Perlen der Informatik 2
Sommersemester 2008

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
Standard ML	implementation language

HOL

HOL = Higher-Order Logic

HOL = Functional programming + Logic

HOL has

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

HOL is a programming language!

Higher-order = functions are values, too!

Formulae

Syntax (in decreasing priority):

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

Scope of quantifiers: as far to the right as possible

Examples

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

Formulae

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

Hiding and renaming:

$\forall\, x\ y.\ (\forall\, x.\ P\ x\ y) \wedge Q\ x\ y \ \equiv\ \forall\, x_0\ y.\ (\forall\, x_1.\ P\ x_1\ y) \wedge G\ x_0\ y$

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:

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

Types and Terms

Types

Syntax:

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

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

Terms: Basic syntax

Syntax:

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

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:

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

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, *+ :: 'a* $\Rightarrow$ *'a* $\Rightarrow$ *'a*

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

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

Type list

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

Large library:
hd, *tl*, *map*, *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`

$\vdots$

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

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

Finding theorems

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

Demo: terms and types

An introduction to recursion and induction

A recursive datatype: toy lists

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

***Nil*:** empty list

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

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

Predefined lists: *[False, True]*

Concrete syntax

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

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

"..." normally not shown on slides

Structural induction on lists

P xs holds for all lists *xs* if

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

A recursive function: append

Declaration

consts *app :: 'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list*

and definition by *primitive recursion*:

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

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

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

sorry

“completes” any proof.

Suitable for top down developments:

Assume lemmas first, prove them later.

Disproving

quickcheck

tries to find counterexample by random testing

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

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

abbreviates

$$A_1 \Longrightarrow \dots \Longrightarrow A_n \Longrightarrow B$$

$; \quad \approx \quad$ “and”

The proof state

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

$x_1 \dots x_p$	Local constants
$A_1 \dots A_n$	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: $\mathit{Nil} :: \mathit{'a\ list}$

  $\mathit{Cons} :: \mathit{'a} \Rightarrow \mathit{'a\ list} \Rightarrow \mathit{'a\ list}$

- Distinctness: $\mathit{Nil} \neq \mathit{Cons}\ x\ xs$
- Injectivity: $(\mathit{Cons}\ x\ xs = \mathit{Cons}\ y\ ys) = (x = y \wedge xs = ys)$

The general case

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

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

Distinctness and Injectivity are applied automatically
Induction must be applied explicitly

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

$$app\ Nil\ ys = ys$$

$$app\ (Cons\ x\ xs)\ ys = Cons\ x\ (app\ xs\ ys)$$

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

More predefined types and functions

Type option

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

Important application:

$$
\begin{array}{rcl}
\ldots \Rightarrow \textit{'a option} & \approx & \text{partial function:} \\[1ex]
\textit{None} & \approx & \text{no result} \\
\textit{Some a} & \approx & \text{result } a
\end{array}
$$

Example:

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

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

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:

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

Rewriting:

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

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:

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

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

Can be mixed: $\forall b.\ f\ ?a\ y = b$

- 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** and **datatype**
- additional lemmas *eq*$_1$ $\ldots$ *eq*$_n$
- assumptions *P*$_1$ $\ldots$ *P*$_m$

Variations:

- *(simp* $\ldots$ *del:* $\ldots$*)* 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: $f(x) = g(x), g(x) = f(x)$

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

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

How to ignore assumptions

Assumptions sometimes cause problems, e.g. nontermination. How to exclude them from *simp*:

apply*(simp (no_asm_simp) . . .)*
Simplify only conclusion

apply*(simp (no_asm_use) . . .)*
Simplify but do not use assumptions

apply*(simp (no_asm) . . .)*
Ignore assumptions completely

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 $A \longrightarrow B$:

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

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

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/Isar $\rightarrow$ Settings $\rightarrow$ 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

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

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

lemma *itrev xs [] = rev xs*

Why in this direction?

Because the lhs is "more complex" than the rhs.

Demo: first proof attempt

Generalisation (1)

Replace constants by variables

lemma *itrev xs ys = rev xs @ ys*

Demo: second proof attempt

Generalisation (2)

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

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

HOL: Propositional Logic

Overview

- Natural deduction
- Rule application in Isabelle/HOL

Rule notation

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

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

subgoal: *1.* $A \wedge B$

Result: *1.* A

2. B

Rule application: the details

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

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

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

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

$\vdots$

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

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!)

