Isabelle/FOL — First-Order Logic

Larry Paulson and Markus Wenzel

April 15, 2020

Contents

1 Intuitionistic first-order logic

theory *IFOL*
imports *Pure*
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*›

typedecl *o*

judgment
Trueprop :: ‹$o \Rightarrow prop$› (‹(-)› *5*)

1.1.1 Equality

axiomatization

  eq :: ‹['a, 'a] ⇒ o› (**infixl** ‹=› *50*)
where
  refl: ‹$a = a$› **and**
  subst: ‹$a = b \Longrightarrow P(a) \Longrightarrow P(b)$›

1.1.2 Propositional logic

axiomatization
  False :: ‹o› **and**
  conj :: ‹[o, o] => o› (**infixr** ‹∧› *35*) **and**
  disj :: ‹[o, o] => o› (**infixr** ‹∨› *30*) **and**
  imp :: ‹[o, o] => o› (**infixr** ‹⟶› *25*)
where
  conjI: ‹$[\![P;\ Q]\!] \Longrightarrow P \land Q$› **and**
  conjunct1: ‹$P \land Q \Longrightarrow P$› **and**
  conjunct2: ‹$P \land Q \Longrightarrow Q$› **and**

  disjI1: ‹$P \Longrightarrow P \lor Q$› **and**
  disjI2: ‹$Q \Longrightarrow P \lor Q$› **and**
  disjE: ‹$[\![P \lor Q;\ P \Longrightarrow R;\ Q \Longrightarrow R]\!] \Longrightarrow R$› **and**

  impI: ‹$(P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q$› **and**
  mp: ‹$[\![P \longrightarrow Q;\ P]\!] \Longrightarrow Q$› **and**

  FalseE: ‹$\mathit{False} \Longrightarrow P$›

1.1.3 Quantifiers

axiomatization
  All :: ‹('a ⇒ o) ⇒ o› (**binder** ‹∀› *10*) **and**
  Ex :: ‹('a ⇒ o) ⇒ o› (**binder** ‹∃› *10*)
where
  allI: ‹$(\bigwedge x.\ P(x)) \Longrightarrow (\forall x.\ P(x))$› **and**
  spec: ‹$(\forall x.\ P(x)) \Longrightarrow P(x)$› **and**
  exI: ‹$P(x) \Longrightarrow (\exists x.\ P(x))$› **and**
  exE: ‹$[\![\exists x.\ P(x);\ \bigwedge x.\ P(x) \Longrightarrow R]\!] \Longrightarrow R$›

1.1.4 Definitions

definition ‹$\mathit{True} \equiv \mathit{False} \longrightarrow \mathit{False}$›

definition *Not* (‹¬ -› [*40*] *40*)
  where *not-def*: ‹$\neg P \equiv P \longrightarrow \mathit{False}$›

definition *iff* (**infixr** ‹⟷› *25*)
  where ‹$P \longleftrightarrow Q \equiv (P \longrightarrow Q) \land (Q \longrightarrow P)$›

definition *Ex1* :: ‹('a ⇒ o) ⇒ o› (**binder** ‹∃!› *10*)
  where *ex1-def*: ‹$\exists! x.\ P(x) \equiv \exists x.\ P(x) \land (\forall y.\ P(y) \longrightarrow y = x)$›

axiomatization where — Reflection, admissible
eq-reflection: ‹$(x = y) \Longrightarrow (x \equiv y)$› **and**
iff-reflection: ‹$(P \longleftrightarrow Q) \Longrightarrow (P \equiv Q)$›

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

1.1.5 Old-style ASCII syntax

notation (*ASCII*)
not-equal (**infixl** ‹~=› *50*) **and**
Not (‹~ -› [*40*] *40*) **and**
conj (**infixr** ‹&› *35*) **and**
disj (**infixr** ‹|› *30*) **and**
All (**binder** ‹*ALL* › *10*) **and**
Ex (**binder** ‹*EX* › *10*) **and**
Ex1 (**binder** ‹*EX*! › *10*) **and**
imp (**infixr** ‹-->› *25*) **and**
iff (**infixr** ‹<->› *25*)

1.2 Lemmas and proof tools

lemmas *strip* = *impI allI*

lemma *TrueI*: ‹*True*›
unfolding *True-def* **by** (*rule impI*)

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

lemma *conjE*:
assumes *major*: ‹$P \wedge Q$›
and *r*: ‹$[\![P; Q]\!] \Longrightarrow R$›
shows ‹R›
apply (*rule r*)
apply (*rule major* [*THEN conjunct1*])
apply (*rule major* [*THEN conjunct2*])
done

lemma *impE*:
assumes *major*: ‹$P \longrightarrow Q$›
and ‹P›
and *r*: ‹$Q \Longrightarrow R$›
shows ‹R›
apply (*rule r*)
apply (*rule major* [*THEN mp*])
apply (*rule* ‹P›)
done

lemma *allE*:
assumes *major*: ‹$\forall x.\ P(x)$›

and r: ‹$P(x) \Longrightarrow R$›
shows ‹R›
apply ($rule\ r$)
apply ($rule\ major\ [THEN\ spec]$)
done

Duplicates the quantifier; for use with `eresolve_tac`.

lemma *all-dupE*:
assumes *major*: ‹$\forall x.\ P(x)$›
and r: ‹$[\![P(x); \forall x.\ P(x)]\!] \Longrightarrow R$›
shows ‹R›
apply ($rule\ r$)
apply ($rule\ major\ [THEN\ spec]$)
apply ($rule\ major$)
done

