Isabelle/FOL — First-Order Logic

Larry Paulson and Markus Wenzel

June 8, 2008

Contents

1 Intuitionistic first-order logic

theory *IFOL*
imports *Pure*
uses
~~/*src*/*Provers*/*splitter.ML*
~~/*src*/*Provers*/*hypsubst.ML*
~~/*src*/*Tools*/*IsaPlanner*/*zipper.ML*
~~/*src*/*Tools*/*IsaPlanner*/*isand.ML*
~~/*src*/*Tools*/*IsaPlanner*/*rw-tools.ML*
~~/*src*/*Tools*/*IsaPlanner*/*rw-inst.ML*
~~/*src*/*Provers*/*eqsubst.ML*
~~/*src*/*Provers*/*quantifier1.ML*
~~/*src*/*Provers*/*project-rule.ML*
~~/*src*/*Tools*/*atomize-elim.ML*

(*fologic.ML*)
(*hypsubstdata.ML*)
(*intprover.ML*)
begin

1.1 Syntax and axiomatic basis

setup *PureThy.old-appl-syntax-setup*

global

classes *term*
defaultsort *term*

typedecl *o*

judgment
Trueprop :: *o* => *prop* ((-) *5*)

consts
True :: *o*
False :: *o*

op = :: ['*a*, '*a*] => *o* (**infixl** = *50*)

Not :: *o* => *o* (~ - [*40*] *40*)
op & :: [*o*, *o*] => *o* (**infixr** & *35*)
op | :: [*o*, *o*] => *o* (**infixr** | *30*)
op --> :: [*o*, *o*] => *o* (**infixr** --> *25*)
op <-> :: [*o*, *o*] => *o* (**infixr** <-> *25*)

All :: ('*a* => *o*) => *o* (**binder** *ALL* *10*)
Ex :: ('*a* => *o*) => *o* (**binder** *EX* *10*)
Ex1 :: ('*a* => *o*) => *o* (**binder** *EX*! *10*)

abbreviation
not-equal :: ['*a*, '*a*] => *o* (**infixl** ~= *50*) **where**
x ~= *y* == ~ (*x* = *y*)

notation (*xsymbols*)
not-equal (**infixl** ≠ *50*)

notation (*HTML* **output**)
not-equal (**infixl** ≠ *50*)

notation (*xsymbols*)
Not (¬ - [*40*] *40*) **and**
op & (**infixr** ∧ *35*) **and**
op | (**infixr** ∨ *30*) **and**
All (**binder** ∀ *10*) **and**
Ex (**binder** ∃ *10*) **and**
Ex1 (**binder** ∃! *10*) **and**
op −−> (**infixr** ⟶ *25*) **and**
op <−> (**infixr** ⟷ *25*)

notation (*HTML* **output**)
Not (¬ - [*40*] *40*) **and**
op & (**infixr** ∧ *35*) **and**
op | (**infixr** ∨ *30*) **and**
All (**binder** ∀ *10*) **and**
Ex (**binder** ∃ *10*) **and**
Ex1 (**binder** ∃! *10*)

local

finalconsts
False All Ex
op =
op &
op |
op −−>

axioms

refl: $a=a$

conjI: $[\!|\ P;\ \ Q\ |\!] ==> P\&Q$
conjunct1: $P\&Q ==> P$
conjunct2: $P\&Q ==> Q$

disjI1: $P ==> P|Q$
disjI2: $Q ==> P|Q$
disjE: $[\!|\ P|Q;\ \ P ==> R;\ \ Q ==> R\ |\!] ==> R$

impI: $(P ==> Q) ==> P-->Q$
mp: $[\!|\ P-->Q;\ \ P\ |\!] ==> Q$

FalseE: $False ==> P$

allI: $(!!x.\ P(x)) ==> (ALL\ x.\ P(x))$
spec: $(ALL\ x.\ P(x)) ==> P(x)$

exI: $P(x) ==> (EX\ x.\ P(x))$
exE: $[|\ EX\ x.\ P(x);\ !!x.\ P(x) ==> R\ |] ==> R$

eq-reflection: $(x=y) ==> (x==y)$
iff-reflection: $(P<->Q) ==> (P==Q)$

lemmas *strip* = *impI allI*

Thanks to Stephan Merz

theorem *subst*:
  assumes *eq*: $a = b$ **and** *p*: $P(a)$
  shows $P(b)$
proof −
  from *eq* **have** *meta*: $a \equiv b$
    by (*rule eq-reflection*)
  from *p* **show** *?thesis*
    by (*unfold meta*)
qed

defs

True-def: $True == False-->False$
not-def: $~P == P-->False$
iff-def: $P<->Q == (P-->Q)\ \&\ (Q-->P)$

ex1-def: $Ex1(P) == EX\ x.\ P(x)\ \&\ (ALL\ y.\ P(y) --> y=x)$

1.2 Lemmas and proof tools

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

lemma *conjE*:
  assumes *major*: $P\ \&\ Q$

and r: $[|\ P;\ Q\ |] ==> R$
shows R
apply (*rule r*)
apply (*rule major* [*THEN conjunct1*])
apply (*rule major* [*THEN conjunct2*])
done

