Logic

An Idris port of Coq.Init.Logic

Eric Bailey
https://github.com/yurrriq
eric@ericb.me

ABSTRACT

Here I present an Idris port of the `Coq.Init.Logic` module from the Coq standard library.

Keywords

logic, coq, idris

```
module Logic

import Data.Bifunctor

%default total

%access export
```

1. PROPOSITIONAL CONNECTIVES

1.1 ⊤

`()` is the always true proposition.

```
%elim data Unit = MkUnit
```

1.2 ⊥

`Void` is the always false proposition.

```
%elim data Void : Type where
```

1.3 ¬

`Not a`, written `~a`, is the negation of `a`.

```
Not : Type -> Type
Not a = a -> Void
```

```
syntax "~" [x] = (Not x)
```

1.4 ∧

`And a b`, written `(a, b)`, is the conjunction of `a` and `b`.

`Conj p q` is a proof of `(a, b)` as soon as `p` is a proof of `a` and `q` a proof of `b`.

`proj1` and `proj2` are first and second projections of a conjunction.

```
syntax "(" [a] "," [b] ")" = (And a b)

data And : Type -> Type -> Type where
     Conj : a -> b -> (a, b)

implementation Bifunctor And where
    bimap f g (Conj a b) = Conj (f a) (g b)

proj1 : (a, b) -> a
proj1 (Conj a _) = a

proj2 : (a, b) -> b
proj2 (Conj _ b) = b
```

1.5 Biconditional

$$\frac{\varphi \vdash \psi \quad \psi \vdash \varphi}{\varphi \leftrightarrow \psi}$$

`iff a b`, written `a <-> b`, expresses the equivalence of `a` and `b`.

```
infixl 9 <->

public export
(<->) : Type -> Type -> Type
(<->) a b = (a -> b, b -> a)
```

1.5.1 *Biconditional is Reflexive*

Proof Wiki

$\vdash \varphi \leftrightarrow \varphi$

```
iffRefl : a <-> a
iffRefl = Conj id id
```

1.5.2 Biconditional is Transitive

Proof Wiki

$$\frac{\varphi \leftrightarrow \psi \qquad \psi \leftrightarrow \chi}{\vdash \varphi \leftrightarrow \chi}$$

```
iffTrans : (a <-> b) -> (b <-> c) -> (a <-> c)
iffTrans (Conj ab ba) (Conj bc cb) =
    Conj (bc . ab) (ba . cb)
```

1.5.3 Biconditional is Commutative

Proof Wiki

$\varphi \leftrightarrow \psi \dashv\vdash \psi \leftrightarrow \varphi$

or

$\vdash (\varphi \leftrightarrow \psi) \leftrightarrow (\psi \leftrightarrow \varphi)$

```
iffSym : (a <-> b) -> (b <-> a)
iffSym (Conj ab ba) = Conj ba ab
```

1.5.4 andIffCompatLeft

$\psi \leftrightarrow \chi \dashv\vdash (\varphi \wedge \psi) \leftrightarrow (\varphi \wedge \chi)$

```
andIffCompatLeft : (b <-> c) -> ((a, b) <-> (a, c))
andIffCompatLeft = bimap second second
```

1.5.5 andIffCompatRight

$\psi \leftrightarrow \chi \dashv\vdash (\psi \wedge \varphi) \leftrightarrow (\chi \wedge \varphi)$

```
andIffCompatRight : (b <-> c) -> ((b, a) <-> (c, a))
andIffCompatRight = bimap first first
```

1.5.6 orIffCompatLeft

$\psi \leftrightarrow \chi \vdash (\varphi \vee \psi) \leftrightarrow (\varphi \vee \chi)$

```
orIffCompatLeft : (b <-> c) ->
                  (Either a b <-> Either a c)
orIffCompatLeft = bimap second second
```

$\psi \leftrightarrow \chi \vdash (\psi \vee \varphi) \leftrightarrow (\chi \vee \varphi)$

```
orIffCompatRight : (b <-> c) ->
                   (Either b a <-> Either c a)
orIffCompatRight = bimap first first
```

$\neg\varphi \dashv\vdash \varphi \leftrightarrow \bot$

or

$\vdash \neg\varphi \leftrightarrow (\varphi \leftrightarrow \bot)$

```
negVoid : Not a <-> (a <-> Void)
negVoid = Conj (flip Conj void) proj1
```

$$\frac{\begin{array}{l}\psi \to \varphi \\ \chi \to \varphi\end{array}}{((\varphi \wedge \psi) \leftrightarrow (\varphi \wedge \chi)) \leftrightarrow (\psi \leftrightarrow \chi)}$$

```
andCancelLeft : (b -> a) ->
                (c -> a) ->
                (((a, b) <-> (a, c)) <-> (b <-> c))
andCancelLeft ba ca = Conj (bimap f g) andIffCompatLeft
  where
    f h b = proj2 . h $ Conj (ba b) b
    g h c = proj2 . h $ Conj (ca c) c
```

$$\frac{\vdash \to \varphi, \psi}{\vdash \psi \wedge \varphi}$$

$$\frac{\begin{array}{l}\psi \to \varphi \\ \chi \to \varphi\end{array}}{((\psi \wedge \varphi) \leftrightarrow (\chi \wedge \varphi)) \leftrightarrow (\psi \leftrightarrow \chi)}$$

$$\frac{\psi \wedge \varphi}{\psi}$$

$$\frac{\begin{array}{l}\psi \to \varphi \\ \psi\end{array}}{\psi \wedge \varphi}$$

