional logic

$$\frac{A \quad B}{A \wedge B}\ \texttt{conjI} \qquad \frac{A \wedge B}{A}\ \texttt{conjE} \qquad \frac{A \wedge B}{B}\ \texttt{conjE}$$

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

$$\frac{\begin{array}{c}[A] \\ B\end{array}}{A \longrightarrow B}\ \texttt{impI} \qquad \frac{A \longrightarrow B \quad A}{B}\ \texttt{impE}$$

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

$$\frac{\begin{array}{c}[A] \\ \bot\end{array}}{\neg A}\ \texttt{notI} \qquad \frac{\neg A \quad A}{\bot}\ \texttt{notE}$$

Natural deduction for propositional logic

$$\frac{A \quad B}{A \wedge B}\ \texttt{conjI} \qquad \frac{A \wedge B}{A}\ \texttt{conjE}$$

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

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

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

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

Natural deduction for propositional logic

$$\frac{A \quad B}{A \wedge B}\ \texttt{conjI} \qquad \frac{A \wedge B}{A}\ \texttt{conjE} \qquad \frac{A \wedge B}{B}$$

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

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

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

$$\frac{A \longrightarrow \mathit{False}}{\neg A}\ \texttt{notI} \qquad \frac{\neg A \quad A}{\mathit{False}}\ \texttt{notE}$$

Natural deduction for propositional logic

$$\frac{A \quad B}{A \wedge B}\ \texttt{conjI} \qquad \frac{A \wedge B}{A}\ \texttt{conjE} \qquad \frac{A \wedge B}{B}\ \texttt{conjE}$$

$$\frac{A}{A \vee B}\ \texttt{disjI1} \qquad \frac{B}{A \vee B}\ \texttt{disjI2} \qquad \frac{A \vee B \quad A \Longrightarrow C \quad B \Longrightarrow C}{C}\ \texttt{disjE}$$

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

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

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

Natural deduction for propositional logic

$$\frac{A \quad B}{A \wedge B}\ \texttt{conjI} \qquad \frac{A \wedge B}{A}\ \texttt{conjE} \qquad \frac{A \wedge B}{B}\ \texttt{conjE}$$

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

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

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

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

Natural deduction for propositional logic

$$\frac{A \quad B}{A \wedge B}\ \texttt{conjI} \qquad \frac{A \wedge B}{A}\ \texttt{conjE} \qquad \frac{A \wedge B}{B}$$

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

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

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

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

Natural deduction for propositional logic

$$\frac{A \quad B}{A \wedge B}\ \texttt{conjI} \qquad \frac{A \wedge B}{C}\ \texttt{conjE}$$

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

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

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

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

Natural deduction for propositional logic

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

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

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

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

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

Natural deduction for propositional logic

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

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

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

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

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

Natural deduction for propositional logic

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

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

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

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

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

Natural deduction for propositional logic

$$\frac{A \quad B}{A \wedge B}\ \texttt{conjI}$$

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

$$\frac{A}{A \vee B}\ \frac{B}{A \vee B}\ \texttt{disjI1/2}$$

$$\frac{A \vee B \quad A \Longrightarrow C \quad B \Longrightarrow C}{C}\ \texttt{disjE}$$

$$\frac{A \Longrightarrow B}{A \longrightarrow B}\ \texttt{impI}$$

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

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

$$\frac{\qquad\qquad}{\qquad\qquad}\ \texttt{iffD1}\qquad \frac{\qquad\qquad}{\qquad\qquad}\ \texttt{iffD2}$$

$$\frac{A \Longrightarrow \mathit{False}}{\neg A}\ \texttt{notI}$$

$$\frac{\qquad\qquad}{\qquad\qquad}\ \texttt{notE}$$

Natural deduction for propositional logic

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

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

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

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

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

Natural deduction for propositional logic

$$\frac{A \quad B}{A \wedge B}\ \texttt{conjI}$$

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

$$\frac{A}{A \vee B}\ \frac{B}{A \vee B}\ \texttt{disjI1/2}$$

$$\frac{A \vee B \quad A \Longrightarrow C \quad B \Longrightarrow C}{C}\ \texttt{disjE}$$

$$\frac{A \Longrightarrow B}{A \longrightarrow B}\ \texttt{impI}$$

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

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

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

$$\frac{A \Longrightarrow \mathit{False}}{\neg A}\ \texttt{notI}$$

$$\frac{}{}\ \texttt{notE}$$

Natural deduction for propositional logic

$$\frac{A \quad B}{A \wedge B}\ \texttt{conjI}$$

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

$$\frac{A}{A \vee B}\ \frac{B}{A \vee B}\ \texttt{disjI1/2}$$

$$\frac{A \vee B \quad A \Longrightarrow C \quad B \Longrightarrow C}{C}\ \texttt{disjE}$$

