An Introduction to Isabelle/HOL 2009-1

Tobias Nipkow

TU München

Overview of Isabelle/HOL

System Architecture

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

HOL

HOL = Higher-Order Logic

HOL = Functional programming + Logic

HOL has

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

HOL is a programming language!

Higher-order = functions are values, too!

Formulae

Syntax (in decreasing priority):

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

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

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:

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

Types and Terms

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:

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

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

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

λ-*calculus on one slide*

Informal notation: $t[x]$

- *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)

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

Type inference

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

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

User can help with type annotations inside the term.

Example: *f (x::nat)*

Currying

Thou shalt curry your functions

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

Advantage: *partial application* *f* a_1 with a_1 :: τ_1

Terms: Syntactic sugar

Some predefined syntactic sugar:

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

Prefix binds more strongly than infix:

! $f\ x + y \equiv (f\ x) + y \neq f\ (x + y)$!

Enclose *if* and *case* in parentheses:

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

Base types: bool, nat, list

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

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

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

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*]*

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.

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	$\forall$	$\exists$	λ	$\neg$	$\wedge$	$\vee$	$\longrightarrow$	$\Rightarrow$
ascii (1)	\<forall>	\<exists>	\<lambda>	\<not>	/\	\/	-->	=>
ascii (2)	ALL	EX	%	~	&	$\vert$		

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

Demo: terms and types

An introduction to recursion and induction

A recursive datatype: toy lists

datatype *'a list = Nil | Cons 'a ('a list)*

***Nil*:** empty list

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

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

Predefined lists: *[False, True]*

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 = ??

1 rule per constructor
Recursive calls must drop the constructor ⟹ Termination

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 `(...)`
$\vdots$
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.

Allows top down development:

Assume lemma first, prove it later.

Some useful tools

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

Equality $\equiv$ (`==`)

For definitions

Universal quantifier $\bigwedge$ (`!!`)

For binding local variables

Do not use *inside* HOL formulae

Notation

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

abbreviates

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

$;\quad \approx \quad$ "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

Example:

typedecl *addr* — An abstract type of addresses

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: $Nil :: 'a\ list$
  $Cons :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$
- Distinctness: $Nil \neq Cons\ x\ xs$
- Injectivity: $(Cons\ x\ xs = Cons\ y\ ys) = (x = y \wedge xs = ys)$

The general case

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

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

Distinctness and Injectivity are applied automatically
Induction must be applied explicitly

Function definition in Isabelle/HOL

Why nontermination can be harmful

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

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
- 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* $\Rightarrow$ *nat* **where** *sq n* = *n* * *n*

Definitions: pitfalls

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

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

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*:

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

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*

Functions on *nat* definable by primrec!

primrec *f :: nat* $\Rightarrow$ *...*
f 0 = ...
f(Suc n) = ... f n ...

More predefined types and functions

Type option

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

Important application:

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

$$\begin{aligned} \textit{None} \quad &\approx \quad \text{no result} \\ \textit{Some a} \quad &\approx \quad \text{result } a \end{aligned}$$

Example:

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

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

Wildcards:

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

Nested patterns:

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

Complicated patterns mean complicated proofs!

Needs *( )* in context

Proof by case distinction

If t :: τ 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* $\Rightarrow$ *'a list* $\Rightarrow$ *'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
- Order of equations matters
- Termination must be provable
  by lexicographic combination of size measures

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:

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

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)

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 $x :: \tau$:

> 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)

$\leadsto$ 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

As long as possible

Terminology: equation $\leadsto$ *rewrite rule*

An example

Equations:

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

Rewriting:

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

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:

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

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

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

Interlude: Variables in Isabelle

Schematic variables

Three kinds of variables:

- bound: $\forall x.\ x = x$
- free: $x = x$
- schematic: $?x = ?x$ (“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*

⋮

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.* $\llbracket P_1; \ldots ; P_m \rrbracket \Longrightarrow C$

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.

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.

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

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

Automatic

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

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

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

Ordered rewriting

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

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

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

For types *nat*, *int* etc:

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

Example: *(simp add: add_ac)* yields

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

Preprocessing

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

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

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 → Settings → Trace simplifier

Output in separate `trace` buffer

Demo: simp

Induction heuristics

Basic heuristics

Theorems about recursive functions are proved by induction

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

A tail recursive reverse

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
- 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 $\oplus$:

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

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

Natural deduction for propositional logic

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

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

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

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

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

Operational reading

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

Introduction rule:

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

Elimination rule

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

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

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

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

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:

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

Working backwards, like in Prolog!

