Contents

1 Introduction to Isabelle

This is also a comment but it generates nice LaTeX-text!

Command **thy_deps** demonstrates a graph of dependecies between Isabelle/HOL theories:

1.1 Terms & Types

Note that free variables (eg `x`), bound variables (eg `λn. n`) and constants (eg `Suc`) are displayed differently.

```
term "x"
term "Suc x"
term "Succ x"
term "Suc x = Succ y"
term "λx. x"
```

A bound variable:

```
term "λx. Suc x < y"
prop "map (λn. Suc n + 1) [0, 1] = [2, 3]"
```

To display types inside terms:

```
declare [[show_types]]
term "Suc x = Succ y"
```

To switch off again:

```
declare [[show_types=false]]
term "Suc x = Succ y"
```

Numbers and + are overloaded:

```
prop "n + n = 0"
prop "(n::nat) + n = 0"
prop "(n::int) + n = 0"
prop "n + n = Suc m"
```

Terms must be type correct! Try this: `term "True + False"`

Displaying theorems, schematic variables

```
thm conjI
```

Schematic variables have question marks and can be instantiated:

```
thm conjI [where ?Q = "x"]
thm conjI [no_vars]
thm impI
thm conjE
```

You can use `term`, `prop` and `thm` in LaTeXsections, too! The lemma conjI is: `⟦?P; ?Q⟧ ⟹ ?P ∧ ?Q`. Nicer version, without schematic variables: `⟦P; Q⟧ ⟹ P ∧ Q`.

Finding theorems

Searching for constants/functions:

```
find_theorems "map"
```

A list of search criteria finds thms that match all of them:

```
find_theorems "map" "zip"
```

To search for patterns, use underscore:

```
find_theorems "_ + _ = _ - _"
find_theorems "_ + _" "_ < _" "Suc"
find_theorems "Suc (Suc _)"
```

Searching for theorem names:

```
find_theorems name: "conj"
```

They can all be combined, theorem names include the theory name:

```
find_theorems "_ ∧ _" name: "HOL." -name: "conj"
```

1.2 Basic proofs

Stating theorems and a first proof

```
lemma "A ⟶ A"
 apply (rule impI)
 apply assumption
 done
```

A proof is a list of `apply` statements, terminated by `done`.

`apply` takes a proof method as argument (assumption, rule, etc.). It needs parentheses when the method consists of more than one word.

Isabelle doesn't care if you call it lemma, theorem or corollary

```
theorem "A ⟶ A"
 apply (rule impI)
 apply assumption
 done

corollary "A ⟶ A"
 apply (rule impI)
 apply assumption
 done
```

You can give it a name

```
lemma mylemma: "A ⟶ A" by (rule impI)
```

Abandoning a proof

```
lemma "P = NP"
 — this is too hard
 oops
```

Isabelle forgets the lemma and you cannot use it later

Faking a proof

```
lemma name1: "P ≠ NP"
 — have an idea, will show this later
 sorry
```

`sorry` only works interactively, and Isabelle keeps track of what you have faked.

Proof styles

```
theorem Cantor: "∃S. S ∉ range (f :: 'a ⇒ 'a set)" by best

— exploring, but unstructured
theorem Cantor': "∃S. S ∉ range (f :: 'a ⇒ 'a set)"
 apply (rule_tac x = "{x. x ∉ f x}" in exI)
 apply (rule notI)
 apply clarsimp
 apply blast
 done

— structured, explaining
theorem Cantor'': "∃S. S ∉ range (f :: 'a ⇒ 'a set)"
proof
 let ?S = "{x. x ∉ f x}"
 show "?S ∉ range f"
 proof
  assume "?S ∈ range f"
  then obtain y where fy: "?S = f y" ..
  show False
```

```
    proof cases
      assume yin: "y ∈ ?S"
      hence "y ∉ f y" by simp
      hence "y ∉ ?S" by(simp add:fy)
      thus False using yin by contradiction
    next
      assume yout: "y ∉ ?S"
      hence "y ∈ f y" by simp
      hence "y ∈ ?S" by(simp add:fy)
      thus False using yout by contradiction
    qed
  qed
qed

end
```

2 Lambda alculus

```
theory Lambda
  imports Main
begin
```

lambda terms

```
term "λx. x + 3"

term "λx a b. x + a + b + 3"
```

alpha-conversion

```
thm refl

lemma "(λx. x) = (λy. y)"
  apply (rule refl)
  done
```

eta-conversion

```
term "λx. f x"
```

beta-reduction

```
term "(λx. x y) t"
```

beta with renaming

```
term "(λz. (λx. f x z)) x"
```

example

((λ a. (λ b. b a) c) b) ((λ c. (c b)) (λ a. a)) = ((λ a. (λ b. b a) c) b) ((λ a. a) b) = ((λ a. (λ b. b a) c) b) b = ((λ a. c a) b) b = c b b

Isabelle performs this automatically:

```
term "((λ a. (λ b. b a) c) b) ((λ c. (c b)) (λ a. a))"
```

basic definitions

```
definition
  succ :: "(('a ⇒ 'a) ⇒ 'a ⇒ 'a) ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a" where
  "succ ≡ λn f x. f (n f x)"

thm succ_def

definition
  add :: "(('a ⇒ 'a) ⇒ 'a ⇒ 'a) ⇒ (('a ⇒ 'a) ⇒ 'a ⇒ 'a) ⇒
      ('a ⇒ 'a) ⇒ 'a ⇒ 'a" where
  "add ≡ λm n f x. m f (n f x)"

definition
  mult :: "(('a ⇒ 'a) ⇒ 'a ⇒ 'a) ⇒ (('a ⇒ 'a) ⇒ 'a ⇒ 'a) ⇒
```

```
    ('a ⇒ 'a) ⇒ 'a ⇒ 'a" where
 "mult ≡ λm n f x. m (n f) x"
```

unfolding a definition

```
definition
 c_0 :: "('a ⇒ 'a) ⇒ 'a ⇒ 'a" where
 "c_0 ≡ λf x. x"

definition
 c_1 :: "('a ⇒ 'a) ⇒ 'a ⇒ 'a" where
 "c_1 ≡ λf x. f x"

definition
 c_2 :: "('a ⇒ 'a) ⇒ 'a ⇒ 'a" where
 "c_2 ≡ λf x. f (f x)"

definition
 c_3 :: "('a ⇒ 'a) ⇒ 'a ⇒ 'a" where
 "c_3 ≡ λf x. f (f (f x))"

thm c_0_def
lemma "c_0 = (λf x. x)"
 apply (unfold c_0_def)
 apply (rule refl)
 done

lemma "succ (succ c_0) = c_2"
 apply (unfold succ_def c_0_def c_2_def)
 apply (rule refl)
 done

lemma "add c_2 c_2 = succ c_3"
 apply (unfold add_def succ_def c_3_def c_2_def)
 apply (rule refl)
 done

lemma "add x c_0 = x"
 apply (unfold add_def c_0_def)
 apply (rule refl)
 done

lemma "mult c_1 x = x"
 apply (unfold mult_def c_1_def)
 apply (rule refl)
 done
```

