An Introduction to Isabelle/HOL
2009-1

Tobias Nipkow
TU München
Overview of Isabelle/HOL
<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProofGeneral</td>
<td>(X)Emacs based interface</td>
</tr>
<tr>
<td>Isabelle/HOL</td>
<td>Isabelle instance for HOL</td>
</tr>
<tr>
<td>Isabelle</td>
<td>Generic theorem prover</td>
</tr>
</tbody>
</table>
HOL

HOL = Higher-Order Logic
HOL = Functional programming + Logic

HOL has

- datatypes
- recursive functions
- logical operators ($\land$, $\rightarrow$, $\forall$, $\exists$, ...)

HOL is a programming language!

Higher-order = functions are values, too!
**Formulae**

**Syntax** (in decreasing priority):

\[ \text{form} ::= (\text{form}) \quad | \quad \text{term} = \text{term} \quad | \quad \neg \text{form} \]
\[ | \quad \text{form} \land \text{form} \quad | \quad \text{form} \lor \text{form} \quad | \quad \text{form} \rightarrow \text{form} \]
\[ | \quad \forall x. \text{form} \quad | \quad \exists x. \text{form} \]

**Examples**

- \( \neg A \land B \lor C \equiv ((\neg A) \land B) \lor C \)
- \( A = B \land C \equiv (A = B) \land C \)
- \( \forall x. P x \land Q x \equiv \forall x. (P x \land Q x) \)

**Scope** of quantifiers: as far to the right as possible
Abbreviation: $\forall x \ y. \ P x \ y \equiv \forall x. \ \forall y. \ P x \ y$  
($\forall$, $\exists$, $\lambda$, …)

Parentheses:

- $\land$, $\lor$ and $\rightarrow$ associate to the right:
  $A \land B \land C \equiv A \land (B \land C)$

- $A \rightarrow B \rightarrow C \equiv A \rightarrow (B \rightarrow C) \neq (A \rightarrow B) \rightarrow C$ !
Warning

Quantifiers have low priority and need to be parenthesized:

\[ \neg P \wedge \forall x. Q x \quad \sim \quad P \wedge (\forall x. Q x) \quad \neg \]
Types and Terms
Types

Syntax:

\[ \tau ::= (\tau) \]

<table>
<thead>
<tr>
<th>\textit{base types}</th>
<th>\textit{type variables}</th>
<th>\textit{total functions}</th>
<th>\textit{pairs (ascii: \texttt{*})}</th>
<th>\textit{lists}</th>
<th>\textit{user-defined types}</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>nat</td>
<td>...</td>
<td>'a</td>
<td>'b</td>
<td>...</td>
</tr>
</tbody>
</table>

Parentheses: \[ T1 \Rightarrow T2 \Rightarrow T3 \equiv T1 \Rightarrow (T2 \Rightarrow T3) \]
Terms: Basic syntax

Syntax:

\[
\text{term} ::= \ (\text{term}) \\
| \ a \quad \text{constant or variable (identifier)} \\
| \ \text{term} \ \text{term} \quad \text{function application} \\
| \ \lambda x. \ \text{term} \quad \text{function “abstraction”} \\
| \ \ldots \quad \text{lots of syntactic sugar}
\]

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

Parantheses: \( f \ a_1 \ a_2 \ a_3 \equiv (f \ a_1) \ a_2) \ a_3 \)
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 ("$\beta$-reduction"): $(\lambda x. t[x])\ a \longrightarrow_\beta t[a]$  

Example: $(\lambda\ x. x + 5)\ 3 \longrightarrow_\beta (3 + 5)$
Isabelle performs $\beta$-reduction automatically.

Isabelle considers $(\lambda x.t[x])a$ and $t[a]$ equivalent.
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 \( \tau \).
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: \( f (x::\text{nat}) \)
Currying

Thou shalt curry your functions

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

Advantage: *partial application* $f a_1$ with $a_1 :: \tau_1$
Some predefined syntactic sugar:

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

Prefix binds more strongly than infix:

\[
! f \cdot x + y \equiv (f \cdot x) + y \neq f (x + y)
\]

Enclose *if* and *case* in parentheses:

\[
! (if _ \text{ then } _ \text{ else } _)
\]
Base types: bool, nat, list
Type bool

Formulae = terms of type bool

True :: bool
False :: bool
∧, ∨, . . . :: bool ⇒ bool ⇒ bool

if-and-only-if: =
**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)

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

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

Large library:

*hd, tl, map, length, filter, set, nth, take, drop, distinct, \ldots*

Don’t reinvent, reuse!

