

Isabelle/FOL — First-Order Logic

Larry Paulson and Markus Wenzel

March 13, 2025

Contents

1	Intuitionistic first-order logic	1
1.1	Syntax and axiomatic basis	1
1.1.1	Equality	1
1.1.2	Propositional logic	2
1.1.3	Quantifiers	2
1.1.4	Definitions	2
1.1.5	Old-style ASCII syntax	3
1.2	Lemmas and proof tools	3
1.2.1	Sequent-style elimination rules for $\wedge \longrightarrow$ and \forall	3
1.2.2	Negation rules, which translate between $\neg P$ and $P \longrightarrow \text{False}$	4
1.2.3	Modus Ponens Tactics	5
1.3	If-and-only-if	5
1.3.1	Destruct rules for \longleftrightarrow similar to Modus Ponens	5
1.4	Unique existence	6
1.4.1	\longleftrightarrow congruence rules for simplification	7
1.5	Equality rules	8
1.6	Simplifications of assumed implications	9
1.7	Intuitionistic Reasoning	11
1.8	Polymorphic congruence rules	12
1.8.1	Congruence rules for predicate letters	13
1.9	Atomizing meta-level rules	13
1.10	Atomizing elimination rules	14
1.11	Calculational rules	14
1.12	“Let” declarations	15
1.13	Intuitionistic simplification rules	15
1.13.1	Conversion into rewrite rules	17
1.13.2	More rewrite rules	18

2	Classical first-order logic	18
2.1	The classical axiom	18
2.2	Lemmas and proof tools	19
2.2.1	Classical introduction rules for \vee and \exists	19
2.3	Special elimination rules	20
2.3.1	Tactics for implication and contradiction	21
3	Classical Reasoner	22
3.1	Classical simplification rules	23
3.1.1	Miniscoping: pushing quantifiers in	23
3.1.2	Named rewrite rules proved for IFOL	24
3.2	Other simple lemmas	25
3.2.1	Monotonicity of implications	26
3.3	Proof by cases and induction	26

1 Intuitionistic first-order logic

```
theory IFOL
  imports Pure
  abbrevs ?< =  $\exists_{\leq 1}$ 
begin
```

```
ML-file <~~/src/Tools/misc-legacy.ML>
ML-file <~~/src/Provers/splitter.ML>
ML-file <~~/src/Provers/hypsubst.ML>
ML-file <~~/src/Tools/IsaPlanner/zipper.ML>
ML-file <~~/src/Tools/IsaPlanner/isand.ML>
ML-file <~~/src/Tools/IsaPlanner/rw-inst.ML>
ML-file <~~/src/Provers/quantifier1.ML>
ML-file <~~/src/Tools/intuitionistic.ML>
ML-file <~~/src/Tools/project-rule.ML>
ML-file <~~/src/Tools/atomize-elim.ML>
```

1.1 Syntax and axiomatic basis

```
setup Pure-Thy.old-appl-syntax-setup
setup <Proofterm.set-preproc (Proof-Rewrite-Rules.standard-preproc [])>

class term
default-sort <term>

typedec1 o

judgment
  Trueprop :: <o  $\Rightarrow$  prop> (⟨⟨notation=judgment⟩-⟩ 5)
```

1.1.1 Equality

axiomatization

$eq :: \langle ['a, 'a] \Rightarrow o \rangle$ (**infixl** $\langle \Rightarrow \rangle$ 50)

where

$refl :: \langle a = a \rangle$ **and**

$subst :: \langle a = b \Longrightarrow P(a) \Longrightarrow P(b) \rangle$

1.1.2 Propositional logic

axiomatization

$False :: \langle o \rangle$ **and**

$conj :: \langle [o, o] \Rightarrow o \rangle$ (**infixr** $\langle \wedge \rangle$ 35) **and**

$disj :: \langle [o, o] \Rightarrow o \rangle$ (**infixr** $\langle \vee \rangle$ 30) **and**

$imp :: \langle [o, o] \Rightarrow o \rangle$ (**infixr** $\langle \longrightarrow \rangle$ 25)

where

$conjI :: \langle \llbracket P; Q \rrbracket \Longrightarrow P \wedge Q \rangle$ **and**

$conjunct1 :: \langle P \wedge Q \Longrightarrow P \rangle$ **and**

$conjunct2 :: \langle P \wedge Q \Longrightarrow Q \rangle$ **and**

$disjI1 :: \langle P \Longrightarrow P \vee Q \rangle$ **and**

$disjI2 :: \langle Q \Longrightarrow P \vee Q \rangle$ **and**

$disjE :: \langle \llbracket P \vee Q; P \Longrightarrow R; Q \Longrightarrow R \rrbracket \Longrightarrow R \rangle$ **and**

$impI :: \langle (P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q \rangle$ **and**

$mp :: \langle \llbracket P \longrightarrow Q; P \rrbracket \Longrightarrow Q \rangle$ **and**

$FalseE :: \langle False \Longrightarrow P \rangle$

1.1.3 Quantifiers

axiomatization

$All :: \langle ('a \Rightarrow o) \Rightarrow o \rangle$ (**binder** $\langle \forall \rangle$ 10) **and**

$Ex :: \langle ('a \Rightarrow o) \Rightarrow o \rangle$ (**binder** $\langle \exists \rangle$ 10)

where

$allI :: \langle (\bigwedge x. P(x)) \Longrightarrow (\forall x. P(x)) \rangle$ **and**

$spec :: \langle (\forall x. P(x)) \Longrightarrow P(x) \rangle$ **and**

$exI :: \langle P(x) \Longrightarrow (\exists x. P(x)) \rangle$ **and**

$exE :: \langle \llbracket \exists x. P(x); \bigwedge x. P(x) \Longrightarrow R \rrbracket \Longrightarrow R \rangle$

1.1.4 Definitions

definition $\langle True \equiv False \longrightarrow False \rangle$

definition $Not \ (\langle \langle open-block notation = \langle prefix \neg \rangle \neg \rangle \ [40] \ 40 \rangle$

where $not-def :: \langle \neg P \equiv P \longrightarrow False \rangle$

definition $iff \ (\text{infixr} \ \langle \longleftrightarrow \rangle \ 25)$

where $\langle P \longleftrightarrow Q \equiv (P \longrightarrow Q) \wedge (Q \longrightarrow P) \rangle$

definition *Uniq* :: $\langle 'a \Rightarrow o \rangle \Rightarrow o$
where $\langle \text{Uniq}(P) \equiv (\forall x y. P(x) \longrightarrow P(y) \longrightarrow y = x) \rangle$

definition *Ex1* :: $\langle 'a \Rightarrow o \rangle \Rightarrow o$ (**binder** $\langle \exists ! \rangle$ 10)
where *ex1-def*: $\langle \exists ! x. P(x) \equiv \exists x. P(x) \wedge (\forall y. P(y) \longrightarrow y = x) \rangle$

axiomatization where — Reflection, admissible

eq-reflection: $\langle (x = y) \Longrightarrow (x \equiv y) \rangle$ **and**
iff-reflection: $\langle (P \longleftrightarrow Q) \Longrightarrow (P \equiv Q) \rangle$

abbreviation *not-equal* :: $\langle ['a, 'a] \Rightarrow o \rangle$ (**infixl** $\langle \neq \rangle$ 50)
where $\langle x \neq y \equiv \neg (x = y) \rangle$

syntax *-Uniq* :: $pttrn \Rightarrow o \Rightarrow o$ ($\langle (\langle \text{indent}=2 \text{ notation}=\langle \text{binder } \exists_{\leq 1} \rangle \exists_{\leq 1} \text{ -./ -} \rangle$
 $[0, 10] \text{ } 10) \rangle$

syntax-consts *-Uniq* \rightleftharpoons *Uniq*

translations $\exists_{\leq 1} x. P \rightleftharpoons \text{CONST } \text{Uniq } (\lambda x. P)$

typed-print-translation \langle

$[(\text{const-syntax } \langle \text{Uniq} \rangle, \text{Syntax-Trans.preserve-binder-abs-tr}' \text{ syntax-const } \langle \text{-Uniq} \rangle)]$