3 Simply Typed λ-calculus

Try \texttt{term "λ a. a a"}

An example with a free variable. In this case Isabelle infers the needed type of the free variable.

```
term "λ y f. f (x y)"


end
theory UnificationAndRules
 imports Main
begin
```

4 Types and Terms in Isabelle

```
term "True"
term "x"
term "x :: int ⇒ int"
term "{0} :: nat set"
term "x :: int set"
```

```
term "{x} :: 'a set"

datatype my_nat = Z | S my_nat
```

Turn on displaying typeclass information

```
thm order.refl
declare [[show_sorts]]
term "x::'a::order"

thm order.refl
declare [[show_sorts=false]]
```

Isabelle has free variables (eg `x`),bound variables (eg `λn. n`) and constants (eg `Suc`). Only schematic variables can be instantiated. Free converted into schematic after proof is finished.

```
thm conjI
thm conjI [where ?P = "x" and ?Q = "y"]
```

5 Higher Order Unification

Unify schematic ?t with `y`

```
thm refl
lemma "y = y"
 apply(rule refl)
 done

thm add.commute
```

Unify schematics ?a and ?b with `x` and `y` respectively.

```
lemma "(x::nat) + y = y + x"
 apply(rule add.commute)
 done
```

`schematic_goal command used to state lemmas that involve schematic variables which may be instantiated during their proofs. Used quite rarely.`

```
thm TrueI
```

Unify schematic ?P with `λx. True`

```
schematic_goal mylemma: "?P x"
 apply(rule TrueI)

 done

thm mylemma

lemma "P x"
 oops
```

Note that the theorem `True` contains just what was proved (namely the proposition `True`) not the more general result as originally stated (which isn't true for all ?P, as consider if ?P were instantiated with `λx. False`.

6 Propositional logic

6.1 Basic rules

$\wedge$

```
thm conjI
thm conjE
thm conjunct1 conjunct2
```

$\vee$

```
thm disjI1
thm disjI2
thm disjE
```

thm disjCI

$\longrightarrow$

thm impI impE

6.2 Examples

a simple backward step:

```
lemma "A ∧ B" thm conjI
 apply(rule conjI)
 oops
```

a simple backward proof:

```
lemma "B ∧ A ⟶ A ∧ B"
 apply (rule impI) thm impI
 apply (erule conjE)
 apply (rule conjI)
  apply assumption
 apply assumption
 done


lemma "A ∨ B ⟶ B ∨ A"
 apply (rule impI) thm disjE
 apply (erule disjE)
  apply (rule disjI2)
  apply assumption
 apply (rule disjI1)
 apply assumption
 done

lemma "⟦ B ⟶ C; A ⟶ B ⟧ ⟹ A ⟶ C"
 apply (rule impI)
 apply (erule impE) thm impE
  apply (erule impE)
   apply assumption
  apply assumption
 apply assumption
 done

thm notI notE

lemma "¬A ∨ ¬B ⟹ ¬(A ∧ B)"
 apply (rule notI)
 apply (erule disjE)
  apply (erule conjE) thm notE
  apply (erule notE)
  apply assumption
 apply (erule conjE)
 apply (erule notE)
 apply assumption
 done
```

Case distinctions. Isabelle can do case distinctions on arbitrary terms

```
lemma "P ∨ ¬P"
 apply (case_tac "P")
  apply (rule disjI1)
  apply (assumption)
  apply (rule disjI2)
apply (assumption)
 done

thm FalseE

lemma "(¬P ⟶ False) ⟶ P"
 apply (rule impI)
 apply (case_tac P)
```

```
  apply assumption
 apply (erule impE)
  apply assumption
 apply (erule FalseE)
 done
```

Explicit backtracking:

```
lemma "⟦ P ∧ Q; A ∧ B ⟧ ⟹ A"
 apply(erule conjE)
 back
 apply(assumption)
 done
```

Implicit backtracking: chaining with ,

```
lemma "⟦ P ∧ Q; A ∧ B ⟧ ⟹ A"

 apply (erule conjE, assumption)
 done
```

OR: use `rule_tac` or `erule_tac` to instantiate the schematic variables of the rule

```
lemma "⟦ P ∧ Q; A ∧ B ⟧ ⟹ A"
 apply (erule_tac ?P=A and ?Q=B in conjE)
 apply assumption
 done
```

= (iff)

```
thm iffI
thm iffE
thm iffD1
thm iffD2

lemma "A ⟶ B = (B ∨ ¬ A)"
 apply (rule impI)
 apply (rule iffI)
  apply (case_tac A)
   apply (rule disjI1)
  apply (assumption)
  apply (rule disjI2)
  apply (assumption)
 apply (erule disjE)
  apply assumption
 apply (erule notE)
 apply assumption
 done
```

— more rules

⟶

```
thm mp
```

¬

```
thm notI
thm notE
```

True & False

```
thm TrueI
thm TrueE
thm FalseE
```

Equality

```
thm refl
thm sym
thm trans
thm subst
```

classical (contradictions)

```
thm classical
thm ccontr
thm excluded_middle
```

classical propositional logic:

```
lemma Peirce: "((A ⟶ B) ⟶ A) ⟶ A"
 apply (rule impI) thm classical
 apply (rule classical)
 apply (erule impE)
  apply (rule impI)
  apply (erule notE)
  apply assumption
 apply assumption
 done
```

defer and prefer

```
lemma "(A ∨ A) = (A ∧ A)"
 apply (rule iffI)
 prefer 2
 defer
 oops
```

Example

```
lemma "A ⟶ (B ∨ C) ⟶ (¬ A ∨ ¬ B) ⟶ C"
 apply (rule impI)+
 apply (erule disjE)
  defer apply assumption
 apply (erule disjE)
  apply (erule notE)
  apply assumption
  apply (erule notE)
 by assumption
```

Exercises

```
lemma "A ∧ B ⟶ A ⟶ B"
 oops

lemma "(A ∧ B) ⟶ (A ∨ B)"
 oops

lemma "((A ⟶ B) ∧ (B ⟶ A)) = (A = B)"
 oops

lemma "(A ⟶ (B ∧ C)) ⟶ (A ⟶ C)"
 oops

end
theory MoreRules
 imports Main
begin
```

7 First Order Logic

Quantifier reasoning

```
thm allI
thm allE
thm exI
thm exE
```

Successful proofs

```
lemma "∀a. ∃b. a = b"
 apply (rule allI)
 thm exI
 apply (rule_tac x="a" in exI)
 thm refl
```

```
apply (rule refl)
done

lemma "∀a. ∃b. a = b"
apply (rule allI)
apply (rule exI)
apply (rule refl)
done
```