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

lemma *notI*: ‹$(P \Longrightarrow False) \Longrightarrow \neg P$›
unfolding *not-def* **by** ($erule\ impI$)

lemma *notE*: ‹$[\![\neg P; P]\!] \Longrightarrow R$›
unfolding *not-def* **by** ($erule\ mp\ [THEN\ FalseE]$)

lemma *rev-notE*: ‹$[\![P; \neg P]\!] \Longrightarrow R$›
by ($erule\ notE$)

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

lemma *not-to-imp*:
assumes ‹$\neg P$›
and r: ‹$P \longrightarrow False \Longrightarrow Q$›
shows ‹Q›
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*: ‹$[\![P; P \longrightarrow Q]\!] \Longrightarrow Q$›
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*: ‹$\llbracket P \Longrightarrow Q;\ Q \Longrightarrow P\rrbracket \Longrightarrow P \longleftrightarrow Q$›
apply (*unfold iff-def*)
apply (*rule conjI*)
apply (*erule impI*)
apply (*erule impI*)
done

lemma *iffE*:
assumes *major*: ‹$P \longleftrightarrow Q$›
and *r*: ‹$P \longrightarrow Q \Longrightarrow Q \longrightarrow P \Longrightarrow R$›
shows ‹R›
apply (*insert major*, *unfold iff-def*)
apply (*erule conjE*)
apply (*erule r*)
apply *assumption*
done

1.3.1 Destruct rules for $\longleftrightarrow$ similar to Modus Ponens

lemma *iffD1*: ‹$\llbracket P \longleftrightarrow Q;\ P\rrbracket \Longrightarrow Q$›
apply (*unfold iff-def*)
apply (*erule conjunct1* [*THEN mp*])
apply *assumption*
done

lemma *iffD2*: ‹$\llbracket P \longleftrightarrow Q;\ Q\rrbracket \Longrightarrow P$›
apply (*unfold iff-def*)
apply (*erule conjunct2* [*THEN mp*])
apply *assumption*
done

lemma *rev-iffD1*: ‹$\llbracket P;\ P \longleftrightarrow Q\rrbracket \Longrightarrow Q$›
apply (*erule iffD1*)
apply *assumption*

done

lemma *rev-iffD2*: ‹$[\![Q;\ P \longleftrightarrow Q]\!] \Longrightarrow P$›
apply (*erule iffD2*)
apply *assumption*
done

lemma *iff-refl*: ‹$P \longleftrightarrow P$›
by (*rule iffI*)

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

lemma *iff-trans*: ‹$[\![P \longleftrightarrow Q;\ Q \longleftrightarrow R]\!] \Longrightarrow P \longleftrightarrow R$›
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 [$\exists !y$ 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*: ‹$P(a) \Longrightarrow (\bigwedge x.\ P(x) \Longrightarrow x = a) \Longrightarrow \exists !x.\ P(x)$›
apply (*unfold ex1-def*)
apply (*assumption* | *rule exI conjI allI impI*)+
done

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

lemma *ex-ex1I*: ‹$\exists x.\ P(x) \Longrightarrow (\bigwedge x\ y.\ [\![P(x);\ P(y)]\!] \Longrightarrow x = y) \Longrightarrow \exists !x.\ P(x)$›
apply (*erule exE*)
apply (*rule ex1I*)
apply *assumption*
apply *assumption*
done

lemma *ex1E*: ‹$\exists !\ x.\ P(x) \Longrightarrow (\bigwedge x.\ [\![P(x);\ \forall y.\ P(y) \longrightarrow y = x]\!] \Longrightarrow R) \Longrightarrow R$›
apply (*unfold 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 ‹
  fun iff-tac ctxt prems i =
    resolve-tac ctxt (prems RL @{thms iffE}) i THEN
    REPEAT1 (eresolve-tac ctxt @{thms asm-rl mp} i);
›

method-setup iff =
  ‹Attrib.thms >>
    (fn prems => fn ctxt => SIMPLE-METHOD′ (iff-tac ctxt prems))›

lemma conj-cong:
  assumes ‹P ⟷ P′›
    and ‹P′ ⟹ Q ⟷ Q′›
  shows ‹(P ∧ Q) ⟷ (P′ ∧ Q′)›
  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 ‹P ⟷ P′›
    and ‹P′ ⟹ Q ⟷ Q′›
  shows ‹(Q ∧ P) ⟷ (Q′ ∧ P′)›
  apply (insert assms)
  apply (assumption | rule iffI conjI | erule iffE conjE mp | iff assms)+
  done

lemma disj-cong:
  assumes ‹P ⟷ P′› and ‹Q ⟷ Q′›
  shows ‹(P ∨ Q) ⟷ (P′ ∨ Q′)›
  apply (insert assms)
  apply (erule iffE disjE disjI1 disjI2 |
    assumption | rule iffI | erule (1) notE impE)+
  done

lemma imp-cong:
  assumes ‹P ⟷ P′›
    and ‹P′ ⟹ Q ⟷ Q′›
  shows ‹(P ⟶ Q) ⟷ (P′ ⟶ Q′)›
  apply (insert assms)
 apply (assumption | rule iffI impI | erule iffE | erule (1) notE impE | iff assms)+
  done