\rangle — to avoid eta-contraction of body

1.1.5 Old-style ASCII syntax

notation (*ASCII*)

not-equal (**infixl** $\langle \sim = \rangle$ 50) **and**

Not ($\langle (\langle \text{open-block notation}=\langle \text{prefix } \sim \rangle \rangle \sim \text{ -}) \rangle$ [40] 40) **and**

conj (**infixr** $\langle \& \rangle$ 35) **and**

disj (**infixr** $\langle | \rangle$ 30) **and**

All (**binder** $\langle ALL \rangle$ 10) **and**

Ex (**binder** $\langle EX \rangle$ 10) **and**

Ex1 (**binder** $\langle EX! \rangle$ 10) **and**

imp (**infixr** $\langle --> \rangle$ 25) **and**

iff (**infixr** $\langle <-> \rangle$ 25)

1.2 Lemmas and proof tools

lemmas *strip* = *impI allI*

lemma *TrueI*: $\langle \text{True} \rangle$

unfolding *True-def* **by** (*rule impI*)

1.2.1 Sequent-style elimination rules for $\wedge \longrightarrow$ and \forall

lemma *conjE*:

assumes *major*: $\langle P \wedge Q \rangle$

and *r*: $\langle \llbracket P; Q \rrbracket \Longrightarrow R \rangle$

shows $\langle R \rangle$

proof (*rule r*)

show *P*

by (*rule major [THEN conjunct1]*)

```

  show  $Q$ 
  by (rule major [THEN conjunct2])
qed

```

```

lemma impE:
  assumes major:  $\langle P \longrightarrow Q \rangle$ 
  and  $\langle P \rangle$ 
  and  $r: \langle Q \Longrightarrow R \rangle$ 
  shows  $\langle R \rangle$ 
proof (rule r)
  show  $Q$ 
  by (rule mp [OF major  $\langle P \rangle$ ])
qed

```

```

lemma allE:
  assumes major:  $\langle \forall x. P(x) \rangle$ 
  and  $r: \langle P(x) \Longrightarrow R \rangle$ 
  shows  $\langle R \rangle$ 
proof (rule r)
  show  $P(x)$ 
  by (rule major [THEN spec])
qed

```

Duplicates the quantifier; for use with `eresolve_tac`.

```

lemma all-dupE:
  assumes major:  $\langle \forall x. P(x) \rangle$ 
  and  $r: \langle \llbracket P(x); \forall x. P(x) \rrbracket \Longrightarrow R \rangle$ 
  shows  $\langle R \rangle$ 
proof (rule r)
  show  $P(x)$ 
  by (rule major [THEN spec])
qed (rule major)

```

1.2.2 Negation rules, which translate between $\neg P$ and $P \longrightarrow False$

```

lemma notI:  $\langle (P \Longrightarrow False) \Longrightarrow \neg P \rangle$ 
  unfolding not-def by (erule impI)

```

```

lemma notE:  $\langle \llbracket \neg P; P \rrbracket \Longrightarrow R \rangle$ 
  unfolding not-def by (erule mp [THEN FalseE])

```

```

lemma rev-notE:  $\langle \llbracket P; \neg P \rrbracket \Longrightarrow R \rangle$ 
  by (erule notE)

```

This is useful with the special implication rules for each kind of P .

```

lemma not-to-imp:
  assumes  $\langle \neg P \rangle$ 
  and  $r: \langle P \longrightarrow False \Longrightarrow Q \rangle$ 
  shows  $\langle Q \rangle$ 

```

```

apply (rule r)
apply (rule impI)
apply (erule notE [OF  $\neg P$ ])
done

```

For substitution into an assumption P , reduce Q to $P \longrightarrow Q$, substitute into this implication, then apply *impI* to move P back into the assumptions.

```

lemma rev-mp:  $\langle \llbracket P; P \longrightarrow Q \rrbracket \Longrightarrow Q \rangle$ 
  by (erule mp)

```

Contrapositive of an inference rule.

```

lemma contrapos:
  assumes major:  $\neg Q$ 
  and minor:  $P \Longrightarrow Q$ 
  shows  $\neg P$ 
  apply (rule major [THEN notE, THEN notI])
  apply (erule minor)
  done

```

1.2.3 Modus Ponens Tactics

Finds $P \longrightarrow Q$ and P in the assumptions, replaces implication by Q .

```

ML <
  fun mp-tac ctxt i =
    eresolve-tac ctxt @{thms notE impE} i THEN assume-tac ctxt i;
  fun eq-mp-tac ctxt i =
    eresolve-tac ctxt @{thms notE impE} i THEN eq-assume-tac i;
  >

```

1.3 If-and-only-if

```

lemma iffI:  $\langle \llbracket P \Longrightarrow Q; Q \Longrightarrow P \rrbracket \Longrightarrow P \longleftrightarrow Q \rangle$ 
  unfolding iff-def
  by (rule conjI; erule impI)

```

```

lemma iffE:
  assumes major:  $P \longleftrightarrow Q$ 
  and r:  $\langle \llbracket P \longrightarrow Q; Q \longrightarrow P \rrbracket \Longrightarrow R \rangle$ 
  shows  $\langle R \rangle$ 
  using major
  unfolding iff-def
  apply (rule conjE)
  apply (erule r)
  apply assumption
  done

```

1.3.1 Destruct rules for \longleftrightarrow similar to Modus Ponens

```

lemma iffD1:  $\langle \llbracket P \longleftrightarrow Q; P \rrbracket \Longrightarrow Q \rangle$ 

```

```

unfolding iff-def
apply (erule conjunct1 [THEN mp])
apply assumption
done

lemma iffD2:  $\langle \llbracket P \longleftrightarrow Q; Q \rrbracket \Longrightarrow P \rangle$ 
unfolding iff-def
apply (erule conjunct2 [THEN mp])
apply assumption
done

lemma rev-iffD1:  $\langle \llbracket P; P \longleftrightarrow Q \rrbracket \Longrightarrow Q \rangle$ 
apply (erule iffD1)
apply assumption
done

lemma rev-iffD2:  $\langle \llbracket Q; P \longleftrightarrow Q \rrbracket \Longrightarrow P \rangle$ 
apply (erule iffD2)
apply assumption
done

lemma iff-refl:  $\langle P \longleftrightarrow P \rangle$ 
by (rule iffI)

lemma iff-sym:  $\langle Q \longleftrightarrow P \Longrightarrow P \longleftrightarrow Q \rangle$ 
apply (erule iffE)
apply (rule iffI)
apply (assumption | erule mp)+
done

lemma iff-trans:  $\langle \llbracket P \longleftrightarrow Q; Q \longleftrightarrow R \rrbracket \Longrightarrow P \longleftrightarrow R \rangle$ 
apply (rule iffI)
apply (assumption | erule iffE | erule (1) notE impE)+
done

```

1.4 Unique existence

NOTE THAT the following 2 quantifications:

- $\exists!x$ such that $[\exists!y \text{ such that } P(x,y)]$ (sequential)
- $\exists!x,y$ such that $P(x,y)$ (simultaneous)

do NOT mean the same thing. The parser treats $\exists!x y.P(x,y)$ as sequential.

```

lemma ex1I:  $\langle P(a) \Longrightarrow (\bigwedge x. P(x) \Longrightarrow x = a) \Longrightarrow \exists!x. P(x) \rangle$ 
unfolding ex1-def
apply (assumption | rule exI conjI allI impI)+
done

```

Sometimes easier to use: the premises have no shared variables. Safe!

```

lemma ex-ex1I:  $\langle \exists x. P(x) \implies (\bigwedge x y. \llbracket P(x); P(y) \rrbracket \implies x = y) \implies \exists !x. P(x) \rangle$ 
  apply (erule exE)
  apply (rule ex1I)
  apply assumption
  apply assumption
  done

```

```

lemma ex1E:  $\langle \exists ! x. P(x) \implies (\bigwedge x. \llbracket P(x); \forall y. P(y) \longrightarrow y = x \rrbracket \implies R) \implies R \rangle$ 
  unfolding ex1-def
  apply (assumption | erule exE conjE) +
  done