An unsuccessful proof

```
lemma "∃y. ∀x. x = y"
apply(rule exI)
apply(rule allI)
oops
```

Intro and elim reasoning

```
lemma "∃b. ∀a. P a b ⟹ ∀a. ∃b. P a b"

apply (rule allI)
apply (erule exE)

  thm allE
  apply (erule_tac x="a" in allE)
  thm exI
apply (rule_tac x="b" in exI)
apply assumption
done
```

What happens if an unsafe rule is tried too early

```
lemma "∃y. ∀x. P x y ⟹ ∀x. ∃y. P x y"
apply(rule allI) thm exI
apply(rule exI)
apply(erule exE)
apply(erule_tac x="x" in allE)
  oops
```

Instantiating allE and exI

```
lemma "∀x. P x ⟹ P 37"
  thm allE
apply (erule_tac x = "37" in allE)
apply assumption
done

lemma "∃n. P (f n) ⟹ ∃m. P m"
  apply(erule exE)
  thm exI
apply(rule_tac x = "f n" in exI)
apply assumption
done
```

Instantiation removes ambiguity

```
lemma "⟦ A ∧ B; C ∧ D ⟧ ⟹ D"
  thm conjE
apply(erule_tac P = "C" and Q="D" in conjE)

apply assumption
done
```

Renaming parameters

```
lemma "⋀x y z. P x y z"
apply(rename_tac a B)
oops

lemma "∀x. P x ⟹ ∀x. ∀x. P x"
apply(rule allI)
apply(rule allI)
  apply(rename_tac y)
```

```
 thm allE
apply(erule_tac x=y in allE)
apply assumption
 done
```

where and of attributes

```
thm conjI
thm conjI [of "A"]
thm conjI [of "A" "B"]
thm conjI [where Q = "f x" and P = "y"]

lemma "⟦ A ∧ B; C ∧ D ⟧ ⟹ D"
 thm conjE
apply(erule conjE[where P = "C" and Q="D"])

apply assumption
done
```

Forward reasoning: drule/frule

```
lemma "A ∧ B ⟹ ¬ ¬ A"
 thm conjunct1
 thm conjunct2
 thm mp
apply (drule conjunct1)
 apply (rule notI)
 thm notE
apply (erule notE)
apply assumption
done


lemma "∀x. P x ⟹ P t ∧ P t'"
thm spec
apply (frule_tac x="t" in spec)
apply (drule_tac x="t'" in spec)
apply (rule conjI)
 apply assumption
apply assumption
done
```

OF and THEN attributes

```
thm dvd_add dvd_refl
thm dvd_add [OF dvd_refl]
thm dvd_add [OF dvd_refl dvd_refl]
```

8 Automation

```
lemma "∀x y. P x y ∧ Q x y ∧ R x y"
apply (intro allI conjI)
 oops

lemma "∀x y. P x y ∧ Q x y ∧ R x y"
 apply clarify
 apply safe
oops

lemma "∃y. ∀x. P x y ⟹ ∀x. ∃y. P x y"
apply blast
done

lemma "∃y. ∀x. P x y ⟹ ∀x. ∃y. P x y"
apply fast
 done
```

More about automation

```
definition
 xor :: "bool ⇒ bool ⇒ bool" where
```

```
  "xor A B ≡ (A ∧ ¬B) ∨ (¬A ∧ B)"

thm xor_def

lemma xorI:
  "A ∨ B ⟹ ¬ (A ∧ B) ⟹ xor A B"
 apply (unfold xor_def)
 by blast

lemma xorE[elim!]:
  "⟦ xor A B; ⟦A; ¬B⟧ ⟹ R; ⟦¬A; B⟧ ⟹ R ⟧ ⟹ R"
 apply (unfold xor_def)
 by blast

lemma "xor A A = False"
 apply (blast elim! : xorE)
 done
```

Example

```
lemma "(∀x. P x ⟶ Q) = ((∃x. P x) ⟶ Q)"
 apply (rule iffI)
  apply (rule impI)
  apply (erule exE)
  apply (erule_tac x= x in allE)
  apply (erule impE)
   apply assumption+
 apply (rule allI)
 apply (rule impI)
 apply (erule impE)
  defer
  apply assumption
 apply (rule_tac x=x in exI)
 apply assumption
 done
```

Exercises

Prove using the xor definition and the proof methods: `rule`, `erule`, `rule_tac`, `erule_tac` and assumption:

```
lemma "xor A B = xor B A"
 oops

lemma "((∀x. P x ) ∧ (∀x. Q x)) = (∀x. (P x ∧ Q x))"
 oops
end
theory HOL_Intro
 imports Main
begin
```

9 HOL

9.1 Epsilon

Epsilon

```
lemma "(∃x. P x) = P (SOME x. P x)"
apply (rule iffI)
 apply (erule exE) thm someI
 apply (rule_tac x=x in someI)
  apply assumption thm exI
 thm exI
apply (rule_tac x="SOME x. P x" in exI)
apply assumption
done
```

Derive the axiom of choice from the SOME operator (using the rule someI), i.e. using only the rules: allI, allE, exI, exE and someI; with only the proof methods: rule, erule, `rule_tac`, `erule_tac` and assumption, prove:

```
lemma choice:
```

```
"∀x. ∃y. R x y ⟹ ∃f. ∀x. R x (f x)"
apply (rule exI)
apply (rule allI)
apply (erule_tac x=x in allE)
apply (erule exE)
thm someI
apply (erule someI)
done
```

9.2 HOL

You can find definition of HOL at `$ISABELLE_HOME/src/HOL/HOL.thy`

`we want to show "⟦ P ⟶ Q; P; Q ⟹ R ⟧ ⟹ R"`

```
lemma impE:
  assumes PQ: "P ⟶ Q"
  assumes P: "P"
  assumes QR: "Q ⟹ R"
  shows "R"

  apply (rule QR)
  apply (insert PQ) thm mp
  apply (drule mp)
   apply (rule P)
  apply assumption
  done
```

10 Simplification

Lists: `[]` empty list `x # xs` cons (list with head x and tail xs) `xs @ ys` append xs and ys
datatype 'a list = Nil | Cons 'a ('a list)

```
print_simpset

lemma "ys @ [] = []"
  apply simp
  oops

definition
  a :: "nat list" where
  "a ≡ []"

definition
  b :: "nat list" where
  "b ≡ []"
```

simp add, rewriting with definitions

```
lemma n[simp]: "xs @ a = xs"

  apply (simp add: a_def)
  done
```

simp only

```
lemma "xs @ a @ a @ a = xs"
  thm append_Nil2
  apply (simp only: a_def append_Nil2) thm a_def append_Nil2
  done

lemma ab [simp]: "a = b" using [[simp_trace]] by (simp only: a_def b_def)
lemma ba [simp]: "b = a" using [[simp_trace]] by simp
```

simp del, termination

```
lemma "a = []"


  apply (simp add: a_def del: ab)
```

```
  done
```