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

lemma *allE*:
assumes *major*: $ALL\ x.\ P(x)$
and r: $P(x) ==> R$
shows R
apply (*rule r*)
apply (*rule major* [*THEN spec*])
done

lemma *all-dupE*:
assumes *major*: $ALL\ x.\ P(x)$
and r: $[|\ P(x);\ ALL\ x.\ P(x)\ |] ==> R$
shows R
apply (*rule r*)
apply (*rule major* [*THEN spec*])
apply (*rule major*)
done

lemma *notI*: $(P ==> False) ==> \sim P$
unfolding *not-def* **by** (*erule impI*)

lemma *notE*: $[|\ \sim P;\ \ P\ |] ==> R$
unfolding *not-def* **by** (*erule mp* [*THEN FalseE*])

lemma *rev-notE*: $[|\ P;\ \sim P\ |] ==> R$
by (*erule notE*)

lemma *not-to-imp*:

assumes $\sim P$
and r: P $-->$ *False* $==>$ Q
shows Q
apply (*rule* r)
apply (*rule impI*)
apply (*erule notE* [*OF* ‹$\sim P$›])
done

lemma *rev-mp*: [| P; P $-->$ Q |] $==>$ Q
by (*erule mp*)

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

ML ⟪
fun mp-tac i = eresolve-tac [@{*thm notE*}, @{*thm impE*}] *i* *THEN* *assume-tac*
i
fun eq-mp-tac i = eresolve-tac [@{*thm notE*}, @{*thm impE*}] *i* *THEN* *eq-assume-tac*
i
⟫

lemma *iffI*: [| P $==>$ Q; Q $==>$ P |] $==>$ $P<->Q$
apply (*unfold iff-def*)
apply (*rule conjI*)
apply (*erule impI*)
apply (*erule impI*)
done

lemma *iffE*:
assumes *major*: P $<->$ Q
and r: $P-->Q$ $==>$ $Q-->P$ $==>$ R
shows R
apply (*insert major*, *unfold iff-def*)

apply (*erule conjE*)
apply (*erule r*)
apply *assumption*
done

lemma *iffD1*: [| *P* <-> *Q*; *P* |] ==> *Q*
apply (*unfold iff-def*)
apply (*erule conjunct1* [*THEN mp*])
apply *assumption*
done

lemma *iffD2*: [| *P* <-> *Q*; *Q* |] ==> *P*
apply (*unfold iff-def*)
apply (*erule conjunct2* [*THEN mp*])
apply *assumption*
done

lemma *rev-iffD1*: [| *P*; *P* <-> *Q* |] ==> *Q*
apply (*erule iffD1*)
apply *assumption*
done

lemma *rev-iffD2*: [| *Q*; *P* <-> *Q* |] ==> *P*
apply (*erule iffD2*)
apply *assumption*
done

lemma *iff-refl*: *P* <-> *P*
by (*rule iffI*)

lemma *iff-sym*: *Q* <-> *P* ==> *P* <-> *Q*
apply (*erule iffE*)
apply (*rule iffI*)
apply (*assumption* | *erule mp*)+
done

lemma *iff-trans*: [| *P* <-> *Q*; *Q*<-> *R* |] ==> *P* <-> *R*
apply (*rule iffI*)
apply (*assumption* | *erule iffE* | *erule* (*1*) *notE impE*)+
done

lemma *ex1I*:
P(*a*) ⟹ (!!*x*. *P*(*x*) ==> *x*=*a*) ⟹ *EX*! *x*. *P*(*x*)
apply (*unfold ex1-def*)

apply (*assumption* | *rule exI conjI allI impI*)+
done

lemma *ex-ex1I*:
$EX\ x.\ P(x) \Longrightarrow (!!x\ y.\ [|\ P(x);\ P(y)\ |] ==> x{=}y) \Longrightarrow EX!\ x.\ P(x)$
apply (*erule exE*)
apply (*rule ex1I*)
apply *assumption*
apply *assumption*
done

lemma *ex1E*:
$EX!\ x.\ P(x) \Longrightarrow (!!x.\ [|\ P(x);\ \ ALL\ y.\ P(y)\ --> y{=}x\ |] ==> R) \Longrightarrow R$
apply (*unfold ex1-def*)
apply (*assumption* | *erule exE conjE*)+
done

