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.

```
Theorem 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.

```
Theorem 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`:

```
Theorem 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.

```
Theorem or_commut : forall A B : Prop,
  A ∨ B  → B ∨ A.
Proof.
  intros A B H.
  inversion H as [HA | HB].
    Case "left". apply or_intror. apply HA.
    Case "right". apply or_introl. apply HB.  □
```

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

```
Theorem or_commut' : forall A B : Prop,
  A ∨ B  → B ∨ A.
Proof.
  intros A B H.
  inversion H as [HA | HB].
    Case "left". right. apply HA.
    Case "right". left. apply HB.  □
```

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.

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

9.3.4 EXERCISE [★]: 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.

```
Theorem 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`:

```
Theorem 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:

```
Theorem 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.

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

9.6.1 EXERCISE [★★]: `double_neg_in` FIX ME

9.6.2 EXERCISE [★★]: Prove the following fact:

```
Theorem contrapositive : forall A B : Prop,
  (A → B) → (~B → ~A).
```

9.6.3 EXERCISE [★]: Prove the following simple fact:

```
Theorem not_both_true_and_false : forall A : Prop,
  ~ (A ∧ ~A).
```

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

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

9.6.5 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 is "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.

```
Theorem 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.

```
Theorem not_eq_false_beq : forall n n′ : nat,
     n <> n′
  → false = beq_nat n n′.
```

9.7.2 EXERCISE [★★★★, OPTIONAL]: The converse of `beq_false_not_eq` says that if `beq_nat` yields `false` then its arguments are unequal. Prove it.

```
Theorem beq_false_not_eq : forall n m,
  false = beq_nat n m → n <> m.
```

9.8 Existential Quantification

Another extremely important logical connective is *existential quantification*.

```
Inductive ex (X : Set) (P : X → Prop) : Prop :=
  ex_intro : forall witness:X, P witness → ex X P.
```

The intuition behind this definition is that, in order to give evidence for the assertion "there exists an `x` for which the proposition `P` holds" we must actually name a *witness*—a specific value `x`—and then give evidence for `P x`.

We can use Coq's notation definition facility to introduce more standard notation for writing existentially quantified propositions, exactly parallel to the built-in syntax for universally quantified propositions. Instead of writing `(ex nat (fun x => ev x))` to express the proposition that there exists some number that is even, for example, we can write `exists x:nat, ev x`. (The exact definition of the notation is in `Logic.v`, of course, but it is not necessary to understand the details).

We can use the same tactics as always for manipulate existentials. For example, if to prove an existential, we `apply` the constructor `ex_intro`. Since the premise of `ex_intro` involves a variable (`witness`) that does not appear in its conclusion, we need to explicitly give its value when we use `apply`.

```
Example exists_example_1 : exists n, plus n (mult n n)
   = 6.
Proof.
  apply ex_intro with (witness:=2).
  reflexivity.  □
```

Or, instead of writing `apply ex_intro with (witness:=...)`, we can use the shorthand tactic `exists ...`.

```
Proof.
  exists 2.
  reflexivity.  □
```

Conversely, if we have an existential hypothesis in the context, we can eliminate it with `destruct`.

```
Theorem exists_example_2 : forall n,
     (exists m, n = plus 4 m)
  → (exists o, n = plus 2 o).
Proof.
  intros n H.
```

```
  inversion H as [m Hm].
  exists (plus 2 m).
  apply Hm.  □
```

Note the use of the `as...` pattern to name the variable that Coq introduces to name the witness value. (If we don't explicitly choose one, Coq will just call it `witness`, which makes proofs confusing.) *)

9.8.1 EXERCISE [★]: Prove that "`P` holds for all `x`" and "there is no `x` for which `P` does not hold" are equivalent assertions.

```
Theorem dist_not_exists : forall (X:Set) (P : X → Prop),
  (forall x, P x) → ~ (exists x, ~ P x).
```

9.8.2 EXERCISE [★★]: Prove that existential quantification distributes over disjunction.

```
Theorem dist_exists_or : forall (X:Set) (P Q : X → Prop),
  (exists x, P x ∨ Q x) ↔ (exists x, P x) ∨ (exists x, Q x).
```

9.9 Equality

Even Coq's equality relation is not actually built in. It has the following inductive definition:

```
Inductive eq (A:Set) : A → A → Prop :=
  refl_equal : forall x, eq A x x.

Notation "x = y" := (eq _ x y) : type_scope.
```

This definition is a bit subtle. The way to think about it is that, given a set `A`, it defines a *family* of propositions "`x` is equal to `y`," indexed by pairs of values (`x` and `y`) from `A`. There is just one way of constructing evidence for members of this family: by applying the constructor `refl_equal` to two identical arguments.

Actually, the Coq library defines equality in a slightly different way:

```
Inductive eq' (A:Set) (x:A) : A → Prop :=
    refl_equal' : eq' A x x.
```

Although this definition probably looks even more puzzling than the first one, they are actually equivalent.

9.9.1 EXERCISE [★★★, OPTIONAL]: Verify that the two definitions are equivalent (theorem `two_defs_of_eq_coincide` in `Logic.v`).

The advantage of the second definition is that the induction principle that Coq derives for it is precisely the familiar principle of *Leibniz equality*: what we mean when we say "x and y are equal" is that every property on A that is true of x is also true of y.