\(\sim\) HOL/List.thy
Isabelle Theories
Theory = Module

Syntax:

theory MyTh
imports ImpTh₁ ... ImpThₙ
begin
(declarations, definitions, theorems, proofs, ...)*
end

• MyTh: name of theory. Must live in file MyTh.thy
• ImpThᵢ: 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: `/\`, `\`, `--->`, ...

<table>
<thead>
<tr>
<th>x-symbol</th>
<th>( \forall )</th>
<th>( \exists )</th>
<th>( \lambda )</th>
<th>( \neg )</th>
<th>( \land )</th>
<th>( \lor )</th>
<th>( \rightarrow )</th>
<th>( \Rightarrow )</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascii (1)</td>
<td><code>\forall</code></td>
<td><code>\exists</code></td>
<td><code>\lambda</code></td>
<td><code>\neg</code></td>
<td><code>\land</code></td>
<td><code>\lor</code></td>
<td><code>\rightarrow</code></td>
<td><code>\Rightarrow</code></td>
</tr>
<tr>
<td>ascii (2)</td>
<td>ALL</td>
<td>EX</td>
<td>%</td>
<td>~</td>
<td>&amp;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(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*:

```plaintext
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 (...)  
...
done
```

If the lemma is suitable as a simplification rule:

```
lemma name [simp] : "..."
```
Proof methods

• Structural induction
  • Format: \( \text{induct } x \)
    \( x \) 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: \textit{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 \( \implies \) (\(\Rightarrow\))
For separating premises and conclusion of theorems

Equality \( \equiv \) (\(==\))
For definitions

Universal quantifier \( \forall \) (\(!!\))
For binding local variables

Do not use inside HOL formulae
Notation

\[
[ A_1; \ldots ; A_n ] \implies B
\]

abbreviates

\[
A_1 \implies \ldots \implies A_n \implies B
\]

; \quad \approx \quad \text{“and”}
The proof state

1. $\bigwedge x_1 \ldots x_p. \left[ A_1; \ldots ; A_n \right] \Rightarrow B$

- $x_1 \ldots x_p$  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
Introduces an *abbreviation* `name` for type `τ`

Examples:

```plaintext
types
  name = string
  (’a,’b)foo = ’a list × ’b list
```

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

**types**

```plaintext
name = τ
```
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 \((\alpha_1, \ldots, \alpha_n)\tau\) = \(C_1 \tau_{1,1} \cdots \tau_{1,n_1}\)
\[
\begin{array}{c}
| \\
| \\
| \\
| \\
| \\
| \\
| \\
| \\
| \\
| \\
| \\
\end{array}
\]
\(\cdots\)
\(C_k \tau_{k,1} \cdots \tau_{k,n_k}\)

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

Distinctness and Injectivity are applied automatically
Induction must be applied explicitly
Function definition in Isabelle/HOL
Why nontermination can be harmful

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

Subtract $f \times$ on both sides.

$\Rightarrow 0 = 1$

! All functions in HOL must be total!
Function definition schemas in Isabelle/HOL

- Non-recursive with \texttt{definition}
  No problem
- Primitive-recursive with \texttt{primrec}
  Terminating by construction
- Well-founded recursion with \texttt{fun}
  Automatic termination proof
- Well-founded recursion with \texttt{function}
  User-supplied termination proof
definition
Definition (non-recursive) by example

\[
\text{definition } \texttt{sq} :: \text{nat} \Rightarrow \text{nat} \text{ where } \texttt{sq} \ n = n \ast n
\]
**Definitions: pitfalls**

Definition: $\text{prime :: nat } \Rightarrow \text{bool where}$

$\text{prime } p = (1 < p \land (m \text{ dvd } p \rightarrow m = 1 \lor m = p))$

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

![Every free variable on the rhs must occur on the lhs](image)

$\text{prime } p = (1 < p \land (\forall m. m \text{ dvd } p \rightarrow m = 1 \lor m = p))$
Definitions are not used automatically