ML ⟪
fun iff-tac prems i =
resolve-tac (*prems RL* @{*thms iffE*}) *i THEN*
REPEAT1 (*eresolve-tac* [@{*thm asm-rl*}, @{*thm mp*}] *i*)
⟫

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* |
tactic ⟪ *iff-tac* (*thms assms*) *1* ⟫)+
done

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* |
tactic ⟪ *iff-tac* (*thms assms*) *1* ⟫)+
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 |*
tactic ⟪ iff-tac (thms assms) 1 ⟫)+
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*: $P <-> P' ==> \sim P <-> \sim P'$
apply (*assumption | rule iffI notI | erule (1) notE impE | erule iffE notE*)+
done

lemma *all-cong*:
assumes $!!x.\ P(x) <-> Q(x)$
shows $(ALL\ x.\ P(x)) <-> (ALL\ x.\ Q(x))$
apply (*assumption | rule iffI allI | erule (1) notE impE | erule allE |*
tactic ⟪ iff-tac (thms assms) 1 ⟫)+
done

lemma *ex-cong*:
assumes $!!x.\ P(x) <-> Q(x)$
shows $(EX\ x.\ P(x)) <-> (EX\ x.\ Q(x))$
apply (*erule exE | assumption | rule iffI exI | erule (1) notE impE |*
tactic ⟪ iff-tac (thms assms) 1 ⟫)+
done

lemma *ex1-cong*:
assumes $!!x.\ P(x) <-> Q(x)$
shows $(EX!\ x.\ P(x)) <-> (EX!\ x.\ Q(x))$
apply (*erule ex1E spec [THEN mp] | assumption | rule iffI ex1I | erule (1) notE impE |*
tactic ⟪ iff-tac (thms assms) 1 ⟫)+
done

lemma *sym*: $a=b ==> b=a$
apply (*erule subst*)

apply (*rule refl*)
done

lemma *trans*: [| *a*=*b*; *b*=*c* |] ==> *a*=*c*
apply (*erule subst*, *assumption*)
done

lemma *not-sym*: *b* ~= *a* ==> *a* ~= *b*
apply (*erule contrapos*)
apply (*erule sym*)
done

lemma *def-imp-iff*: (*A* == *B*) ==> *A* <-> *B*
apply *unfold*
apply (*rule iff-refl*)
done

lemma *meta-eq-to-obj-eq*: (*A* == *B*) ==> *A* = *B*
apply *unfold*
apply (*rule refl*)
done

lemma *meta-eq-to-iff*: *x*==*y* ==> *x*<->*y*
by *unfold* (*rule iff-refl*)

lemma *ssubst*: [| *b* = *a*; *P*(*a*) |] ==> *P*(*b*)
apply (*drule sym*)
apply (*erule* (*1*) *subst*)
done

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

lemma *subst-context*: [| *a*=*b* |] ==> *t*(*a*)=*t*(*b*)
apply (*erule ssubst*)
apply (*rule refl*)
done

lemma *subst-context2*: [| a=b; c=d |] ==> $t(a,c)$=$t(b,d)$
apply (*erule ssubst*)+
apply (*rule refl*)
done

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

lemma *box-equals*: [| a=b; a=c; b=d |] ==> c=d
apply (*rule trans*)
apply (*rule trans*)
apply (*rule sym*)
apply *assumption*+
done

lemma *simp-equals*: [| a=c; b=d; c=d |] ==> a=b
apply (*rule trans*)
apply (*rule trans*)
apply *assumption*+
apply (*erule sym*)
done

lemma *pred1-cong*: a=a' ==> $P(a)$ <-> $P(a')$
apply (*rule iffI*)
apply (*erule* (*1*) *subst*)
apply (*erule* (*1*) *ssubst*)
done

lemma *pred2-cong*: [| a=a'; b=b' |] ==> $P(a,b)$ <-> $P(a',b')$
apply (*rule iffI*)
apply (*erule subst*)+
apply *assumption*
apply (*erule ssubst*)+
apply *assumption*
done

lemma *pred3-cong*: [| a=a'; b=b'; c=c' |] ==> $P(a,b,c)$ <-> $P(a',b',c')$
apply (*rule iffI*)
apply (*erule subst*)+
apply *assumption*
apply (*erule ssubst*)+
apply *assumption*

done

ML ⟪
bind-thms (*pred-congs*,
  List.concat (*map* (*fn c =>*
            map (*fn th => read-instantiate* [(*P*,*c*)] *th*)
              [@{*thm pred1-cong*}, @{*thm pred2-cong*}, @{*thm pred3-cong*}])
            (*explodePQRS*)))
⟫

lemma *eq-cong*: [| *a* = *a'*; *b* = *b'* |] ==> *a* = *b* <-> *a'* = *b'*
  apply (*erule* (*1*) *pred2-cong*)
  done