```

1.4.1 \longleftrightarrow congruence rules for simplification

Use *iffE* on a premise. For *conj-cong*, *imp-cong*, *all-cong*, *ex-cong*.

```

ML  $\langle$ 
  fun iff-tac ctxt prems i =
    resolve-tac ctxt (prems RL @ {thms iffE}) i THEN
    REPEAT1 (eresolve-tac ctxt @ {thms asm-rl mp} i);
 $\rangle$ 

```

```

method-setup iff =
   $\langle$  Attrib.thms  $\rangle\langle$ 
    (fn prems  $\implies$  fn ctxt  $\implies$  SIMPLE-METHOD' (iff-tac ctxt prems))  $\rangle$ 

```

```

lemma conj-cong:
  assumes  $\langle P \longleftrightarrow P' \rangle$ 
  and  $\langle P' \implies Q \longleftrightarrow Q' \rangle$ 
  shows  $\langle (P \wedge Q) \longleftrightarrow (P' \wedge Q') \rangle$ 
  apply (insert assms)
  apply (assumption | rule iffI conjI | erule iffE conjE mp | iff assms) +
  done

```

Reversed congruence rule! Used in ZF/Order.

```

lemma conj-cong2:
  assumes  $\langle P \longleftrightarrow P' \rangle$ 
  and  $\langle P' \implies Q \longleftrightarrow Q' \rangle$ 
  shows  $\langle (Q \wedge P) \longleftrightarrow (Q' \wedge P') \rangle$ 
  apply (insert assms)
  apply (assumption | rule iffI conjI | erule iffE conjE mp | iff assms) +
  done

```

```

lemma disj-cong:
  assumes  $\langle P \longleftrightarrow P' \rangle$  and  $\langle Q \longleftrightarrow Q' \rangle$ 
  shows  $\langle (P \vee Q) \longleftrightarrow (P' \vee Q') \rangle$ 
  apply (insert assms)
  apply (erule iffE disjE disjI1 disjI2 |

```



```

    assumption | rule iffI | erule (1) notE impE)+
  done

lemma imp-cong:
  assumes  $\langle P \longleftrightarrow P' \rangle$ 
  and  $\langle P' \Longrightarrow Q \longleftrightarrow Q' \rangle$ 
  shows  $\langle (P \longrightarrow Q) \longleftrightarrow (P' \longrightarrow Q') \rangle$ 
  apply (insert assms)
  apply (assumption | rule iffI impI | erule iffE | erule (1) notE impE | iff assms)+
  done

lemma iff-cong:  $\langle \llbracket P \longleftrightarrow P'; Q \longleftrightarrow Q' \rrbracket \Longrightarrow (P \longleftrightarrow Q) \longleftrightarrow (P' \longleftrightarrow Q') \rangle$ 
  apply (erule iffE | assumption | rule iffI | erule (1) notE impE)+
  done

lemma not-cong:  $\langle P \longleftrightarrow P' \Longrightarrow \neg P \longleftrightarrow \neg P' \rangle$ 
  apply (assumption | rule iffI notI | erule (1) notE impE | erule iffE notE)+
  done

lemma all-cong:
  assumes  $\langle \bigwedge x. P(x) \longleftrightarrow Q(x) \rangle$ 
  shows  $\langle (\forall x. P(x)) \longleftrightarrow (\forall x. Q(x)) \rangle$ 
  apply (assumption | rule iffI allI | erule (1) notE impE | erule allE | iff assms)+
  done

lemma ex-cong:
  assumes  $\langle \bigwedge x. P(x) \longleftrightarrow Q(x) \rangle$ 
  shows  $\langle (\exists x. P(x)) \longleftrightarrow (\exists x. Q(x)) \rangle$ 
  apply (erule exE | assumption | rule iffI exI | erule (1) notE impE | iff assms)+
  done

lemma ex1-cong:
  assumes  $\langle \bigwedge x. P(x) \longleftrightarrow Q(x) \rangle$ 
  shows  $\langle (\exists !x. P(x)) \longleftrightarrow (\exists !x. Q(x)) \rangle$ 
  apply (erule ex1E spec [THEN mp] | assumption | rule iffI ex1I | erule (1) notE
    impE | iff assms)+
  done

```

1.5 Equality rules

```

lemma sym:  $\langle a = b \Longrightarrow b = a \rangle$ 
  apply (erule subst)
  apply (rule refl)
  done

lemma trans:  $\langle \llbracket a = b; b = c \rrbracket \Longrightarrow a = c \rangle$ 
  apply (erule subst, assumption)
  done

```

```

lemma not-sym:  $\langle b \neq a \implies a \neq b \rangle$ 
  apply (erule contrapos)
  apply (erule sym)
  done

```

Two theorems for rewriting only one instance of a definition: the first for definitions of formulae and the second for terms.

```

lemma def-imp-iff:  $\langle (A \equiv B) \implies A \longleftrightarrow B \rangle$ 
  apply unfold
  apply (rule iff-refl)
  done

```

```

lemma meta-eq-to-obj-eq:  $\langle (A \equiv B) \implies A = B \rangle$ 
  apply unfold
  apply (rule refl)
  done

```

```

lemma meta-eq-to-iff:  $\langle x \equiv y \implies x \longleftrightarrow y \rangle$ 
  by unfold (rule iff-refl)

```

Substitution.

```

lemma ssubst:  $\langle \llbracket b = a; P(a) \rrbracket \implies P(b) \rangle$ 
  apply (erule sym)
  apply (erule (1) subst)
  done

```

A special case of *ex1E* that would otherwise need quantifier expansion.

```

lemma ex1-equalsE:  $\langle \llbracket \exists !x. P(x); P(a); P(b) \rrbracket \implies a = b \rangle$ 
  apply (erule ex1E)
  apply (rule trans)
  apply (rule-tac [2] sym)
  apply (assumption | erule spec [THEN mp])+
  done

```

1.6 Simplifications of assumed implications

Roy Dyckhoff has proved that *conj-impE*, *disj-impE*, and *imp-impE* used with *mp_tac* (restricted to atomic formulae) is COMPLETE for intuitionistic propositional logic.

See R. Dyckhoff, Contraction-free sequent calculi for intuitionistic logic (preprint, University of St Andrews, 1991).

```

lemma conj-impE:
  assumes major:  $\langle (P \wedge Q) \longrightarrow S \rangle$ 
  and r:  $\langle P \longrightarrow (Q \longrightarrow S) \implies R \rangle$ 
  shows  $\langle R \rangle$ 
  by (assumption | rule conjI impI major [THEN mp] r)+

```

lemma *disj-impE*:
assumes *major*: $\langle (P \vee Q) \longrightarrow S \rangle$
and *r*: $\langle \llbracket P \longrightarrow S; Q \longrightarrow S \rrbracket \Longrightarrow R \rangle$
shows $\langle R \rangle$
by (*assumption* | *rule disjI1 disjI2 impI major [THEN mp] r*) +

Simplifies the implication. Classical version is stronger. Still UNSAFE since Q must be provable – backtracking needed.

lemma *imp-impE*:
assumes *major*: $\langle (P \longrightarrow Q) \longrightarrow S \rangle$
and *r1*: $\langle \llbracket P; Q \longrightarrow S \rrbracket \Longrightarrow Q \rangle$
and *r2*: $\langle S \Longrightarrow R \rangle$
shows $\langle R \rangle$
by (*assumption* | *rule impI major [THEN mp] r1 r2*) +

Simplifies the implication. Classical version is stronger. Still UNSAFE since P must be provable – backtracking needed.

lemma *not-impE*: $\langle \neg P \longrightarrow S \Longrightarrow (P \Longrightarrow \text{False}) \Longrightarrow (S \Longrightarrow R) \Longrightarrow R \rangle$
apply (*drule mp*)
apply (*rule notI* | *assumption*) +
done

Simplifies the implication. UNSAFE.

lemma *iff-impE*:
assumes *major*: $\langle (P \longleftrightarrow Q) \longrightarrow S \rangle$
and *r1*: $\langle \llbracket P; Q \longrightarrow S \rrbracket \Longrightarrow Q \rangle$
and *r2*: $\langle \llbracket Q; P \longrightarrow S \rrbracket \Longrightarrow P \rangle$
and *r3*: $\langle S \Longrightarrow R \rangle$
shows $\langle R \rangle$
by (*assumption* | *rule iffI impI major [THEN mp] r1 r2 r3*) +