Unfolding the definition of $sq$

\texttt{apply(\textit{unfold \textit{sq\_def})}}
primrec
The example

primrec app :: 'a list ⇒ 'a list ⇒ 'a list where
app Nil ys = ys |
app (Cons x xs) ys = Cons x (app xs ys)
The general case

If \( \tau \) 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*:

\[
\begin{align*}
  f \mathrel{ x_1 \cdots (C_1 y_{1,1} \cdots y_{1,n_1}) \cdots x_p} & = r_1 \\
  \vdots \\
  f \mathrel{ x_1 \cdots (C_k y_{k,1} \cdots y_{k,n_k}) \cdots x_p} & = r_k
\end{align*}
\]

The recursive calls in \( r_i \) must be *structurally smaller*, i.e. of the form \( f \mathrel{ a_1 \cdots y_{i,j} \cdots a_p} \)
nat is a datatype

datatype \texttt{nat} = 0 | Suc \texttt{nat}

Functions on \texttt{nat} definable by primrec!

\texttt{primrec} \ f :: \texttt{nat} \Rightarrow ... 
\ f \ 0 = ... 
\ f(Suc \ n) = ... \ f \ n ...
More predefined types and functions
**Type option**

```plaintext
datatype 'a option = None | Some 'a
```

Important application:

\[ \ldots \Rightarrow 'a \text{ option} \approx \text{partial function:} \]

\[ \quad \text{None} \approx \text{no result} \]
\[ \quad \text{Some } a \approx \text{result } a \]

Example:

```plaintext
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)
```
Datatype values can be taken apart with case expressions:

\[(\text{case } xs \text{ of } [] \Rightarrow \ldots \mid y\#ys \Rightarrow \ldots y \ldots ys \ldots)\]

Wildcards:

\[(\text{case } xs \text{ of } [] \Rightarrow [] \mid y\#_\Rightarrow [y])\]

Nested patterns:

\[(\text{case } xs \text{ of } [0] \Rightarrow 0 \mid [\text{Suc } n] \Rightarrow n \mid _\Rightarrow 2)\]

Complicated patterns mean complicated proofs!

Needs ( ) in context
Proof by case distinction

If \( t :: \tau \) and \( \tau \) is a datatype

\[
\text{apply(} \text{case_tac } t \text{)}
\]

creates \( k \) subgoals

\[
t = C_i \ x_1 \ldots x_p \implies \ldots
\]

one for each constructor \( C_i \) of type \( \tau \).
Demo: trees
fun

From primitive recursion
to arbitrary pattern matching
fun fib :: nat ⇒ nat where

fib 0 = 0  |
fib (Suc 0) = 1  |
fib (Suc(Suc n)) = fib (n+1) + fib n
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)
fun ack :: nat ⇒ nat ⇒ 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

- \( \text{size}(n::\text{nat}) = n \)
- \( \text{size}(xs) = \text{length } xs \)
- \( \text{size} \) 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

\[
\begin{align*}
ack 0 n &= \text{Suc } n \\
ack (\text{Suc } m) 0 &= \ack m \ (\text{Suc } 0) \\
ack (\text{Suc } m) \ (\text{Suc } n) &= \ack m \ (\ack (\text{Suc } m) \ n)
\end{align*}
\]

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 \texttt{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
fun div2 :: nat ⇒ nat where
  div2 0 = 0 |
  div2 (Suc 0) = 0 |
  div2(Suc(Suc n)) = Suc(div2 n)

\(\Rightarrow\) induction rule div2.induct:

\[
\begin{align*}
P(0) & \quad P(Suc\ 0) & \quad P(n) \implies P(Suc(Suc\ n)) \\
& \quad P(m) \quad
\end{align*}
\]
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:

\[ 0 + n = n \quad (1) \]
\[ (\text{Suc } m) + n = \text{Suc } (m + n) \quad (2) \]
\[ (\text{Suc } m \leq \text{Suc } n) = (m \leq n) \quad (3) \]
\[ (0 \leq m) = \text{True} \quad (4) \]