lemma *conj-impE*:
  assumes *major*: (*P*&*Q*)-->*S*
    and *r*: *P*-->(*Q*-->*S*) ==> *R*
  shows *R*
  by (*assumption* | *rule conjI impI major* [*THEN mp*] *r*)+

lemma *disj-impE*:
  assumes *major*: (*P*|*Q*)-->*S*
    and *r*: [| *P*-->*S*; *Q*-->*S* |] ==> *R*
  shows *R*
  by (*assumption* | *rule disjI1 disjI2 impI major* [*THEN mp*] *r*)+

lemma *imp-impE*:
  assumes *major*: (*P*-->*Q*)-->*S*
    and *r1*: [| *P*; *Q*-->*S* |] ==> *Q*
    and *r2*: *S* ==> *R*
  shows *R*
  by (*assumption* | *rule impI major* [*THEN mp*] *r1 r2*)+

lemma *not-impE*:
  ~*P* --> *S* ⟹ (*P* ==> *False*) ⟹ (*S* ==> *R*) ⟹ *R*
  apply (*drule mp*)
   apply (*rule notI*)
   apply *assumption*
  apply *assumption*
  done

lemma *iff-impE*:
assumes *major*: $(P<->Q)-->S$
and *r1*: $[|\ P;\ Q-->S\ |] ==>\ Q$
and *r2*: $[|\ Q;\ P-->S\ |] ==>\ P$
and *r3*: $S ==>\ R$
shows R
apply (*assumption* | *rule iffI impI major* [*THEN mp*] *r1 r2 r3*)+
done

lemma *all-impE*:
assumes *major*: $(ALL\ x.\ P(x))-->S$
and *r1*: $!!x.\ P(x)$
and *r2*: $S ==>\ R$
shows R
apply (*rule allI impI major* [*THEN mp*] *r1 r2*)+
done

lemma *ex-impE*:
assumes *major*: $(EX\ x.\ P(x))-->S$
and *r*: $P(x)-->S ==>\ R$
shows R
apply (*assumption* | *rule exI impI major* [*THEN mp*] *r*)+
done

lemma *disj-imp-disj*:
$P|Q \Longrightarrow (P==>R) \Longrightarrow (Q==>S) \Longrightarrow R|S$
apply (*erule disjE*)
apply (*rule disjI1*) **apply** *assumption*
apply (*rule disjI2*) **apply** *assumption*
done

ML ⟪
structure ProjectRule = ProjectRuleFun
(*struct*
val conjunct1 = @{*thm conjunct1*}
val conjunct2 = @{*thm conjunct2*}
val mp = @{*thm mp*}
end)
⟫

use *fologic.ML*

lemma *thin-refl*: $!!X.\ [|x=x;\ PROP\ W|] ==>\ PROP\ W$ **.**

use *hypsubstdata.ML*
setup *hypsubst-setup*
use *intprover.ML*

1.3 Intuitionistic Reasoning

lemma *impE'*:
  assumes *1*: P −−> Q
    and *2*: Q ==> 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*: *ALL x*. $P(x)$
    and *2*: $P(x)$ ==> *ALL 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 ⟪ *ContextRules.addSWrapper* (*fn tac => hyp-subst-tac 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

1.4 Atomizing meta-level rules

lemma *atomize-all* [*atomize*]: $(!!x.\ P(x)) == \mathit{Trueprop}\ (\mathit{ALL}\ x.\ P(x))$
proof
  assume $!!x.\ P(x)$
  then show $\mathit{ALL}\ x.\ P(x)$ **..**
next
  assume $\mathit{ALL}\ x.\ P(x)$
  then show $!!x.\ P(x)$ **..**
qed

lemma *atomize-imp* [*atomize*]: $(A ==> B) == \mathit{Trueprop}\ (A --> B)$
proof
  assume $A ==> B$
  then show $A --> B$ **..**
next
  assume $A --> B$ **and** A
  then show B **by** (*rule mp*)
qed

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

lemma *atomize-iff* [*atomize*]: $(A == B) == \mathit{Trueprop}\ (A <-> B)$
proof
  assume $A == B$
  show $A <-> B$ **unfolding** ‹$A == B$› **by** (*rule iff-refl*)
next
  assume $A <-> B$
  then show $A == B$ **by** (*rule iff-reflection*)
qed

lemma *atomize-conj* [*atomize*]:
  includes *meta-conjunction-syntax*
  shows $(A \&\& B) == \mathit{Trueprop}\ (A \ \&\ B)$
proof
  assume *conj*: $A \&\& B$

show *A* & *B*
proof (*rule conjI*)
from *conj* **show** *A* **by** (*rule conjunctionD1*)
from *conj* **show** *B* **by** (*rule conjunctionD2*)
qed
next
assume *conj*: *A* & *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*