$$\frac{A \Longrightarrow B}{A \longrightarrow B}\ \texttt{impI}$$

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

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

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

$$\frac{A \Longrightarrow \mathit{False}}{\neg A}\ \texttt{notI}$$

$$\frac{}{}\ \texttt{notE}$$

Natural deduction for propositional logic

$$\frac{A \quad B}{A \wedge B}\ \texttt{conjI}$$

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

$$\frac{A}{A \vee B}\ \frac{B}{A \vee B}\ \texttt{disjI1/2}$$

$$\frac{A \vee B \quad A \Longrightarrow C \quad B \Longrightarrow C}{C}\ \texttt{disjE}$$

$$\frac{A \Longrightarrow B}{A \longrightarrow B}\ \texttt{impI}$$

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

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

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

$$\frac{A \Longrightarrow \mathit{False}}{\neg A}\ \texttt{notI}$$

$$\frac{\neg A}{C}\ \texttt{notE}$$

Natural deduction for propositional logic

$$\frac{A \quad B}{A \wedge B}\ \texttt{conjI}$$

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

$$\frac{A}{A \vee B}\ \frac{B}{A \vee B}\ \texttt{disjI1/2}$$

$$\frac{A \vee B \quad A \Longrightarrow C \quad B \Longrightarrow C}{C}\ \texttt{disjE}$$

$$\frac{A \Longrightarrow B}{A \longrightarrow B}\ \texttt{impI}$$

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

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

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

$$\frac{A \Longrightarrow \mathit{False}}{\neg A}\ \texttt{notI}$$

$$\frac{\neg A \quad A}{C}\ \texttt{notE}$$

Operational reading

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

Operational reading

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

Introduction rule:

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

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 $A_2 \ldots A_n$.

Equality

$$\frac{}{t = t}\ \texttt{refl} \qquad \frac{s = t}{t = s}\ \texttt{sym} \qquad \frac{r = s \quad s = t}{r = t}\ \texttt{trans}$$

Equality

$$\frac{}{t = t}\ \texttt{refl} \qquad \frac{s = t}{t = s}\ \texttt{sym} \qquad \frac{r = s \quad s = t}{r = t}\ \texttt{trans}$$

$$\frac{s = t \quad A(s)}{A(t)}\ \texttt{subst}$$

Equality

$$\frac{}{t = t}\ \texttt{refl} \qquad \frac{s = t}{t = s}\ \texttt{sym} \qquad \frac{r = s \quad s = t}{r = t}\ \texttt{trans}$$

$$\frac{s = t \quad A(s)}{A(t)}\ \texttt{subst}$$

Rarely needed explicitly — used implicitly by *simp*

More rules

$$\frac{A \longrightarrow B \quad A}{B}\ \mathtt{mp}$$

More rules

$$\frac{A \longrightarrow B \quad A}{B}\ \texttt{mp}$$

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

More rules

$$\frac{A \longrightarrow B \quad A}{B}\ \texttt{mp}$$

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

Remark:

`ccontr` and `classical` are not derivable from the ND-rules.

More rules

$$\frac{A \longrightarrow B \quad A}{B}\ \texttt{mp}$$

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

Remark:

`ccontr` and `classical` are not derivable from the ND-rules.

They make the logic “classical”, i.e. “non-constructive”.

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:

Rule application: the rough idea

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

- Unify A and C

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 $A_1 \ldots A_n$

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!

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: $1.\ A \wedge B$

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

Rule application: the details

Rule: $[\![A_1; \ldots ; A_n]\!] \Longrightarrow A$

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

Substitution: $\sigma(A) \equiv \sigma(C)$

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:

$1.\ \sigma([\![B_1; \ldots ; B_m]\!] \Longrightarrow A_1)$

$\vdots$

$n.\ \sigma([\![B_1; \ldots ; B_m]\!] \Longrightarrow A_n)$

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

Proof by assumption

apply *assumption*

proves

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

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

Applying elimination rules

apply(*erule <elim-rule>*)

Like *rule* but also

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

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: $1.\ [\![\ X;\ A \wedge B;\ Y\ ]\!] \Longrightarrow Z$

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 $?R \equiv Z$

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:	$1.\ [\![\ X;\ Y\ ]\!] \Longrightarrow [\![\ A;\ B\ ]\!] \Longrightarrow Z$

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>*)

