

Isabelle/FOL — First-Order Logic

Larry Paulson and Markus Wenzel

May 22, 2012

Contents

1	Intuitionistic first-order logic	1
1.1	Syntax and axiomatic basis	2
1.1.1	Equality	2
1.1.2	Propositional logic	2
1.1.3	Quantifiers	3
1.1.4	Definitions	3
1.1.5	Additional notation	3
1.2	Lemmas and proof tools	4
1.3	Intuitionistic Reasoning	13
1.4	Atomizing meta-level rules	14
1.5	Atomizing elimination rules	15
1.6	Calculational 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	26

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 => prop          ((-) 5)

```

1.1.1 Equality

```

axiomatization
  eq :: ['a, 'a] => o (infixl = 50)
where
  refl:      a=a and
  subst:     a=b ==> P(a) ==> 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 |] ==> P&Q and
  conjunct1: P&Q ==> P and
  conjunct2: P&Q ==> Q and

  disjI1: P ==> P|Q and
  disjI2: Q ==> P|Q and
  disjE: [| P|Q; P ==> R; Q ==> R |] ==> R and

  impI: (P ==> Q) ==> P-->Q and
  mp: [| P-->Q; P |] ==> Q and

```

FalseE: $False ==> P$

1.1.3 Quantifiers

axiomatization

All :: $('a ==> o) ==> o$ (**binder ALL 10**) and

Ex :: $('a ==> o) ==> o$ (**binder EX 10**)

where

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

spec: $(ALL x. P(x)) ==> P(x)$ and

exI: $P(x) ==> (EX x. P(x))$ and

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

1.1.4 Definitions

definition *True* == $False --> False$

definition *Not* (\sim - [40] 40) **where** *not-def*: $\sim P == P --> False$

definition *iff* (**infixr** $<->$ 25) **where** $P <-> Q == (P --> Q) \& (Q --> P)$

definition *Ex1* :: $('a ==> o) ==> o$ (**binder EX! 10**)

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

axiomatization where — Reflection, admissible

eq-reflection: $(x=y) ==> (x==y)$ and

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

1.1.5 Additional notation

abbreviation *not-equal* :: $['a, 'a] ==> o$ (**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)

1.2 Lemmas and proof tools

lemmas *strip* = *impI allI*

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* $\langle P \rangle$)
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: ( $P \implies False$ )  $\implies \sim P$ 
unfolding not-def by (erule impI)

```

```

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

```

```

lemma rev-notE: [ $P$ ;  $\sim P$ ]  $\implies R$ 
by (erule notE)

```

```

lemma not-to-imp:
assumes  $\sim P$ 
and  $r: P \longrightarrow False \implies Q$ 
shows  $Q$ 
apply (rule r)
apply (rule impI)
apply (erule notE [OF  $\sim P$ ])
done

```

```

lemma rev-mp: [ $P$ ;  $P \longrightarrow Q$ ]  $\implies Q$ 
by (erule mp)

```

```

lemma contrapos:
assumes major:  $\sim Q$ 
and minor:  $P \implies Q$ 
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:  $Q \leftrightarrow P \implies P \leftrightarrow Q$ 
  apply (erule iffE)
  apply (rule iffI)
  apply (assumption | erule mp)+
  done

```

```

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

```

```

lemma ex1I:
   $P(a) \implies (!x. P(x) \implies x=a) \implies \exists x. P(x)$ 
  apply (unfold ex1-def)
  apply (assumption | rule exI conjI allI impI)+
  done

```

```

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

```

```

lemma ex1E:
   $\exists x. P(x) \implies (!x. [P(x); \forall y. P(y) \implies y=x] \implies R) \implies R$ 
  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  $P \leftrightarrow P'$ 
  and  $P' \implies Q \leftrightarrow Q'$ 

```

```

shows  $(P \& Q) \leftrightarrow (P' \& Q')$ 
apply (insert assms)
apply (assumption | rule iffI conjI | erule iffE conjE mp |
  tactic  $\ll$  iff-tac  $\@$ {thms assms} 1  $\gg$ )+
done

```

```

lemma conj-cong2:
assumes  $P \leftrightarrow P'$ 
  and  $P' \implies Q \leftrightarrow Q'$ 
shows  $(Q \& P) \leftrightarrow (Q' \& P')$ 
apply (insert assms)
apply (assumption | rule iffI conjI | erule iffE conjE mp |
  tactic  $\ll$  iff-tac  $\@$ {thms assms} 1  $\gg$ )+
done