1.5 Atomizing elimination rules

setup *AtomizeElim.setup*

lemma *atomize-exL*[*atomize-elim*]: (!!*x*. *P*(*x*) ==> *Q*) == ((*EX x*. *P*(*x*)) ==> *Q*)
by *rule iprover+*

lemma *atomize-conjL*[*atomize-elim*]: (*A* ==> *B* ==> *C*) == (*A* & *B* ==> *C*)
by *rule iprover+*

lemma *atomize-disjL*[*atomize-elim*]: ((*A* ==> *C*) ==> (*B* ==> *C*) ==> *C*) == ((*A* | *B* ==> *C*) ==> *C*)
by *rule iprover+*

lemma *atomize-elimL*[*atomize-elim*]: (!!*B*. (*A* ==> *B*) ==> *B*) == *Trueprop*(*A*)
..

1.6 Calculational rules

lemma *forw-subst*: *a* = *b* ==> *P*(*b*) ==> *P*(*a*)
by (*rule ssubst*)

lemma *back-subst*: *P*(*a*) ==> *a* = *b* ==> *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

1.7 "Let" declarations

nonterminals *letbinds letbind*

constdefs
Let :: [*'a*::{}, *'a* => *'b*] => (*'b*::{})
Let(*s*, *f*) == *f*(*s*)

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

translations
-Let(*-binds*(*b*, *bs*), *e*) == *-Let*(*b*, *-Let*(*bs*, *e*))
let x = a in e == *Let*(*a*, %*x*. *e*)

lemma *LetI*:
assumes !!*x*. *x*=*t* ==> *P*(*u*(*x*))
shows *P*(*let x*=*t in u*(*x*))
apply (*unfold Let-def*)
apply (*rule refl* [*THEN assms*])
done

1.8 Intuitionistic simplification rules

lemma *conj-simps*:
P & *True* <-> *P*
True & *P* <-> *P*
P & *False* <-> *False*
False & *P* <-> *False*
P & *P* <-> *P*
P & *P* & *Q* <-> *P* & *Q*
P & ~*P* <-> *False*
~*P* & *P* <-> *False*
(*P* & *Q*) & *R* <-> *P* & (*Q* & *R*)
by *iprover*+

lemma *disj-simps*:
P | *True* <-> *True*
True | *P* <-> *True*
P | *False* <-> *P*
False | *P* <-> *P*
P | *P* <-> *P*
P | *P* | *Q* <-> *P* | *Q*
(*P* | *Q*) | *R* <-> *P* | (*Q* | *R*)
by *iprover*+

lemma *not-simps*:
~(*P*|*Q*) <-> ~*P* & ~*Q*
~ *False* <-> *True*
~ *True* <-> *False*
by *iprover*+

lemma *imp-simps*:
(*P* --> *False*) <-> ~*P*
(*P* --> *True*) <-> *True*
(*False* --> *P*) <-> *True*
(*True* --> *P*) <-> *P*
(*P* --> *P*) <-> *True*
(*P* --> ~*P*) <-> ~*P*
by *iprover*+

lemma *iff-simps*:
(*True* <-> *P*) <-> *P*
(*P* <-> *True*) <-> *P*
(*P* <-> *P*) <-> *True*
(*False* <-> *P*) <-> ~*P*
(*P* <-> *False*) <-> ~*P*
by *iprover*+

lemma *quant-simps*:
!!*P*. (*ALL x*. *P*) <-> *P*
(*ALL x*. *x*=*t* --> *P*(*x*)) <-> *P*(*t*)
(*ALL x*. *t*=*x* --> *P*(*x*)) <-> *P*(*t*)
!!*P*. (*EX x*. *P*) <-> *P*
EX x. *x*=*t*
EX x. *t*=*x*
(*EX x*. *x*=*t* & *P*(*x*)) <-> *P*(*t*)
(*EX x*. *t*=*x* & *P*(*x*)) <-> *P*(*t*)
by *iprover*+

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*+

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*: *P* ==> (*P* <-> *True*) **by** *iprover*
lemma *iff-reflection-T*: *P* ==> (*P* == *True*) **by** (*rule P-iff-T* [*THEN iff-reflection*])