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: propositional proofs

$\Longrightarrow$ *vs* $\longrightarrow$

To facilitate application of theorems:

write them like this $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$

not like this $A_1 \wedge \ldots \wedge A_n \longrightarrow A$

HOL: Predicate Logic

Parameters

Subgoal:

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

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

Parameters

Subgoal:

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

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

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

Scope

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

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

$$\frac{}{\forall x.\ P(x)}\ \texttt{allI} \qquad \frac{\qquad}{\qquad}\ \texttt{allE}$$

$$\frac{}{\qquad}\ \texttt{exI} \qquad \frac{\qquad}{\qquad}\ \texttt{exE}$$

Natural deduction for quantifiers

$$\frac{\bigwedge x.\ P(x)}{\forall x.\ P(x)}\ \texttt{allI} \qquad \frac{\forall x.\ P(x)}{P(t)}\ \texttt{allE}$$

$$\frac{P(t)}{\exists x.\ P(x)}\ \texttt{exI} \qquad \frac{\exists x.\ P(x) \qquad \bigwedge x.\ P(x) \Longrightarrow Q}{Q}\ \texttt{exE}$$

Natural deduction for quantifiers

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

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

Natural deduction for quantifiers

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

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

Natural deduction for quantifiers

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

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

Natural deduction for quantifiers

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

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

Natural deduction for quantifiers

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

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

Natural deduction for quantifiers

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

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

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 ($\bigwedge x$).

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.

Instantiating rules

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

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

Similar: *erule_tac*

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 **!**

A quantifier proof

1. $\forall a.\ \exists b.\ a = b$

A quantifier proof

1. $\forall a.\ \exists b.\ a = b$

apply*(rule alll)*

A quantifier proof

1. $\forall a.\ \exists b.\ a = b$

apply*(rule alll)*

1. $\bigwedge a.\ \exists b.\ a = b$

A quantifier proof

1. ∀a. ∃b. a = b

apply*(rule alll)*

1. ⋀a. ∃b. a = b

apply*(rule_tac x = "a" in exI)*

A quantifier proof

1. ∀a. ∃b. a = b

apply*(rule alll)*

1. ⋀a. ∃b. a = b

apply*(rule_tac x = "a" in exI)*

1. ⋀a. a = a

A quantifier proof

1. ∀a. ∃b. a = b

apply*(rule alll)*

1. ⋀a. ∃b. a = b

apply*(rule_tac x = "a" in exI)*

1. ⋀a. a = a

apply*(rule refl)*

Demo: quantifier proofs

More proof methods

Forward proofs: frule and drule

"Forward" rule: $A_1 \Longrightarrow A$

Subgoal: *1.* $[\![B_1; \ldots ; B_n]\!] \Longrightarrow C$

Forward proofs: frule and drule

"Forward" rule: $A_1 \Longrightarrow A$

Subgoal: $1.\ [\![B_1; \ldots ; B_n]\!] \Longrightarrow C$

Substitution: $\sigma(B_i) \equiv \sigma(A_1)$

Forward proofs: frule and drule

"Forward" rule:	$A_1 \Longrightarrow A$
Subgoal:	$1.\ [\![B_1; \ldots ; B_n]\!] \Longrightarrow C$
Substitution:	$\sigma(B_i) \equiv \sigma(A_1)$
New subgoal:	$1.\ \sigma([\![B_1; \ldots ; B_n; A]\!] \Longrightarrow C)$

Forward proofs: frule and drule

"Forward" rule:	$A_1 \Longrightarrow A$
Subgoal:	$1.\ [\![B_1; \ldots ; B_n]\!] \Longrightarrow C$
Substitution:	$\sigma(B_i) \equiv \sigma(A_1)$
New subgoal:	$1.\ \sigma([\![B_1; \ldots ; B_n; A]\!] \Longrightarrow C)$

Command:

apply(*frule* $rulename$)

Forward proofs: frule and drule

"Forward" rule:	$A_1 \Longrightarrow A$
Subgoal:	$1.\ [\![B_1; \ldots ; B_n]\!] \Longrightarrow C$
Substitution:	$\sigma(B_i) \equiv \sigma(A_1)$
New subgoal:	$1.\ \sigma([\![B_1; \ldots ; B_n; A]\!] \Longrightarrow C)$

Command:

apply(*frule* $rulename$)

Like *frule* but also deletes B_i:

apply(*drule* $rulename$)

frule and drule: the general case

Rule: $[\![A_1; \ldots ; A_m]\!] \Longrightarrow A$

Creates additional subgoals:

$$
\begin{aligned}
&1.\ \sigma([\![B_1; \ldots ; B_n]\!] \Longrightarrow A_2) \\
&\vdots \\
&m\text{-}1.\ \sigma([\![B_1; \ldots ; B_n]\!] \Longrightarrow A_m) \\
&m.\ \sigma([\![B_1; \ldots ; B_n; A]\!] \Longrightarrow C)
\end{aligned}
$$

Forward proofs: OF

r[OF $r_1 \ldots r_n$*]*

Prove assumption 1 of theorem *r* with theorem r_1, and assumption 2 with theorem r_2, and . . .