```

```

lemma disj-cong:
assumes  $P \leftrightarrow P'$  and  $Q \leftrightarrow Q'$ 
shows  $(P | Q) \leftrightarrow (P' | Q')$ 
apply (insert assms)
apply (erule iffE disjE disjI1 disjI2 | assumption | rule iffI | erule (1) notE
impE)+
done

```

```

lemma imp-cong:
assumes  $P \leftrightarrow P'$ 
  and  $P' \implies Q \leftrightarrow Q'$ 
shows  $(P \dashrightarrow Q) \leftrightarrow (P' \dashrightarrow Q')$ 
apply (insert assms)
apply (assumption | rule iffI impI | erule iffE | erule (1) notE impE |
  tactic  $\ll$  iff-tac  $\@$ {thms assms} 1  $\gg$ )+
done

```

```

lemma iff-cong:  $\ll$   $P \leftrightarrow P'; Q \leftrightarrow Q'$   $\gg \implies (P \leftrightarrow Q) \leftrightarrow (P' \leftrightarrow Q')$ 
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  $\ll$  iff-tac  $\@$ {thms assms} 1  $\gg$ )+
done

```

```

lemma ex-cong:

```

```

assumes !! $x$ .  $P(x) \leftrightarrow Q(x)$ 
shows  $(\exists x. P(x)) \leftrightarrow (\exists x. Q(x))$ 
apply (erule exE | assumption | rule iffI exI | erule (1) notE impE |
  tactic << iff-tac @ {thms assms} 1 >>)+
done

```

```

lemma ex1-cong:
assumes !! $x$ .  $P(x) \leftrightarrow Q(x)$ 
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 << iff-tac @ {thms assms} 1 >>)+
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:  $(A == B) \implies A = B$ 
apply unfold
apply (rule refl)
done

```

```

lemma meta-eq-to-iff:  $x==y \implies x \leftrightarrow y$ 
by unfold (rule iff-refl)

```

```

lemma ssubst: [ $b = a$ ;  $P(a)$ ]  $\implies P(b)$ 
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)-->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:
   $\sim P \dashrightarrow S \implies (P \implies \text{False}) \implies (S \implies R) \implies R$ 
  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:  $!!x. P(x)$ 
  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:
   $P|Q \implies (P \implies R) \implies (Q \implies S) \implies R|S$ 
  apply (erule disjE)
  apply (rule disjI1) apply assumption
  apply (rule disjI2) apply assumption
  done

ML ⟨⟨
  structure Project-Rule = Project-Rule
  (
    val conjunct1 =  $\@ \{ \text{thm conjunct1} \}$ 
    val conjunct2 =  $\@ \{ \text{thm conjunct2} \}$ 
  )
  ⟩⟩