More rewrite rules

lemma *conj-commute*: $P\&Q <-> Q\&P$ **by** *iprover*
lemma *conj-left-commute*: $P\&(Q\&R) <-> Q\&(P\&R)$ **by** *iprover*
lemmas *conj-comms* = *conj-commute conj-left-commute*

lemma *disj-commute*: $P|Q <-> Q|P$ **by** *iprover*
lemma *disj-left-commute*: $P|(Q|R) <-> Q|(P|R)$ **by** *iprover*
lemmas *disj-comms* = *disj-commute disj-left-commute*

lemma *conj-disj-distribL*: $P\&(Q|R) <-> (P\&Q \mid P\&R)$ **by** *iprover*
lemma *conj-disj-distribR*: $(P|Q)\&R <-> (P\&R \mid Q\&R)$ **by** *iprover*

lemma *disj-conj-distribL*: $P|(Q\&R) <-> (P|Q) \& (P|R)$ **by** *iprover*
lemma *disj-conj-distribR*: $(P\&Q)|R <-> (P|R) \& (Q|R)$ **by** *iprover*

lemma *imp-conj-distrib*: $(P \dashrightarrow (Q\&R)) <-> (P \dashrightarrow Q) \& (P \dashrightarrow R)$ **by** *iprover*
lemma *imp-conj*: $((P\&Q) \dashrightarrow R) <-> (P \dashrightarrow (Q \dashrightarrow R))$ **by** *iprover*
lemma *imp-disj*: $(P|Q \dashrightarrow R) <-> (P \dashrightarrow R) \& (Q \dashrightarrow R)$ **by** *iprover*

lemma *de-Morgan-disj*: $(\sim(P \mid Q)) <-> (\sim P \,\&\, \sim Q)$ **by** *iprover*

lemma *not-ex*: $(\sim (EX\ x.\ P(x))) <-> (ALL\ x.\sim P(x))$ **by** *iprover*
lemma *imp-ex*: $((EX\ x.\ P(x)) \dashrightarrow Q) <-> (ALL\ x.\ P(x) \dashrightarrow Q)$ **by** *iprover*

lemma *ex-disj-distrib*:
$(EX\ x.\ P(x) \mid Q(x)) <-> ((EX\ x.\ P(x)) \mid (EX\ x.\ Q(x)))$ **by** *iprover*

lemma *all-conj-distrib*:
$(ALL\ x.\ P(x) \,\&\, Q(x)) <-> ((ALL\ x.\ P(x)) \,\&\, (ALL\ x.\ Q(x)))$ **by** *iprover*

1.9 Legacy ML bindings

ML ⟪
val refl = @{*thm refl*}
val trans = @{*thm trans*}
val sym = @{*thm sym*}
val subst = @{*thm subst*}
val ssubst = @{*thm ssubst*}
val conjI = @{*thm conjI*}
val conjE = @{*thm conjE*}
val conjunct1 = @{*thm conjunct1*}
val conjunct2 = @{*thm conjunct2*}
val disjI1 = @{*thm disjI1*}
val disjI2 = @{*thm disjI2*}
val disjE = @{*thm disjE*}
val impI = @{*thm impI*}
val impE = @{*thm impE*}
val mp = @{*thm mp*}
val rev-mp = @{*thm rev-mp*}

val TrueI = @{thm TrueI}
val FalseE = @{thm FalseE}
val iff-refl = @{thm iff-refl}
val iff-trans = @{thm iff-trans}
val iffI = @{thm iffI}
val iffE = @{thm iffE}
val iffD1 = @{thm iffD1}
val iffD2 = @{thm iffD2}
val notI = @{thm notI}
val notE = @{thm notE}
val allI = @{thm allI}
val allE = @{thm allE}
val spec = @{thm spec}
val exI = @{thm exI}
val exE = @{thm exE}
val eq-reflection = @{thm eq-reflection}
val iff-reflection = @{thm iff-reflection}
val meta-eq-to-obj-eq = @{thm meta-eq-to-obj-eq}
val meta-eq-to-iff = @{thm meta-eq-to-iff}
⟩

end

2 Classical first-order logic

theory *FOL*
imports *IFOL*
uses
~~/src/Provers/classical.ML
~~/src/Provers/blast.ML
~~/src/Provers/clasimp.ML
~~/src/Tools/induct.ML
(*cladata.ML*)
(*blastdata.ML*)
(*simpdata.ML*)
begin

2.1 The classical axiom

axioms
classical: (*~P ==> P*) *==> P*

2.2 Lemmas and proof tools

lemma *ccontr*: (*¬ P ⟹ False*) *⟹ P*
by (*erule FalseE* [*THEN classical*])

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

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

lemma *exCI*:
assumes *r*: $ALL\ x.\ \sim P(x) \Longrightarrow P(a)$
shows $EX\ x.\ P(x)$
apply (*rule ex-classical*)
apply (*rule notI* [*THEN allI*, *THEN r*])
apply (*erule notE*)
apply (*erule exI*)
done

lemma *excluded-middle*: $\sim P \mid P$
apply (*rule disjCI*)
apply *assumption*
done