Simplification in assumption:

```
lemma "⟦ xs @ zs = ys @ xs; [] @ xs = [] @ [] ⟧ ⟹ ys = zs"
  apply simp
  done

end
theory Simplification
  imports Main
begin
```

Given the simplification rules: 0 + n = n (1) Suc m + n = Suc (m + n) (2) (Suc m ≤ Suc n) = (m ≤ n) (3) (0 ≤ m) = True (4) Then 0 + Suc 0 ≤ Suc 0 + x is simplified to True as follows: (0 + Suc 0 ≤ Suc 0 + x) (1) (Suc 0 ≤ Suc 0 + x) (2) (Suc 0 ≤ Suc (0 + x)) (3) (0 ≤ 0 + x) (4) True

implicit backtracking

```
lemma "⟦P ∧ Q; R ∧ S⟧ ⟹ S"

  apply (erule conjE, assumption)
  done

lemma "⟦P ∧ Q; R ∧ S⟧ ⟹ S"
```

do (n) assumption steps

```
  apply (erule (1) conjE)
  done

lemma "⟦P ∧ Q; R ∧ S⟧ ⟹ S"
```

'by' does extra assumption steps implicitly

```
  by (erule conjE)
```

more automation

clarsimp: repeated clarify and simp

```
lemma "ys = []  ⟹ ∀xs. xs @ ys = []"
  apply clarsimp
  oops
```

auto: simplification, classical reasoning, more

Method "auto" can be modified almost like "simp": instead of "add" use "simp add".

```
lemma "(∃u::nat. x = y + u) ⟶ a * (b + c) + y ≤ x + a * b + a * c"
  thm add_mult_distrib2
  apply (auto simp add: add_mult_distrib2)
  done
```

limit auto (and any other tactic) to first [n] goals

```
lemma "(True ∧ True)"
  apply (rule conjI)
   apply (auto) [1]
  apply (rule TrueI)
  done
```

Fastforce method

```
definition sq :: "nat ⇒ nat" where
  "sq n = n*n"

lemma "⟦ ∀xs ∈ A. ∃ys. xs = ys @ ys; us ∈ A ⟧ ⟹ ∃n. length us = n+n"
  by (fastforce simp add: sq_def)
```

simplification with assumptions, more control:

```
lemma "∀x. f x = g x ∧ g x = f x ⟹ f [] = f [] @ []"
```

would diverge:

```
  apply (simp (no_asm_use))
  done
```

let expressions

```
thm Let_def
term "let a = f x in g a"

lemma "let a = f x in g a = x"
  apply (simp)
  oops
```

splitting if: automatic in conclusion

```
lemma "(A ∧ B) = (if A then B else False)"
  by simp
```

splitting manually

```
thm list.split

lemma "1 ≤ (case ns of [] ⇒ 1 | n#_ ⇒ Suc n)"
  by (simp split: list.split)
```

splitting if: manual in assumptions

```
thm if_splits
thm if_split_asm

lemma "⟦ (if x then A else B) = z; Q ⟧ ⟹ P"

   apply (simp split: if_split_asm)
  oops

lemma " ((if x then A else B) = z) ⟶ (z = A ∨ z = B)"
  apply (rule impI)
  apply (simp split:if_splits)
done
```

Congruence rules

```
thm conj_cong

lemma "A ∧ (A ⟶ B)"
  apply (simp cong: conj_cong)
  oops

thm if_cong

lemma "⟦ (if x then x ⟶ z else B) = z; Q ⟧ ⟹ P"
  apply (simp cong: if_cong)
  oops

thm if_weak_cong
```

Term Rewriting Confluence

```
thm add_ac
thm mult_ac

lemmas all_ac = add_ac mult_ac
print_theorems

lemma
  fixes f :: "'a ⇒ 'a ⇒ 'a" (infix "⊙" 70)
  assumes A: "⋀x y z. (x ⊙ y) ⊙ z = x ⊙ (y ⊙ z)"
  assumes C: "⋀x y. x ⊙ y = y ⊙ x"
  assumes AC: "⋀x y z. x ⊙ (y ⊙ z) = y ⊙ (x ⊙ z)"
  shows "(z ⊙ x) ⊙ (y ⊙ v) = t"
  apply (simp only: C)
  apply (simp only: A C)
```

No confluence. We want v ⊙ (x ⊙ (y ⊙ z)) but got v ⊙ (y ⊙ (x ⊙ z))

```
 apply (simp only: AC)
 oops
```

when all else fails: tracing the simplifier

typing declare `[[simp_trace]]` turns tracing on, declare `[[simp_trace=false]]` turns tracing off (within a proof, write `'using'` rather than `'declare'`)

```
declare [[simp_trace]]
lemma "A ∧ (A ⟶ B)"
  apply (simp cong: conj_cong)
 oops
declare [[simp_trace=false]]


lemma "A ∧ (A ⟶ B)"
 using [[simp_trace]] apply (simp cong: conj_cong)
 oops
```

single stepping: subst

```
thm add.commute
thm add.assoc
declare add.assoc [simp]
thm add.assoc [symmetric]

lemma "a + b = x + ((y::nat) + z)"
thm add.assoc[symmetric]
 apply (simp  add: add.assoc [symmetric] del: add.assoc)
thm add.commute
 apply (subst add.commute [where a=a and b = b])
 oops

end
theory DatatypesAndInduction
 imports Main
begin
```

11 Types Declaration

```
type_synonym string = "char list"

typedecl myType
```

A recursive data type:

```
datatype ('a,'b) tree = Tip | Node "('a,'b) tree" 'b "('a,'b) tree"

print_theorems
```

Distincteness of constructors automatic:

```
lemma "Tip ~= Node l x r" by simp
```

Lists

Large library: HOL/List.thy included in Main. Don't reinvent, reuse! Predefined: xs @ ys (append), length, and map

```
term "Nil"
term "Cons"

thm list.induct
```

Enumeration

```
datatype Status = Inactive | InProgress | Finished
```

Mutual Recursion

```
datatype even = EvenZero | EvenSuc odd
 and odd = OddZero | OddSuc even

thm even.induct odd.induct
```

Case sytax, case distinction manual

```
lemma "(1::nat) ≤ (case t of Tip ⇒ 1 | Node l x r ⇒ x+1)"
 apply(case_tac t)
 apply auto
 done
```

Partial cases and dummy patterns:

```
term "case t of Node _ b _ => b"
```

Partial case maps to 'undefined':

```
lemma "(case Tip of Node _ _ _ => 0) = undefined" by simp
```

Nested case and default pattern:

```
term "case t of Node (Node _ _ _) x Tip => 0 | _ => 1"
```