lemma iff-cong: ‹⟦P ⟷ P′; Q ⟷ Q′⟧ ⟹ (P ⟷ Q) ⟷ (P′ ⟷ Q′)›
  apply (erule iffE | assumption | rule iffI | erule (1) notE impE)+
  done
```

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

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

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

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

1.5 Equality rules

lemma *sym*: ‹$a = b \Longrightarrow b = a$›
  apply (*erule subst*)
  apply (*rule refl*)
  done

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

lemma *not-sym*: ‹$b \neq a \Longrightarrow a \neq b$›
  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*: ‹$(A \equiv B) \Longrightarrow A \longleftrightarrow B$›
  apply *unfold*
  apply (*rule iff-refl*)
  done

lemma *meta-eq-to-obj-eq*: ‹$(A \equiv B) \Longrightarrow A = B$›
  apply *unfold*

apply (*rule refl*)
done

lemma *meta-eq-to-iff*: ‹$x \equiv y \Longrightarrow x \longleftrightarrow y$›
by *unfold* (*rule iff-refl*)

Substitution.

lemma *ssubst*: ‹$[\![b = a;\ P(a)]\!] \Longrightarrow P(b)$›
apply (*drule sym*)
apply (*erule* (*1*) *subst*)
done

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

lemma *ex1-equalsE*: ‹$[\![\exists!x.\ P(x);\ P(a);\ P(b)]\!] \Longrightarrow a = b$›
apply (*erule ex1E*)
apply (*rule trans*)
apply (*rule-tac* [*2*] *sym*)
apply (*assumption* | *erule spec* [*THEN mp*])+
done

1.5.1 Polymorphic congruence rules

lemma *subst-context*: ‹$a = b \Longrightarrow t(a) = t(b)$›
apply (*erule ssubst*)
apply (*rule refl*)
done

lemma *subst-context2*: ‹$[\![a = b;\ c = d]\!] \Longrightarrow t(a,c) = t(b,d)$›
apply (*erule ssubst*)+
apply (*rule refl*)
done

lemma *subst-context3*: ‹$[\![a = b;\ c = d;\ e = f]\!] \Longrightarrow t(a,c,e) = t(b,d,f)$›
apply (*erule ssubst*)+
apply (*rule refl*)
done

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

a = b — — c = d

lemma *box-equals*: ‹$[\![a = b;\ a = c;\ b = d]\!] \Longrightarrow c = d$›
apply (*rule trans*)
apply (*rule trans*)
apply (*rule sym*)
apply *assumption*+
done

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

lemma *simp-equals*: ‹$[\![a = c;\ b = d;\ c = d]\!] \Longrightarrow a = b$›

apply (*rule trans*)
 apply (*rule trans*)
  apply *assumption*+
apply (*erule sym*)
done

1.5.2 Congruence rules for predicate letters

lemma *pred1-cong*: ‹$a = a' \Longrightarrow P(a) \longleftrightarrow P(a')$›
apply (*rule iffI*)
 apply (*erule* (*1*) *subst*)
apply (*erule* (*1*) *ssubst*)
done

lemma *pred2-cong*: ‹$[\![a = a'; b = b']\!] \Longrightarrow P(a,b) \longleftrightarrow P(a',b')$›
apply (*rule iffI*)
 apply (*erule subst*)+
 apply *assumption*
apply (*erule ssubst*)+
apply *assumption*
done

lemma *pred3-cong*: ‹$[\![a = a'; b = b'; c = c']\!] \Longrightarrow P(a,b,c) \longleftrightarrow P(a',b',c')$›
apply (*rule iffI*)
 apply (*erule subst*)+
 apply *assumption*
apply (*erule ssubst*)+
apply *assumption*
done

Special case for the equality predicate!

lemma *eq-cong*: ‹$[\![a = a'; b = b']\!] \Longrightarrow a = b \longleftrightarrow a' = b'$›
apply (*erule* (*1*) *pred2-cong*)
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*: ‹$(P \wedge Q) \longrightarrow S$›
 and *r*: ‹$P \longrightarrow (Q \longrightarrow S) \Longrightarrow R$›
shows ‹R›
by (*assumption* | *rule conjI impI major* [*THEN mp*] *r*)+

lemma *disj-impE*:
  assumes *major*: ‹$(P \vee Q) \longrightarrow S$›
    and *r*: ‹$[\![P \longrightarrow S;\ Q \longrightarrow S]\!] \Longrightarrow R$›
  shows ‹R›
  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*: ‹$(P \longrightarrow Q) \longrightarrow S$›
    and *r1*: ‹$[\![P;\ Q \longrightarrow S]\!] \Longrightarrow Q$›
    and *r2*: ‹$S \Longrightarrow R$›
  shows ‹R›
  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*: ‹$\neg\ P \longrightarrow S \Longrightarrow (P \Longrightarrow \mathit{False}) \Longrightarrow (S \Longrightarrow R) \Longrightarrow R$›
  apply (*drule mp*)
   apply (*rule notI*)
   apply *assumption*
  apply *assumption*
  done

Simplifies the implication. UNSAFE.

lemma *iff-impE*:
  assumes *major*: ‹$(P \longleftrightarrow Q) \longrightarrow S$›
    and *r1*: ‹$[\![P;\ Q \longrightarrow S]\!] \Longrightarrow Q$›
    and *r2*: ‹$[\![Q;\ P \longrightarrow S]\!] \Longrightarrow P$›
    and *r3*: ‹$S \Longrightarrow R$›
  shows ‹R›
  apply (*assumption* | *rule iffI impI major* [*THEN mp*] *r1 r2 r3*)+
  done

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

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

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

lemma *ex-impE*:
  assumes *major*: ‹$(\exists x.\ P(x)) \longrightarrow S$›
    and *r*: ‹$P(x) \longrightarrow S \Longrightarrow R$›
  shows ‹R›

apply (*assumption* | *rule exI impI major* [*THEN mp*] *r*)+
done

Courtesy of Krzysztof Grabczewski.

lemma *disj-imp-disj*: ‹$P \vee Q \Longrightarrow (P \Longrightarrow R) \Longrightarrow (Q \Longrightarrow S) \Longrightarrow R \vee S$›
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*: ‹$[\![x = x; PROP\ W]\!] \Longrightarrow PROP\ W$› **.**