ML ⟪
fun excluded-middle-tac sP =
res-inst-tac [(*Q*,*sP*)] (@{*thm excluded-middle*} *RS* @{*thm disjE*})
⟫

lemma *case-split-thm*:
assumes *r1*: $P \Longrightarrow Q$
and *r2*: $\sim P \Longrightarrow Q$
shows *Q*
apply (*rule excluded-middle* [*THEN disjE*])
apply (*erule r2*)
apply (*erule r1*)
done

lemmas *case-split* = *case-split-thm* [*case-names True False*]

ML ⟪

fun case-tac a = res-inst-tac [(P,a)] @{thm case-split-thm}
⟫

lemma *impCE*:
assumes *major*: *P−−>Q*
and *r1*: *~P ==> R*
and *r2*: *Q ==> R*
shows *R*
apply (*rule excluded-middle* [*THEN disjE*])
apply (*erule r1*)
apply (*rule r2*)
apply (*erule major* [*THEN mp*])
done

lemma *impCE'*:
assumes *major*: *P−−>Q*
and *r1*: *Q ==> R*
and *r2*: *~P ==> R*
shows *R*
apply (*rule excluded-middle* [*THEN disjE*])
apply (*erule r2*)
apply (*rule r1*)
apply (*erule major* [*THEN mp*])
done

lemma *notnotD*: *~~P ==> P*
apply (*rule classical*)
apply (*erule notE*)
apply *assumption*
done

lemma *contrapos2*: *[| Q; ~ P ==> ~ Q |] ==> P*
apply (*rule classical*)
apply (*drule* (*1*) *meta-mp*)
apply (*erule* (*1*) *notE*)
done

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: EX! x. P(x)
  and r: !!x. [| P(x);  ALL 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 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