Rewriting:

\[ 0 + \text{Suc } 0 \leq \text{Suc } 0 + x \quad (1) \]
\[ \text{Suc } 0 \leq \text{Suc } 0 + x \quad (2) \]
\[ \text{Suc } 0 \leq \text{Suc } (0 + x) \quad (3) \]
\[ 0 \leq 0 + x \quad (4) \]
\[ \text{True} \]
More formally

substitution = mapping from variables to terms

- \( l = r \) is applicable to term \( t[s] \) if there is a substitution \( \sigma \) 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:

\[ [P_1 \ldots P_n] \implies l = r \]

is *applicable* to term \( t[s] \) with \( \sigma \) if

- \( \sigma(l) = s \) and
- \( \sigma(P_1), \ldots, \sigma(P_n) \) 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

\[ \text{rev}(a @ []) = \text{rev} a \]

using app NIL2 with \( \sigma = \{ ?xs \mapsto a \} \)
Term rewriting in Isabelle
Basic simplification

Goal: 1. \( [ P_1; \ldots ; P_m ] \rightarrow C \)

\textbf{apply}(simp add: eq_1 \ldots eq_n)

Simplify \( P_1 \ldots P_m \) and \( C \) using

- lemmas with attribute simp
- rules from \texttt{primrec}, \texttt{fun} and \texttt{datatype}
- additional lemmas \( eq_1 \ldots eq_n \)
- assumptions \( P_1 \ldots P_m \)

Variations:

- \((simp \ldots del: \ldots)\) removes simp-lemmas
- \texttt{add} and \texttt{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: \( f(x) = g(x), \ g(x) = f(x) \)

\[
[P_1 \ldots P_n] \implies l = r
\]

is suitable as a simp-rule only if \( l \) is “bigger” than \( r \) and each \( P_i \)

\[
n < m \implies (n < Suc m) = True \quad YES
\]
\[
Suc n < m \implies (n < m) = True \quad NO
\]
Rewriting with definitions

Definitions do not have the $simp$ attribute.

They must be used explicitly: $(simp\ add:\ f\_def\ \ldots)$
Extensions of rewriting
Local assumptions

Simplification of \( A \rightarrow B \):  
1. Simplify \( A \) to \( A' \)  
2. Simplify \( B \) using \( A' \)
Case splitting with simp

\[
P(\text{if } A \text{ then } s \text{ else } t) = (A \rightarrow P(s)) \land (\neg A \rightarrow P(t))
\]

Automatic

\[
P(\text{case } e \text{ of } 0 \Rightarrow a \mid \text{Suc } n \Rightarrow b) = (e = 0 \rightarrow P(a)) \land (\forall n. e = \text{Suc } n \rightarrow P(b))
\]

By hand: \texttt{(simp split: nat.split)}

Similar for any datatype \( t \): \texttt{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:

\[-A \leftrightarrow A = False\]

\[A \rightarrow B \leftrightarrow A \Rightarrow B\]

\[A \land B \leftrightarrow A, B\]

\[\forall x.A(x) \leftrightarrow A(?x)\]

\[A \leftrightarrow A = True\]

Example:

\[(p \rightarrow q \land \neg r) \land s \leftrightarrow \begin{cases} p \rightarrow q = True \\ p \rightarrow r = False \\ s = True \end{cases}\]
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
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

\[ A_1 \ldots A_n \]
\[ \frac{A}{A} \]

instead of

\[ [A_1 \ldots A_n] \implies A \]
Natural Deduction
Natural deduction

Two kinds of rules for each logical operator ⊕:

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

Elimination: what can I prove from $A \oplus B$?
Natural deduction for propositional logic

\[
\begin{align*}
\frac{A \quad B}{A \land B} & \text{ conjI} \\
\frac{A}{A \lor B} \quad \frac{B}{A \lor B} & \text{ disjI1/2} \\
\frac{A \rightarrow B}{A \rightarrow B} & \text{ impI} \\
\frac{A \rightarrow B}{A \rightarrow B} & \text{ iffI} \\
\frac{A \rightarrow \text{False}}{\neg A} & \text{ notI} \\
\frac{A \land B}{[A;B] \rightarrow C} & \text{ conjE} \\
\frac{A \lor B}{A \rightarrow C} \quad \frac{B \rightarrow C}{C} & \text{ disjE} \\
\frac{A \rightarrow B}{A \rightarrow B} \quad \frac{A \rightarrow B}{B \rightarrow C} & \text{ impE} \\
\frac{A = B}{A \rightarrow B} & \text{ iffD1} \\
\frac{\neg A}{A \rightarrow B} & \text{ notE} \\
\frac{A}{B \rightarrow A} & \text{ iffD2}
\end{align*}
\]
**Operational reading**

\[
\begin{array}{c}
A_1 \ldots A_n \\
\hline
A
\end{array}
\]