```
ML ‹
structure Hypsubst = Hypsubst
(
  val dest-eq = FOLogic.dest-eq
  val dest-Trueprop = FOLogic.dest-Trueprop
  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*: ‹$Q \Longrightarrow R$›

```
  and 3: ‹P ⟶ Q ⟹ P›
 shows ‹R›
proof -
 from 3 and 1 have ‹P› .
 with 1 have ‹Q› by (rule impE)
 with 2 show ‹R› .
qed

lemma allE':
 assumes 1: ‹∀x. P(x)›
  and 2: ‹P(x) ⟹ ∀x. P(x) ⟹ Q›
 shows ‹Q›
proof -
 from 1 have ‹P(x)› by (rule spec)
 from this and 1 show ‹Q› by (rule 2)
qed

lemma notE':
 assumes 1: ‹¬ P›
  and 2: ‹¬ P ⟹ P›
 shows ‹R›
proof -
 from 2 and 1 have ‹P› .
 with 1 show ‹R› 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 ‹
 Context-Rules.addSWrapper
  (fn ctxt => fn tac => hyp-subst-tac ctxt ORELSE' tac)
›


lemma iff-not-sym: ‹¬ (Q ⟷ P) ⟹ ¬ (P ⟷ Q)›
 by iprover

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


lemma eq-commute: ‹a = b ⟷ b = a›
 apply (rule iffI)
 apply (erule sym)+
 done
```

1.8 Atomizing meta-level rules

lemma *atomize-all* [*atomize*]: ‹$(\bigwedge x.\ P(x)) \equiv$ *Trueprop* $(\forall x.\ P(x))$›
proof
  assume ‹$\bigwedge x.\ P(x)$›
  then show ‹$\forall x.\ P(x)$› **..**
next
  assume ‹$\forall x.\ P(x)$›
  then show ‹$\bigwedge x.\ P(x)$› **..**
qed

lemma *atomize-imp* [*atomize*]: ‹$(A \Longrightarrow B) \equiv$ *Trueprop* $(A \longrightarrow B)$›
proof
  assume ‹$A \Longrightarrow B$›
  then show ‹$A \longrightarrow B$› **..**
next
  assume ‹$A \longrightarrow B$› **and** ‹A›
  then show ‹B› **by** (*rule mp*)
qed

lemma *atomize-eq* [*atomize*]: ‹$(x \equiv y) \equiv$ *Trueprop* $(x = y)$›
proof
  assume ‹$x \equiv y$›
  show ‹$x = y$› **unfolding** ‹$x \equiv y$› **by** (*rule refl*)
next
  assume ‹$x = y$›
  then show ‹$x \equiv y$› **by** (*rule eq-reflection*)
qed

lemma *atomize-iff* [*atomize*]: ‹$(A \equiv B) \equiv$ *Trueprop* $(A \longleftrightarrow B)$›
proof
  assume ‹$A \equiv B$›
  show ‹$A \longleftrightarrow B$› **unfolding** ‹$A \equiv B$› **by** (*rule iff-refl*)
next
  assume ‹$A \longleftrightarrow B$›
  then show ‹$A \equiv B$› **by** (*rule iff-reflection*)
qed

lemma *atomize-conj* [*atomize*]: ‹$(A\ \&\&\&\ B) \equiv$ *Trueprop* $(A \land B)$›
proof
  assume *conj*: ‹$A\ \&\&\&\ B$›
  show ‹$A \land B$›
  proof (*rule conjI*)
    from *conj* **show** ‹A› **by** (*rule conjunctionD1*)
    from *conj* **show** ‹B› **by** (*rule conjunctionD2*)
  qed
next
  assume *conj*: ‹$A \land B$›
  show ‹$A\ \&\&\&\ B$›
  proof −

    from *conj* **show** ‹A› **..**
    from *conj* **show** ‹B› **..**
  qed
qed

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

1.9 Atomizing elimination rules

lemma *atomize-exL*[*atomize-elim*]: ‹$(\bigwedge x.\ P(x) \Longrightarrow Q) \equiv ((\exists x.\ P(x)) \Longrightarrow Q)$›
  by *rule iprover+*

lemma *atomize-conjL*[*atomize-elim*]: ‹$(A \Longrightarrow B \Longrightarrow C) \equiv (A \wedge B \Longrightarrow C)$›
  by *rule iprover+*

lemma *atomize-disjL*[*atomize-elim*]: ‹$((A \Longrightarrow C) \Longrightarrow (B \Longrightarrow C) \Longrightarrow C) \equiv ((A \vee B \Longrightarrow C) \Longrightarrow C)$›
  by *rule iprover+*

lemma *atomize-elimL*[*atomize-elim*]: ‹$(\bigwedge B.\ (A \Longrightarrow B) \Longrightarrow B) \equiv Trueprop(A)$› **..**