Forward proofs: OF

r[OF $r_1 \ldots r_n$]

Prove assumption 1 of theorem *r* with theorem r_1, and assumption 2 with theorem r_2, and ...

Rule *r*	$[\![A_1; \ldots ; A_m]\!] \Longrightarrow A$
Rule r_1	$[\![B_1; \ldots ; B_n]\!] \Longrightarrow B$
Substitution	$\sigma(B) \equiv \sigma(A_1)$

r[OF r_1]

Forward proofs: OF

r[OF $r_1 \ldots r_n$]

Prove assumption 1 of theorem *r* with theorem r_1, and assumption 2 with theorem r_2, and ...

Rule *r*	$[\![A_1; \ldots ; A_m]\!] \Longrightarrow A$
Rule r_1	$[\![B_1; \ldots ; B_n]\!] \Longrightarrow B$
Substitution	$\sigma(B) \equiv \sigma(A_1)$
r[OF r_1]	$\sigma([\![B_1; \ldots; B_n; A_2; \ldots; A_m]\!] \Longrightarrow A)$

Clarifying the goal

Clarifying the goal

- **apply***(clarify)*
  Repeated application of safe rules
  without splitting the goal

Clarifying the goal

- **apply***(clarify)*
  Repeated application of safe rules
  without splitting the goal
- **apply***(clarsimp simp add: . . .)*
  Combination of *clarify* and *simp*.

Demo: proof methods

Sets

Overview

- Set notation
- Inductively defined sets

Set notation

Sets

Sets over type *'a*:

'a set

Sets

Sets over type *'a*:

$$\textit{'a set} \quad = \quad \textit{'a} \Rightarrow \textit{bool}$$

Sets

Sets over type *'a*:

$$\textit{'a set} \quad = \quad \textit{'a} \Rightarrow \textit{bool}$$

- $\{\}$, $\{e_1,\ldots,e_n\}$, $\{x.\ P\ x\}$

Sets

Sets over type *'a*:

$$\textit{'a set} \quad = \quad \textit{'a} \Rightarrow \textit{bool}$$

- $\{\}$, $\{e_1,\ldots,e_n\}$, $\{x.\ P\ x\}$
- $e \in A$, $A \subseteq B$

Sets

Sets over type *'a*:

$$\textit{'a set} \quad = \quad \textit{'a} \Rightarrow \textit{bool}$$

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

Sets

Sets over type *'a*:

$$\textit{'a set} \quad = \quad \textit{'a} \Rightarrow \textit{bool}$$

- $\{\}$, $\{e_1,\ldots,e_n\}$, $\{x.\ P\ x\}$
- $e \in A$, $A \subseteq B$
- $A \cup B$, $A \cap B$, A - B, - A
- $\bigcup x{\in}A\ B\ x$, $\bigcap x{\in}A\ B\ x$

Sets

Sets over type *'a*:

$$\textit{'a set} \quad = \quad \textit{'a} \Rightarrow \textit{bool}$$

- $\{\}$, $\{e_1,\ldots,e_n\}$, $\{x.\ P\ x\}$
- $e \in A$, $A \subseteq B$
- $A \cup B$, $A \cap B$, A - B, - A
- $\bigcup x{\in}A\ B\ x$, $\bigcap x{\in}A\ B\ x$
- *{i..j}*

Sets

Sets over type *'a*:

$$\textit{'a set} \quad = \quad \textit{'a} \Rightarrow \textit{bool}$$

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

Sets

Sets over type *'a*:

$$\textit{'a set} \quad = \quad \textit{'a} \Rightarrow \textit{bool}$$

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

Proofs about sets

Natural deduction proofs:

- `equalityI`: $[\![A \subseteq B; B \subseteq A]\!] \Longrightarrow A = B$

Proofs about sets

Natural deduction proofs:

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

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

Bounded quantifiers

- $\forall x \in A.\ P\ x$

Bounded quantifiers

- $\forall x{\in}A.\ P\ x \quad \equiv \quad \forall x.\ x{\in}A \longrightarrow P\ x$