```

```

    val mp = @{thm mp}
  )
  >>

```

```

use fologic.ML

```

```

lemma thin-refl: [|x=x; PROP W|] ==> 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 --> 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: $\sim P$
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*]: $(!!x. P(x)) \implies \text{Trueprop } (ALL x. P(x))$
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*]: $(A \implies B) \implies \text{Trueprop } (A \dashrightarrow B)$
proof
assume $A \implies B$
then show $A \dashrightarrow B$..
next
assume $A \dashrightarrow B$ 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*]: $(!!B. (A ==> B) ==> B) == \text{Trueprop}(A)$
..

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*::{*t*}, '*a* => '*b*] => ('*b*::{*t*}) **where**
 $\text{Let}(s, f) == f(s)$

syntax

-bind :: [*pttrn*, '*a*] => *letbind* ((*?* =/ -) 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*))
 $\text{let } x = a \text{ in } e == \text{CONST } \text{Let}(a, \%x. e)$

lemma *LetI*:

assumes $!!x. x=t ==> P(u(x))$
shows $P(\text{let } x=t \text{ in } u(x))$
apply (*unfold Let-def*)
apply (*rule refl [THEN assms]*)
done

1.8 Intuitionistic simplification rules

lemma *conj-simps*:

$P \& \text{True} \leftrightarrow P$
 $\text{True} \& P \leftrightarrow P$
 $P \& \text{False} \leftrightarrow \text{False}$
 $\text{False} \& P \leftrightarrow \text{False}$
 $P \& P \leftrightarrow P$
 $P \& P \& Q \leftrightarrow P \& Q$
 $P \& \sim P \leftrightarrow \text{False}$
 $\sim P \& P \leftrightarrow \text{False}$
 $(P \& Q) \& R \leftrightarrow P \& (Q \& R)$
by *iprover+*

lemma *disj-simps*:

$P \mid \text{True} \leftrightarrow \text{True}$
 $\text{True} \mid P \leftrightarrow \text{True}$
 $P \mid \text{False} \leftrightarrow P$
 $\text{False} \mid P \leftrightarrow P$
 $P \mid P \leftrightarrow P$
 $P \mid P \mid Q \leftrightarrow P \mid Q$
 $(P \mid Q) \mid R \leftrightarrow P \mid (Q \mid R)$
by *iprover+*

lemma *not-simps*:

$\sim(P \mid Q) \leftrightarrow \sim P \& \sim Q$
 $\sim \text{False} \leftrightarrow \text{True}$
 $\sim \text{True} \leftrightarrow \text{False}$
by *iprover+*

lemma *imp-simps*:

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

lemma *iff-simps*:

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

lemma *quant-simps*:

$!!P. (\text{ALL } x. P) \leftrightarrow P$

$(\text{ALL } x. x=t \text{ ---} \rightarrow P(x)) \text{ <-> } P(t)$
 $(\text{ALL } x. t=x \text{ ---} \rightarrow P(x)) \text{ <-> } P(t)$
 $!!P. (\text{EX } x. P) \text{ <-> } P$
 $\text{EX } x. x=t$
 $\text{EX } x. t=x$
 $(\text{EX } x. x=t \ \& \ P(x)) \text{ <-> } P(t)$
 $(\text{EX } x. t=x \ \& \ P(x)) \text{ <-> } P(t)$
by iprover+

lemma *distrib-simps*:

$P \ \& \ (Q \ | \ R) \text{ <-> } P \ \& \ Q \ | \ P \ \& \ R$
 $(Q \ | \ R) \ \& \ P \text{ <-> } Q \ \& \ P \ | \ R \ \& \ P$
 $(P \ | \ Q \ \text{---} \rightarrow R) \text{ <-> } (P \ \text{---} \rightarrow R) \ \& \ (Q \ \text{---} \rightarrow R)$
by iprover+

Conversion into rewrite rules

lemma *P-iff-F*: $\sim P \text{ ==> } (P \text{ <-> } \text{False})$ **by iprover**
lemma *iff-reflection-F*: $\sim P \text{ ==> } (P \text{ == } \text{False})$ **by (rule P-iff-F [THEN iff-reflection])**
lemma *P-iff-T*: $P \text{ ==> } (P \text{ <-> } \text{True})$ **by iprover**
lemma *iff-reflection-T*: $P \text{ ==> } (P \text{ == } \text{True})$ **by (rule P-iff-T [THEN iff-reflection])**

More rewrite rules

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

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

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

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

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

lemma *de-Morgan-disj*: $(\sim(P \ | \ Q)) \text{ <-> } (\sim P \ \& \ \sim Q)$ **by iprover**

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

lemma *ex-disj-distrib*:

$(EX x. P(x) \mid Q(x)) \leftrightarrow ((EX x. P(x)) \mid (EX x. Q(x)))$ **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*

keywords *print-claset print-induct-rules :: diag*

uses

~~/src/Provers/classical.ML

~~/src/Provers/blast.ML

~~/src/Provers/clasimp.ML

~~/src/Tools/induct.ML

