

Isabelle/FOL — First-Order Logic

Larry Paulson and Markus Wenzel

October 10, 2011

Contents

1	Intuitionistic first-order logic	1
1.1	Syntax and axiomatic basis	2
1.2	Lemmas and proof tools	4
1.3	Intuitionistic Reasoning	13
1.4	Atomizing meta-level rules	15
1.5	Atomizing elimination rules	16
1.6	Computational rules	16
1.7	“Let” declarations	16
1.8	Intuitionistic simplification rules	17
2	Classical first-order logic	19
2.1	The classical axiom	19
2.2	Lemmas and proof tools	19
3	Classical Reasoner	22
3.1	Other simple lemmas	26
3.2	Proof by cases and induction	27

1 Intuitionistic first-order logic

```
theory IFOL
imports Pure
uses
  ~/src/Tools/misc-legacy.ML
  ~/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/Tools/eqsubst.ML
  ~/src/Provers/quantifier1.ML
```

```

~~/src/Tools/intuitionistic.ML
~~/src/Tools/project-rule.ML
~~/src/Tools/atomize-elim.ML
(fologic.ML)
(intprover.ML)

```

begin

1.1 Syntax and axiomatic basis

setup *Pure-Thy.old-appl-syntax-setup*

classes *term*

default-sort *term*

typedecl *o*

judgment

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

consts

True :: o

False :: o

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

Not :: $o \Rightarrow o$ (**(~ - [40] 40)**)

conj :: $[o, o] \Rightarrow o$ (**infixr** & 35)

disj :: $[o, o] \Rightarrow o$ (**infixr** | 30)

imp :: $[o, o] \Rightarrow o$ (**infixr** --> 25)

iff :: $[o, o] \Rightarrow o$ (**infixr** <-> 25)

All :: $('a \Rightarrow o) \Rightarrow o$ (**binder ALL** 10)

Ex :: $('a \Rightarrow o) \Rightarrow o$ (**binder EX** 10)

Ex1 :: $('a \Rightarrow o) \Rightarrow o$ (**binder EX!** 10)

abbreviation

not-equal :: $['a, 'a] \Rightarrow o$ (**infixl** $\sim =$ 50) **where**

$x \sim = y == \sim (x = y)$

notation (*xsymbols*)

not-equal (**infixl** \neq 50)

notation (*HTML output*)

not-equal (**infixl** \neq 50)

notation (*xsymbols*)

Not (\neg - [40] 40) **and**
conj (**infixr** \wedge 35) **and**
disj (**infixr** \vee 30) **and**
All (**binder** \forall 10) **and**
Ex (**binder** \exists 10) **and**
Ex1 (**binder** $\exists!$ 10) **and**
imp (**infixr** \longrightarrow 25) **and**
iff (**infixr** \longleftrightarrow 25)

notation (*HTML output*)

Not (\neg - [40] 40) **and**
conj (**infixr** \wedge 35) **and**
disj (**infixr** \vee 30) **and**
All (**binder** \forall 10) **and**
Ex (**binder** \exists 10) **and**
Ex1 (**binder** $\exists!$ 10)

finalconsts

False All Ex eq conj disj imp

axiomatization where

refl: $a=a$ **and**
subst: $a=b \implies P(a) \implies P(b)$

axiomatization where

conjI: $[| P; Q |] \implies P \& Q$ **and**
conjunct1: $P \& Q \implies P$ **and**
conjunct2: $P \& Q \implies Q$ **and**

disjI1: $P \implies P | Q$ **and**
disjI2: $Q \implies P | Q$ **and**
disjE: $[| P | Q; P \implies R; Q \implies R |] \implies R$ **and**

impI: $(P \implies Q) \implies P \longrightarrow Q$ **and**
mp: $[| P \longrightarrow Q; P |] \implies Q$ **and**

FalseE: $False \implies P$

axiomatization where

allI: $(!!x. P(x)) \implies (ALL x. P(x))$ **and**
spec: $(ALL x. P(x)) \implies P(x)$ **and**

exI: $P(x) \implies (EX x. P(x))$ **and**

exE: $\llbracket \text{EX } x. P(x); \text{!!}x. P(x) \implies R \rrbracket \implies R$

axiomatization where

eq-reflection: $(x=y) \implies (x==y)$ **and**
iff-reflection: $(P<->Q) \implies (P==Q)$

lemmas *strip = impI allI*

defs

True-def: $\text{True} == \text{False} \dashrightarrow \text{False}$
not-def: $\sim P == P \dashrightarrow \text{False}$
iff-def: $P <-> Q == (P \dashrightarrow Q) \ \& \ (Q \dashrightarrow P)$

ex1-def: $\text{Ex1}(P) == \text{EX } x. P(x) \ \& \ (\text{ALL } y. P(y) \dashrightarrow y=x)$