1.10 Calculational rules

lemma *forw-subst*: ‹$a = b \Longrightarrow P(b) \Longrightarrow P(a)$›
  by (*rule ssubst*)

lemma *back-subst*: ‹$P(a) \Longrightarrow a = b \Longrightarrow P(b)$›
  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

1.11 "Let" declarations

nonterminal *letbinds* **and** *letbind*

definition *Let* :: ‹['*a*::{}, '*a* => '*b*] ⇒ ('*b*::{})›
  where ‹$Let(s, f) \equiv f(s)$›

syntax
  -bind :: ‹[*pttrn*, '*a*] => *letbind*› (‹(*2*- =/ -)› *10*)
  :: ‹*letbind* => *letbinds*› (‹-›)
  -binds :: ‹[*letbind*, *letbinds*] => *letbinds*› (‹-;/ -›)
  -Let :: ‹[*letbinds*, '*a*] => '*a*› (‹(*let* (-)/ *in* (-))› *10*)

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

lemma *LetI*:
  assumes ‹$\bigwedge x.\ x = t \Longrightarrow P(u(x))$›
  shows ‹$P(\textit{let}\ x = t\ \textit{in}\ u(x))$›
  apply (*unfold Let-def*)
  apply (*rule refl* [*THEN assms*])
  done

1.12 Intuitionistic simplification rules

lemma *conj-simps*:
  ‹$P \wedge \textit{True} \longleftrightarrow P$›
  ‹$\textit{True} \wedge P \longleftrightarrow P$›
  ‹$P \wedge \textit{False} \longleftrightarrow \textit{False}$›
  ‹$\textit{False} \wedge P \longleftrightarrow \textit{False}$›
  ‹$P \wedge P \longleftrightarrow P$›
  ‹$P \wedge P \wedge Q \longleftrightarrow P \wedge Q$›
  ‹$P \wedge \neg P \longleftrightarrow \textit{False}$›
  ‹$\neg P \wedge P \longleftrightarrow \textit{False}$›
  ‹$(P \wedge Q) \wedge R \longleftrightarrow P \wedge (Q \wedge R)$›
  by *iprover+*

lemma *disj-simps*:
  ‹$P \vee \textit{True} \longleftrightarrow \textit{True}$›
  ‹$\textit{True} \vee P \longleftrightarrow \textit{True}$›
  ‹$P \vee \textit{False} \longleftrightarrow P$›
  ‹$\textit{False} \vee P \longleftrightarrow P$›
  ‹$P \vee P \longleftrightarrow P$›
  ‹$P \vee P \vee Q \longleftrightarrow P \vee Q$›
  ‹$(P \vee Q) \vee R \longleftrightarrow P \vee (Q \vee R)$›
  by *iprover+*

lemma *not-simps*:
  ‹$\neg (P \vee Q) \longleftrightarrow \neg P \wedge \neg Q$›
  ‹$\neg \textit{False} \longleftrightarrow \textit{True}$›
  ‹$\neg \textit{True} \longleftrightarrow \textit{False}$›
  by *iprover+*

lemma *imp-simps*:
  ‹$(P \longrightarrow \textit{False}) \longleftrightarrow \neg P$›
  ‹$(P \longrightarrow \textit{True}) \longleftrightarrow \textit{True}$›
  ‹$(\textit{False} \longrightarrow P) \longleftrightarrow \textit{True}$›
  ‹$(\textit{True} \longrightarrow P) \longleftrightarrow P$›
  ‹$(P \longrightarrow P) \longleftrightarrow \textit{True}$›
  ‹$(P \longrightarrow \neg P) \longleftrightarrow \neg P$›

by *iprover+*

lemma *iff-simps*:
$\langle(True \longleftrightarrow P) \longleftrightarrow P\rangle$
$\langle(P \longleftrightarrow True) \longleftrightarrow P\rangle$
$\langle(P \longleftrightarrow P) \longleftrightarrow True\rangle$
$\langle(False \longleftrightarrow P) \longleftrightarrow \neg P\rangle$
$\langle(P \longleftrightarrow False) \longleftrightarrow \neg P\rangle$
by *iprover+*

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

lemma *quant-simps*:
$\langle\bigwedge P.\ (\forall x.\ P) \longleftrightarrow P\rangle$
$\langle(\forall x.\ x = t \longrightarrow P(x)) \longleftrightarrow P(t)\rangle$
$\langle(\forall x.\ t = x \longrightarrow P(x)) \longleftrightarrow P(t)\rangle$
$\langle\bigwedge P.\ (\exists x.\ P) \longleftrightarrow P\rangle$
$\langle\exists x.\ x = t\rangle$
$\langle\exists x.\ t = x\rangle$
$\langle(\exists x.\ x = t \wedge P(x)) \longleftrightarrow P(t)\rangle$
$\langle(\exists x.\ t = x \wedge P(x)) \longleftrightarrow P(t)\rangle$
by *iprover+*

These are NOT supplied by default!

lemma *distrib-simps*:
$\langle P \wedge (Q \vee R) \longleftrightarrow P \wedge Q \vee P \wedge R\rangle$
$\langle(Q \vee R) \wedge P \longleftrightarrow Q \wedge P \vee R \wedge P\rangle$
$\langle(P \vee Q \longrightarrow R) \longleftrightarrow (P \longrightarrow R) \wedge (Q \longrightarrow R)\rangle$
by *iprover+*

