Chapter 4

Proof system of Isabelle/HOL

Methods and Rules

Formulas, sequents, and rules revisited

Propositions can represent:

- formulas, generalized sequents: lemmas/theorems to be proven
- rules: to be applied in a proof step
- proof (sub-)goals, i.e., open leaves in a proof tree

Example: from Lecture.thy

- SPEC, SCHEMATIC (Warning)
- ARULE
- GOAL

A proven lemma/theorem is automatically transformed into a rule. That is, the set of rules is not fixed in Isabelle/HOL.E.g. ARULE.

Variables

Six kinds of variables:

- (logical) variables bound by the logic-quantifiers
- (logical) variables bound by the meta-quantifier
- free (logical) variables
- schematic variables (in rules and proofs)
- type variables
- schematic type variables

Format of Goals and Rules

Format of Goals

- $\bigwedge x1...xk.\ [|A1; ...; Am|] \Longrightarrow C$
- xi are variables local to the subgoal (possibly none)
- Ai are called the assumptions (possibly none)
- C is called the conclusion
- usually first three types of variables sometimes also schematic variables.

Format of Rules

- $[|P1; ...; Pn|] \Longrightarrow Q$
- Pi are called the premises (possibly none)
- P1 is called the major premise
- Q is called the consequent (not standard)
- Schematic variables in Pi, Q.

Application of rules

Methods are commands to work on the proof state

In particular, methods allow to apply rules. Whereas the set of rules is not fixed, the basic methods are fixed in Isabelle/HOL.

Rule application:

- Applying rules is based on unification.
- Unification is done w.r.t. the schematic variables.
- The unifier is applied to the complete proof state!
- Unification may involve renaming of bound variables.

Example: (general idea of rule application)

- rule: $[|P1; P2|] \Longrightarrow Q$
- subgoal: $A \Longrightarrow C$
- if U unifies C and Q, then sufficient subgoals are:
- $U(A) \Longrightarrow U(P1), U(A) \Longrightarrow U(P2)$

Methods

Command: apply(method <parameters>)

Application of a rule to a subgoal depends on the method:

Methods are (for convenience) be classified into:

- introduction methods: decompose formulae to the right of $\Longrightarrow$
- elimination methods: decompose formulae to the left of $\Longrightarrow$

Method rule <rulename> :

- unify Q with C; fails if no unifier exists; otherwise unifier U
- remaining subgoals: For $i = 1, ..., n$
- $\bigwedge x1...xk.\ U([|A1; ...; Am|] \Longrightarrow Pi)$
- Example GOAL

Methods

Method assumption:

- unify C with first possible Aj; fails if no Aj exists for unification
- subgoal is closed (discharged)
- Example GOAL

Method erule <rulename> :

- unify Q with C and simultanneously unify P1 with some Aj; fails if no unifier exists; otherwise unifier U
- remaining subgoals: For $i = 2, ..., n$
- $\bigwedge x1...xk.\ U([|A1; ...; Am \backslash Aj|] \Longrightarrow Pi)$
- Example GOAL

Methods

Method drule <rulename> :

- unify P1 some Aj; fails if no unifier exists; otherwise unifier U
- remaining subgoals:
- For $i = 2, ..., n$ $\quad \bigwedge x1...xk.\ U([|A1; ...; Am \backslash Aj|] \Longrightarrow Pi)$
- $\bigwedge x1...xk.\ U([|A1; ...; Am \backslash Aj; Q|] \Longrightarrow C)$
- Example C1

Method frule <rulename> :

- unify P1 some Aj; fails if no unifier exists; otherwise unifier U
- remaining subgoals:
- For $i = 2, ..., n$ $\quad \bigwedge x1...xk.\ U([|A1; ...; Am|] \Longrightarrow Pi)$
- $\bigwedge x1...xk.\ U([|A1; ...; Am; Q|] \Longrightarrow C)$
- Example C1

Methods

Method [edf]rule_tac x= term in <rule> :

- are similar to the version above but allow to influence the unification
- Example 5.8.2, p. 79, TAC
- FIXAX2

Method unfold <name_def> :

- unfolds the definition of a constant in all subgoals
- Example SPEC

Method induct_tac <freevar...> :

- uses the inductive definition of a function
- generates the corresponding subgoals

Fundamental rules of Isabelle/HOl

See IsabelleHOLMain, Sect. 2.2

Remark

- Safe rules preserve provability
- e.g. conjI, impI, notI, iffI, refl, ccontr, classical, conjE, disjE
- Unsafe rules can turn a provable goal into an unprovable one
- e.g. disjI1, disjI2, impE, iffD1, iffD2, notE
- $\rightsquigarrow$ Apply safe rules before unsafe ones

Example

- lemma UNSAFE: "$A \vee \neg A$"
- apply (rule disI1)
- sorry

An overview of theory Main

The structure of theory Main: p. 23

Set construction in Isabelle/HOL: Sect. 6

Natural numbers in Isabelle/HOL: Sect. 15

Remark

Working with theory Main:

- The programmer cannot know the complete library
- The “verificator” cannot know all rules.