Demo: application of introduction rule

Applying elimination rules

apply(*erule <elim-rule>*)

Like *rule* but also

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

Example:

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

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

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

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

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

How to prove it by natural deduction

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

  apply *(rule <intro-rule>)*

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

  apply *(erule <elim-rule>)*

Demo: examples

Safe and unsafe rules

Safe rules preserve provability

`conjI, impI, notI, iffI, refl, ccontr, classical, conjE, disjE`

Unsafe rules can turn a provable goal into an unprovable one

`disjI1, disjI2, impE, iffD1, iffD2, notE`

Apply safe rules before unsafe ones

$\Longrightarrow$ vs $\longrightarrow$

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

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

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

Demo: further techniques

HOL: Predicate Logic

Parameters

Subgoal:

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

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

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

- $\forall\, x.\ P(x)$: x can appear in $P(x)$.

  Example: $\forall\, x.\ x = x$ is obtained by $P \mapsto \lambda u.\ u = u$

- $\forall\, x.\ P$: x cannot appear in P.

  Example: $P \mapsto x = x$ yields $\forall\, x'.\ x = x$

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

Natural deduction for quantifiers

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

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

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

Instantiating rules

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

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

Similar: *erule_tac*

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

Two successful proofs

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

apply *(rule allI)*

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

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

Two unsuccessful proofs

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

apply *(rule_tac x = ??? in exI)*

apply *(rule exI)*

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

apply *(rule allI)*

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

apply *(rule refl)*

$?y \mapsto x$ yields $\bigwedge x'.\ x' = x$

Principle:

$?f\ x_1\ \ldots\ x_n$ can only be replaced by term t
if $params(t) \subseteq \{x_1, \ldots, x_n\}$

Demo: quantifier proofs

Safe and unsafe rules

Safe `allI, exE`

Unsafe `allE, exI`

Create parameters first, unknowns later

Proof methods

Parameter names

Parameter names are chosen by Isabelle

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

apply(*rule allI*)

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

apply(*rule_tac x = x in exI*)

Brittle!

Renaming parameters

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

apply*(rule allI)*

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

apply*(rename_tac xxx)*

1. $\bigwedge$*xxx.* $\exists$ *y. xxx = y*

apply*(rule_tac x = xxx in exI)*

In general:

(rename_tac $x_1 \ldots x_n$*)* renames the rightmost (inner) n parameters to $x_1 \ldots x_n$

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:

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

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

Forward proofs: THEN

r_1[THEN r_2]   means r_2[OF r_1]

Clarifying the goal

- **apply** *(intro . . .)*
  Repeated application of intro rules
  Example: **apply** *(intro allI)*
- **apply** *(elim . . .)*
  Repeated application of elim rules
  Example: **apply** *(elim conjE)*
- **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

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

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

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

Format of inductive definitions

inductive_set $S :: \tau\ set$

where

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

$\vdots$

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

Proving properties of even numbers

Easy: $4 \in Ev$

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

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

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

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

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

Rule induction for Ev

To prove

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

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

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

Rule `Ev.induct`:

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

An elimination rule

Rule induction in general

Set S is defined inductively.
To prove

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

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

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

that P is preserved:

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

In Isabelle/HOL:

apply *(erule S.induct)*

Demo: inductively defined sets

Inductive predicates

$$\tau \textit{ set} \quad \rightsquigarrow \quad \tau \Rightarrow \textit{bool}$$

Example:

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

where

Ev 0 $\quad|$

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

Comparison:

predicate simpler syntax

set direct usage of $\cup$ etc

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

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
- Propositional logic
- Predicate logic

Isar core syntax

proof	=	**proof** [method] statement* **qed**
	$\mid$	**by** method

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

statement	=	**fix** variables	($\bigwedge$)
	$\mid$	**assume** proposition	($\Longrightarrow$)
	$\mid$	[**from** name$^+$] (**have** | **show**) proposition proof	
	$\mid$	**next**	(separates subgoals)

proposition = [name:] formula

Demo: propositional logic, introduction rules

Basic proof methods

Basic atomic proof:

by $method$
apply $method$, then prove all subgoals by assumption

Basic proof method:

rule $\vec{a}$
apply a rule in $\vec{a}$;
if $\vec{a}$ is empty: apply a standard elim or intro rule.