1.2 Lemmas and proof tools

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

lemma *conjE*:
assumes *major: P & Q*
and *r: $\llbracket P; Q \rrbracket \implies R$*
shows *R*
apply (*rule r*)
apply (*rule major [THEN conjunct1]*)
apply (*rule major [THEN conjunct2]*)
done

lemma *impE*:
assumes *major: P \dashrightarrow Q*
and *P*
and *r: Q \implies R*
shows *R*
apply (*rule r*)
apply (*rule major [THEN mp]*)
apply (*rule $\langle P \rangle$*)
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: [|  $P$ ;  $P \leftrightarrow Q$  |] ==>  $Q$ 
apply (erule iffD1)
apply assumption
done

lemma rev-iffD2: [|  $Q$ ;  $P \leftrightarrow Q$  |] ==>  $P$ 
apply (erule iffD2)
apply assumption
done

lemma iff-refl:  $P \leftrightarrow P$ 
by (rule iffI)

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

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

lemma exI1:
 $P(a) ==> (!x. P(x) ==> x=a) ==> EX! x. P(x)$ 
apply (unfold ex1-def)
apply (assumption | rule exI conjI allI impI)+
done

lemma ex-exI1:
 $EX x. P(x) ==> (!x y. [| P(x); P(y) |] ==> x=y) ==> EX! x. P(x)$ 
apply (erule exE)
apply (rule exI1)
apply assumption
apply assumption
done

lemma ex1E:
 $EX! x. P(x) ==> (!x. [| P(x); ALL y. P(y) --> y=x |] ==> R) ==> R$ 
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  $(P \rightarrow Q) <-> (P' \rightarrow Q')$ 
  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:  $P \leftrightarrow P' \implies \sim P \leftrightarrow \sim P'$ 
apply (assumption | rule iffI notI | erule (1) notE impE | erule iffE notE)+
done
```

```
lemma all-cong:
assumes  $\forall x. P(x) \leftrightarrow Q(x)$ 
shows  $(\forall x. P(x)) \leftrightarrow (\forall x. Q(x))$ 
apply (assumption | rule iffI allI | erule (1) notE impE | erule allE |
tactic  $\langle\langle$  iff-tac @{thms assms} 1  $\rangle\rangle$ )+
done
```

```
lemma ex-cong:
assumes  $\forall x. P(x) \leftrightarrow Q(x)$ 
shows  $(\exists x. P(x)) \leftrightarrow (\exists x. Q(x))$ 
apply (erule exE | assumption | rule iffI exI | erule (1) notE impE |
tactic  $\langle\langle$  iff-tac @{thms assms} 1  $\rangle\rangle$ )+
done
```

```
lemma ex1-cong:
assumes  $\forall x. P(x) \leftrightarrow Q(x)$ 
shows  $(\exists! x. P(x)) \leftrightarrow (\exists! x. Q(x))$ 
apply (erule ex1E spec [THEN mp] | assumption | rule iffI ex1I | erule (1) notE
impE |
tactic  $\langle\langle$  iff-tac @{thms assms} 1  $\rangle\rangle$ )+
done
```

```
lemma sym:  $a=b \implies b=a$ 
apply (erule subst)
apply (rule refl)
done
```

```
lemma trans:  $[| a=b; b=c |] \implies a=c$ 
apply (erule subst, assumption)
done
```

```
lemma not-sym:  $b \sim a \implies a \sim b$ 
apply (erule contrapos)
apply (erule sym)
done
```

```
lemma def-imp-iff:  $(A == B) \implies A \leftrightarrow B$ 
```

```

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

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

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

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

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

lemma *not-impE*:
 $\sim P \dashrightarrow S \implies (P \implies \text{False}) \implies (S \implies R) \implies R$
apply (*drule mp*)
apply (*rule notI*)
apply *assumption*
apply *assumption*
done

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

lemma *all-impE*:
assumes *major*: $(\text{ALL } x. P(x)) \dashrightarrow S$
and *r1*: $\forall x. P(x)$
and *r2*: $S \implies R$
shows *R*
apply (*rule allI impI major [THEN mp] r1 r2*)
done

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

```

lemma disj-imp-disj:
   $P \mid Q \implies (P \implies R) \implies (Q \implies S) \implies R \mid S$ 
  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}
  )
  >>

```

```

use fologic.ML

```

```

lemma thin-refl:  $\llbracket x=x; PROP W \rrbracket \implies PROP W$  .