**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

\[
\begin{align*}
\text{refl} & : t = t \\
\text{sym} & : s = t \\
\text{trans} & : r = s \quad s = t \\
\text{subst} & : A(s) = t \\
\end{align*}
\]

Rarely needed explicitly — used implicitly by \textit{simp}
More rules

\[
\begin{array}{c}
A \rightarrow B \\
\hline
B \\
A
\end{array}
\]  \quad \text{mp}

\[
\begin{array}{c}
\neg A \rightarrow False \\
\hline
A
\end{array}
\]  \quad ccontr

\[
\begin{array}{c}
\neg A \rightarrow A \\
A
\end{array}
\]  \quad \text{classical}

Remark:

\text{ccontr and classical are not derivable from the ND-rules. They make the logic “classical”, i.e. “non-constructive”.}
Proof by assumption

\[
\frac{A_1 \ldots A_n}{A_i} \text{ assumption}
\]
Rule application: the rough idea

Applying rule $[A_1; \ldots ; A_n] \Rightarrow A$ 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: $[?P; ?Q] \Rightarrow ?P \land ?Q$
subgoal: $1. A \land B$
Result: $1. A$
$2. B$
Rule application: the details

Rule:
\[ [A_1; \ldots; A_n] \implies A \]

Subgoal:
\[ 1. [B_1; \ldots; B_m] \implies C \]

Substitution:
\[ \sigma(A) \equiv \sigma(C) \]

New subgoals:
\[ 1. \sigma([B_1; \ldots; B_m] \implies A_1) \]
\[ \vdots \]
\[ n. \sigma([B_1; \ldots; B_m] \implies A_n) \]

Command:
apply(rule <rulename>)
Proof by assumption

apply assumption

proves

1. \[ [B_1; \ldots; B_m] \implies C \]

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: \[ (?P \land ?Q; (?P; ?Q) \implies ?R) \implies ?R \]

Subgoal: 1. [ X; A \land B; Y ] \implies Z

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

New subgoal: 1. [ X; Y ] \implies [ A; B ] \implies Z

same as: 1. [ X; Y; A; B ] \implies Z
How to prove it by natural deduction

• **Intro** rules decompose formulae to the right of $\Rightarrow$.
  
  apply(\text{rule} <\text{intro-rule}>)

• **Elim** rules decompose formulae on the left of $\Rightarrow$.
  
  apply(\text{erule} <\text{elim-rule}>)
Demo: propositional proofs
To facilitate application of theorems:

write them like this \([A_1; \ldots; A_n] \implies A\)

not like this \(A_1 \land \ldots \land A_n \implies A\)
HOL: Predicate Logic
Subgoal:

1. $\wedge x_1 \ldots x_n$. Formula

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

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

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

$$\bigwedge x\ y. \left[ \forall y. P y \implies Q z y; \ Q x\ y \right] \implies \exists x. \ Q x\ y$$

means

$$\bigwedge x\ y. \left[ (\forall y_1. P y_1 \implies Q z y_1); \ Q x\ y \right] \implies \exists x_1. \ Q x_1\ y$$
Bound variables are renamed automatically to avoid name clashes with other variables.
Natural deduction for quantifiers

\[ \forall x. P(x) \quad \exists x. P(x) \]

- **allI** and **exI** introduce new parameters (\(\forall x\)).
- **allE** and **exE** introduce new unknowns (\(?x\)).
Instantiating rules

\[ \text{apply}(\text{rule_tac } x = \text{term in rule}) \]

Like \textit{rule}, but \(?x\) in \textit{rule} is instantiated by \textit{term} before application.