Bounded quantifiers

- $\forall x{\in}A.\ P\ x \quad \equiv \quad \forall x.\ x{\in}A \longrightarrow P\ x$
- $\exists x{\in}A.\ P\ x$

Bounded quantifiers

- $\forall x{\in}A.\ P\ x \quad \equiv \quad \forall x.\ x{\in}A \longrightarrow P\ x$
- $\exists x{\in}A.\ P\ x \quad \equiv \quad \exists x.\ x{\in}A \wedge P\ x$

Bounded quantifiers

- $\forall x{\in}A.\ P\ x \quad \equiv \quad \forall x.\ x{\in}A \longrightarrow P\ x$
- $\exists x{\in}A.\ P\ x \quad \equiv \quad \exists x.\ x{\in}A \wedge P\ x$
- `ballI`: $(\bigwedge x.\ x \in A \Longrightarrow P\ x) \Longrightarrow \forall x{\in}A.\ P\ x$
- `bspec`: $[\![\forall x{\in}A.\ P\ x; x \in A]\!] \Longrightarrow P\ x$

Bounded quantifiers

- $\forall x{\in}A.\ P\ x \quad \equiv \quad \forall x.\ x{\in}A \longrightarrow P\ x$
- $\exists x{\in}A.\ P\ x \quad \equiv \quad \exists x.\ x{\in}A \wedge P\ x$
- `ballI`: $(\bigwedge x.\ x \in A \Longrightarrow P\ x) \Longrightarrow \forall x{\in}A.\ P\ x$
- `bspec`: $[\![\forall x{\in}A.\ P\ x; x \in A]\!] \Longrightarrow P\ x$
- `bexI`: $[\![P\ x; x \in A]\!] \Longrightarrow \exists x{\in}A.\ P\ x$
- `bexE`: $[\![\exists x{\in}A.\ P\ x; \bigwedge x.\ [\![x \in A; P\ x]\!] \Longrightarrow Q]\!] \Longrightarrow Q$

Inductively defined sets

Example: even numbers

Informally:

Example: even numbers

Informally:

- 0 is even

Example: even numbers

Informally:

- 0 is even
- If n is even, so is $n + 2$

Example: even numbers

Informally:

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

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_set *Ev :: nat set* — The set of all even numbers

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_set *Ev :: nat set* — The set of all even numbers

where

0 ∈ Ev |

n ∈ Ev ⟹ n + 2 ∈ Ev

Format of inductive definitions

inductive_set $S :: \tau$ *set*

Format of inductive definitions

inductive_set $S :: \tau\ set$

where

$\llbracket\ a_1 \in S; \ldots ; a_n \in S; A_1; \ldots; A_k\ \rrbracket \Longrightarrow a \in S$ |

$\vdots$

Format of inductive definitions

inductive_set $S :: \tau$ *set*

where

$$[\![\, a_1 \in S; \ldots ; a_n \in S; A_1; \ldots; A_k \,]\!] \Longrightarrow a \in S \;\;|$$

$$\vdots$$

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

Proving properties of even numbers

Easy: $4 \in Ev$

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

Proving properties of even numbers

Easy: $4 \in Ev$

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

Trickier: $m \in Ev \Longrightarrow m+m \in Ev$

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 $m \in Ev$

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

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 $0 \in Ev$

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 $0 \in Ev$
  $\Longrightarrow m = 0 \Longrightarrow 0+0 \in Ev$

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 $n \in Ev \Longrightarrow n+2 \in Ev$

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.!)

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

Rule induction for Ev

To prove

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

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

- $P\ 0$

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 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`:

$$\llbracket\, n \in Ev;\ P\ 0;\ \bigwedge n.\ P\ n \Longrightarrow P(n+2) \,\rrbracket \Longrightarrow P\ n$$

Rule induction in general

Set S is defined inductively.

Rule induction in general

Set S is defined inductively.
To prove

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

by *rule induction* on $x \in S$

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

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

that P is preserved:

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

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

that P is preserved:

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

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

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

that P is preserved:

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

In Isabelle/HOL:

apply*(induct rule: S.induct)*

Demo: inductively defined sets

Inductive predicates

$$x \in S \quad \rightsquigarrow \quad S\,x$$

Inductive predicates

$$x \in S \quad \rightsquigarrow \quad S\,x$$

Example:

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

where

Ev 0 |

Ev n $\Longrightarrow$ *Ev (n + 2)*

Inductive predicates

$$x \in S \quad \rightsquigarrow \quad S\,x$$

Example:

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

where

Ev 0 |

Ev n $\Longrightarrow$ *Ev (n + 2)*

Comparison:

predicate: simpler syntax

set: direct usage of $\cup$ etc

Inductive predicates

$$x \in S \quad \rightsquigarrow \quad S\,x$$

Example:

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

where

Ev 0 |

Ev n $\Longrightarrow$ *Ev (n + 2)*

Comparison:

predicate: simpler syntax

set: direct usage of $\cup$ etc

Inductive predicates can be of type $\tau_1 \Rightarrow ... \Rightarrow \tau_n \Rightarrow$ *bool*

Automating it

simp and auto

simp rewriting and a bit of arithmetic

auto rewriting and a bit of arithmetic, logic & sets

simp and auto

simp rewriting and a bit of arithmetic

auto rewriting and a bit of arithmetic, logic & sets

- Show you where they got stuck

simp and auto

simp rewriting and a bit of arithmetic

auto rewriting and a bit of arithmetic, logic & sets

- Show you where they got stuck
- highly incomplete wrt logic

blast

- A complete (for FOL) tableaux calculus implementation

blast

- A complete (for FOL) tableaux calculus implementation
- Covers logic, sets, relations, . . .

blast

- A complete (for FOL) tableaux calculus implementation
- Covers logic, sets, relations, . . .
- Extensible with intro/elim rules

blast

- A complete (for FOL) tableaux calculus implementation
- Covers logic, sets, relations, . . .
- Extensible with intro/elim rules
- Almost no “=”

Demo: blast

blast: A 2-stage implementation

1. Search for proof with quick, dirty and possibly buggy program

blast: A 2-stage implementation

1. Search for proof with quick, dirty and possibly buggy program (while recording proof tree)

blast: A 2-stage implementation

1. Search for proof with quick, dirty and possibly buggy program (while recording proof tree)
2. Check proof tree with Isabelle kernel

blast: A 2-stage implementation

1. Search for proof with quick, dirty and possibly buggy program (while recording proof tree)
2. Check proof tree with Isabelle kernel

This is the *LCF-architecture* by Robin Milner:

blast: A 2-stage implementation

1. Search for proof with quick, dirty and possibly buggy program (while recording proof tree)
2. Check proof tree with Isabelle kernel

This is the *LCF-architecture* by Robin Milner:

- Proofs only constructable via small trustworthy kernel

blast: A 2-stage implementation

1. Search for proof with quick, dirty and possibly buggy program (while recording proof tree)
2. Check proof tree with Isabelle kernel

This is the *LCF-architecture* by Robin Milner:

- Proofs only constructable via small trustworthy kernel
- Proof search programmable in ML on top

blast: A 2-stage implementation

1. Search for proof with quick, dirty and possibly buggy program (while recording proof tree)
2. Check proof tree with Isabelle kernel

This is the *LCF-architecture* by Robin Milner:

- Proofs only constructable via small trustworthy kernel
- Proof search programmable in ML on top

Isabelle follows the LCF architecture

fast and friends

fast slow and incomplete version of *blast*

fast and friends

fast slow and incomplete version of *blast*

fastsimp rewriting and logic

fast and friends

fast slow and incomplete version of *blast*

fastsimp rewriting and logic

force slower but completer version of *fastsimp*

metis

- An fast resolution theorem prover in ML

metis

- An fast resolution theorem prover in ML
- Can deal with bidirectional “=”

metis

- An fast resolution theorem prover in ML
- Can deal with bidirectional “=”
- Knows only pure logic, not sets etc

Demo: beyond blast

Problems

- Most proofs require additional lemmas.

Problems

- Most proofs require additional lemmas.
- Adding arbitrary lemmas slows *blast* down significantly;

Problems

- Most proofs require additional lemmas.
- Adding arbitrary lemmas slows *blast* down significantly; *metis* copes better.

Problems

- Most proofs require additional lemmas.
- Adding arbitrary lemmas slows *blast* down significantly; *metis* copes better.
- Finding the right lemmas in a library of thousands of lemmas is light years beyond *blast* and *metis*.

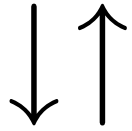
Problems

- Most proofs require additional lemmas.
- Adding arbitrary lemmas slows *blast* down significantly; *metis* copes better.
- Finding the right lemmas in a library of thousands of lemmas is light years beyond *blast* and *metis*.
- There are highly optimized *ATPs* (automatic theorem provers) for FOL that can deal with large libraries . . .