```

```

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;
  >>

```

```

setup hypsubst-setup
use intprover.ML

```

1.3 Intuitionistic Reasoning

```

setup << Intuitionistic.method-setup @{binding iprover} >>

```

```

lemma impE':
  assumes 1:  $P \dashv\vdash Q$ 

```

```

    and 2:  $Q \implies R$ 
    and 3:  $P \dashv\vdash Q \implies P$ 
  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:  $\forall x. P(x)$ 
    and 2:  $P(x) \implies \forall x. P(x) \implies Q$ 
  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:  $\sim P$ 
    and 2:  $\sim P \implies P$ 
  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  $\ll$  Context-Rules.addSWrapper (fn tac => hyp-subst-tac ORELSE' tac)  $\gg$ 

```

```

lemma iff-not-sym:  $\sim (Q \leftrightarrow P) \implies \sim (P \leftrightarrow Q)$ 
  by iprover

```

```

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

```

```

lemma eq-commute:  $a=b \leftrightarrow b=a$ 
apply (rule iffI)
apply (erule sym)+
done

```

1.4 Atomizing meta-level rules

lemma *atomize-all* [*atomize*]: $(!!x. P(x)) == \text{Trueprop } (ALL\ x. P(x))$

proof

assume $!!x. P(x)$

then show $ALL\ x. P(x)$..

next

assume $ALL\ x. P(x)$

then show $!!x. P(x)$..

qed

lemma *atomize-imp* [*atomize*]: $(A ==> B) == \text{Trueprop } (A --> B)$

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*]: $(x == y) == \text{Trueprop } (x = y)$

proof

assume $x == y$

show $x = y$ **unfolding** $\langle x == y \rangle$ **by** (*rule refl*)

next

assume $x = y$

then show $x == y$ **by** (*rule eq-reflection*)

qed

lemma *atomize-iff* [*atomize*]: $(A == B) == \text{Trueprop } (A <-> B)$

proof

assume $A == B$

show $A <-> B$ **unfolding** $\langle A == B \rangle$ **by** (*rule iff-refl*)

next

assume $A <-> B$

then show $A == B$ **by** (*rule iff-reflection*)

qed

lemma *atomize-conj* [*atomize*]: $(A \&\&\& B) == \text{Trueprop } (A \& B)$

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

```

nonterminal letbinds and letbind

```

```

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

```

```

syntax

```

$-bind \quad :: [pttrn, 'a] \Rightarrow letbind \quad ((\lambda - \Rightarrow -) 10)$
 $\quad \quad \quad :: letbind \Rightarrow letbinds \quad (-)$
 $-binds \quad :: [letbind, letbinds] \Rightarrow letbinds \quad (-; / -)$
 $-Let \quad \quad :: [letbinds, 'a] \Rightarrow 'a \quad ((let (-) / in (-)) 10)$

translations

$-Let(-binds(b, bs), e) == -Let(b, -Let(bs, e))$
 $let\ x = a\ in\ e \quad ==\ CONST\ Let(a, \%x. e)$

lemma LetI:

assumes $!!x. x=t \implies P(u(x))$
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:

$P \ \& \ True \ \longleftrightarrow \ P$
 $True \ \& \ P \ \longleftrightarrow \ P$
 $P \ \& \ False \ \longleftrightarrow \ False$
 $False \ \& \ P \ \longleftrightarrow \ False$
 $P \ \& \ P \ \longleftrightarrow \ P$
 $P \ \& \ P \ \& \ Q \ \longleftrightarrow \ P \ \& \ Q$
 $P \ \& \ \sim P \ \longleftrightarrow \ False$
 $\sim P \ \& \ P \ \longleftrightarrow \ False$
 $(P \ \& \ Q) \ \& \ R \ \longleftrightarrow \ P \ \& \ (Q \ \& \ R)$
by $iprover+$

lemma disj-simps:

$P \ | \ True \ \longleftrightarrow \ True$
 $True \ | \ P \ \longleftrightarrow \ True$
 $P \ | \ False \ \longleftrightarrow \ P$
 $False \ | \ P \ \longleftrightarrow \ P$
 $P \ | \ P \ \longleftrightarrow \ P$
 $P \ | \ P \ | \ Q \ \longleftrightarrow \ P \ | \ Q$
 $(P \ | \ Q) \ | \ R \ \longleftrightarrow \ P \ | \ (Q \ | \ R)$
by $iprover+$

lemma not-simps:

$\sim(P|Q) \ \longleftrightarrow \ \sim P \ \& \ \sim Q$
 $\sim False \ \longleftrightarrow \ True$
 $\sim True \ \longleftrightarrow \ False$
by $iprover+$

lemma imp-simps:

$(P \ \longrightarrow \ False) \ \longleftrightarrow \ \sim P$

$(P \dashrightarrow True) \leftrightarrow True$
 $(False \dashrightarrow P) \leftrightarrow True$
 $(True \dashrightarrow P) \leftrightarrow P$
 $(P \dashrightarrow P) \leftrightarrow True$
 $(P \dashrightarrow \sim P) \leftrightarrow \sim P$
by *iprover+*

lemma *iff-simps*:

$(True \leftrightarrow P) \leftrightarrow P$
 $(P \leftrightarrow True) \leftrightarrow P$
 $(P \leftrightarrow P) \leftrightarrow True$
 $(False \leftrightarrow P) \leftrightarrow \sim P$
 $(P \leftrightarrow False) \leftrightarrow \sim P$
by *iprover+*

lemma *quant-simps*:

$!!P. (ALL x. P) \leftrightarrow P$
 $(ALL x. x=t \dashrightarrow P(x)) \leftrightarrow P(t)$
 $(ALL x. t=x \dashrightarrow P(x)) \leftrightarrow P(t)$
 $!!P. (EX x. P) \leftrightarrow P$
 $EX x. x=t$
 $EX x. t=x$
 $(EX x. x=t \ \& \ P(x)) \leftrightarrow P(t)$
 $(EX x. t=x \ \& \ P(x)) \leftrightarrow P(t)$
by *iprover+*

lemma *distrib-simps*:

$P \ \& \ (Q \ | \ R) \leftrightarrow P \ \& \ Q \ | \ P \ \& \ R$
 $(Q \ | \ R) \ \& \ P \leftrightarrow Q \ \& \ P \ | \ R \ \& \ P$
 $(P \ | \ Q \ \dashrightarrow R) \leftrightarrow (P \ \dashrightarrow R) \ \& \ (Q \ \dashrightarrow R)$
by *iprover+*

Conversion into rewrite rules

lemma *P-iff-F*: $\sim P \implies (P \leftrightarrow False)$ **by** *iprover*

lemma *iff-reflection-F*: $\sim P \implies (P == False)$ **by** (rule *P-iff-F* [THEN *iff-reflection*])

lemma *P-iff-T*: $P \implies (P \leftrightarrow True)$ **by** *iprover*

lemma *iff-reflection-T*: $P \implies (P == True)$ **by** (rule *P-iff-T* [THEN *iff-reflection*])

More rewrite rules

lemma *conj-commute*: $P \ \& \ Q \leftrightarrow Q \ \& \ P$ **by** *iprover*

lemma *conj-left-commute*: $P \ \& \ (Q \ \& \ R) \leftrightarrow Q \ \& \ (P \ \& \ R)$ **by** *iprover*

lemmas *conj-comms* = *conj-commute conj-left-commute*

lemma *disj-commute*: $P \ | \ Q \leftrightarrow Q \ | \ P$ **by** *iprover*

lemma *disj-left-commute*: $P \ | \ (Q \ | \ R) \leftrightarrow Q \ | \ (P \ | \ R)$ **by** *iprover*

lemmas *disj-comms* = *disj-commute disj-left-commute*

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

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

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

lemma *de-Morgan-disj*: $(\sim(P | Q)) \leftrightarrow (\sim P \& \sim Q)$ **by** *iprover*

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

lemma *ex-disj-distrib*:
 $(EX x. P(x) | Q(x)) \leftrightarrow ((EX x. P(x)) | (EX x. Q(x)))$ **by** *iprover*