Example: rule: $[\![?P; ?Q]\!] \Longrightarrow ?P \land ?Q$

subgoal: 1. $A \land B$

Result: 1. A

2. B

Rule application: the details

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

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

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

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

$\vdots$

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

Command:

apply(*rule <rulename>*)

Proof by assumption

apply *assumption*

proves

$$1.\ \llbracket\, B_1; \ldots ; B_m \,\rrbracket \Longrightarrow C$$

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

Example:

Rule: $[\![?P \wedge ?Q; [\![?P; ?Q]\!] \Longrightarrow ?R]\!] \Longrightarrow ?R$

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

Unification: $?P \wedge ?Q \equiv A \wedge B$ and $?R \equiv Z$

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

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

How to prove it by natural deduction

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

  apply(*rule <intro-rule>*)

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

  apply(*erule <elim-rule>*)

Demo: 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. $\bigwedge x_1 \dots x_n$*. Formula*

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

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

Scope

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

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

means

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

α-Conversion

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

Natural deduction for quantifiers

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

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

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

Instantiating rules

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

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

Similar: *erule_tac*

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

A quantifier proof

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

apply(*rule allI*)

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

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

1. $\bigwedge a. a = a$

apply(*rule refl*)

Demo: quantifier proofs

More proof methods

Forward proofs: frule and drule

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

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:

1. $\sigma([\![B_1; \ldots ; B_n]\!] \Longrightarrow A_2)$

$\vdots$

m-1. $\sigma([\![B_1; \ldots ; B_n]\!] \Longrightarrow A_m)$

m. $\sigma([\![B_1; \ldots ; B_n; A]\!] \Longrightarrow C)$

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

- **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*:

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

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

Proofs about sets

Natural deduction proofs:

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

Demo: proofs about sets

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:

- 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
$0 \in Ev \quad |$
$n \in Ev \Longrightarrow n + 2 \in Ev$

Format of inductive definitions

inductive_set *S* :: τ *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: $4 \in Ev$

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

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

Idea: induction on the length of the derivation of $m \in Ev$

Better: induction on the *structure* of the derivation

Two cases: $m \in Ev$ is proved by

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

Rule induction for Ev

To prove

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

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

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

Rule `Ev.induct`:

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

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

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

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

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

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 (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*

fastsimp rewriting and logic

force slower but completer version of *fastsimp*

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.
- 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

Formula
& filtered library

$\downarrow \uparrow$

Proof
=
lemmas used

ATPs

Empirical study:

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

Demo: Sledghehammer

Automating 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.

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
- hard to maintain
- do not scale

No structure!

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

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

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)*
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

=

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

Example: Structured lemma statement

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

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

- **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 $\ldots$
next
  assume Q
  $\vdots$
  show P $\ldots$
qed

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

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

Propositional proof patterns

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

Case distinction

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

Case distinction

show P
proof *(rule ccontr)*
  assume $\neg P$
  ⋮
  show *False* $\ldots$
qed

Contradiction

Quantifier introduction proof patterns

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

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

$\exists$ *elimination:* obtain

have $\exists x.\ P(x)$
then obtain x **where** p: $P(x)$ **by** blast
$\vdots$ 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)*
  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

Applies method and generates subgoal(s):

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

How to prove each subgoal:

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

Separated by **next**

Demo: proof

Streamlining proofs: Pattern matching and Quotations

Example: pattern matching

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

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

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

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

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

Induction

Overview

- Structural induction
- Rule induction
- Induction with fun

Structural induction for type nat

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

Demo: structural induction

Structural induction with $\Longrightarrow$

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

case 0	$\equiv$	**fix** x **assume** $0{:}\ A(0)$
. . .		**let** $?case = P(0)$
show $?case$		

next

case $(Suc\ n)$	$\equiv$	**fix** n
. . .		**assume** $Suc{:}\ A(n) \Longrightarrow P(n)$
$\cdots\ n\ \cdots$		$A(Suc\ n)$
. . .		**let** $?case = P(Suc\ n)$
show $?case$		

qed

A remark on style

- **case** *(Suc n)* . . . **show** *?case*
  is easy to write and maintain
- **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$_1$*:* $\llbracket$ $s \in S$; A $\rrbracket \Longrightarrow s' \in S$

$\vdots$

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: $x \in S$

$\vdots$

show $P(x)$

using A **proof** *induct*

$\vdots$

qed

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

using A **proof** *induct*

$\vdots$

qed

Renaming free variables in rule

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

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

Demo: rule induction

Induction with fun

Definition:

fun *f*

$\vdots$

Proof:

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

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

  case *1*

  $\vdots$

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

also

$\vdots$

also

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

finally show . . .

— like **from** ‘t_0 = t_n‘ **show**

also

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

Demo: also

Accumulating facts

moreover

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

Demo: moreover