1.12.1 Conversion into rewrite rules

lemma *P-iff-F*: $\langle\neg P \Longrightarrow (P \longleftrightarrow False)\rangle$
by *iprover*
lemma *iff-reflection-F*: $\langle\neg P \Longrightarrow (P \equiv False)\rangle$
by (*rule P-iff-F* [*THEN iff-reflection*])

lemma *P-iff-T*: $\langle P \Longrightarrow (P \longleftrightarrow True)\rangle$
by *iprover*
lemma *iff-reflection-T*: $\langle P \Longrightarrow (P \equiv True)\rangle$
by (*rule P-iff-T* [*THEN iff-reflection*])

1.12.2 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*: ‹$P \wedge (Q \vee R) \longleftrightarrow (P \wedge Q \vee P \wedge R)$› **by** *iprover*
lemma *conj-disj-distribR*: ‹$(P \vee Q) \wedge R \longleftrightarrow (P \wedge R \vee Q \wedge R)$› **by** *iprover*

lemma *disj-conj-distribL*: ‹$P \vee (Q \wedge R) \longleftrightarrow (P \vee Q) \wedge (P \vee R)$› **by** *iprover*
lemma *disj-conj-distribR*: ‹$(P \wedge Q) \vee R \longleftrightarrow (P \vee R) \wedge (Q \vee R)$› **by** *iprover*

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

lemma *de-Morgan-disj*: ‹$(\neg (P \vee Q)) \longleftrightarrow (\neg P \wedge \neg Q)$› **by** *iprover*

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

lemma *ex-disj-distrib*: ‹$(\exists x.\ P(x) \vee Q(x)) \longleftrightarrow ((\exists x.\ P(x)) \vee (\exists x.\ Q(x)))$›
  by *iprover*

lemma *all-conj-distrib*: ‹$(\forall x.\ P(x) \wedge Q(x)) \longleftrightarrow ((\forall x.\ P(x)) \wedge (\forall x.\ Q(x)))$›
  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 ‹~~/*src*/*Provers*/*blast.ML*›
ML-file ‹~~/*src*/*Provers*/*clasimp.ML*›

2.1 The classical axiom

axiomatization where
  classical: ‹$(\neg P \Longrightarrow P) \Longrightarrow P$›

2.2 Lemmas and proof tools

lemma *ccontr*: ‹$(\neg P \Longrightarrow \mathit{False}) \Longrightarrow P$›
  by (*erule FalseE* [*THEN classical*])

2.2.1 Classical introduction rules for ∨ and ∃

lemma *disjCI*: ‹$(\neg\ Q \Longrightarrow P) \Longrightarrow P \lor Q$›
  apply (*rule classical*)
  apply (*assumption* | *erule meta-mp* | *rule disjI1 notI*)+
  apply (*erule notE disjI2*)+
  done

Introduction rule involving only ∃

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

Version of above, simplifying ¬∃ to ∀¬.

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

lemma *excluded-middle*: ‹$\neg\ P \lor P$›
  apply (*rule disjCI*)
  apply *assumption*
  done

lemma *case-split* [*case-names True False*]:
  assumes *r1*: ‹$P \Longrightarrow Q$›
    and *r2*: ‹$\neg\ P \Longrightarrow Q$›
  shows ‹Q›
  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* (*Args.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*: ⟨$\neg\ P \Longrightarrow R$⟩
    and *r2*: ⟨$Q \Longrightarrow R$⟩
  shows ⟨R⟩
  apply (*rule excluded-middle* [*THEN disjE*])
   apply (*erule r1*)
  apply (*rule 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*: ⟨$\neg\ P \Longrightarrow R$⟩
  shows ⟨R⟩
  apply (*rule excluded-middle* [*THEN disjE*])
   apply (*erule r2*)
  apply (*rule r1*)
  apply (*erule major* [*THEN mp*])
  done

Double negation law.

lemma *notnotD*: ⟨$\neg\ \neg\ P \Longrightarrow P$⟩
  apply (*rule classical*)
  apply (*erule notE*)
  apply *assumption*
  done

lemma *contrapos2*: ⟨$[\![Q; \neg\ P \Longrightarrow \neg\ Q]\!] \Longrightarrow P$⟩
  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 $\neg\ P \Longrightarrow \neg\ Q$ and $P \Longrightarrow Q$.

```
lemma iffCE:
  assumes major: ‹P ⟷ Q›
    and r1: ‹⟦P; Q⟧ ⟹ R›
    and r2: ‹⟦¬ P; ¬ Q⟧ ⟹ R›
  shows ‹R›
  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: ‹∃! x. P(x)›
    and r: ‹⋀x. ⟦P(x);  ∀ y y'. P(y) ∧ P(y') ⟶ y = y'⟧ ⟹ R›
  shows ‹R›
  using major
proof (rule ex1E)
  fix x
  assume * : ‹∀ y. P(y) ⟶ y = x›
  assume ‹P(x)›
  then show ‹R›
  proof (rule r)
    {
      fix y y'
      assume ‹P(y)› and ‹P(y')›
      with * have ‹x = y› and ‹x = y'›
        by − (tactic IntPr.fast-tac context 1)+
      then have ‹y = y'› by (rule subst)
    } note r' = this
    show ‹∀ y y'. P(y) ∧ P(y') ⟶ y = y'›
      by (intro strip, elim conjE) (rule r')
  qed
qed

lemma imp-elim: ‹P ⟶ Q ⟹ (¬ R ⟹ P) ⟹ (Q ⟹ R) ⟹ R›
  by (rule classical) iprover

lemma swap: ‹¬ P ⟹ (¬ R ⟹ P) ⟹ R›
  by (rule classical) iprover
```