Similar: \textit{erule_tac}

\[ \downarrow \quad x \text{ is in rule, not in the goal} \]
A quantifier proof

1. \( \forall a. \exists b. \ a = b \)

apply \((\text{rule allI})\)

1. \( \land a. \exists b. \ a = b \)

apply \((\text{rule_tac } x = "a" \text{ in exl})\)

1. \( \land a. \ a = a \)

apply \((\text{rule refl})\)
Demo: quantifier proofs
More proof methods
Forward proofs: frule and drule

“Forward” rule: \[ A_1 \Rightarrow A \]
Subgoal: \[ 1. [ B_1; \ldots; B_n ] \Rightarrow C \]
Substitution: \[ \sigma(B_i) \equiv \sigma(A_1) \]
New subgoal: \[ 1. \sigma([ B_1; \ldots; B_n; A ] \Rightarrow C) \]

Command:

\[
\text{apply}(\text{frule } \text{rule name})
\]

Like \text{frule} but also deletes \( B_i \):

\[
\text{apply}(\text{drule } \text{rule name})
\]
frule and drule: the general case

Rule: $[ A_1; \ldots ; A_m ] \Rightarrow A$

Creates additional subgoals:

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

  : 

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

m. $\sigma([ B_1; \ldots ; B_n; A ] \Rightarrow C)$
Forward proofs: OF

\[ r[\text{OF } r_1 \ldots r_n] \]

Prove assumption 1 of theorem \( r \) with theorem \( r_1 \), and assumption 2 with theorem \( r_2 \), and \( \ldots \)

Rule \( r \)
\[ [ A_1; \ldots ; A_m ] \implies A \]

Rule \( r_1 \)
\[ [ B_1; \ldots ; B_n ] \implies B \]

Substitution
\[ \sigma(B) \equiv \sigma(A_1) \]

\( r[\text{OF } r_1] \)
\[ \sigma([ B_1; \ldots ; B_n; A_2; \ldots ; A_m ] \implies A) \]
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:

\[ \text{‘a set} = \text{‘a } \Rightarrow \text{ bool} \]

- \{\}, \{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\}
- \(\text{insert :: ‘a } \Rightarrow \text{ ‘a set } \Rightarrow \text{ ‘a set}\)
- \ldots \)
Proofs about sets

Natural deduction proofs:

- **equalityI**: \([A \subseteq B; B \subseteq A] \implies A = B\)
- **subsetI**: \((\forall x. x \in A \implies x \in B) \implies A \subseteq B\)
- ... (see Tutorial)
Demo: proofs about sets
Bounded quantifiers

- $\forall x \in A. \ P x \equiv \ \forall x. \ x \in A \rightarrow P x$
- $\exists x \in A. \ P x \equiv \ \exists x. \ x \in A \land P x$
- ballI: $(\land x. \ x \in A \rightarrow P x) \rightarrow \ \forall x \in A. \ P x$
- bspec: $[\forall x \in A. \ P x; x \in A] \rightarrow P x$
- bexI: $[P x; x \in A] \rightarrow \ \exists x \in A. \ P x$
- bexE: $[\exists x \in A. \ P x; \land x. \ [x \in A; P x] \rightarrow Q] \rightarrow Q$
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:

\[
\text{inductive_set } Ev :: \text{nat set} \\
\text{where} \\
0 \in Ev \\
\text{if } n \in Ev \implies n + 2 \in Ev
\]
Format of inductive definitions

inductive_set $S :: \tau$ set

where

\[
[ a_1 \in S; \ldots ; a_n \in S; A_1; \ldots ; A_k ] \Rightarrow a \in S
\]

where $A_1; \ldots ; A_k$ are side conditions not involving $S$. 
Proving properties of even numbers

Easy: \( 4 \in Ev \)

\[
0 \in Ev \implies 2 \in Ev \implies 4 \in Ev
\]

Trickier: \( m \in Ev \implies m+m \in Ev \)

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 \( 0 \in Ev \)
  \[
  \implies m = 0 \implies 0+0 \in Ev
  \]
- rule \( n \in Ev \implies n+2 \in Ev \)
  \[
  \implies m = n+2 \text{ and } n+n \in Ev \text{ (ind. hyp.!)}
  \]
  \[
  \implies m+m = (n+2)+(n+2) = ((n+n)+2)+2 \in Ev
  \]
Rule induction for Ev

To prove

\[ n \in \text{Ev} \implies P \, n \]

by rule induction on \( n \in \text{Ev} \) we must prove

- \( P \, 0 \)
- \( P \, n \implies P(n+2) \)

Rule \( \text{Ev.induct} : \)

\[ [ n \in \text{Ev}; P \, 0; \bigwedge n. P \, n \implies P(n+2) ] \implies P \, n \]
Rule induction in general