What if $(\forall x. \neg \neg P(x)) \longrightarrow \neg \neg (\forall x. P(x))$ is an assumption? UNSAFE.

lemma *all-impE*:
assumes *major*: $\langle (\forall x. P(x)) \longrightarrow S \rangle$
and *r1*: $\langle \bigwedge x. P(x) \rangle$
and *r2*: $\langle S \Longrightarrow R \rangle$
shows $\langle R \rangle$
by (*rule allI impI major [THEN mp] r1 r2*) +

Unsafe: $\exists x. P(x) \longrightarrow S$ is equivalent to $\forall x. P(x) \longrightarrow S$.

lemma *ex-impE*:
assumes *major*: $\langle (\exists x. P(x)) \longrightarrow S \rangle$
and *r*: $\langle P(x) \longrightarrow S \Longrightarrow R \rangle$
shows $\langle R \rangle$
by (*assumption* | *rule exI impI major [THEN mp] r*) +

Courtesy of Krzysztof Grabczewski.

lemma *disj-imp-disj*: $\langle P \vee Q \Longrightarrow (P \Longrightarrow R) \Longrightarrow (Q \Longrightarrow S) \Longrightarrow R \vee S \rangle$

```

apply (erule disjE)
apply (rule disjI1) apply assumption
apply (rule disjI2) apply assumption
done

```

```

ML <
structure Project-Rule = Project-Rule
(
  val conjunct1 = @{thm conjunct1}
  val conjunct2 = @{thm conjunct2}
  val mp = @{thm mp}
)
>

```

ML-file <*fologic.ML*>

lemma *thin-refl*: <[$x = x$; *PROP W*] \implies *PROP W*> .

```

ML <
structure Hypsubst = Hypsubst
(
  val dest-eq = FOLogic.dest-eq
  val dest-Trueprop = dest-judgment
  val dest-imp = FOLogic.dest-imp
  val eq-reflection = @{thm eq-reflection}
  val rev-eq-reflection = @{thm meta-eq-to-obj-eq}
  val imp-intr = @{thm impI}
  val rev-mp = @{thm rev-mp}
  val subst = @{thm subst}
  val sym = @{thm sym}
  val thin-refl = @{thm thin-refl}
);
open Hypsubst;
>

```

ML-file <*intprover.ML*>

1.7 Intuitionistic Reasoning

setup <*Intuitionistic.method-setup* **binding** <*iprover*>>

```

lemma impE':
  assumes 1: < $P \longrightarrow Q$ >
    and 2: < $Q \implies R$ >
    and 3: < $P \longrightarrow Q \implies P$ >
  shows < $R$ >
proof -
  from 3 and 1 have < $P$ > .
  with 1 have < $Q$ > by (rule impE)

```

with 2 show $\langle R \rangle$.
qed

lemma *allE'*:
assumes 1: $\langle \forall x. P(x) \rangle$
and 2: $\langle P(x) \implies \forall x. P(x) \implies Q \rangle$
shows $\langle Q \rangle$
proof –
from 1 have $\langle P(x) \rangle$ by (rule spec)
from this and 1 show $\langle Q \rangle$ by (rule 2)
qed

lemma *notE'*:
assumes 1: $\langle \neg P \rangle$
and 2: $\langle \neg P \implies P \rangle$
shows $\langle R \rangle$
proof –
from 2 and 1 have $\langle P \rangle$.
with 1 show $\langle R \rangle$ by (rule notE)
qed

lemmas [*Pure.elim!*] = *disjE iffE FalseE conjE exE*
and [*Pure.intro!*] = *iffI conjI impI TrueI notI allI refl*
and [*Pure.elim 2*] = *allE notE' impE'*
and [*Pure.intro*] = *exI disjI2 disjI1*

setup \langle
Context-Rules.addSWrapper
(fn ctxt => fn tac => hyp-subst-tac ctxt ORELSE' tac)
 \rangle

lemma *iff-not-sym*: $\langle \neg (Q \longleftrightarrow P) \implies \neg (P \longleftrightarrow Q) \rangle$
by iprover

lemmas [*sym*] = *sym iff-sym not-sym iff-not-sym*
and [*Pure.elim?*] = *iffD1 iffD2 impE*

lemma *eq-commute*: $\langle a = b \longleftrightarrow b = a \rangle$
by iprover

1.8 Polymorphic congruence rules

lemma *subst-context*: $\langle a = b \implies t(a) = t(b) \rangle$
by iprover

lemma *subst-context2*: $\langle \llbracket a = b; c = d \rrbracket \implies t(a,c) = t(b,d) \rangle$
by iprover

lemma *subst-context3*: $\langle \llbracket a = b; c = d; e = f \rrbracket \implies t(a,c,e) = t(b,d,f) \rangle$
by *iprover*

Useful with **eresolve_tac** for proving equalities from known equalities.

$a = b \mid \mid c = d$

lemma *box-equals*: $\langle \llbracket a = b; a = c; b = d \rrbracket \implies c = d \rangle$
by *iprover*

Dual of *box-equals*: for proving equalities backwards.

lemma *simp-equals*: $\langle \llbracket a = c; b = d; c = d \rrbracket \implies a = b \rangle$
by *iprover*

1.8.1 Congruence rules for predicate letters

lemma *pred1-cong*: $\langle a = a' \implies P(a) \longleftrightarrow P(a') \rangle$
by *iprover*

lemma *pred2-cong*: $\langle \llbracket a = a'; b = b' \rrbracket \implies P(a,b) \longleftrightarrow P(a',b') \rangle$
by *iprover*

lemma *pred3-cong*: $\langle \llbracket a = a'; b = b'; c = c' \rrbracket \implies P(a,b,c) \longleftrightarrow P(a',b',c') \rangle$
by *iprover*

Special case for the equality predicate!

lemma *eq-cong*: $\langle \llbracket a = a'; b = b' \rrbracket \implies a = b \longleftrightarrow a' = b' \rangle$
by *iprover*

1.9 Atomizing meta-level rules

lemma *atomize-all* [*atomize*]: $\langle (\bigwedge x. P(x)) \equiv \text{Trueprop } (\forall x. P(x)) \rangle$

proof

assume $\langle \bigwedge x. P(x) \rangle$

then show $\langle \forall x. P(x) \rangle$..

next

assume $\langle \forall x. P(x) \rangle$

then show $\langle \bigwedge x. P(x) \rangle$..

qed

lemma *atomize-imp* [*atomize*]: $\langle (A \implies B) \equiv \text{Trueprop } (A \longrightarrow B) \rangle$

proof

assume $\langle A \implies B \rangle$

then show $\langle A \longrightarrow B \rangle$..

next

assume $\langle A \longrightarrow B \rangle$ **and** $\langle A \rangle$

then show $\langle B \rangle$ **by** (*rule mp*)

qed

lemma *atomize-eq* [*atomize*]: $\langle (x \equiv y) \equiv \text{Trueprop } (x = y) \rangle$

```

proof
  assume  $\langle x \equiv y \rangle$ 
  show  $\langle x = y \rangle$  unfolding  $\langle x \equiv y \rangle$  by (rule refl)
next
  assume  $\langle x = y \rangle$ 
  then show  $\langle x \equiv y \rangle$  by (rule eq-reflection)
qed

lemma atomize-iff [atomize]:  $\langle (A \equiv B) \equiv \text{Trueprop } (A \longleftrightarrow B) \rangle$ 
proof
  assume  $\langle A \equiv B \rangle$ 
  show  $\langle A \longleftrightarrow B \rangle$  unfolding  $\langle A \equiv B \rangle$  by (rule iff-refl)
next
  assume  $\langle A \longleftrightarrow B \rangle$ 
  then show  $\langle A \equiv B \rangle$  by (rule iff-reflection)
qed

lemma atomize-conj [atomize]:  $\langle (A \&\&\& B) \equiv \text{Trueprop } (A \wedge B) \rangle$ 
proof
  assume conj:  $\langle A \&\&\& B \rangle$ 
  show  $\langle A \wedge B \rangle$ 
  proof (rule conjI)
    from conj show  $\langle A \rangle$  by (rule conjunctionD1)
    from conj show  $\langle B \rangle$  by (rule conjunctionD2)
  qed
next
  assume conj:  $\langle A \wedge B \rangle$ 
  show  $\langle A \&\&\& B \rangle$ 
  proof –
    from conj show  $\langle A \rangle$  ..
    from conj show  $\langle B \rangle$  ..
  qed
qed

lemmas [symmetric, rulify] = atomize-all atomize-imp
  and [symmetric, defn] = atomize-all atomize-imp atomize-eq atomize-iff