Sledgehammer

Architecture

Isabelle

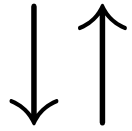


ATPs

Architecture

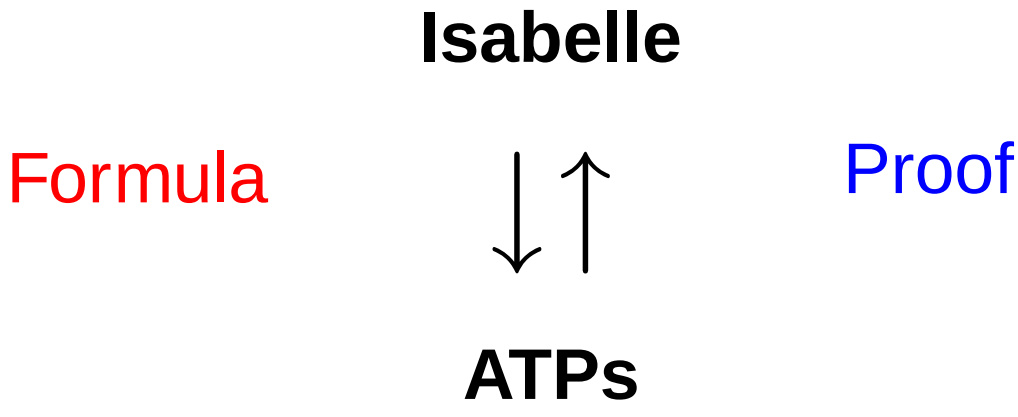
Isabelle

Formula

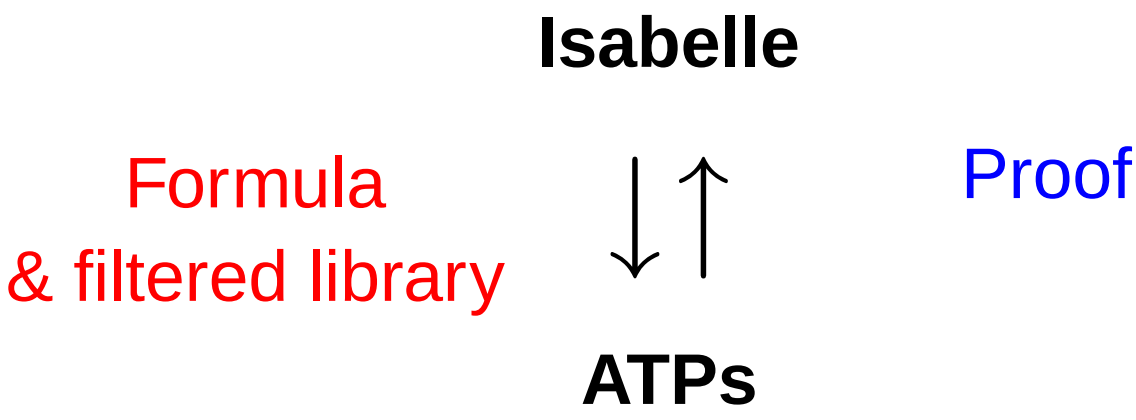


ATPs

Architecture



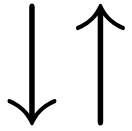
Architecture



Architecture

Isabelle

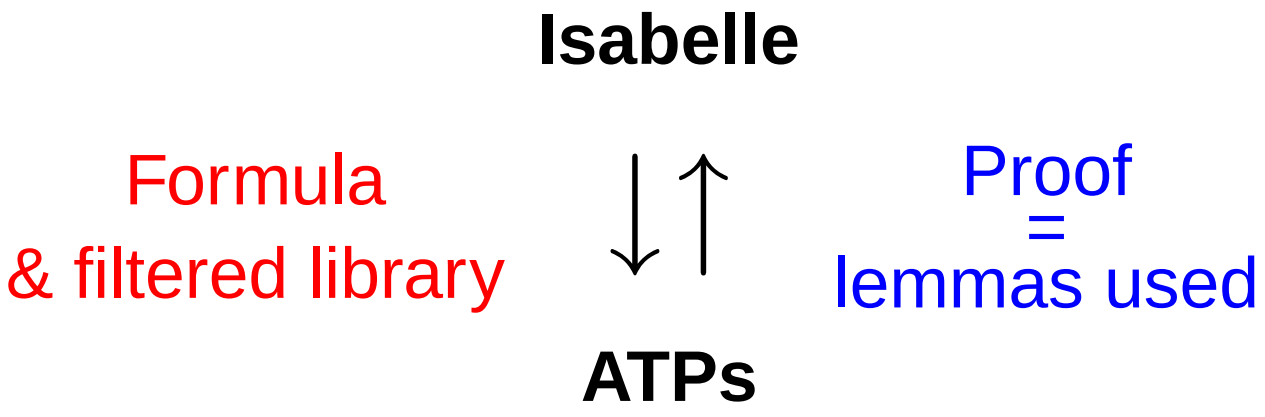
Formula
& filtered library



Proof
=
lemmas used

ATPs

Architecture



Empirical study:

Sledgehammer works for 1/3 of non-trivial Isabelle proofs

Demo: Sledghehammer

Automating Arithmetic

Automating arithmetic

arith:

Automating arithmetic

arith:

- proves linear formulas (no “$*$”)

Automating arithmetic

arith:

- proves linear formulas (no "$*$")
- complete for quantifier-free *real* arithmetic

Automating arithmetic

arith:

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

Automating arithmetic

Theorem (Tarski) $Th(\mathbb{R}, +, -, *, <, =)$ is decidable.

Automating arithmetic

Theorem (Tarski) $Th(\mathbb{R}, +, -, *, <, =)$ is decidable.

An incomplete but (often) fast method for the quantifier-free fragement:

sos

Automating arithmetic

Theorem (Tarski) $Th(\mathbb{R}, +, -, *, <, =)$ is decidable.

An incomplete but (often) fast method for the quantifier-free fragement:

sos

Idea: (re)write polynomials as sums-of-squares to prove non-negativity

Demo: Arithmetic

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*

$\vdots$

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 (⋀)
| **assume** prop (⟹)
| [**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* $\Rightarrow$ *'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* $\Rightarrow$ *'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*)

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* $\Rightarrow$ *'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*

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:* $\neg$ *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*
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 ⇒ 'a set*
  assumes *s: surj f*
  shows *False*

Example: Structured lemma statement

lemma *Cantor':*
  fixes *f :: 'a ⇒ '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 ⇒ '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 $surj\ f \Longrightarrow False$

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
  $\vdots$
  show Q ...
next
  assume Q
  $\vdots$
  show P ...
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

Propositional proof patterns

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

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

show $A \subseteq B$
proof
  fix x
  assume $x \in A$
  ⋮
  show $x \in B$ ...
qed

Propositional proof patterns

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

show P
proof *(rule ccontr)*
  assume $\neg P$
  ⋮
  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 $\exists x.\ P(x)$
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'':* $\neg$ *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*
  hence *a $\notin$ f a $\longleftrightarrow$ a $\in$ f a* **by** *blast*

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*
  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. $\bigwedge x_1 \ldots x_n \llbracket A_1; \ldots ; A_m \rrbracket \Longrightarrow A$

proof method

Applies method and generates subgoal(s):

1. $\bigwedge x_1 \ldots x_n \llbracket A_1; \ldots ; A_m \rrbracket \Longrightarrow A$

How to prove each subgoal:

proof method

Applies method and generates subgoal(s):

1. $\bigwedge x_1 \ldots x_n \llbracket A_1; \ldots ; A_m \rrbracket \Longrightarrow A$

How to prove each subgoal:

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

proof method

Applies method and generates subgoal(s):

1. $\bigwedge x_1 \ldots x_n \llbracket A_1; \ldots ; A_m \rrbracket \Longrightarrow A$

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 ($C_1\ \vec{x}$)
  $\ldots\ \vec{x}\ \ldots$
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)*
  . . .
  . . . *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)* $\equiv$ **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)*
  . . .
  . . . n . . .
  . . .
  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)*
  . . .
  . . . *n* . . .
  . . .
  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)*
  . . . *n* . . . *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*$_1$ ... *x*$_k$*)*

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

proof *(induct* x_1 . . . 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*(. . .) . . .

proof *(induct* x_1 . . . 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 "$t_0 = t_1$" . . .

also

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

also

have "$t_0 = t_1$" ...

also

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

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

also

⋮

also

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

also

have "$t_0 = t_1$" ...
also
have "... = t_2" ... ... $\equiv t_1$
also
⋮
also
have "... = t_n" ... ... $\equiv t_{n-1}$
finally show ...

also

have "$t_0 = t_1$" ...
also
have "... = t_2" ... ... $\equiv t_1$
also
⋮
also
have "... = t_n" ... ... $\equiv t_{n-1}$
finally show ...
— like **from** ‘$t_0 = t_n$‘ **show**

also

- “. . .” is merely an abbreviation

also

- “. . .” is merely an abbreviation
- **also** works for other transitive relations ($<$, $\leq$, . . .)

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
⋮
moreover
have $formula_n$ **. . .**
ultimately show . . .

— like **from** $f_1 \ldots f_n$ **show** but needs no labels

Demo: moreover