Set $S$ is defined inductively. To prove

$$x \in S \implies P x$$

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

$$[ a_1 \in S; \ldots ; a_n \in S ] \implies a \in S$$

that $P$ is preserved:

$$[ P a_1; \ldots ; P a_n ] \implies P a$$

In Isabelle/HOL:

```
apply (induct rule: S.induct)
```
Demo: inductively defined sets
Inductive predicates

\[ x \in S \implies S \cdot x \]

Example:

\textbf{inductive} \( Ev :: nat \Rightarrow bool \)

\textbf{where}

\[ Ev \ 0 \mid \]
\[ Ev \ n \implies Ev \ (n + 2) \]

Comparison:

\textbf{predicate:} simpler syntax

\textbf{set:} direct usage of \( \cup \) etc

Inductive predicates can be of type \( \tau_1 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow bool \)
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
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
Empirical study:

*Sledgehammer works for 1/3 of non-trivial Isabelle proofs*
Demo: Sledgehammer
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 fragment:

\textbf{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 \( \text{formula}_0 \)
  have \( \text{formula}_1 \) by simp
  
  
  have \( \text{formula}_n \) by blast
  show \( \text{formula}_{n+1} \) by \ldots
  qed

proves \( \text{formula}_0 \Rightarrow \text{formula}_{n+1} \)
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  (∧)
            |  assume prop  (⇒)
            |  [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: ¬ surj(f :: 'a ⇒ 'a set)
proof  assume surj, show False
  assume a: surj f
  from a have b: ∀ A. ∃ a. A = f a
    by (simp add: surj_def)
  from b have c: ∃ a. {x. x ∉ f x} = f a
    by blast
  from c show False
    by blast
qed
Demo: this, then etc
### Abbreviations

- **this**: the previous proposition proved or assumed
- **then**: from this
- **thus**: then show
- **hence**: then have
First the what, then the how:

\[(\text{have} | \text{show}) \text{ prop using facts} = \text{from facts (have} | \text{show}) \text{ prop}\]
Example: Structured lemma statement

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

Proves $\text{surj } f \implies \text{False}$
but $\text{surj } f$ 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**

\[ \text{show } P \iff Q \]

proof
  \[ \text{assume } P \]
  \[ : \]
  \[ \text{show } Q \ldots \]
  next
  \[ \text{assume } Q \]
  \[ : \]
  \[ \text{show } P \ldots \]
  qed

\[ \text{show } A = B \]

proof
  \[ \text{show } A \subseteq B \ldots \]
  next
  \[ \text{show } B \subseteq A \ldots \]
  qed

\[ \text{show } A \subseteq B \]

proof
  \[ \text{fix } x \]
  \[ \text{assume } x \in A \]
  \[ : \]
  \[ \text{show } x \in B \ldots \]
  qed
Propositional proof patterns

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

have $P \lor Q$ ...
then show $R$
proof
  assume $P$
  :
  :
  show $R$ ...
next
  assume $Q$
  :
  :
  show $R$ ...
qed

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

Case distinction  Case distinction  Contradiction
Quantifier introduction proof patterns

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

show $\exists x. P(x)$
proof
  \ldots
  show $P(\text{witness})$  \ldots
qed
\( \exists \ \text{elimination: obtain} \)

\[
\begin{align*}
\text{have } & \exists x. \ P(x) \\
\text{then obtain } & x \ \text{where } p: \ P(x) \ \text{by blast} \\
\therefore & \quad x \ \text{local fixed variable}
\end{align*}
\]

Works for one or more \( x \)
lemma "Cantor": \( \neg \text{surj}(f :: 'a \Rightarrow 'a \text{ set}) \)

proof
  assume \( \text{surj } f \)
  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 \( a \notin f a \leftrightarrow a \in f a \) by blast
  thus \( \text{False} \) by blast
qed
Applies method and generates subgoal(s):

1. \( \forall x_1 \ldots x_n \ [ A_1; \ldots ; A_m ] \Rightarrow A \)

How to prove each subgoal:

- fix \( x_1 \ldots x_n \)
- assume \( A_1 \ldots A_m \)
- show \( A \)