$$\frac{\begin{array}{l}\psi \wedge \varphi \vdash \psi \\ \psi \to \varphi, \psi \vdash \psi \wedge \varphi\end{array}}{\psi \leftrightarrow (\psi \wedge \varphi)}$$

$$\frac{\chi \wedge \varphi}{\chi}$$

$$\frac{\begin{array}{l}\chi \to \varphi \\ \chi\end{array}}{\chi \wedge \varphi}$$

$$\frac{\begin{array}{l}\chi \wedge \varphi \vdash \chi \\ \chi \to \varphi, \chi \vdash \chi \wedge \varphi\end{array}}{\chi \leftrightarrow (\chi \wedge \varphi)}$$

$$\frac{\begin{array}{l}\psi \leftrightarrow (\psi \wedge \varphi) \\ \chi \leftrightarrow (\chi \wedge \varphi) \\ (\psi \wedge \varphi) \to (\chi \wedge \varphi)\end{array}}{\psi \to \chi}$$

$$\frac{\psi \wedge \varphi}{\varphi \wedge \psi}$$

$$\frac{\chi \wedge \varphi}{\chi \wedge \psi}$$

$$\frac{\begin{array}{l}\psi \leftrightarrow (\psi \wedge \varphi) \\ \chi \leftrightarrow (\chi \wedge \varphi) \\ (\chi \wedge \varphi) \to (\psi \wedge \varphi)\end{array}}{\chi \to \psi}$$

$$\frac{\begin{array}{l}\psi \to \chi \\ \chi \to \psi\end{array}}{\psi \leftrightarrow \chi}$$

```
andCancelRight : (b -> a) ->
                 (c -> a) ->
                 (((b, a) <-> (c, a)) <-> (b <-> c))
andCancelRight ba ca = Conj (bimap f g) andIffCompatRight
  where
    f h b = ?rhs -- proj1 . h £ Conj b (ba b)
    g h c = proj1 . h $ Conj c (ca c)
```

1.5.7 Conjunction is Commutative

Proof Wiki

Formulation 1. $a \wedge b \dashv\vdash b \wedge a$

Formulation 1. $\vdash (a \wedge b) \leftrightarrow (b \wedge a)$

Source.

```
andComm : (a, b) <-> (b, a)
andComm = Conj swap swap
  where
    swap : (p, q) -> (q, p)
    swap (Conj p q) = Conj q p
```

1.5.8 Conjunction is Associative

Proof Wiki

Formulation 1. $(a \wedge b) \wedge c \dashv\vdash a \wedge (b \wedge c)$

Formulation 2. $\vdash ((a \wedge b) \wedge c) \leftrightarrow (a \wedge (b \wedge c))$

Source.

```
andAssoc : ((a, b), c) <-> (a, (b, c))
andAssoc = Conj f g
  where
    f abc@(Conj (Conj a b) c) = Conj a (first proj2 abc)
    g abc@(Conj a (Conj b c)) = Conj (second proj1 abc) c
```

1.5.9 orCancelLeft

$(b \to \neg a) \to (c \to \neg a) \to (((a \vee b) \leftrightarrow (a \vee c)) \leftrightarrow (b \leftrightarrow c))$

```
orCancelLeft : (b -> Not a) -> (c -> Not a) ->
               ((Either a b <-> Either a c) <-> (b <-> c))
orCancelLeft bNotA cNotA = Conj (bimap f g) orIffCompatLeft
  where
    f ef b = go (bNotA b) (ef (Right b))
    g eg c = go (cNotA c) (eg (Right c))
    go : Not a -> Either a b -> b
    go lf = either (void . lf) id
```

1.5.10 orCancelRight

$$\begin{array}{l} \psi \vdash \neg\varphi \\ \chi \vdash \neg\varphi \\ \hline ((\psi \vee \varphi) \leftrightarrow (\chi \vee \varphi)) \leftrightarrow (\psi \leftrightarrow \chi) \end{array}$$

```
or_cancel_r : (b -> Not a)
            -> (c -> Not a)
            -> ((Either b a <-> Either c a) <-> (b <-> c))
-- or_cancel_r bNotA cNotA = (bimap f g, or_iff_compat_r)
--   where
--     f ef b = go (bNotA b) (ef (Left b))
--     g eg c = go (cNotA c) (eg (Left c))
--     go : Not p -> Either q p -> q
--     go rf = either id (void . rf)
```

$(A \vee B) \leftrightarrow (B \vee A)$

```
or_comm : Either a b <-> Either b a
-- or_comm = (mirror, mirror)
```

$((A \vee B) \vee C) \to (A \vee (B \vee C))$

```
or_assoc_lemma1 : Either (Either a b) c -> Either a (Either b c
-- or_assoc_lemma1 = either (second Left) (pure . pure)
```

$(A \vee (B \vee C)) \to ((A \vee B) \vee C)$

```
or_assoc_lemma2 : Either a (Either b c) -> Either (Either a b)
-- or_assoc_lemma2 = either (Left . Left) (first Right)
```

$((A \vee B) \vee C) \leftrightarrow (A \vee (B \vee C))$

```
or_assoc : Either (Either a b) c <-> Either a (Either b c)
-- or_assoc = (or_assoc_lemma1, or_assoc_lemma2)
```

$(A \leftrightarrow B) \to ((A \to B) \wedge (B \to A))$

```
iff_and : (a <-> b) -> (a -> b, b -> a)
-- iff_and = id
```

$(A \leftrightarrow B) \leftrightarrow ((A \to B) \wedge (B \to A))$

```
iff_to_and : (a <-> b) <-> (a -> b, b -> a)
-- iff_to_and = (id, id)
```