```

1.10 Atomizing elimination rules

```

lemma atomize-exL[atomize-elim]:  $\langle (\bigwedge x. P(x) \implies Q) \equiv ((\exists x. P(x)) \implies Q) \rangle$ 
  by rule iprover+

lemma atomize-conjL[atomize-elim]:  $\langle (A \implies B \implies C) \equiv (A \wedge B \implies C) \rangle$ 
  by rule iprover+

lemma atomize-disjL[atomize-elim]:  $\langle ((A \implies C) \implies (B \implies C) \implies C) \equiv ((A \vee B \implies C) \implies C) \rangle$ 
  by rule iprover+

```

lemma *atomize-elimL*[*atomize-elim*]: $\langle (\bigwedge B. (A \implies B) \implies B) \equiv \text{Trueprop}(A) \rangle \dots$

1.11 Calculational rules

lemma *forw-subst*: $\langle a = b \implies P(b) \implies P(a) \rangle$
by (*rule ssubst*)

lemma *back-subst*: $\langle P(a) \implies a = b \implies P(b) \rangle$
by (*rule subst*)

Note that this list of rules is in reverse order of priorities.

lemmas *basic-trans-rules* [*trans*] =
forw-subst
back-subst
rev-mp
mp
trans

1.12 “Let” declarations

nonterminal *letbinds* and *letbind*

definition *Let* :: $\langle [a::\{\}, 'a \Rightarrow 'b] \Rightarrow ('b::\{\}) \rangle$
where $\langle \text{Let}(s, f) \equiv f(s) \rangle$

syntax

-bind :: $\langle [pttrn, 'a] \Rightarrow \text{letbind} \rangle \quad (\langle (\langle \text{indent}=2 \text{ notation}=\langle \text{infix let binding} \rangle) -$
 $= / - \rangle 10)$
:: $\langle \text{letbind} \Rightarrow \text{letbinds} \rangle \quad (\langle - \rangle)$
-binds :: $\langle [\text{letbind}, \text{letbinds}] \Rightarrow \text{letbinds} \rangle \quad (\langle -; / - \rangle)$
-Let :: $\langle [\text{letbinds}, 'a] \Rightarrow 'a \rangle \quad (\langle (\langle \text{notation}=\langle \text{mixfix let expression} \rangle) \text{let} (-) /$
 $\text{in} (-) \rangle 10)$

syntax-consts

-Let \Rightarrow *Let*

translations

-Let(*-binds*(*b*, *bs*), *e*) == *-Let*(*b*, *-Let*(*bs*, *e*))
 $\text{let } x = a \text{ in } e$ == *CONST* *Let*(*a*, $\lambda x. e$)

lemma *LetI*:

assumes $\langle \bigwedge x. x = t \implies P(u(x)) \rangle$
shows $\langle P(\text{let } x = t \text{ in } u(x)) \rangle$
unfolding *Let-def*
apply (*rule refl* [*THEN assms*])
done

1.13 Intuitionistic simplification rules

lemma *conj-simps*:

$\langle P \wedge \text{True} \longleftrightarrow P \rangle$
 $\langle \text{True} \wedge P \longleftrightarrow P \rangle$

$\langle P \wedge \text{False} \longleftrightarrow \text{False} \rangle$
 $\langle \text{False} \wedge P \longleftrightarrow \text{False} \rangle$
 $\langle P \wedge P \longleftrightarrow P \rangle$
 $\langle P \wedge P \wedge Q \longleftrightarrow P \wedge Q \rangle$
 $\langle P \wedge \neg P \longleftrightarrow \text{False} \rangle$
 $\langle \neg P \wedge P \longleftrightarrow \text{False} \rangle$
 $\langle (P \wedge Q) \wedge R \longleftrightarrow P \wedge (Q \wedge R) \rangle$
by *iprover*+

lemma *disj-simps*:

$\langle P \vee \text{True} \longleftrightarrow \text{True} \rangle$
 $\langle \text{True} \vee P \longleftrightarrow \text{True} \rangle$
 $\langle P \vee \text{False} \longleftrightarrow P \rangle$
 $\langle \text{False} \vee P \longleftrightarrow P \rangle$
 $\langle P \vee P \longleftrightarrow P \rangle$
 $\langle P \vee P \vee Q \longleftrightarrow P \vee Q \rangle$
 $\langle (P \vee Q) \vee R \longleftrightarrow P \vee (Q \vee R) \rangle$
by *iprover*+

lemma *not-simps*:

$\langle \neg (P \vee Q) \longleftrightarrow \neg P \wedge \neg Q \rangle$
 $\langle \neg \text{False} \longleftrightarrow \text{True} \rangle$
 $\langle \neg \text{True} \longleftrightarrow \text{False} \rangle$
by *iprover*+

lemma *imp-simps*:

$\langle (P \longrightarrow \text{False}) \longleftrightarrow \neg P \rangle$
 $\langle (P \longrightarrow \text{True}) \longleftrightarrow \text{True} \rangle$
 $\langle (\text{False} \longrightarrow P) \longleftrightarrow \text{True} \rangle$
 $\langle (\text{True} \longrightarrow P) \longleftrightarrow P \rangle$
 $\langle (P \longrightarrow P) \longleftrightarrow \text{True} \rangle$
 $\langle (P \longrightarrow \neg P) \longleftrightarrow \neg P \rangle$
by *iprover*+

lemma *iff-simps*:

$\langle (\text{True} \longleftrightarrow P) \longleftrightarrow P \rangle$
 $\langle (P \longleftrightarrow \text{True}) \longleftrightarrow P \rangle$
 $\langle (P \longleftrightarrow P) \longleftrightarrow \text{True} \rangle$
 $\langle (\text{False} \longleftrightarrow P) \longleftrightarrow \neg P \rangle$
 $\langle (P \longleftrightarrow \text{False}) \longleftrightarrow \neg P \rangle$
by *iprover*+

The $x = t$ versions are needed for the simplification procedures.

lemma *quant-simps*:

$\langle \bigwedge P. (\forall x. P) \longleftrightarrow P \rangle$
 $\langle (\forall x. x = t \longrightarrow P(x)) \longleftrightarrow P(t) \rangle$
 $\langle (\forall x. t = x \longrightarrow P(x)) \longleftrightarrow P(t) \rangle$
 $\langle \bigwedge P. (\exists x. P) \longleftrightarrow P \rangle$
 $\langle \exists x. x = t \rangle$

```

⟨∃ x. t = x⟩
⟨(∃ x. x = t ∧ P(x)) ⟷ P(t)⟩
⟨(∃ x. t = x ∧ P(x)) ⟷ P(t)⟩
by iprover +

```

These are NOT supplied by default!

lemma *distrib-simps*:

```

⟨P ∧ (Q ∨ R) ⟷ P ∧ Q ∨ P ∧ R⟩
⟨(Q ∨ R) ∧ P ⟷ Q ∧ P ∨ R ∧ P⟩
⟨(P ∨ Q ⟶ R) ⟷ (P ⟶ R) ∧ (Q ⟶ R)⟩
by iprover +

```

lemma *subst-all*:

```

⟨(∧ x. x = a ⟹ PROP P(x)) ≡ PROP P(a)⟩
⟨(∧ x. a = x ⟹ PROP P(x)) ≡ PROP P(a)⟩
proof –
  show ⟨(∧ x. x = a ⟹ PROP P(x)) ≡ PROP P(a)⟩
  proof (rule equal-intr-rule)
    assume *: ⟨∧ x. x = a ⟹ PROP P(x)⟩
    show ⟨PROP P(a)⟩
    by (rule *) (rule refl)
  next
    fix x
    assume ⟨PROP P(a)⟩ and ⟨x = a⟩
    from ⟨x = a⟩ have ⟨x ≡ a⟩
    by (rule eq-reflection)
    with ⟨PROP P(a)⟩ show ⟨PROP P(x)⟩
    by simp
  qed
  show ⟨(∧ x. a = x ⟹ PROP P(x)) ≡ PROP P(a)⟩
  proof (rule equal-intr-rule)
    assume *: ⟨∧ x. a = x ⟹ PROP P(x)⟩
    show ⟨PROP P(a)⟩
    by (rule *) (rule refl)
  next
    fix x
    assume ⟨PROP P(a)⟩ and ⟨a = x⟩
    from ⟨a = x⟩ have ⟨a ≡ x⟩
    by (rule eq-reflection)
    with ⟨PROP P(a)⟩ show ⟨PROP P(x)⟩
    by simp
  qed
qed

