Natural Deduction Rules for Quantifiers

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

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

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

Safe and Unsafe Rules

Safe: `allI`, `exE`

Unsafe: `allE`, `exI`

Create parameters first, unknowns later

Instantiating Variables in Rules

`proof (rule_tac x = "term" in rule)`

Like `rule`, but $?x$ in *rule* is instantiated with *term* before application.
$?x$ must be schematic variable occurring in statement of *rule*.

Similar: `erule_tac`

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

Two Apply-Style Successful Proofs

1. $\forall x.\ \exists y.\ x = y$
`apply (rule allI)`
1. $\bigwedge x. \exists y.\ x = y$

Better practice:	Exploration:
`apply (rule exI)`	`apply(rule_tac x = "x" in exI)`
1. $\bigwedge x.\ x = ?yx$	1. $\bigwedge x.\ x = x$
`apply (rule refl)`	`apply (rule refl)`
$?y \mapsto \lambda u.\ u$	
simpler & cleaner	shorter & trickier

Successful Attempt in Isar

```
lemma shows "∀(x::'a). ∃y. x = y"
proof (rule allI)
 fix x::'a
 show "∃y. x = y"
 proof (rule exI)
  show "x = x" by (rule refl)
 qed
qed
```


Two Unsuccessful Apply-Style Proof Attempts

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 $\bigwedge x'.\ x' = x$

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

Parameter Names

Parameter names are chosen by Isabelle

1. ∀x. ∃y. x = y
apply(rule allI)
1. ⋀x. ∃y. x = y
apply(rule_tac x = "x" in exI)

Works, but is brittle!!

Better to use Isar, where you choose the name.

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:

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

In Isar style, use have

Forward Proofs: OF and THEN

r [OF $r_1 \ldots r_n$]

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

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]	$\sigma([\![B_1; \ldots; B_n; A_2; \ldots; A_m]\!] \Longrightarrow A)$

r_1[THEN r_2] means r_2[OF r_1]

Forward Proofs: of

Given a theorem like gcd_mult_distrib2:

?k * gcd (?m, ?n) = gcd (?k * ?m, ?k * ?n)

- We want to replace ?m by 1.
- of instantiates variables left to right
- In above the order is ?k, ?m, and ?n
- [of k 1] replaces ?k by k, and ?m by 1.
- gcd_mult_distrib2 [of k 1] yields

k * gcd (1, ?n) = gcd (k * 1, k * ?n)

Forward Proofs: where

Alternately, with where you can specify the variable to get the term:

gcd_mult_distrib2 [where m = "1"] yields

?k * gcd (1, ?n) = gcd (?k * 1, ?k * ?n)

Same result given by gcd_mult_distrib2 [of _ 1]

gcd_mult_distrib2 [where m = "1" and k = "k"] yields

k * gcd (1, ?n) = gcd (k * 1, k * ?n)

Caution: of and where cannot use goal parameters

Forward Proofs: lemmas

- Can use `lemmas` to capture result of forward proof:

  `lemmas gcd_mult0 = gcd_mult_distrib2 [of k 1]`

- Can follow on with more forward reasoning:

  `lemmas gcd_mult1 = gcd_mult0 [simplified]` yields
  `k = gcd (k, k * ?n)`

- `[simplified]` applies `simp` to theorem

Forward Proofs: lemmas

Can combine multiple steps together:

`lemmas gcd_mult =`

`gcd_mult_distrib2 [of _ 1, simplified, THEN sym]`

yields

`gcd (?k, ?k * ?n) = ?k`

Adding Assumptions to Goals

- `cut_tac` *thm* insert *thm* as new assumption to current subgoal

`lemma relprime_dvd_mult:`

"$[\![\mathtt{gcd}(k, n) = 1;\ k \ \mathtt{dvd}\ m * n]\!] \Longrightarrow k \ \mathtt{dvd}\ m$"

`apply (cut_tac gcd_mult_distrib2 [of m k n])`

yields:

$[\![\mathtt{gcd}(k, n) = 1;\ k\ \mathtt{dvd}\ m * n;\ m * \mathtt{gcd}(k, n) = \mathtt{gcd}(m * k, m * n)]\!] \Longrightarrow$
k dvd m

Adding Assumptions to Goals

Note: `of` and `where` can use only original user variables, but not Isabelle generated parameters

`cut_tac k="m" and m="k" and n="n" in gcd_mult_distrib2` yields same result as above

`cut_tac` can use parameters

Adding Assumptions to Goals: subgoal_tac

- Can always add assumption *asm* to current subgoal with

  `apply (subgoal_tac "`*asm*`")`

- Statement can use Isabelle parameters
- Adds new subgoal *asm* with same assumptions as current subgoal

Adding Assumptions to Goals: subgoal_tac

1. $[\![A_1; \ldots ; A_n]\!] \Longrightarrow A$

`apply (subgoal_tac "`*asm*`")`

yields

1. $[\![A_1; \ldots ; A_n; \mathtt{asm}]\!] \Longrightarrow A$
2. $[\![A_1; \ldots ; A_n]\!] \Longrightarrow \mathtt{asm}$

Removing Assumptions: thin_tac

- Can remove unwanted assumption *asm* from current subgoal with `apply (thin_tac "asm")`

  1. $[\![A_1; \ldots; A_{i-1}; A_i; A_{i+1}; \ldots; A_n]\!] \Longrightarrow A$
  `apply (thin_tac "A_i")`

  yields

  1. $[\![A_1; \ldots; A_{i-1}; A_{i+1}; \ldots; A_n; asm]\!] \Longrightarrow A$

"Clarifying" the Goal

- `proof (intro ... )`
  Repeated application of intro rules
  Example: `proof (intro allI)`
- `proof (elim ... )`
  Repeated application of elim rules
  Example: `proof (elim conjE)`
- `proof (clarify)`
  Repeated application of safe rules without splitting goal
- `proof (clarsimp simp add: ... )`
  Combination of `clarify` and `simp`

Other Automated Proof Methods

- `blast` Isabelle's most powerful classical reasoner.
  Useful for goals stated using only predicate logic and set theory
  Can be extended with rules (with `[iff]` attribute) to handler broader classes of goals
- `auto`
  Applies to all subgoals.
  Combines classical reasoning with simplification
  Does what it can; leaves unfinished subgoals
  Splits subgoals
- `force`
  Similar to `auto`, but only applies to one goal, and either finishes or fails.
- `safe`
  Like `clarify` but also splits goals

Demo: Proof Methods