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

end

2 Classical first-order logic

theory *FOL*
imports *IFOL*
uses
 $\sim\sim$ */src/Provers/classical.ML*
 $\sim\sim$ */src/Provers/blast.ML*
 $\sim\sim$ */src/Provers/clasimp.ML*
 $\sim\sim$ */src/Tools/induct.ML*
 $\sim\sim$ */src/Tools/case-product.ML*
(simpdata.ML)
begin

2.1 The classical axiom

axiomatization **where**
classical: $(\sim P \implies P) \implies P$

2.2 Lemmas and proof tools

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

```

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

```

```

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

```

```

lemma exCI:
  assumes r:  $ALL\ x.\ \sim P(x) \implies P(a)$ 
  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

```

```

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

```

```

ML <<
  fun case-tac ctxt a = res-inst-tac ctxt [((P, 0), a)] @ {thm case-split}
  >>

```

```

method-setup case-tac = <<
  Args.goal-spec -- Scan.lift Args.name-source >>
  (fn (quant, s) => fn ctxt => SIMPLE-METHOD'' quant (case-tac ctxt s))
  >> case-tac emulation (dynamic instantiation!)

```

```

lemma impCE:
  assumes major:  $P \dashrightarrow Q$ 
    and r1:  $\sim P \implies R$ 
    and r2:  $Q \implies R$ 
  shows R
  apply (rule excluded-middle [THEN disjE])
  apply (erule r1)
  apply (rule r2)
  apply (erule major [THEN mp])
  done

```

```

lemma impCE':
  assumes major:  $P \dashrightarrow Q$ 
    and r1:  $Q \implies R$ 
    and r2:  $\sim P \implies R$ 
  shows R
  apply (rule excluded-middle [THEN disjE])
  apply (erule r2)
  apply (rule r1)
  apply (erule major [THEN mp])
  done

```

```

lemma notnotD:  $\sim\sim P \implies P$ 
  apply (rule classical)
  apply (erule notE)
  apply assumption
  done

```

```

lemma contrapos2:  $[[ Q; \sim P \implies \sim Q ]] \implies P$ 
  apply (rule classical)
  apply (drule (1) meta-mp)
  apply (erule (1) notE)
  done

```

```

lemma iffCE:
  assumes major:  $P \leftrightarrow Q$ 
    and r1:  $[[ P; Q ]] \implies R$ 
    and r2:  $[[ \sim P; \sim Q ]] \implies R$ 
  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 * :  $\forall y. P(y) \longrightarrow y = x$ 
  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  $\forall y y'. P(y) \wedge P(y') \longrightarrow y = y'$  by (intro strip, elim conjE) (rule r')
qed
qed

```

```

lemma imp-elim:  $P \longrightarrow Q \implies (\sim R \implies P) \implies (Q \implies R) \implies R$ 
  by (rule classical) iprover

