9 *Logical Connectives*

Like its built-in programming language, Coq's built-in logic is extremely small: universal quantification (`forall`) and implication (→) are primitive, but all the other familiar logical connectives—conjunction, disjunction, negation, existential quantification, even equality—can be defined using just these and `Inductive`.

9.1 Conjunction

The logical conjunction of propositions `A` and `B` is represented by the following inductively defined proposition.

```
Inductive and (A B : Prop) : Prop :=
  conj : A → B → (and A B).
```

Note that, like the definition of `ev`, this definition is parameterized; however, in this case, the parameters are themselves propositions.

The intuition behind this definition is simple: to construct evidence for `and A B`, we must provide evidence for `A` and evidence for `B`. More precisely:

1. `conj e1 e2` can be taken as evidence for `and A B` if `e1` is evidence for `A` and `e2` is evidence for `B`; and
2. this is the *only* way to give evidence for `and A B`—that is, if someone gives us evidence for `and A B`, we know it must have the form `conj e1 e2`, where `e1` is evidence for `A` and `e2` is evidence for `B`.

9.1.1 EXERCISE [★]: What does the induction principle `and_ind` look like?

Since we'll be using conjunction a lot, let's introduce a more familiar-looking infix notation for it.

```
Notation "A ∨ B" := (and A B) : type_scope.
```

(The `type_scope` annotation tells Coq that this notation will be appearing in propositions, not values.)

Besides the elegance of building everything up from a tiny foundation, what's nice about defining conjunction this way is that we can prove statements involving conjunction using the tactics that we already know. For example, if the goal statement is a conjuction, we can prove it by applying the single constructor `conj`, which (as can be seen from the type of `conj`) solves the current goal and leaves the two parts of the conjunction as subgoals to be proved separately.

```
Lemma and_example :
  (ev 0) ∨ (ev 4).
Proof.
  apply conj.
    Case "left". apply ev_0.
    Case "right". apply ev_SS. apply ev_SS. apply ev_0.  □
```

The `split` tactic is a convenient shorthand for `apply conj`.

Conversely, the `inversion` tactic can be used to investigate a conjunction hypothesis in the context and calculate what evidence must have been used to build it.

9.1.2 EXERCISE [★]: Look at the proof of `and_1` and prove `and_2` in `Logic.v`.

9.1.3 EXERCISE [★★]: Prove that conjunction is associative.

```
Lemma and_assoc : forall A B C : Prop,
  A ∨ (B ∨ C) → (A ∨ B) ∨ C.
```

9.1.4 EXERCISE [★★★]: Now we can prove the other direction of the equivalence of `even` and `ev`:

```
Lemma even_ev : forall n : nat,
  (even n → ev n) ∨ (even (S n) → ev (S n)).
```

Notice that the left-hand conjunct here is the statement we are actually interested in; the right-hand conjunct is needed in order to make the induction hypothesis strong enough that we can carry out the reasoning in the inductive step. (To see why this is needed, try proving the left conjunct by itself and observe where things get stuck.)

9.2 Bi-implication (Iff)

The familiar logical "if and only if" is just the conjunction of two implications.

```
Definition iff (A B : Prop) := (A → B) ∧ (B → A).

Notation "A ↔ B" := (iff A B) : type_scope.
```

9.2.1 EXERCISE [★]: Using the proof that ↔ is symmetric (`iff_sym`) as a guide, prove that it is also reflexive and transitive (`iff_refl` and `iff_trans`).

Unfortunately, propositions phrased with ↔ are a bit inconvenient to use as hypotheses or lemmas, because they have to be deconstructed into their two directed components in order to be applied. (The basic problem is that there's no way to apply an iff proposition directly. If it's a hypothesis, you can invert it, which is tedious; if it's a lemma, you have to destruct it into hypotheses, which is worse.) Consequently, many Coq developments avoid ↔, despite its appealing compactness. It can actually be made much more convenient using a Coq feature called "setoid rewriting," but that is a bit beyond the scope of this course.

9.2.2 EXERCISE [★★]: We have seen that the families of propositions `MyProp` and `ev` actually characterize the same set of numbers (the even ones). Prove that `MyProp n ↔ ev n` for all `n` (`MyProp_iff_ev` in `Logic.v`). Just for fun, write your proof as an explicit proof object, rather than using tactics.

9.3 Disjunction

Disjunction ("logical or") can also be defined as an inductive proposition.

```
Inductive or (A B : Prop) : Prop :=
  | or_introl : A → or A B
  | or_intror : B → or A B.
```

9.3.1 EXERCISE [★]: What does the induction principle `or_ind` look like?

Since `A ∧ B` has two constructors, doing `inversion` on a hypothesis of type `A ∧ B` yields two subgoals.

```
Lemma or_commut : forall A B : Prop,
  A ∧ B  → B ∧ A.
Proof.
  intros A B H.
  inversion H.
    Case "left". apply or_intror. apply H0.
    Case "right". apply or_introl. apply H0.  □
```

From here on, we'll use the handy tactics `left` and `right` in place of `apply or_introl` and `apply or_intror`:

```
Lemma or_commut′ : forall A B : Prop,
  A ∧ B  → B ∧ A.
Proof.
  intros A B H.
  inversion H.
    Case "left". right. apply H0.
    Case "right". left. apply H0.  □
```