Rewriting and simplification

taken from IsabelleTutorial, Sect. 3.1) »> slidesNipkow:

apply(simp add: eq1 . . . eqn)

»> Demo: MyDemo, Simp

Overview

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

Term rewriting foundations

Term rewriting means . . .

Using equations $l = r$ from left to right

Term rewriting means . . .

Using equations $l = r$ from left to right

As long as possible

Term rewriting means . . .

Using equations $l = r$ from left to right

As long as possible

Terminology: equation $\rightsquigarrow$ *rewrite rule*

An example

Equations:

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

An example

Equations:

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

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

Rewriting:

An example

Equations:

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

Rewriting:

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

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:

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

An example

Equations:

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

Rewriting:

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

An example

Equations:

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

Rewriting:

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

More formally

substitution = mapping from variables to terms

More formally

substitution = mapping from variables to terms

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

More formally

substitution = mapping from variables to terms

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

More formally

substitution = mapping from variables to terms

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

More formally

substitution = mapping from variables to terms

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

Example:

Equation: $0 + n = n$

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

More formally

substitution = mapping from variables to terms

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

Example:

Equation: $0 + n = n$

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

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

More formally

substitution = mapping from variables to terms

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

Example:

Equation: $0 + n = n$

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

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

Result: $a + (b + c)$

Extension: conditional rewriting

Rewrite rules can be conditional:

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

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
- $\sigma(P_1)$, ..., $\sigma(P_n)$ are provable (again by rewriting).

Interlude: Variables in Isabelle

Schematic variables

Three kinds of variables:

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

Schematic variables

Three kinds of variables:

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

Schematic variables

Three kinds of variables:

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

Schematic variables:

Schematic variables

Three kinds of variables:

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

Schematic variables:

- Logically: free = schematic

Schematic variables

Three kinds of variables:

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

Schematic variables:

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

From x to ?x

State lemmas with free variables:

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

From x to ?x

State lemmas with free variables:

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

⋮

done

From x to ?x

State lemmas with free variables:

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

⋮

done

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

?xs @ [] = ?xs

Now usable with arbitrary values for *?xs*

From x to ?x

State lemmas with free variables:

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

⋮

done

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

?xs @ [] = ?xs

Now usable with arbitrary values for *?xs*

Example: rewriting

rev(a @ []) = rev a

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

Term rewriting in Isabelle

Basic simplification

Goal: *1.* $\llbracket P_1; \ldots ; P_m \rrbracket \Longrightarrow C$

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

Basic simplification

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

apply *(simp add: eq*$_1$ $\ldots$ *eq*$_n$*)*

Simplify P_1 $\ldots$ P_m and C using

- lemmas with attribute *simp*

Basic simplification

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

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

Simplify $P_1 \ldots P_m$ and C using

- lemmas with attribute *simp*
- rules from **primrec**, **fun** and **datatype**

Basic simplification

Goal: *1.* $\llbracket$ *P_1; ... ; P_m* $\rrbracket \Longrightarrow$ *C*

apply *(simp add: eq_1 ... eq_n)*

Simplify P_1 ... P_m and *C* using

- lemmas with attribute *simp*
- rules from **primrec**, **fun** and **datatype**
- additional lemmas eq_1 ... eq_n

Basic simplification

Goal: *1.* $\llbracket$ P_1; ... ; P_m $\rrbracket \Longrightarrow C$

apply *(simp add: eq_1 ... eq_n)*

Simplify $P_1 \ldots P_m$ and C using

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

Basic simplification

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

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

Simplify $P_1 \ldots P_m$ and C using

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

Variations:

- *(simp ... del: ...)* removes *simp*-lemmas
- *add* and *del* are optional

auto versus simp

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

Termination

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

Termination

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

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

Termination

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

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

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

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

Termination

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

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

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

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

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

Termination

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

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

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

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

Rewriting with definitions

Definitions do not have the *simp* attribute.

Rewriting with definitions

Definitions do not have the *simp* attribute.