Infinitely branching:

```
datatype 'a inftree = Tp | Nd 'a "nat ⇒ 'a inftree"
```

Mutually recursive

```
datatype
 ty = Integer | Real | RefT ref
 and
 ref = Class | Array ty
```

12 Primitive recursion and Functions

Non-recursive functions

```
definition sq :: "nat ⇒ nat" where
"sq n = n * n"

abbreviation sq' :: "nat ⇒ nat" where
"sq' n ≡ n * n"

thm sq_def
```

but there is no such a `thm sq'_def`

```
primrec add :: "nat ⇒ nat ⇒ nat" where
"add 0 n = n" |
"add (Suc m) n = Suc(add m n)"

primrec
 app :: "'a list => 'a list => 'a list"
where
 "app Nil ys = ys" |
 "app (Cons x xs) ys = Cons x (app xs ys)"

print_theorems

fun rev1 :: "'a list ⇒ 'a list" where
"rev1 Nil = Nil" |
"rev1 (Cons x xs) = (rev1 xs) @ (Cons x Nil)"

value "rev1(Cons True (Cons False Nil))"
value "rev1(Cons a (Cons b Nil))"
```

On trees:

```
primrec
 mirror :: "('a,'b) tree => ('a,'b) tree"
where
 "mirror Tip = Tip" |
 "mirror (Node l x r) = Node (mirror r) x (mirror l)"

print_theorems
```

Mutual recursion

```
primrec
 has_int :: "ty ⇒ bool" and
 has_int_ref :: "ref ⇒ bool"
where
 "has_int Integer     = True" |
 "has_int Real       = False" |
 "has_int (RefT T)    = has_int_ref T" |

 "has_int_ref Class    = False" |
 "has_int_ref (Array T) = has_int T"
```

— ————————————————————————

13 Structural induction

Basic examples with nats

```
lemma add_02: "add m 0 = m"
 apply(induction m)
 apply(auto)
done

fun double :: "nat ⇒ nat" where
"double 0 = 0" |
"double (Suc n) = Suc (Suc (double n))"

lemma nSm_Snm[simp]: "add n (Suc m) = add (Suc n) m"
 apply (induction n)
 apply (auto)
done

theorem double_add: "add n n = double n"
 apply(induction n)
 apply(auto)
 done

fun sum :: "nat ⇒ nat" where
"sum 0 = 0" |
"sum (Suc n) = (Suc n) + (sum n)"

lemma sum_is: "sum n = n * (n + 1) div 2"
 apply(induction n)
 apply(auto)
 done
```

Structural induction for lists

```
find_theorems "rev (rev _ )"
```

Discovering lemmas needed to prove a theorem

```
lemma app_nil[simp]: "xs @ [] = xs"
 apply (induction xs)
  apply (auto)
 done

lemma app_assoc: "(xs @ ys) @ zs = xs @ (ys @ zs)"
 apply (induction xs)
 apply (auto)
 done

thm List.append.append_Cons

lemma rev_app: "rev1 (xs @ ys) = (rev1 ys) @ (rev1 xs)"
 apply (induction xs)
 using [[simp_trace]]
  apply (simp)
 apply (simp add: app_assoc)
 done
```

```
theorem rev_rev: "rev1 (rev1 xs) = xs"
 apply (induction xs)
 apply (auto simp add:rev_app)
 done
```

One more example

```
fun count :: "'a ⇒ 'a list ⇒ nat" where
"count y [] = 0" |
"count y (x # xs) = (if x = y then Suc(count y xs) else count y xs)"

lemma count_less: "count y xs ≤ length xs"
 apply(induction xs)
 apply(auto)
 done
```

Induction heuristics

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

lemma itrev_rev_app: "itrev xs ys = rev xs @ ys"
 apply (induct xs arbitrary: ys)
  apply simp
 using [[simp_trace]]
 apply auto
 done

lemma "itrev xs [] = rev xs"
 apply (induct xs)
  apply simp
 apply (clarsimp simp: itrev_rev_app)
 done
```

— ————————————————————-

```
primrec
 lsum :: "nat list ⇒ nat"
where
 "lsum [] = 0" |
 "lsum (x#xs) = x + lsum xs"

find_theorems "(_ # _) @ _"

lemma lsum_app: "lsum (xs @ ys) = lsum xs + lsum ys"
 apply (induct xs)
 by auto

lemma
  "2 * lsum [0 ..< Suc n] = n * (n + 1)"
 apply (induct n)
  apply simp
 apply simp
 apply (auto simp: lsum_app)
 done

lemma
  "lsum (replicate n a) = n * a"
 apply (induct n)
  apply simp
 apply simp
 done
```

Tail recursive version:

```
primrec
 lsumT :: "nat list ⇒ nat ⇒ nat"
where
```

```
  "lsumT [] s = s" |
  "lsumT (x#xs) s = lsumT xs (x + s)"

lemma lsumT_gen:
  "lsumT xs s = lsum xs + s"
  by (induct xs arbitrary: s, auto)

lemma lsum_correct:
  "lsumT xs 0 = lsum xs"
  apply (induct xs)
   apply (simp)
  apply (simp add :lsumT_gen)
  done

end
theory Inductive
  imports Main
begin
```

14 Inductive

14.1 Inductive definition of the even numbers

```
inductive ev :: "nat ⇒ bool" where
ev0: "ev 0" |
evSS: "ev n ⟹ ev (Suc(Suc n))"

thm ev0 evSS
thm ev.intros
thm ev.intros(1)
thm ev.intros(2)

print_theorems
```

Using the introduction rules:

```
lemma "ev (Suc(Suc(Suc(Suc 0))))"
apply(rule evSS)+
apply(rule ev0)
done

thm evSS[OF evSS[OF ev0]]
```

A recursive definition of evenness:

```
fun evn :: "nat ⇒ bool" where
"evn 0 = True" |
"evn (Suc 0) = False" |
"evn (Suc(Suc n)) = evn n"
```

A simple example of rule induction:

```
thm ev.induct
lemma "ev n ⟹ evn n"
  apply(induction rule: ev.induct)
  apply(simp)
  apply(simp)
  done
```

An induction on the computation of evn:

```
thm evn.induct
lemma "evn n ⟹ ev n"
  apply(induction n rule: evn.induct)
  apply (simp add: ev0)
  apply simp
  apply(simp add: evSS)
  done
```

No problem with termination because the premises are always smaller than the conclusion:

```
thm ev.intros
declare ev.intros[simp,intro]
```

A shorter proof:

```
lemma "evn n ⟹ ev n"
 apply(induction n rule: evn.induct)
 apply(simp_all)
 done
```

The power of "arith":

```
lemma "ev n ⟹ ∃k. n = 2*k"
 apply(induction rule: ev.induct)
 apply(simp)
 apply arith
 done
```

14.2 Inductive definition of the reflexive transitive closure