Separated by \texttt{next}
Demo: proof
Streamlining proofs: Pattern matching and Quotations
Example: pattern matching

\begin{align*}
&\text{show } formula_1 \leftrightarrow formula_2 \quad (\text{is } ?L \leftrightarrow ?R) \\
&\text{proof} \\
&\quad \text{assume } ?L \\
&\quad : \\
&\quad \text{show } ?R \ldots \\
&\text{next} \\
&\quad \text{assume } ?R \\
&\quad : \\
&\quad \text{show } ?L \ldots \\
&\text{qed}
\end{align*}
\textbf{show} \textit{formula} \ (\textit{is} \ ?thesis)\\
\textbf{proof} -\\
\hspace{1em}:\textbf{\hspace{1em}}\\
\hspace{2em}\textbf{show} \ ?thesis \ \ldots\\
\textbf{qed}\\

\textit{Every show implicitly defines} \ ?\textit{thesis}
Quoting facts by value

By name:

```plaintext
have x0: "x > 0" ... 
: 
from x0 ...
```

By value:

```plaintext
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 \( t = C_1 \vec{x} \mid \ldots \)

proof (cases term)
  case \((C_1 \vec{x})\)
  \(\ldots \vec{x} \ldots\)
next
::
qed

where \[\text{case } (C_i \vec{x}) \equiv\]

fix \(\vec{x}\)

assume \[C_i : \begin{cases} \text{label} \text{ term} = (C_i \vec{x}) \end{cases}\]
Induction
Overview

- Structural induction
- Rule induction
- Induction with fun
Structural induction for type nat

show $P(n)$
proof (induct $n$)
  case 0 $\equiv$ let $?\text{case} = P(0)$
  \ldots
  show $?\text{case}$
next
  case $(\text{Suc } n)$ $\equiv$ fix $n$ assume Suc: $P(n)$
  \ldots
  let $?\text{case} = P(\text{Suc } n)$
  \ldots $n$ \ldots
  show $?\text{case}$
qed
Demo: structural induction
Structural induction with $\Rightarrow$

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

proof (induct $n$)

  case 0

  ... show ?case

next

  case $(\text{Suc } n)$

  ... show ?case

qed

\[
\begin{array}{l}
\text{\textit{proof}} \quad \text{(induct } n) \\
\text{\textit{case 0}} \quad \equiv \quad \text{fix } x \text{ assume } 0: A(0) \\
\text{} \quad \text{let } ?\text{case} = P(0) \\
\text{\textit{next}} \quad \equiv \quad \text{fix } n \\
\text{\textit{case } (\text{Suc } n)} \quad \equiv \quad \text{assume } \text{Suc: } A(n) \Rightarrow P(n) \\
\text{} \quad A(\text{Suc } n) \\
\text{} \quad \text{let } ?\text{case} = P(\text{Suc } n) \\
\end{array}
\]
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
Rule induction
Inductive definition

inductive_set $S$

intros

$\text{rule}_1 : [ s \in S; A ] \implies s' \in S$

: $

\text{rule}_n : \ldots$
Rule induction

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

proof (*induct rule: S.induct*)

  case rule$_1$

  ...  

  show ?case

next

next

  case rule$_n$

  ... 

  show ?case

qed
Implicit selection of induction rule

assume $A: x \in S$

: 

show $P(x)$

using $A$ proof \textit{induct}

: 

qed
**Renaming free variables in rule**

\[
\text{case } (\text{rule}_i \ x_1 \ldots \ x_k)
\]

Renames the (alphabetically!) first \(k\) variables in \(\text{rule}_i\) to \(x_1 \ldots x_k\).
Demo: rule induction
Induction with fun

Definition:

\begin{verbatim}
fun f
::
\end{verbatim}

Proof:

\begin{verbatim}
show \ldots f(\ldots) \ldots
proof (induct x_1 \ldots x_k, rule: f.induct)
  case 1
  ::
\end{verbatim}

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 "\( \ldots = t_2 \)" . . . \( \ldots \equiv t_1 \)
also
:  
also
have "\( \ldots = t_n \)" . . . \( \ldots \equiv t_{n-1} \)
finally show . . .
— like from \( t_0 = t_n \) show
• “…” is merely an abbreviation

• also works for other transitive relations (〈, ≤, …)
Demo: also
Accumulating facts
moreover

have $\text{formula}_1$ . . .
moresover
have $\text{formula}_2$ . . .
moresover
::
moresover
have $\text{formula}_n$ . . .
ultimately show . . .

— like from $f_1 \ldots f_n$ show but needs no labels
Demo: moreover