~~/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 *contr*: $(\neg P \implies False) \implies P$

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:  $P \implies Q$ 
  and r1:  $\sim P \implies R$ 
  and r2:  $Q \implies R$ 
  shows  $R$ 
  apply (rule excluded-middle [THEN disjE])
  apply (erule r1)
  apply (erule 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:  $!!x. [[ P(x); ALL y y'. P(y) \& P(y') \dashrightarrow y=y' ]] \implies R$ 
  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  $r' = \text{this}$ 
  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 --> 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: $(P \dashrightarrow Q) \ \& \ (\sim P \dashrightarrow Q) \ \leftrightarrow \ Q$ **by blast**

lemma int-ex-simps:

$!!P \ Q. \ (EX \ x. \ P(x) \ \& \ Q) \ \leftrightarrow \ (EX \ x. \ P(x)) \ \& \ Q$
 $!!P \ Q. \ (EX \ x. \ P \ \& \ Q(x)) \ \leftrightarrow \ P \ \& \ (EX \ x. \ Q(x))$
 $!!P \ Q. \ (EX \ x. \ P(x) \ | \ Q) \ \leftrightarrow \ (EX \ x. \ P(x)) \ | \ Q$
 $!!P \ Q. \ (EX \ x. \ P \ | \ Q(x)) \ \leftrightarrow \ P \ | \ (EX \ x. \ Q(x))$
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:

$!!P \ Q. \ (ALL \ x. \ P(x) \ | \ Q) \ \leftrightarrow \ (ALL \ x. \ P(x)) \ | \ Q$
 $!!P \ Q. \ (ALL \ x. \ P \ | \ Q(x)) \ \leftrightarrow \ P \ | \ (ALL \ x. \ Q(x))$
by blast+

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

lemma imp-disj1: $(P \dashrightarrow Q) \ | \ R \ \leftrightarrow \ (P \dashrightarrow Q \ | \ R)$ **by blast**

lemma imp-disj2: $Q \ | \ (P \dashrightarrow R) \ \leftrightarrow \ (P \dashrightarrow Q \ | \ R)$ **by blast**

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

lemma not-imp: $\sim(P \dashrightarrow Q) \leftrightarrow (P \ \& \ \sim Q)$ **by blast**

lemma not-iff: $\sim(P \leftrightarrow Q) \leftrightarrow (P \leftrightarrow \sim Q)$ **by blast**

lemma not-all: $(\sim (ALL\ x.\ P(x))) \leftrightarrow (EX\ x.\ \sim P(x))$ **by blast**

lemma imp-all: $((ALL\ x.\ P(x)) \dashrightarrow Q) \leftrightarrow (EX\ x.\ P(x) \dashrightarrow Q)$ **by blast**

lemmas meta-simps =

triv-forall-equality

True-implies-equals

lemmas IFOL-simps =

reft [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:

$\sim(P \ \& \ Q) \leftrightarrow \sim P \ | \ \sim Q$

$P \ | \ \sim P$

$\sim P \ | \ P$

$\sim \sim P \leftrightarrow P$

$(\sim P \dashrightarrow P) \leftrightarrow P$

$(\sim P \leftrightarrow \sim Q) \leftrightarrow (P \leftrightarrow Q)$ **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* $(EX\ x.\ P(x)) = \langle\langle\ fn \dashrightarrow Quantifier1.rearrange-ex \rangle\rangle$

simproc-setup *defined-All* $(ALL\ x.\ P(x)) = \langle\langle\ fn \dashrightarrow Quantifier1.rearrange-all$

$\rangle\rangle$

ML $\langle\langle$

*(*intuitionistic simprules only*)*

val IFOL-ss =

FOL-basic-ss

addsims @{thms meta-simps IFOL-simps int-ex-simps int-all-simps}

addsimprocs [@{simproc defined-All}, @{simproc defined-Ex}]

|> Simplifier.add-cong @{thm imp-cong};

*(*classical simprules too*)*

val FOL-ss = IFOL-ss addsims @{thms cla-simps cla-ex-simps cla-all-simps};

$\rangle\rangle$

setup $\langle\langle\ Simplifier.map-simpset-global (K\ FOL-ss) \rangle\rangle$

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

3.1 Other simple lemmas

lemma [*simp*]: $((P \dashrightarrow R) \leftrightarrow (Q \dashrightarrow R)) \leftrightarrow ((P \leftrightarrow Q) \mid R)$
by *blast*

lemma [*simp*]: $((P \dashrightarrow Q) \leftrightarrow (P \dashrightarrow R)) \leftrightarrow (P \dashrightarrow (Q \leftrightarrow R))$
by *blast*

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

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

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

lemma *imp-mono*: $[[Q1 \dashrightarrow P1; P2 \dashrightarrow Q2]] \implies (P1 \dashrightarrow P2) \dashrightarrow (Q1 \dashrightarrow Q2)$
by *fast*

lemma *imp-refl*: $P \dashrightarrow P$
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*: $(!!x. P(x)) \implies \text{Trueprop}(\text{induct-forall}(\lambda x. P(x)))$
unfolding *atomize-all induct-forall-def* .

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