```

1.13.1 Conversion into rewrite rules

```

lemma P-iff-F: ⟨¬ P ⟹ (P ⟷ False)⟩
  by iprover
lemma iff-reflection-F: ⟨¬ P ⟹ (P ≡ False)⟩

```

```

    by (rule P-iff-F [THEN iff-reflection])

lemma P-iff-T:  $\langle P \implies (P \longleftrightarrow \text{True}) \rangle$ 
  by iprover
lemma iff-reflection-T:  $\langle P \implies (P \equiv \text{True}) \rangle$ 
  by (rule P-iff-T [THEN iff-reflection])

1.13.2 More rewrite rules

lemma conj-commute:  $\langle P \wedge Q \longleftrightarrow Q \wedge P \rangle$  by iprover
lemma conj-left-commute:  $\langle P \wedge (Q \wedge R) \longleftrightarrow Q \wedge (P \wedge R) \rangle$  by iprover
lemmas conj-comms = conj-commute conj-left-commute

lemma disj-commute:  $\langle P \vee Q \longleftrightarrow Q \vee P \rangle$  by iprover
lemma disj-left-commute:  $\langle P \vee (Q \vee R) \longleftrightarrow Q \vee (P \vee R) \rangle$  by iprover
lemmas disj-comms = disj-commute disj-left-commute

lemma conj-disj-distribL:  $\langle P \wedge (Q \vee R) \longleftrightarrow (P \wedge Q \vee P \wedge R) \rangle$  by iprover
lemma conj-disj-distribR:  $\langle (P \vee Q) \wedge R \longleftrightarrow (P \wedge R \vee Q \wedge R) \rangle$  by iprover

lemma disj-conj-distribL:  $\langle P \vee (Q \wedge R) \longleftrightarrow (P \vee Q) \wedge (P \vee R) \rangle$  by iprover
lemma disj-conj-distribR:  $\langle (P \wedge Q) \vee R \longleftrightarrow (P \vee R) \wedge (Q \vee R) \rangle$  by iprover

lemma imp-conj-distrib:  $\langle (P \longrightarrow (Q \wedge R)) \longleftrightarrow (P \longrightarrow Q) \wedge (P \longrightarrow R) \rangle$  by iprover
lemma imp-conj:  $\langle ((P \wedge Q) \longrightarrow R) \longleftrightarrow (P \longrightarrow (Q \longrightarrow R)) \rangle$  by iprover
lemma imp-disj:  $\langle (P \vee Q \longrightarrow R) \longleftrightarrow (P \longrightarrow R) \wedge (Q \longrightarrow R) \rangle$  by iprover

lemma de-Morgan-disj:  $\langle (\neg (P \vee Q)) \longleftrightarrow (\neg P \wedge \neg Q) \rangle$  by iprover

lemma not-ex:  $\langle (\neg (\exists x. P(x))) \longleftrightarrow (\forall x. \neg P(x)) \rangle$  by iprover
lemma imp-ex:  $\langle ((\exists x. P(x)) \longrightarrow Q) \longleftrightarrow (\forall x. P(x) \longrightarrow Q) \rangle$  by iprover

lemma ex-disj-distrib:  $\langle (\exists x. P(x) \vee Q(x)) \longleftrightarrow ((\exists x. P(x)) \vee (\exists x. Q(x))) \rangle$ 
  by iprover

lemma all-conj-distrib:  $\langle (\forall x. P(x) \wedge Q(x)) \longleftrightarrow ((\forall x. P(x)) \wedge (\forall x. Q(x))) \rangle$ 
  by iprover

end

```

2 Classical first-order logic

```

theory FOL
  imports IFOL
  keywords print-claset print-induct-rules :: diag
begin

ML-file <~~/src/Provers/classical.ML>

```

ML-file $\langle \sim\sim/src/Provers/blast.ML \rangle$
 ML-file $\langle \sim\sim/src/Provers/classimp.ML \rangle$

2.1 The classical axiom

axiomatization where

classical: $\langle (\neg P \implies P) \implies P \rangle$

2.2 Lemmas and proof tools

lemma *ccontr*: $\langle (\neg P \implies False) \implies P \rangle$
 by (*erule FalseE [THEN classical]*)

2.2.1 Classical introduction rules for \vee and \exists

lemma *disjCI*: $\langle (\neg Q \implies P) \implies P \vee Q \rangle$
 apply (*rule classical*)
 apply (*assumption | erule meta-mp | rule disjI1 notI*) +
 apply (*erule notE disjI2*) +
 done

Introduction rule involving only \exists

lemma *ex-classical*:
 assumes *r*: $\langle \neg (\exists x. P(x)) \implies P(a) \rangle$
 shows $\langle \exists x. P(x) \rangle$
 apply (*rule classical*)
 apply (*rule exI, erule r*)
 done

Version of above, simplifying $\neg\exists$ to $\forall\neg$.

lemma *exCI*:
 assumes *r*: $\langle \forall x. \neg P(x) \implies P(a) \rangle$
 shows $\langle \exists x. P(x) \rangle$
 apply (*rule ex-classical*)
 apply (*rule notI [THEN allI, THEN r]*)
 apply (*erule notE*)
 apply (*erule exI*)
 done

lemma *excluded-middle*: $\langle \neg P \vee P \rangle$
 apply (*rule disjCI*)
 apply *assumption*
 done

lemma *case-split* [*case-names True False*]:
 assumes *r1*: $\langle P \implies Q \rangle$
 and *r2*: $\langle \neg P \implies Q \rangle$
 shows $\langle Q \rangle$
 apply (*rule excluded-middle [THEN disjE]*)

```

    apply (erule r2)
    apply (erule r1)
  done

ML <
  fun case-tac ctxt a fixes =
    Rule-Insts.res-inst-tac ctxt [(((P, 0), Position.none), a)] fixes @ {thm case-split};
>

method-setup case-tac = <
  Args.goal-spec -- Scan.lift (Parse.embedded-inner-syntax -- Parse.for-fixes)
>>
  (fn (quant, (s, fixes)) => fn ctxt => SIMPLE-METHOD'' quant (case-tac ctxt
s fixes))
> case-tac emulation (dynamic instantiation!)

```

2.3 Special elimination rules

Classical implies (\longrightarrow) elimination.

```

lemma impCE:
  assumes major: <P  $\longrightarrow$  Q>
    and r1: <¬ P  $\Longrightarrow$  R>
    and r2: <Q  $\Longrightarrow$  R>
  shows <R>
  apply (rule excluded-middle [THEN disjE])
  apply (erule r1)
  apply (erule r2)
  apply (erule major [THEN mp])
  done

```

This version of \longrightarrow elimination works on Q before P . It works best for those cases in which P holds “almost everywhere”. Can’t install as default: would break old proofs.

```

lemma impCE':
  assumes major: <P  $\longrightarrow$  Q>
    and r1: <Q  $\Longrightarrow$  R>
    and r2: <¬ P  $\Longrightarrow$  R>
  shows <R>
  apply (rule excluded-middle [THEN disjE])
  apply (erule r2)
  apply (erule r1)
  apply (erule major [THEN mp])
  done

```

Double negation law.

```

lemma notnotD: <¬ ¬ P  $\Longrightarrow$  P>
  apply (rule classical)
  apply (erule notE)

```

```

apply assumption
done

lemma contrapos2:  $\langle \llbracket Q; \neg P \implies \neg Q \rrbracket \implies P \rangle$ 
  apply (rule classical)
  apply (drule (1) meta-mp)
  apply (erule (1) notE)
done