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

Extensions of rewriting

Local assumptions

Simplification of $A \longrightarrow B$:

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

Case splitting with simp

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

Case splitting with simp

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

Automatic

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

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

Case splitting with simp

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

Automatic

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

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

Case splitting with simp

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

Automatic

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

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

Similar for any datatype t: *t.split*

Ordered rewriting

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

Ordered rewriting

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

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

Ordered rewriting

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

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

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

Ordered rewriting

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

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

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

For types *nat*, *int* etc:

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

Ordered rewriting

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

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

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

For types *nat*, *int* etc:

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

Example: *(simp add: add_ac)* yields

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

Preprocessing

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

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

Preprocessing

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

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

Example:

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

Preprocessing

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

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

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

Case analysis and structural induction

taken from IsabelleTutorial, Sect. 2, Sect. 3.2, Sect. 3.5

»> slidesNipkow:»> Demo: MyDemo,Trees

Slides for Session 3.2, 1-12 (slidesNipkow 87-93)

»>MyDemo, Induction Heuristics

Slides for Session 2, 57-79

»>MyDemo, Fun

Basic heuristics

Theorems about recursive functions are proved by induction

Basic heuristics

Theorems about recursive functions are proved by induction

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

A tail recursive reverse

primrec $\mathit{itrev} :: \text{'a list} \Rightarrow \text{'a list} \Rightarrow \text{'a list}$

A tail recursive reverse

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

A tail recursive reverse

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

A tail recursive reverse

primrec *itrev :: 'a list ⇒ 'a list ⇒ 'a list* **where**
itrev [] ys = ys |
itrev (x#xs) ys = itrev xs (x#ys)

lemma *itrev xs [] = rev xs*

A tail recursive reverse

primrec *itrev :: 'a list ⇒ 'a list ⇒ 'a list* **where**
itrev [] ys = ys |
itrev (x#xs) ys = itrev xs (x#ys)

lemma *itrev xs [] = rev xs*

Why in this direction?

A tail recursive reverse

primrec *itrev :: 'a list ⇒ 'a list ⇒ 'a list* **where**
itrev [] ys = ys |
itrev (x#xs) ys = itrev xs (x#ys)

lemma *itrev xs [] = rev xs*

Why in this direction?

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

Demo

Generalisation

- Replace constants by variables

Generalisation

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

Proof search automation

taken from IsabelleTutorial, Sect. 5.12, 5.13

Proof automation tries to apply rules either

- to finish the proof of (sub-)goal
- to simplify the subgoals

We call this the success criterion.

Methods for proof automation are different in

- the success criterion
- the rules they use
- the way in which these rule are applied

Simplification applies rewrite rules repeatedly as long as possible.
Classical reasoning uses search and backtracking with rules from predicate logic.

General Methods (Tactics)

blast:

- tries to finish proof of (sub-)goal
- classical reasoner

clarify:

- tries to perform obvious proof steps
- classical reasoner (only safe rule, no splitting of (sub-)goal)

safe:

- tries to perform obvious proof steps
- classical reasoner (only safe rule, splitting)

General Methods (Tactics)

clarsimp:

- tries to finish proof of (sub-)goal
- classical reasoner interleaved with simplification (only safe rule, no splitting)

force:

- tries to finish proof of (sub-)goal
- classical reasoner and simplification

auto:

- tries to perform proof steps on all subgoals
- classical reasoner and simplification (splitting)

More proof methods

Forward proof step in backward proof:

- apply rules to assumptions

Forward proofs (Hilbert style proofs):

- directly prove a theorem from proven theorems

Directives/attributes:

- of: instantiates the variables of a rule to a list of terms
- OF: applies a rule to a list of theorems
- THEN: gives a theorem to named rule and returns the conclusion
- simplified: applies the simplifier to a theorem

More proof methods

Forward proof step in backward proof:

- apply rules to assumptions

Forward proofs (Hilbert style proofs):

- directly prove a theorem from proven theorems

Directives/attributes:

- of: instantiates the variables of a rule to a list of terms
- OF: applies a rule to a list of theorems
- THEN: gives a theorem to named rule and returns the conclusion
- simplified: applies the simplifier to a theorem

Forward proofs: OF

r[OF $r_1 \ldots r_n$]

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

Forward proofs: OF

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

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

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

Forward proofs: OF

$r[OF\ r_1\ \dots\ r_n]$

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

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

More proof methods

Method insert:

- inserts a theorem as a new assumption into current subgoal

Method subgoal_tac:

- inserts an arbitrary formula F as assumption
- F becomes additional subgoal

»>MyDemo, subgoal_tac