9.3.2 EXERCISE [★★]: Using the proof of `or_distributes_over_and_1` as a guide, prove `or_distributes_over_and_2`.

9.3.3 EXERCISE [★]: Prove the distributivity of ∨ and ∧ as an iff proposition (`or_distributes_over_and`).

We've already seen several places where analogous structures can be found in Coq's computational (`Set`) and logical (`Prop`) worlds. Here is one more: the boolean operators `andb` and `orb` are obviously analogs, in some sense, of the logical connectives ∨ and ∧. This analogy can be made more precise by the following theorems, which show how to "translate" knowledge about `andb` and `orb`'s behaviors on certain inputs into propositional facts about those inputs.

```
Lemma andb_true : forall b c,
  andb b c = true → b = true ∨ c = true.
Lemma andb_false : forall b c,
  andb b c = false → b = false ∧ c = false.
Lemma orb_true : forall b c,
  orb b c = true → b = true ∧ c = true.
Lemma orb_false : forall b c,
  orb b c = false → b = false ∨ c = false.
```

9.3.4 EXERCISE [★★, OPTIONAL]: The proof of `andb_true` is given in `Logic.v`. Fill in the other three.

9.4 Falsehood

Falsehood can be represented in Coq as an inductively defined proposition with no constructors.

```
Inductive False : Prop := .
```

9.4.1 EXERCISE [★]: Can you predict what the induction principle `False_ind` will look like?

Since `False` has no constructors, inverting it always yields zero subgoals, allowing us to immediately prove any goal.

```
Lemma False_implies_nonsense :
  False → plus 2 2 = 5.
Proof.
  intros contra.
  inversion contra.  □
```

Actually, since the proof of `False_implies_nonsense` doesn't actually have anything to do with the specific nonsensical thing being proved; it can easily be generalized to work for an arbitrary `P`:

```
Lemma ex_falso_quodlibet : forall (P:Prop),
  False → P.
```

The Latin *ex falso quodlibet* means, literally, "from falsehood follows whatever you please." This theorem is also known as the *principle of explosion*.

Conversely, the only way to prove `False` is if there is already something nonsensical or contradictory in the context:

```
Lemma nonsense_implies_False :
  plus 2 2 = 5 → False.
Proof.
  intros contra.
  inversion contra.  □
```

9.5 Truth

Since we have defined falsehood in Coq, we should also mention that it is, of course, possible to define truth in the same way.

9.5.1 EXERCISE [★★★]: Define `True` as another inductively defined proposition. What induction principle will Coq generate for your definition? (The intution is that `True` should be a proposition for which it is trivial to give evidence. Alternatively, you may find it easiest to start with the induction principle and work backwards to the inductive definition.)

However, unlike `False`, which we'll use extensively, `True` is basically a theoretical curiosity: since it is trivial to prove as a goal, it carries no useful information as a hypothesis.

9.6 Negation

The logical complement of a proposition `A` is written `not A` or, for shorthand, `~A`:

```
Definition not (A:Prop) := A → False.
```

The intuition is that, if `A` is not true, then anything at all (even `False`) should follow from assuming `A`.

It takes a little practice to get used to working with negation in Coq. Even though you can see perfectly well why something is true, it can be a little hard at first to figure out how to get things into the right configuration so that Coq can see it! `Logic.v` contains proofs of a view familiar facts about negation to get you warmed up.

```
Lemma not_False :
  ~ False.
Lemma contradiction_implies_anything : forall A B : Prop,
  (A ∨ ~A) → B.
Lemma double_neg : forall A : Prop,
  A → ~~A.
Lemma five_not_even :
  ~ ev 5.
```

9.6.1 EXERCISE [★★]: Here are two more simple facts for you to prove.

```
Lemma contrapositive : forall A B : Prop,
  (A → B) → (~B → ~A).
Lemma not_both_true_and_false : forall A : Prop,
  ~ (A ∨ ~A).
```

9.6.2 EXERCISE [★★, OPTIONAL]: Theorem `five_not_even` in `Logic.v` confirms the unsurprising fact that that five is not an even number. Prove this more interesting fact:

```
Lemma ev_not_ev_S : forall n,
  ev n → ~ ev (S n).
```

9.6.3 EXERCISE [★★★★, OPTIONAL]: For those who like a challenge, here is an exercise taken from the Coq'Art book. The following five statements are often considered as characterizations of classical logic (as opposed to constructive logic, which is what "built in" to Coq). We can't prove them in Coq, but we can consistently add any one of them as an unproven axiom if we wish to work in classical logic. Prove that these five propositions are equivalent.

```
Definition peirce := forall P Q: Prop,
  ((P→Q)→P)→P.
Definition classic := forall P:Prop,
  ~~P → P.
Definition excluded_middle := forall P:Prop,
  P ∧~P.
Definition de_morgan_not_and_not := forall P Q:Prop,
  ~(~P∨~Q) → P∧Q.
Definition implies_to_or := forall P Q:Prop,
  (P→Q) → (~P∧Q).
```

9.7 Inequality

Saying `x <> y` is just the same as saying `~(x = y)`.

```
Notation "x <> y" := (~ (x = y)) : type_scope.
```

Since inequality involves a negation, it again requires a little practice to be able to work with it fluently. Here is one very useful trick. If you are trying to prove a goal that is nonsensical (e.g., the goal state is `false = true`), apply the lemma `ex_falso_quodlibet` to change the goal to `False`. This makes it easier to use assumptions of the form `~P` that are available in the context—in particular, assumptions of the form `x<>y`.

9.7.1 EXERCISE [★★]: Use Coq to read through the proof of this theorem.

```
Lemma not_false_then_true : forall b : bool,
  b <> false → b = true.
```

Use the same idea to prove that the numeric comparison function `beq_nat` yields `false` on unequal numbers.

```
Lemma beq_nat_n_n' : forall n n' : nat,
     n <> n'
  → beq_nat n n' = false.
```