```

2.3.1 Tactics for implication and contradiction

Classical \longleftrightarrow elimination. Proof substitutes $P = Q$ in $\neg P \implies \neg Q$ and $P \implies Q$.

```

lemma iffCE:
  assumes major:  $\langle P \longleftrightarrow Q \rangle$ 
    and r1:  $\langle \llbracket P; Q \rrbracket \implies R \rangle$ 
    and r2:  $\langle \llbracket \neg P; \neg Q \rrbracket \implies R \rangle$ 
  shows  $\langle R \rangle$ 
  apply (rule major [unfolded iff-def, THEN conjE])
  apply (elim impCE)
    apply (erule (1) r2)
    apply (erule (1) notE) +
  apply (erule (1) r1)
done

```

```

lemma alt-ex1E:
  assumes major:  $\langle \exists! x. P(x) \rangle$ 
    and r:  $\langle \bigwedge x. \llbracket P(x); \forall y y'. P(y) \wedge P(y') \longrightarrow y = y' \rrbracket \implies R \rangle$ 
  shows  $\langle R \rangle$ 
  using major
proof (rule ex1E)
  fix x
  assume *:  $\langle \forall y. P(y) \longrightarrow y = x \rangle$ 
  assume  $\langle P(x) \rangle$ 
  then show  $\langle R \rangle$ 
  proof (rule r)
  {
    fix y y'
    assume  $\langle P(y) \rangle$  and  $\langle P(y') \rangle$ 
    with * have  $\langle x = y \rangle$  and  $\langle x = y' \rangle$ 
    by - (tactic IntPr.fast-tac context 1) +
    then have  $\langle y = y' \rangle$  by (rule subst)
  } note r' = this
  show  $\langle \forall y y'. P(y) \wedge P(y') \longrightarrow y = y' \rangle$ 
    by (intro strip, elim conjE) (rule r')
qed
qed

```

lemma *imp-elim*: $\langle P \longrightarrow Q \Longrightarrow (\neg R \Longrightarrow P) \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R \rangle$
by (*rule classical*) *iprover*

lemma *swap*: $\langle \neg P \Longrightarrow (\neg R \Longrightarrow P) \Longrightarrow R \rangle$
by (*rule classical*) *iprover*

3 Classical Reasoner

ML \langle

structure Cla = Classical

(
val imp-elim = @{*thm imp-elim*}
val not-elim = @{*thm notE*}
val swap = @{*thm swap*}
val classical = @{*thm classical*}
val sizef = *size-of-thm*
val hyp-subst-tacs = [*hyp-subst-tac*]
);

structure Basic-Classical: BASIC-CLASSICAL = Cla;

open Basic-Classical;

\rangle

lemmas [*intro!*] = *refl TrueI conjI disjCI impI notI iffI*
and [*elim!*] = *conjE disjE impCE FalseE iffCE*

ML \langle *val prop-cs* = *claset-of context* \rangle

lemmas [*intro!*] = *allI ex-ex1I*

and [*intro*] = *exI*

and [*elim!*] = *exE alt-ex1E*

and [*elim*] = *allE*

ML \langle *val FOL-cs* = *claset-of context* \rangle

ML \langle

structure Blast = Blast

(
structure Classical = Cla
val Trueprop-const = *dest-Const Const* \langle *Trueprop* \rangle
val equality-name = *const-name* \langle *eq* \rangle
val not-name = *const-name* \langle *Not* \rangle
val notE = @{*thm notE*}
val ccontr = @{*thm ccontr*}
val hyp-subst-tac = *Hypsubst.blast-hyp-subst-tac*
);

val blast-tac = *Blast.blast-tac*;

\rangle

lemma *ex1-functional*: $\langle \llbracket \exists! z. P(a,z); P(a,b); P(a,c) \rrbracket \implies b = c \rangle$
by *blast*

Elimination of *True* from assumptions:

lemma *True-implies-equals*: $\langle (True \implies PROP P) \equiv PROP P \rangle$

proof

assume $\langle True \implies PROP P \rangle$

from this and TrueI show $\langle PROP P \rangle$.

next

assume $\langle PROP P \rangle$

then show $\langle PROP P \rangle$.

qed

lemma *uncurry*: $\langle P \longrightarrow Q \longrightarrow R \implies P \wedge Q \longrightarrow R \rangle$

by *blast*

lemma *iff-allI*: $\langle (\bigwedge x. P(x) \longleftrightarrow Q(x)) \implies (\forall x. P(x)) \longleftrightarrow (\forall x. Q(x)) \rangle$

by *blast*

lemma *iff-exI*: $\langle (\bigwedge x. P(x) \longleftrightarrow Q(x)) \implies (\exists x. P(x)) \longleftrightarrow (\exists x. Q(x)) \rangle$

by *blast*

lemma *all-comm*: $\langle (\forall x y. P(x,y)) \longleftrightarrow (\forall y x. P(x,y)) \rangle$

by *blast*

lemma *ex-comm*: $\langle (\exists x y. P(x,y)) \longleftrightarrow (\exists y x. P(x,y)) \rangle$

by *blast*

3.1 Classical simplification rules

Avoids duplication of subgoals after *expand-if*, when the true and false cases boil down to the same thing.

lemma *cases-simp*: $\langle (P \longrightarrow Q) \wedge (\neg P \longrightarrow Q) \longleftrightarrow Q \rangle$

by *blast*

3.1.1 Miniscoping: pushing quantifiers in

We do NOT distribute of \forall over \wedge , or dually that of \exists over \vee .

Baaz and Leitsch, On Skolemization and Proof Complexity (1994) show that this step can increase proof length!

Existential miniscoping.

lemma *int-ex-simps*:

$\langle \bigwedge P Q. (\exists x. P(x) \wedge Q) \longleftrightarrow (\exists x. P(x)) \wedge Q \rangle$

$\langle \bigwedge P Q. (\exists x. P \wedge Q(x)) \longleftrightarrow P \wedge (\exists x. Q(x)) \rangle$

$\langle \bigwedge P \ Q. (\exists x. P(x) \vee Q) \longleftrightarrow (\exists x. P(x)) \vee Q \rangle$
 $\langle \bigwedge P \ Q. (\exists x. P \vee Q(x)) \longleftrightarrow P \vee (\exists x. Q(x)) \rangle$
by *iprover*+

Classical rules.

lemma *cla-ex-simps*:

$\langle \bigwedge P \ Q. (\exists x. P(x) \longrightarrow Q) \longleftrightarrow (\forall x. P(x)) \longrightarrow Q \rangle$
 $\langle \bigwedge P \ Q. (\exists x. P \longrightarrow Q(x)) \longleftrightarrow P \longrightarrow (\exists x. Q(x)) \rangle$
by *blast*+

lemmas *ex-simps* = *int-ex-simps* *cla-ex-simps*

Universal miniscoping.

lemma *int-all-simps*:

$\langle \bigwedge P \ Q. (\forall x. P(x) \wedge Q) \longleftrightarrow (\forall x. P(x)) \wedge Q \rangle$
 $\langle \bigwedge P \ Q. (\forall x. P \wedge Q(x)) \longleftrightarrow P \wedge (\forall x. Q(x)) \rangle$
 $\langle \bigwedge P \ Q. (\forall x. P(x) \longrightarrow Q) \longleftrightarrow (\exists x. P(x)) \longrightarrow Q \rangle$
 $\langle \bigwedge P \ Q. (\forall x. P \longrightarrow Q(x)) \longleftrightarrow P \longrightarrow (\forall x. Q(x)) \rangle$
by *iprover*+

Classical rules.

lemma *cla-all-simps*:

$\langle \bigwedge P \ Q. (\forall x. P(x) \vee Q) \longleftrightarrow (\forall x. P(x)) \vee Q \rangle$
 $\langle \bigwedge P \ Q. (\forall x. P \vee Q(x)) \longleftrightarrow P \vee (\forall x. Q(x)) \rangle$
by *blast*+

lemmas *all-simps* = *int-all-simps* *cla-all-simps*

3.1.2 Named rewrite rules proved for IFOL

lemma *imp-disj1*: $\langle (P \longrightarrow Q) \vee R \longleftrightarrow (P \longrightarrow Q \vee R) \rangle$ **by** *blast*

lemma *imp-disj2*: $\langle Q \vee (P \longrightarrow R) \longleftrightarrow (P \longrightarrow Q \vee R) \rangle$ **by** *blast*

lemma *de-Morgan-conj*: $\langle (\neg (P \wedge Q)) \longleftrightarrow (\neg P \vee \neg Q) \rangle$ **by** *blast*

lemma *not-imp*: $\langle \neg (P \longrightarrow Q) \longleftrightarrow (P \wedge \neg Q) \rangle$ **by** *blast*

lemma *not-iff*: $\langle \neg (P \longleftrightarrow Q) \longleftrightarrow (P \longleftrightarrow \neg Q) \rangle$ **by** *blast*

lemma *not-all*: $\langle (\neg (\forall x. P(x))) \longleftrightarrow (\exists x. \neg P(x)) \rangle$ **by** *blast*

lemma *imp-all*: $\langle ((\forall x. P(x)) \longrightarrow Q) \longleftrightarrow (\exists x. P(x) \longrightarrow Q) \rangle$ **by** *blast*

lemmas *meta-simps* =

triv-forall-equality — prunes params
True-implies-equals — prune asms *True*

lemmas *IFOL-simps* =

reft [*THEN* *P-iff-T*] *conj-simps* *disj-simps* *not-simps*
imp-simps *iff-simps* *quant-simps*

lemma *notFalseI*: $\langle \neg \text{False} \rangle$ **by** *iprover*

lemma *cla-simps-misc*:

$\langle \neg (P \wedge Q) \longleftrightarrow \neg P \vee \neg Q \rangle$
 $\langle P \vee \neg P \rangle$
 $\langle \neg P \vee P \rangle$
 $\langle \neg \neg P \longleftrightarrow P \rangle$
 $\langle (\neg P \longrightarrow P) \longleftrightarrow P \rangle$
 $\langle (\neg P \longleftrightarrow \neg Q) \longleftrightarrow (P \longleftrightarrow Q) \rangle$ **by** *blast+*

lemmas *cla-simps* =

de-Morgan-conj de-Morgan-disj imp-disj1 imp-disj2
not-imp not-all not-ex cases-simp cla-simps-misc

ML-file $\langle \text{simpdata.ML} \rangle$

simproc-setup *defined-Ex* $\langle \exists x. P(x) \rangle = \langle K \text{ Quantifier1.rearrange-Ex} \rangle$
simproc-setup *defined-All* $\langle \forall x. P(x) \rangle = \langle K \text{ Quantifier1.rearrange-All} \rangle$
simproc-setup *defined-all* $\langle \bigwedge x. \text{PROP } P(x) \rangle = \langle K \text{ Quantifier1.rearrange-all} \rangle$

ML \langle

*(*intuitionistic simprules only*)*

val *IFOL-ss* =

put-simpset *FOL-basic-ss* **context**
addsims @{*thms meta-simps IFOL-simps int-ex-simps int-all-simps subst-all*}
 $|>$ *Simplifier.add-proc* **simproc** $\langle \text{defined-All} \rangle$
 $|>$ *Simplifier.add-proc* **simproc** $\langle \text{defined-Ex} \rangle$
 $|>$ *Simplifier.add-cong* @{*thm imp-cong*}
 $|>$ *simpset-of*;

*(*classical simprules too*)*

val *FOL-ss* =

put-simpset *IFOL-ss* **context**
addsims @{*thms cla-simps cla-ex-simps cla-all-simps*}
 $|>$ *simpset-of*;

\rangle

setup \langle

map-theory-simpset (*put-simpset* *FOL-ss*) $\#>$
Simplifier.method-setup *Splitter.split-modifiers*

\rangle

ML-file $\langle \sim\sim / \text{src} / \text{Tools} / \text{eqsubst.ML} \rangle$

3.2 Other simple lemmas

lemma [*simp*]: $\langle ((P \longrightarrow R) \longleftrightarrow (Q \longrightarrow R)) \longleftrightarrow ((P \longleftrightarrow Q) \vee R) \rangle$

by *blast*

lemma *[simp]*: $\langle (P \longrightarrow Q) \longleftrightarrow (P \longrightarrow R) \longleftrightarrow (P \longrightarrow (Q \longleftrightarrow R)) \rangle$
by *blast*

lemma *not-disj-iff-imp*: $\langle \neg P \vee Q \longleftrightarrow (P \longrightarrow Q) \rangle$
by *blast*

3.2.1 Monotonicity of implications

lemma *conj-mono*: $\langle \llbracket P1 \longrightarrow Q1; P2 \longrightarrow Q2 \rrbracket \Longrightarrow (P1 \wedge P2) \longrightarrow (Q1 \wedge Q2) \rangle$
by *fast*

lemma *disj-mono*: $\langle \llbracket P1 \longrightarrow Q1; P2 \longrightarrow Q2 \rrbracket \Longrightarrow (P1 \vee P2) \longrightarrow (Q1 \vee Q2) \rangle$
by *fast*

lemma *imp-mono*: $\langle \llbracket Q1 \longrightarrow P1; P2 \longrightarrow Q2 \rrbracket \Longrightarrow (P1 \longrightarrow P2) \longrightarrow (Q1 \longrightarrow Q2) \rangle$
by *fast*

lemma *imp-refl*: $\langle P \longrightarrow P \rangle$
by (*rule impI*)

The quantifier monotonicity rules are also intuitionistically valid.

lemma *ex-mono*: $\langle (\bigwedge x. P(x) \longrightarrow Q(x)) \Longrightarrow (\exists x. P(x)) \longrightarrow (\exists x. Q(x)) \rangle$
by *blast*

lemma *all-mono*: $\langle (\bigwedge x. P(x) \longrightarrow Q(x)) \Longrightarrow (\forall x. P(x)) \longrightarrow (\forall x. Q(x)) \rangle$
by *blast*

3.3 Proof by cases and induction

Proper handling of non-atomic rule statements.

context
begin

qualified definition $\langle \text{induct-forall}(P) \equiv \forall x. P(x) \rangle$

qualified definition $\langle \text{induct-implies}(A, B) \equiv A \longrightarrow B \rangle$

qualified definition $\langle \text{induct-equal}(x, y) \equiv x = y \rangle$

qualified definition $\langle \text{induct-conj}(A, B) \equiv A \wedge B \rangle$

lemma *induct-forall-eq*: $\langle (\bigwedge x. P(x)) \equiv \text{Trueprop}(\text{induct-forall}(\lambda x. P(x))) \rangle$
unfolding *atomize-all induct-forall-def* .

lemma *induct-implies-eq*: $\langle (A \Longrightarrow B) \equiv \text{Trueprop}(\text{induct-implies}(A, B)) \rangle$
unfolding *atomize-imp induct-implies-def* .

lemma *induct-equal-eq*: $\langle (x \equiv y) \equiv \text{Trueprop}(\text{induct-equal}(x, y)) \rangle$
unfolding *atomize-eq induct-equal-def* .

lemma *induct-conj-eq*: $\langle (A \ \&\&\ B) \equiv \text{Trueprop}(\text{induct-conj}(A, B)) \rangle$
unfolding *atomize-conj induct-conj-def* .

lemmas *induct-atomize* = *induct-forall-eq induct-implies-eq induct-equal-eq induct-conj-eq*

lemmas *induct-rulify* [*symmetric*] = *induct-atomize*

lemmas *induct-rulify-fallback* =
induct-forall-def induct-implies-def induct-equal-def induct-conj-def

Method setup.

ML-file $\langle \sim / \text{src} / \text{Tools} / \text{induct.ML} \rangle$

ML \langle
structure Induct = Induct
 \langle
val cases-default = @{thm case-split}
val atomize = @{thms induct-atomize}
val rulify = @{thms induct-rulify}
val rulify-fallback = @{thms induct-rulify-fallback}
val equal-def = @{thm induct-equal-def}
fun dest-def - = NONE
fun trivial-tac - = no-tac
 \rangle ;
 \rangle

declare *case-split* [*cases type: o*]

end

ML-file $\langle \sim / \text{src} / \text{Tools} / \text{case-product.ML} \rangle$

hide-const (**open**) *eq*

end