Abbreviations:

.	=	**by** do-nothing
..	=	**by** *rule*

Demo: propositional logic, elimination rules

Elimination rules / forward reasoning

- Elim rules are triggered by facts fed into a proof:
  from $\vec{a}$ **have** $formula$ **proof**
- **proof** alone abbreviates **proof** *rule*
- *rule*: tries elim rules first (if there are incoming facts $\vec{a}$!)
- **from** $\vec{a}$ **have** $formula$ **proof** *(rule* $rule$*)*

  $\vec{a}$ must prove the first n premises of $rule$, in the right order

  the others are left as new subgoals

Abbreviations

this	=	the previous proposition proved or assumed
then	=	**from** *this*
thus	=	**then show**
hence	=	**then have**
with $\vec{a}$	=	**from** $\vec{a}$ *this*

using

First the what, then the how:

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

=

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

Can be mixed:

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

=

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

Demo: avoiding duplication

Schematic term variables

$$?A$$

- Defined by pattern matching:

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

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

Demo: predicate calculus

obtain

Syntax:

obtain variables **where** proposition proof

Mixing proof styles

from . . .

have . . .

  apply - make incoming facts assumptions

  apply$(\ldots)$

  $\vdots$

  apply$(\ldots)$

  done

Advanced Isar

Overview

- Case distinction
- Induction
- Calculational reasoning

Case distinction

Boolean case distinction

proof *cases*
  assume $formula$
  $\vdots$
next
  assume $\neg formula$
  $\vdots$
qed

proof *(cases* $formula$*)*
  case *True*
  $\vdots$
next
  case *False*
  $\vdots$
qed

case *True* $\quad \equiv$
assume *True:* $formula$

Demo: case distinction

Datatype case distinction

proof (*cases* $term$)

  case *Constructor*$_1$

  ⋮

next

⋮

next

  case (*Constructor*$_k$ $\vec{x}$)

  $\cdots$ $\vec{x}$ $\cdots$

qed

case (*Constructor*$_i$ $\vec{x}$) $\equiv$

fix $\vec{x}$ **assume** *Constructor*$_i$: $term$ = (*Constructor*$_i$ $\vec{x}$)

Induction

Overview

- Structural induction
- Rule induction
- Induction with fun

Structural induction for type nat

show $P($*n*$)$		
proof *(induction n)*		
case *0*	$\equiv$	**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 $\Longrightarrow$ and $\bigwedge$

show $\bigwedge x.\ A(n) \Longrightarrow P(n)$
proof *(induction n)*

case *0*	$\equiv$	**fix** x **assume** *0:* $A(0)$
...		**let** *?case* $= P(0)$
show *?case*		
next		
case *(Suc n)*	$\equiv$	**fix** $n\ x$
...		**assume** *Suc:* $\bigwedge x.\ A(n) \Longrightarrow P(n)$
$\cdots\ n\ \cdots$		$A(Suc\ n)$
...		**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 $\Longrightarrow$ and $\bigwedge$

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

$\vdots$

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*

⋮

Proof:

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

proof *(induction x*$_1$ $\ldots$ *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

Calculational Reasoning

Overview

- Accumulating facts
- Chains of equations and inequations

moreover

have $formula_1$ **...**

moreover

have $formula_2$ **...**

moreover

⋮

moreover

have $formula_n$ **...**

ultimately **show** **...**

— pipes facts $formula_1 \ldots formula_n$ into the proof

proof

⋮

also

have t_0 = t_1 . . . .

also

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

also

$\vdots$

also

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

finally **show** . . . .

— pipes fact t_0 = t_n into the proof

proof

$\vdots$

...

"$\ldots$" is merely an abbreviation

Demo: moreover and also

Variations on also

Transitivity:

have $t_0 = t_1$ . . . .

also have $\ldots = t_2$ . . . .

also/finally $\leadsto t_0 = t_2$

Substitution:

have $P(s)$ . . . .

also have $s = t$ . . . .

also/finally $\leadsto P(t)$

From $=$ to $\leq$ and $<$

Transitivity:

have $t_0 \leq t_1$ **. . . .**

also have $\ldots \leq t_2$ **. . . .**

also/finally $\leadsto t_0 \leq t_2$

Substitution:

have $r \leq f(s)$ **. . . .**

also have $s < t$ **. . . .**

also/finally $\leadsto (\bigwedge x.\ x < y \Longrightarrow f(x) < f(y)) \Longrightarrow r < f(t)$

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

All about also

To view all combinations in Proof General:

Isabelle/Isar $\rightarrow$ Show me $\rightarrow$ Transitivity rules

Demo: monotonicity reasoning