use cladata.ML
setup Cla.setup
setup cla-setup
setup case-setup

use blastdata.ML
setup Blast.setup


lemma ex1-functional: [| EX! z. P(a,z);  P(a,b);  P(a,c) |] ==> b = c
 by blast
```

lemma *True-implies-equals*: (*True* ==> *PROP P*) == *PROP P*
proof
  assume *True* ⟹ *PROP P*
  from *this* **and** *TrueI* **show** *PROP P* **.**
next
  assume *PROP P*
  then show *PROP P* **.**
qed

lemma *uncurry*: *P* --> *Q* --> *R* ==> *P* & *Q* --> *R*
  by *blast*

lemma *iff-allI*: (!!*x*. *P*(*x*) <-> *Q*(*x*)) ==> (*ALL x*. *P*(*x*)) <-> (*ALL x*. *Q*(*x*))
  by *blast*

lemma *iff-exI*: (!!*x*. *P*(*x*) <-> *Q*(*x*)) ==> (*EX x*. *P*(*x*)) <-> (*EX x*. *Q*(*x*))
  by *blast*

lemma *all-comm*: (*ALL x y*. *P*(*x*,*y*)) <-> (*ALL y x*. *P*(*x*,*y*)) **by** *blast*

lemma *ex-comm*: (*EX x y*. *P*(*x*,*y*)) <-> (*EX y x*. *P*(*x*,*y*)) **by** *blast*

lemma *cases-simp*: (*P* --> *Q*) & (~*P* --> *Q*) <-> *Q* **by** *blast*

lemma *int-ex-simps*:
  !!*P Q*. (*EX x*. *P*(*x*) & *Q*) <-> (*EX x*. *P*(*x*)) & *Q*
  !!*P Q*. (*EX x*. *P* & *Q*(*x*)) <-> *P* & (*EX x*. *Q*(*x*))
  !!*P Q*. (*EX x*. *P*(*x*) | *Q*) <-> (*EX x*. *P*(*x*)) | *Q*
  !!*P Q*. (*EX x*. *P* | *Q*(*x*)) <-> *P* | (*EX x*. *Q*(*x*))
  by *iprover+*

lemma *cla-ex-simps*:
  !!*P Q*. (*EX x*. *P*(*x*) --> *Q*) <-> (*ALL x*. *P*(*x*)) --> *Q*
  !!*P Q*. (*EX x*. *P* --> *Q*(*x*)) <-> *P* --> (*EX x*. *Q*(*x*))
  by *blast+*

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

lemma *int-all-simps*:
  !!$P\ Q.\ (ALL\ x.\ P(x)\ \&\ Q) <-> (ALL\ x.\ P(x))\ \&\ Q$
  !!$P\ Q.\ (ALL\ x.\ P\ \&\ Q(x)) <-> P\ \&\ (ALL\ x.\ Q(x))$
  !!$P\ Q.\ (ALL\ x.\ P(x) --> Q) <-> (EX\ x.\ P(x)) --> Q$
  !!$P\ Q.\ (ALL\ x.\ P --> Q(x)) <-> P --> (ALL\ x.\ Q(x))$
  by *iprover+*

lemma *cla-all-simps*:
  !!$P\ Q.\ (ALL\ x.\ P(x)\ |\ Q) <-> (ALL\ x.\ P(x))\ |\ Q$
  !!$P\ Q.\ (ALL\ x.\ P\ |\ Q(x)) <-> P\ |\ (ALL\ x.\ Q(x))$
  by *blast+*

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

lemma *imp-disj1*: $(P-->Q)\ |\ R <-> (P-->Q\ |\ R)$ **by** *blast*
lemma *imp-disj2*: $Q\ |\ (P-->R) <-> (P-->Q\ |\ R)$ **by** *blast*

lemma *de-Morgan-conj*: $(\sim(P\ \&\ Q)) <-> (\sim P\ |\ \sim Q)$ **by** *blast*

lemma *not-imp*: $\sim(P --> Q) <-> (P\ \&\ \sim Q)$ **by** *blast*
lemma *not-iff*: $\sim(P <-> Q) <-> (P <-> \sim Q)$ **by** *blast*

lemma *not-all*: $(\sim\ (ALL\ x.\ P(x))) <-> (EX\ x.\sim P(x))$ **by** *blast*
lemma *imp-all*: $((ALL\ x.\ P(x)) --> Q) <-> (EX\ x.\ P(x) --> Q)$ **by** *blast*

lemmas *meta-simps* =
  triv-forall-equality
  True-implies-equals

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

lemma *notFalseI*: $\sim False$ **by** *iprover*

lemma *cla-simps-misc*:
  $\sim(P\&Q) <-> \sim P\ |\ \sim Q$
  $P\ |\ \sim P$
  $\sim P\ |\ P$
  $\sim\ \sim\ P <-> P$
  $(\sim P --> P) <-> P$
  $(\sim P <-> \sim Q) <-> (P<->Q)$ **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

use *simpdata.ML*
setup *simpsetup*
setup *Simplifier.method-setup Splitter.split-modifiers*
setup *Splitter.setup*
setup *clasimp-setup*
setup *EqSubst.setup*

2.3 Other simple lemmas

lemma [*simp*]: $((P-->R) <-> (Q-->R)) <-> ((P<->Q) \mid R)$
by *blast*

lemma [*simp*]: $((P-->Q) <-> (P-->R)) <-> (P --> (Q<->R))$
by *blast*

lemma *not-disj-iff-imp*: $\sim P \mid Q <-> (P-->Q)$
by *blast*

lemma *conj-mono*: $[\mid P1-->Q1;\ P2-->Q2 \mid] ==> (P1\&P2) --> (Q1\&Q2)$
by *fast*

lemma *disj-mono*: $[\mid P1-->Q1;\ P2-->Q2 \mid] ==> (P1|P2) --> (Q1|Q2)$
by *fast*

lemma *imp-mono*: $[\mid Q1-->P1;\ P2-->Q2 \mid] ==> (P1-->P2)-->(Q1-->Q2)$
by *fast*

lemma *imp-refl*: $P-->P$
by (*rule impI*, *assumption*)

lemma *ex-mono*: $(!!x.\ P(x) --> Q(x)) ==> (EX\ x.\ P(x)) --> (EX\ x.\ Q(x))$
by *blast*

lemma *all-mono*: $(!!x.\ P(x) --> Q(x)) ==> (ALL\ x.\ P(x)) --> (ALL\ x.\ Q(x))$
by *blast*

2.4 Proof by cases and induction

Proper handling of non-atomic rule statements.

constdefs
induct-forall **where** *induct-forall*$(P) == \forall x.\ P(x)$
induct-implies **where** *induct-implies*$(A,\ B) == A \longrightarrow B$
induct-equal **where** *induct-equal*$(x,\ y) == x = y$
induct-conj **where** *induct-conj*$(A,\ B) == A \wedge B$

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

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

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

lemma *induct-conj-eq*:
includes *meta-conjunction-syntax*
shows $(A\ \&\&\ B) == \mathit{Trueprop}(\mathit{induct\text{-}conj}(A,\ B))$
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*, *standard*] = *induct-atomize*
lemmas *induct-rulify-fallback* =
induct-forall-def induct-implies-def induct-equal-def induct-conj-def

hide *const induct-forall induct-implies induct-equal induct-conj*

Method setup.

ML ⟪
structure Induct = InductFun
(
val cases-default = @{*thm case-split*}
val atomize = @{*thms induct-atomize*}
val rulify = @{*thms induct-rulify*}
val rulify-fallback = @{*thms induct-rulify-fallback*}
);
⟫

setup *Induct.setup*
declare *case-split* [*cases type*: *o*]

end