```
inductive
 star :: "('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool"
for r where
refl: "star r x x" |
step: "r x y ⟹ star r y z ⟹ star r x z"

print_theorems
thm star.induct

lemma star_trans:
 "star r x y ⟹ star r y z ⟹ star r x z"
 apply(induction rule: star.induct)
 apply assumption
 apply(rename_tac u x y)
 by (simp add: star.step)

end
theory Isar_Demo
imports Complex_Main
begin
```

15 Isar

An introductory example

```
lemma "¬ surj(f :: 'a ⇒ 'a set)"
proof
 assume 0: "surj f"
 from 0 have 1: "∀A. ∃a. A = f a" by(simp add: surj_def)
 from 1 have 2: "∃a. {x. x ∉ f x} = f a" by blast
 from 2 show "False" by blast
qed
```

A bit shorter:

```
lemma "¬ surj(f :: 'a ⇒ 'a set)"
proof
 assume 0: "surj f"
 from 0 have 1: "∃a. {x. x ∉ f x} = f a" by(auto simp: surj_def)
 from 1 show "False" by blast
qed
```

15.1 "this", "then", "hence" and "thus

Avoid labels, use "this"

```
lemma "¬ surj(f :: 'a ⇒ 'a set)"
proof
 assume "surj f"
 from this have "∃a. {x. x ∉ f x} = f a" by(auto simp: surj_def)
```

```
  from this show "False" by blast
qed
```

"then" = "from this"

```
lemma "¬ surj(f :: 'a ⇒ 'a set)"
proof
  assume "surj f"
  then have "∃a. {x. x ∉ f x} = f a" by(auto simp: surj_def)
  then show "False" by blast
qed
```

"hence" = "then have", "thus" = "then show"

```
lemma "¬ surj(f :: 'a ⇒ 'a set)"
proof
  assume "surj f"
  hence "∃a. {x. x ∉ f x} = f a" by(auto simp: surj_def)
  thus "False" by blast
qed
```

15.2 Structured statements: "fixes", "assumes", "shows"

```
lemma
  fixes f :: "'a ⇒ 'a set"
  assumes s: "surj f"
  shows "False"
proof-
  have "∃ a. {x. x ∉ f x} = f a" using s
    by(auto simp: surj_def)
  thus "False" by blast
qed
```

15.3 Proof patterns

```
lemma "P ⟷ Q"
proof
  assume "P"
  show "Q" sorry
next
  assume "Q"
  show "P" sorry
qed
```

```
lemma "A = (B::'a set)"
proof
  show "A ⊆ B" sorry
next
  show "B ⊆ A" sorry
qed
```

```
lemma "A ⊆ B"
proof
  fix a
  assume "a ∈ A"
  show "a ∈ B" sorry
qed
```

Contradiction

```
lemma P
proof (rule ccontr)
  assume "¬P"
  show "False" sorry
qed
```

Case distinction

```
lemma "R"
proof cases
  assume "P"
```

```
  show "R" sorry
next
  assume "¬ P"
  show "R" sorry
qed

lemma "R"
proof -
  have "P ∨ Q" sorry
  then show "R"
  proof
    assume "P"
    show "R" sorry
  next
    assume "Q"
    show "R" sorry
  qed
qed
```

"obtain" example

```
lemma "¬ surj(f :: 'a ⇒ 'a set)"
proof
  assume "surj f"
  hence "∃a. {x. x ∉ f x} = f a" by(auto simp: surj_def)
  then obtain a where "{x. x ∉ f x} = f a" by blast
  hence "a ∉ f a ⟷ a ∈ f a" by blast
  thus "False" by blast
qed
```

Interactive exercise:

```
lemma assumes "∃x. ∀y. P x y" shows "∀y. ∃x. P x y"
sorry
```

15.4 (In)Equation Chains

```
lemma "(0::real) ≤ x^2 + y^2 - 2*x*y"
proof -
  have "0 ≤ (x - y)^2" by simp
  also have "... = x^2 + y^2 - 2*x*y"
    by(simp add: numeral_eq_Suc algebra_simps)
  finally show "0 ≤ x^2 + y^2 - 2*x*y" .
qed
```

Interactive exercise:

```
lemma
  fixes x y :: real
  assumes "x ≥ y" "y > 0"
  shows "(x - y) ^ 2 ≤ x^2 - y^2"
proof -
  have "(x - y) ^ 2 = x^2 + y^2 - 2*x*y"
    by(simp add: numeral_eq_Suc algebra_simps)
  show "(x - y) ^ 2 ≤ x^2 - y^2" sorry
qed
```

15.5 Streamlining proofs

15.6 Pattern matching and ?-variables

Show ∃

```
lemma "∃ xs. length xs = 0" (is "∃ xs. ?P xs")
proof
  show "?P([])" by simp
qed
```

Multiple EX easier with forward proof:

```
lemma shows "∃ x y :: int. x < z & z < y" (is "∃ x y. ?P x y")
proof -
```

```
  have "?P (z - 1) (z + 1)" by arith
  thus ?thesis by blast
qed
```

15.7 Quoting facts

```
lemma assumes "x < (0::int)" shows "x*x > 0"
proof -
  from ‹x<0› show ?thesis by(metis mult_neg_neg)
qed
```

15.8 Example: Top Down Proof Development

```
lemma "∃ys zs. xs = ys @ zs ∧
      (length ys = length zs ∨ length ys = length zs + 1)"
sorry
```

15.9 Solutions to interactive exercises

```
lemma assumes "∃x. ∀y. P x y" "B c" shows "∀y. ∃x. P x y"
proof
  fix b
  from assms obtain a where 0: "∀y. P a y" by blast
  show "∃x. P x b"
  proof
    show "P a b" using 0 by blast
  qed
qed
```

```
lemma fixes x y :: real assumes "x ≥ y" "y > 0"
shows "(x - y) ^ 2 ≤ x^2 - y^2"
proof -
  have "(x - y) ^ 2 = x^2 + y^2 - 2*x*y"
    by(simp add: numeral_eq_Suc algebra_simps)
  also have "... ≤ x^2 + y^2 - 2*y*y"
    using assms by(simp)
  also have "... = x^2 - y^2"
    by(simp add: numeral_eq_Suc)
  finally show ?thesis .
qed
```

15.10 Example: Top Down Proof Development

The key idea: case distinction on length:

```
lemma "∃ys zs. xs = ys @ zs ∧ (length ys = length zs ∨ length ys = length zs + 1)"
proof cases
  assume "EX n. length xs = n+n"
  show ?thesis sorry
next
  assume "¬ (EX n. length xs = n+n)"
  show ?thesis sorry
qed
```

A proof skeleton:

```
lemma "∃ys zs. xs = ys @ zs ∧ (length ys = length zs ∨ length ys = length zs + 1)"
proof cases
  assume "∃n. length xs = n+n"
  then obtain n where "length xs = n+n" by blast
  let ?ys = "take n xs"
  let ?zs = "take n (drop n xs)"
  have "xs = ?ys @ ?zs ∧ length ?ys = length ?zs" sorry
  thus ?thesis by blast
next
  assume "¬ (∃n. length xs = n+n)"
  then obtain n where "length xs = Suc(n+n)" sorry
  let ?ys = "take (Suc n) xs"
  let ?zs = "take n (drop (Suc n) xs)"
  have "xs = ?ys @ ?zs ∧ length ?ys = length ?zs + 1" sorry
```

```
  then show ?thesis by blast
qed
```

The complete proof:

```
lemma "∃ys zs. xs = ys @ zs ∧ (length ys = length zs ∨ length ys = length zs + 1)"
proof cases
  assume "∃n. length xs = n+n"
  then obtain n where "length xs = n+n" by blast
  let ?ys = "take n xs"
  let ?zs = "take n (drop n xs)"
  have "xs = ?ys @ ?zs ∧ length ?ys = length ?zs"
    using [[simp_trace]]
    find_theorems "length (take _ _)"
    find_theorems "length (drop _ _)"
find_theorems "take _ (drop _ _)"
    by (simp add: `length xs = n + n`)
  thus ?thesis by blast
next
  assume "¬ (∃n. length xs = n+n)"
  hence "∃n. length xs = Suc(n+n)" by arith
  then obtain n where l: "length xs = Suc(n+n)" by blast
  let ?ys = "take (Suc n) xs"
  let ?zs = "take n (drop (Suc n) xs)"
  have "xs = ?ys @ ?zs ∧ length ?ys = length ?zs + 1" by (simp add: l)
  thus ?thesis by blast
qed

end
theory Isar_Induction_Demo
imports Main
begin
```

15.11 Case distinction

Explicit:

```
lemma "length(tl xs) = length xs - 1"
proof (cases xs)
  assume "xs = []" thus ?thesis by simp
next
  fix y ys assume "xs = y#ys"
  thus ?thesis by simp
qed
```

Implicit:

```
lemma "length(tl xs) = length xs - 1"
proof (cases xs)
print_cases
  case Nil
thm Nil
  thus ?thesis by simp
next
  case (Cons y ys)
thm Cons
  thus ?thesis by simp
qed
```

15.12 Structural induction for type nat

Explicit:

```
lemma "∑{0..n::nat} = n*(n+1) div 2" (is "?P n")
proof (induction n)
  show "?P 0" by simp
next
  fix n assume "?P n"
  thus "?P(Suc n)" by simp
qed
```

In more detail:

```
lemma "∑ {0..n::nat} = n*(n+1) div 2" (is "?P n")
proof (induction n)
  show "?P 0" by simp
next
  fix n assume IH: "?P n"
  have "∑ {0..Suc n} = ∑ {0..n} + Suc n" by simp
  also have "... = n*(n+1) div 2 + Suc n" using IH by simp
  also have "... = (Suc n)*((Suc n)+1) div 2" by simp
  finally show "?P(Suc n)" .
qed
```

Implicit:

```
lemma "∑ {0..n::nat} = n*(n+1) div 2"
proof (induction n)
print_cases
  case 0
  show ?case by simp
next
  case (Suc n)
thm Suc
  thus ?case by simp
qed
```

Induction with $\Longrightarrow$:

```
lemma split_list: "x : set xs ⟹ ∃ys zs. xs = ys @ x # zs"
proof (induction xs)
  case Nil thus ?case by simp
next
  case (Cons a xs)
thm Cons.IH
thm Cons.prems
thm Cons
  from Cons.prems have "x = a ∨ x : set xs" by simp
  thus ?case
  proof
    assume "x = a"
    hence "a#xs = [] @ x # xs" by simp
    thus ?thesis by blast
  next
    assume "x : set xs"
    then obtain ys zs where "xs = ys @ x # zs" using Cons.IH by auto
    hence "a#xs = (a#ys) @ x # zs" by simp
    thus ?thesis by blast
  qed
qed
```

15.13 Computation induction

```
fun div2 :: "nat ⇒ nat" where
"div2 0 = 0" |
"div2 (Suc 0) = 0" |
"div2 (Suc(Suc n)) = div2 n + 1"

thm div2.induct

lemma "2 * div2 n ≤ n"
proof(induction n rule: div2.induct)
  case 1
  show ?case by simp
next
  case 2
  show ?case by simp
next
  case (3 n)
  thm 3
  have "2 * div2 (Suc(Suc n)) = 2 * div2 n + 2" by simp
  also have "... ≤ n + 2" using "3.IH" by simp
```

```
  also have "... = Suc(Suc n)" by simp
  finally show ?case .
qed
```

Note that `3.IH` is not a valid name, it needs double quotes: `"3.IH"`.

```
fun sep :: "'a ⇒ 'a list ⇒ 'a list" where
"sep a (x # y # zs) = x # a # sep a (y # zs)" |
"sep a xs = xs"

thm sep.simps

lemma "map f (sep a xs) = sep (f a) (map f xs)"
proof (induction a xs rule: sep.induct)
print_cases
  case (1 a x y zs)
  thus ?case by simp
next
  case ("2_1" a)
  show ?case by simp
next
  case ("2_2" a v)
  show ?case by simp
qed
```

15.14 Rule induction

```
inductive ev :: "nat => bool" where
ev0:  "ev 0" |
evSS:  "ev n ⟹ ev(Suc(Suc n))"

thm ev.induct

declare ev.intros [simp]


lemma "ev n ⟹ ∃k. n = 2*k"
proof (induction rule: ev.induct)
  case ev0 show ?case by simp
next
  case evSS thus ?case by arith
qed


lemma "ev n ⟹ ∃k. n = 2*k"
proof (induction rule: ev.induct)
  case ev0 show ?case by simp
next
  case (evSS m)
thm evSS
thm evSS.IH
thm evSS.hyps
  from evSS.IH obtain k where "m = 2*k" by blast
  hence "Suc(Suc m) = 2*(k+1)" by simp
  thus "∃k. Suc(Suc m) = 2*k" by blast
qed
```

15.15 Inductive definition of the reflexive transitive closure

```
consts step :: "'a ⇒ 'a ⇒ bool" (infix "→" 55)

inductive steps :: "'a ⇒ 'a ⇒ bool" (infix "→*" 55) where
refl: "x →* x" |
step: "⟦ x → y; y →* z ⟧ ⟹ x →* z"

declare refl[simp, intro]
```

Explicit and by hand:

```
lemma "x →* y  ⟹  y →* z ⟹ x →* z"
```

```
proof(induction rule: steps.induct)
  fix x assume "x →* z"
  thus "x →* z" .
next
  fix x' x y :: 'a
  assume "x' → x" and "x →* y"
  assume IH: "y →* z ⟹ x →* z"
  assume "y →* z"
  show "x' →* z" by(rule step[OF `x' → x` IH[OF `y→*z`]])
qed
```

Implicit and automatic:

```
lemma "x →* y  ⟹  y →* z ⟹ x →* z"
proof(induction rule: steps.induct)
  case refl thus ?case .
next
  case (step x' x y)

thm step
thm step.IH
thm step.hyps
thm step.prems
  show ?case
    by (metis step.hyps(1) step.IH step.prems steps.step)
qed
```

15.16 Rule inversion

```
lemma assumes "ev n" shows "ev(n - 2)"
proof-
  from `ev n` show "ev(n - 2)"
  proof cases
    case ev0
thm ev0
    then show ?thesis by simp
  next
    case (evSS k)
thm evSS
    then show ?thesis by simp
  qed
qed
```

Impossible cases are proved automatically:

```
lemma "¬ ev(Suc 0)"
proof
  assume "ev(Suc 0)"
  then show False
  proof cases
  qed
qed


end
theory Locales
  imports "HOL-Library.Countable_Set"
begin
```

16 Locales

Locales are based on context . A context is a formula scheme $\bigwedge x_1 \ldots x_n.\ [[\ A_1 ; \ldots ; A_m\ ]] \Longrightarrow \ldots$

```
locale partial_order =
  fixes le :: "'a ⇒ 'a ⇒ bool" (infixl "⊑" 50)
  assumes refl [intro, simp]: "x ⊑ x"
    and anti_sym [intro]: "⟦ x ⊑ y; y ⊑ x ⟧ ⟹ x = y"
    and trans [trans]: "⟦ x ⊑ y; y ⊑ z ⟧ ⟹ x ⊑ z"
```

The parameter of this locale is le, which is a binary predicate with infix syntax ⊑. The parameter syntax is available in the subsequent assumptions, which are the familiar partial order axioms.

```
print_locales

print_locale! partial_order
```

The assumptions have turned into conclusions, denoted by the keyword notes. Also, there is only one assumption - `partial_order`. The locale declaration has introduced the predicate `partial_order` to the theory. This predicate is the locale predicate.

```
thm partial_order_def
thm partial_order.trans partial_order.anti_sym partial_order.refl
```

16.1 Extending Locales

```
definition (in partial_order)
less :: "'a ⇒ 'a ⇒ bool" (infixl "⊏" 50)
 where "(x ⊏ y) = (x ⊑ y ∧ x ≠ y)"
```

The definition generates a foundational constant `partial_order.less`:

```
thm partial_order.less_def
```

At the same time, the locale is extended by syntax transformations hiding this construction in the context of the locale.

```
print_locale! partial_order

lemma (in partial_order) less_le_trans [trans]:
 "⟦ x ⊏ y; y ⊑ z ⟧ ⟹ x ⊏ z"
 unfolding less_def by (blast intro: trans)
```

16.2 context n begin ... end

Entering locale context:

```
context partial_order

begin

definition
 is_inf where "is_inf x y i =
     (i ⊑ x ∧ i ⊑ y ∧ (∀ z. z ⊑ x ∧ z ⊑ y ⟶ z ⊑ i))"

definition
 is_sup where "is_sup x y s =
     (x ⊑ s ∧ y ⊑ s ∧ (∀ z. x ⊑ z ∧ y ⊑ z ⟶ s ⊑ z))"

theorem is_inf_uniq: "⟦is_inf x y i; is_inf x y i'⟧ ⟹ i = i'"
 oops
theorem is_sup_uniq: "⟦is_sup x y s; is_sup x y s'⟧ ⟹ s = s'"
 oops

end

print_locale! partial_order
```

16.3 Import

```
locale total_order = partial_order +
 assumes total: "x ⊑ y ∨ y ⊑ x"

locale lattice = partial_order +
 assumes ex_inf: "∃ inf. is_inf x y inf"
  and ex_sup: "∃ sup. is_sup x y sup"
begin

definition meet (infixl "⊓" 70) where "x ⊓ y = (THE inf. is_inf x y inf)"
definition join (infixl "⊔" 65) where "x ⊔ y = (THE sup. is_sup x y sup)"

lemma meet_left: "x ⊓ y ⊑ x" oops
```

```
end
```

16.4 Interpretation

The declaration sublocale `l1` ⊆ `l2` causes locale l2 to be interpreted in the context of l1. This means that all conclusions of l2 are made available in l1.

```
sublocale total_order ⊆ lattice
```

The sublocale command generates a goal, which must be discharged by the user:

```
proof unfold_locales
 fix x y
 thm is_inf_def
 from total have "is_inf x y (if x ⊑ y then x else y)" by (auto simp: is_inf_def)
 thus "∃ inf. is_inf x y inf" ..
 from total have "is_sup x y (if x ⊑ y then y else x)" by (auto simp: is_sup_def)
 thus "∃ sup. is_sup x y sup" ..
qed
```

The command interpretation is for the interpretation of locale in theories.

In the following example, the parameter of locale `partial_order` is replaced by `(≤)` and the locale instance is interpreted in the current theory.

```
interpretation int: partial_order "(≤) :: int ⇒ int ⇒ bool"
 apply unfold_locales
  apply auto
 done

thm int.trans
thm int.less_def
```

We want to replace int.less by `<`.

In order to allow for the desired replacement, interpretation accepts equations in addition to the parameter instantiation. These follow the locale expression and are indicated with the keyword rewrites.

```
interpretation int: partial_order "(≤) :: [int, int] ⇒ bool"
 rewrites "int.less x y = (x < y)"

proof-
 show "partial_order ((≤) :: int ⇒ int ⇒ bool)"
  by unfold_locales auto
 show "partial_order.less (≤) x y = (x < y)"
  unfolding partial_order.less_def [OF ‹partial_order (≤)›]
  by auto
qed
```

In the above example, the fact that $\leq$ is a partial order for the integers was used in the second goal to discharge the premise in the definition of $\sqsubseteq$. In general, proofs of the equations not only may involve definitions from the interpreted locale but arbitrarily complex arguments in the context of the locale. Therefore it would be convenient to have the interpreted locale conclusions temporarily available in the proof. This can be achieved by a locale interpretation in the proof body. The command for local interpretations is interpret.

```
interpretation int: partial_order "(≤) :: int ⇒ int ⇒ bool"
 rewrites "int.less x y = (x < y)"
proof -
 show "partial_order ((≤) :: int ⇒ int ⇒ bool)"
  by unfold_locales auto
 then interpret int: partial_order "(≤) :: [int, int] ⇒ bool" .
 show "int.less x y = (x < y)"
  unfolding int.less_def by auto
qed
```

theorems from the local interpretation disappear after leaving the proof context — that is, after the succeeding next or qed statement

```
end
```