```

```

lemma swap:  $\sim P \implies (\sim R \implies P) \implies R$ 
  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;
>>

setup <<
  ML-Antiquote.value @{binding claset}
  (Scan.succeed Cla.claset-of (ML-Context.the-local-context ()))
>>

setup Cla.setup

lemmas [intro!] = refl TrueI conjI disjCI impI notI iffI
  and [elim!] = conjE disjE impCE FalseE iffCE
ML << val prop-cs = @{claset} >>

lemmas [intro!] = allI ex-ex1I
  and [intro] = exI
  and [elim!] = exE alt-ex1E
  and [elim] = allE
ML << val FOL-cs = @{claset} >>

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;
>>

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*: $P \dashrightarrow Q \dashrightarrow R \implies P \& Q \dashrightarrow R$
by *blast*

lemma *iff-allI*: $(\forall x. P(x) \leftrightarrow Q(x)) \implies (ALL\ x. P(x)) \leftrightarrow (ALL\ x. Q(x))$
by *blast*

lemma *iff-exI*: $(\forall x. P(x) \leftrightarrow Q(x)) \implies (EX\ x. P(x)) \leftrightarrow (EX\ x. Q(x))$
by *blast*

lemma *all-comm*: $(ALL\ x\ y. P(x,y)) \leftrightarrow (ALL\ y\ x. P(x,y))$ **by** *blast*

lemma *ex-comm*: $(EX\ x\ y. P(x,y)) \leftrightarrow (EX\ y\ x. P(x,y))$ **by** *blast*

lemma *cases-simp*: $(P \dashrightarrow Q) \& (\sim P \dashrightarrow Q) \leftrightarrow Q$ **by** *blast*

lemma *int-ex-simps*:

$!!P\ Q. (EX\ x. P(x) \& Q) \leftrightarrow (EX\ x. P(x)) \& Q$
 $!!P\ Q. (EX\ x. P \& Q(x)) \leftrightarrow P \& (EX\ x. Q(x))$
 $!!P\ Q. (EX\ x. P(x) \mid Q) \leftrightarrow (EX\ x. P(x)) \mid Q$
 $!!P\ Q. (EX\ x. P \mid Q(x)) \leftrightarrow P \mid (EX\ x. Q(x))$
by *iprover+*

lemma *cla-ex-simps*:

$!!P\ Q. (EX\ x. P(x) \dashrightarrow Q) \leftrightarrow (ALL\ x. P(x)) \dashrightarrow Q$
 $!!P\ Q. (EX\ x. P \dashrightarrow Q(x)) \leftrightarrow P \dashrightarrow (EX\ x. Q(x))$
by *blast+*

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

lemma *int-all-simps*:

$!!P\ Q. (ALL\ x. P(x) \& Q) \leftrightarrow (ALL\ x. P(x)) \& Q$
 $!!P\ Q. (ALL\ x. P \& Q(x)) \leftrightarrow P \& (ALL\ x. Q(x))$
 $!!P\ Q. (ALL\ x. P(x) \dashrightarrow Q) \leftrightarrow (EX\ x. P(x)) \dashrightarrow Q$
 $!!P\ Q. (ALL\ x. P \dashrightarrow Q(x)) \leftrightarrow P \dashrightarrow (ALL\ x. Q(x))$
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*: (~ (P & Q)) <-> (~ P | ~ Q) by blast

lemma *not-imp*: ~ (P --> Q) <-> (P & ~ Q) by blast

lemma *not-iff*: ~ (P <-> Q) <-> (P <-> ~ Q) by blast

lemma *not-all*: (~ (ALL x. P(x))) <-> (EX x. ~ 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*: ~ False by iprover

lemma *cla-simps-misc*:

~ (P & Q) <-> ~ P | ~ Q

P | ~ P

~ P | P

~ ~ P <-> P

(~ P --> P) <-> P

(~ P <-> ~ 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*

```

simproc-setup defined-Ex (EX x. P(x)) = << fn - => Quantifier1.rearrange-ex >>
simproc-setup defined-All (ALL x. P(x)) = << fn - => Quantifier1.rearrange-all
>>

```

```

ML <<
(*intuitionistic simprules only*)
val FOL-ss =
  FOL-basic-ss
  addsimps (@{thms meta-simps} @ @{thms FOL-simps} @ @{thms int-ex-simps}
@ @{thms int-all-simps})
  addsimprocs [@{simproc defined-All}, @{simproc defined-Ex}]
  addcongs [@{thm imp-cong}];

```

```

(*classical simprules too*)
val FOL-ss = FOL-ss addsimps (@{thms cla-simps} @ @{thms cla-ex-simps} @
@{thms cla-all-simps});
>>

```

```

setup << Simplifier.map-simpset-global (K FOL-ss) >>

```

```

setup Simplifier.method-setup Splitter.split-modifiers
setup Splitter.setup
setup clasimp-setup
setup EqSubst.setup

```

3.1 Other simple lemmas

```

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

```

```

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

```

```

lemma not-disj-iff-imp: ~P | Q <-> (P-->Q)
by blast

```

```

lemma conj-mono: [| P1-->Q1; P2-->Q2 |] ==> (P1&P2) --> (Q1&Q2)
by fast

```

```

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

```

```

lemma imp-mono: [| Q1-->P1; P2-->Q2 |] ==> (P1-->P2)-->(Q1-->Q2)
by fast

```

```

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

```

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

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

3.2 Proof by cases and induction

Proper handling of non-atomic rule statements.

definition *induct-forall*(P) == $\forall x. P(x)$

definition *induct-implies*(A, B) == $A \longrightarrow B$

definition *induct-equal*(x, y) == $x = y$

definition *induct-conj*(A, B) == $A \wedge B$

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

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

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

lemma *induct-conj-eq*: $(A \&\&\& B) == \text{Trueprop}(\text{induct-conj}(A, B))$
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 = 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
  );
  >>
```

```
setup Induct.setup  
declare case-split [cases type: o]  
  
setup Case-Product.setup  
  
hide-const (open) eq  
  
end
```