3 Classical Reasoner

```
ML ‹
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;
›
```

lemmas [*intro*!] = *refl TrueI conjI disjCI impI notI iffI*
  and [*elim*!] = *conjE disjE impCE FalseE iffCE*
ML ‹*val prop-cs = claset-of* ***context***›

lemmas [*intro*!] = *allI ex-ex1I*
  and [*intro*] = *exI*
  and [*elim*!] = *exE alt-ex1E*
  and [*elim*] = *allE*
ML ‹*val FOL-cs = claset-of* ***context***›

ML ‹
```
  structure Blast = Blast
  (
    structure Classical = Cla
    val Trueprop-const = dest-Const const‹Trueprop›
    val equality-name = const-name‹eq›
    val not-name = const-name‹Not›
    val notE = @{thm notE}
    val ccontr = @{thm ccontr}
    val hyp-subst-tac = Hypsubst.blast-hyp-subst-tac
  );
  val blast-tac = Blast.blast-tac;
›
```

lemma *ex1-functional*: ‹$[\![\exists!\ z.\ P(a,z);\ P(a,b);\ P(a,c)]\!] \Longrightarrow b = c$›
  by *blast*

Elimination of *True* from assumptions:

lemma *True-implies-equals*: ‹$(True \Longrightarrow PROP\ P) \equiv PROP\ P$›
proof
  assume ‹$True \Longrightarrow PROP\ P$›
  from *this* **and** *TrueI* **show** ‹$PROP\ P$› **.**
next
  assume ‹$PROP\ P$›
  then show ‹$PROP\ P$› **.**

qed

lemma *uncurry*: ‹$P \longrightarrow Q \longrightarrow R \Longrightarrow P \wedge Q \longrightarrow R$›
 by *blast*

lemma *iff-allI*: ‹$(\bigwedge x.\ P(x) \longleftrightarrow Q(x)) \Longrightarrow (\forall x.\ P(x)) \longleftrightarrow (\forall x.\ Q(x))$›
 by *blast*

lemma *iff-exI*: ‹$(\bigwedge x.\ P(x) \longleftrightarrow Q(x)) \Longrightarrow (\exists x.\ P(x)) \longleftrightarrow (\exists x.\ Q(x))$›
 by *blast*

lemma *all-comm*: ‹$(\forall x\ y.\ P(x,y)) \longleftrightarrow (\forall y\ x.\ P(x,y))$›
 by *blast*

lemma *ex-comm*: ‹$(\exists x\ y.\ P(x,y)) \longleftrightarrow (\exists y\ x.\ P(x,y))$›
 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*: ‹$(P \longrightarrow Q) \wedge (\neg P \longrightarrow Q) \longleftrightarrow Q$›
 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*:
 ‹$\bigwedge P\ Q.\ (\exists x.\ P(x) \wedge Q) \longleftrightarrow (\exists x.\ P(x)) \wedge Q$›
 ‹$\bigwedge P\ Q.\ (\exists x.\ P \wedge Q(x)) \longleftrightarrow P \wedge (\exists x.\ Q(x))$›
 ‹$\bigwedge P\ Q.\ (\exists x.\ P(x) \vee Q) \longleftrightarrow (\exists x.\ P(x)) \vee Q$›
 ‹$\bigwedge P\ Q.\ (\exists x.\ P \vee Q(x)) \longleftrightarrow P \vee (\exists x.\ Q(x))$›
 by *iprover+*

Classical rules.

lemma *cla-ex-simps*:
 ‹$\bigwedge P\ Q.\ (\exists x.\ P(x) \longrightarrow Q) \longleftrightarrow (\forall x.\ P(x)) \longrightarrow Q$›
 ‹$\bigwedge P\ Q.\ (\exists x.\ P \longrightarrow Q(x)) \longleftrightarrow P \longrightarrow (\exists x.\ Q(x))$›
 by *blast+*

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

Universal miniscoping.

lemma *int-all-simps*:
‹$\bigwedge P\ Q.\ (\forall x.\ P(x) \wedge Q) \longleftrightarrow (\forall x.\ P(x)) \wedge Q$›
‹$\bigwedge P\ Q.\ (\forall x.\ P \wedge Q(x)) \longleftrightarrow P \wedge (\forall x.\ Q(x))$›
‹$\bigwedge P\ Q.\ (\forall x.\ P(x) \longrightarrow Q) \longleftrightarrow (\exists\ x.\ P(x)) \longrightarrow Q$›
‹$\bigwedge P\ Q.\ (\forall x.\ P \longrightarrow Q(x)) \longleftrightarrow P \longrightarrow (\forall x.\ Q(x))$›
by *iprover+*

Classical rules.

lemma *cla-all-simps*:
‹$\bigwedge P\ Q.\ (\forall x.\ P(x) \vee Q) \longleftrightarrow (\forall x.\ P(x)) \vee Q$›
‹$\bigwedge P\ Q.\ (\forall x.\ P \vee Q(x)) \longleftrightarrow P \vee (\forall x.\ Q(x))$›
by *blast+*

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

3.1.2 Named rewrite rules proved for IFOL

lemma *imp-disj1*: ‹$(P \longrightarrow Q) \vee R \longleftrightarrow (P \longrightarrow Q \vee R)$› **by** *blast*
lemma *imp-disj2*: ‹$Q \vee (P \longrightarrow R) \longleftrightarrow (P \longrightarrow Q \vee R)$› **by** *blast*

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

lemma *not-imp*: ‹$\neg\ (P \longrightarrow Q) \longleftrightarrow (P \wedge \neg\ Q)$› **by** *blast*
lemma *not-iff*: ‹$\neg\ (P \longleftrightarrow Q) \longleftrightarrow (P \longleftrightarrow \neg\ Q)$› **by** *blast*

lemma *not-all*: ‹$(\neg\ (\forall x.\ P(x))) \longleftrightarrow (\exists x.\ \neg\ P(x))$› **by** *blast*
lemma *imp-all*: ‹$((\forall x.\ P(x)) \longrightarrow Q) \longleftrightarrow (\exists x.\ P(x) \longrightarrow Q)$› **by** *blast*

lemmas *meta-simps* =
triv-forall-equality — prunes params
True-implies-equals — prune asms *True*

lemmas *IFOL-simps* =
refl [*THEN P-iff-T*] *conj-simps disj-simps not-simps*
imp-simps iff-simps quant-simps

lemma *notFalseI*: ‹$\neg$ *False*› **by** *iprover*

lemma *cla-simps-misc*:
‹$\neg\ (P \wedge Q) \longleftrightarrow \neg\ P \vee \neg\ Q$›
‹$P \vee \neg\ P$›
‹$\neg\ P \vee P$›
‹$\neg\ \neg\ P \longleftrightarrow P$›
‹$(\neg\ P \longrightarrow P) \longleftrightarrow P$›
‹$(\neg\ P \longleftrightarrow \neg\ Q) \longleftrightarrow (P \longleftrightarrow Q)$› **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 ‹*simpdata.ML*›

simproc-setup *defined-Ex* (‹$\exists x.\ P(x)$›) = ‹*fn - => Quantifier1.rearrange-ex*›
simproc-setup *defined-All* (‹$\forall x.\ P(x)$›) = ‹*fn - => Quantifier1.rearrange-all*›

ML ‹
```
(*intuitionistic simprules only*)
val IFOL-ss =
  put-simpset FOL-basic-ss context
  addsimps @{thms meta-simps IFOL-simps int-ex-simps int-all-simps}
  addsimprocs [simproc ‹defined-All›, simproc ‹defined-Ex›]
  |> Simplifier.add-cong @{thm imp-cong}
  |> simpset-of;

(*classical simprules too*)
val FOL-ss =
  put-simpset IFOL-ss context
  addsimps @{thms cla-simps cla-ex-simps cla-all-simps}
  |> simpset-of;
```
›

setup ‹
```
  map-theory-simpset (put-simpset FOL-ss) #>
  Simplifier.method-setup Splitter.split-modifiers
```
›

ML-file ‹*~~/src/Tools/eqsubst.ML*›

3.2 Other simple lemmas

lemma [*simp*]: ‹$((P \longrightarrow R) \longleftrightarrow (Q \longrightarrow R)) \longleftrightarrow ((P \longleftrightarrow Q) \vee R)$›
  by *blast*

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

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

3.2.1 Monotonicity of implications

lemma *conj-mono*: ‹$[\![P1 \longrightarrow Q1;\ P2 \longrightarrow Q2]\!] \Longrightarrow (P1 \wedge P2) \longrightarrow (Q1 \wedge Q2)$›
  by *fast*

lemma *disj-mono*: ‹$[\![P1 \longrightarrow Q1;\ P2 \longrightarrow Q2]\!] \Longrightarrow (P1 \vee P2) \longrightarrow (Q1 \vee Q2)$›
  by *fast*

lemma *imp-mono*: ‹$[\![Q1 \longrightarrow P1;\ P2 \longrightarrow Q2]\!] \Longrightarrow (P1 \longrightarrow P2) \longrightarrow (Q1 \longrightarrow Q2)$›
  by *fast*

lemma *imp-refl*: ‹$P \longrightarrow P$›
  by (*rule impI*)

The quantifier monotonicity rules are also intuitionistically valid.

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

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

3.3 Proof by cases and induction

Proper handling of non-atomic rule statements.

context
begin

qualified definition ‹$\mathit{induct\text{-}forall}(P) \equiv \forall x.\ P(x)$›
qualified definition ‹$\mathit{induct\text{-}implies}(A,\ B) \equiv A \longrightarrow B$›
qualified definition ‹$\mathit{induct\text{-}equal}(x,\ y) \equiv x = y$›
qualified definition ‹$\mathit{induct\text{-}conj}(A,\ B) \equiv A \wedge B$›

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

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

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

lemma *induct-conj-eq*: ‹$(A\ \&\&\&\ B) \equiv \mathit{Trueprop}(\mathit{induct\text{-}conj}(A,\ B))$›
  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 ‹*~~/src/Tools/induct.ML*›
ML ‹
  structure Induct = Induct
  (
    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
  );
›
```

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

end

ML-file ‹~~/*src*/*Tools*/*case-product.ML*›

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

end