```
Check eq'_ind.
```

```
▶ eq'_ind
    : forall (A : Set) (x : A) (P : A → Prop),
      P x → forall y : A, eq' A x y → P y
```

9.9.2 EXERCISE [★★★]: Prove (again) that equality is transitive without using `rewrite` or `reflexivity`. (If you're stumped, try some of the tactics we've used with other inductively defined datatypes in the past.) This is theorem `trans_eq'` in `Logic.v`.

9.10 Inversion, Again

See `Logic.v` *for the text of this section.*

9.11 Induction principles in `Prop`

See `Logic.v` *for the text of this section.*

9.11.1 EXERCISE [★★★]: Do the exercise marked `p_provability` in `Logic.v`.

10 *Relations*

A proposition parameterized over a number (like `ev`) can be thought of as a *predicate*—i.e., it defines a subset of `nat`, namely those numbers for which the proposition is provable. In the same way, a two-argument proposition can be thought of as a *relation*—i.e., it defines a set of pairs for which the proposition is provable. In this chapter, we explore the consequences of this observation.

10.1 Relations as Propositions

We've already seen an inductive definition of one fundamental relation: equality. Another very useful one is the "less than or equal to" relation on numbers:

```
Inductive le : nat → nat → Prop :=
  | le_n : forall n, le n n
  | le_S : forall n m, (le n m) → (le n (S m)).

Notation "m <= n" := (le m n).
```

This definition should be fairly intuitive. It says that there are two ways to give evidence that one number is less than or equal to another: either observe that they are the same number, or give evidence that the first is less than or equal to the predecessor of the second.

This is a fine definition of the `<=` relation, but we can streamline it a little by observing that the left-hand argument `n` is the same everywhere in the definition, so we can actually make it a "general parameter" to the whole definition, rather than an argument to each constructor.

```
Inductive le (n:nat) : nat → Prop :=
  | le_n : le n n
  | le_S : forall m, (le n m) → (le n (S m)).
```

The reason to prefer the second definition even though it is a little less symmetric, so less intuitive, is that (like the second definition of =) it gives rise to a simpler induction principle:

```
Check le_ind.   (* the second one *)
```

```
▶ le_ind
     : forall (n : nat) (P : nat → Prop),
       P n →
       (forall m : nat, n <= m → P m → P (S m)) →
       forall n0 : nat, n <= n0 → P n0
```

By contrast, the induction principle that Coq calculates for the first definition has a lot of extra quantifiers, which makes it messier to work with when proving things by induction.

```
Check le_ind.  (* the first one *)
```

```
▶ le_ind
     : forall P : nat → nat → Prop,
       (forall n : nat, P n n) →
       (forall n m : nat, FirstLe.le n m → P n m → P n (S m)) →
       forall n n0 : nat, FirstLe.le n n0 → P n n0
```

Proofs of facts about <= using the constructors `le_n` and `le_S` follow the same patterns as proofs about predicates, like `ev` in the previous chapter. We can `apply` the constructors to prove <= goals (e.g., to show that `3<=3` or `3<=6`), and we can use tactics like `inversion` to extract information from <= hypotheses in the context (e.g., to prove that `~(2 <= 1)`.)

Here are some other simple relations on numbers:

```
Inductive square_of : nat → nat → Prop :=
  sq : forall n:nat, square_of n (mult n n).
Inductive next_nat (n:nat) : nat → Prop :=
  | nn : next_nat n (S n).
Inductive next_even (n:nat) : nat → Prop :=
  | ne_1 : ev (S n) → next_even n (S n)
  | ne_2 : ev (S (S n)) → next_even n (S (S n)).
```

10.1.1 EXERCISE [★★]: Define an inductive relation `total_relation` that holds between every pair of natural numbers.

10.1.2 EXERCISE [★★]: Define an inductive relation `empty_relation` (on numbers) that never holds.

10.1.3 EXERCISE [★★★]: We can define three-place relations, four-place relations, etc., in just the same way as binary relations. For example, consider the following three-place relation on numbers:

```
Inductive R : nat → nat → nat → Prop :=
  | c1 : R 0 0 0
  | c2 : forall m n o, R m n o → R (S m) n (S o)
  | c3 : forall m n o, R m n o → R m (S n) (S o)
  | c4 : forall m n o, R (S m) (S n) (S (S o)) → R m n o
  | c5 : forall m n o, R m n o → R n m o.
```

1. Which of the following propositions are provable?
   (a) `R 1 1 2`
   (b) `R 2 2 6`
2. If we dropped constructor `c5` from the definition of `R`, would the set of provable propositions change? Briefly (1 sentence) explain your answer.
3. If we dropped constructor `c4` from the definition of `R`, would the set of provable propositions change? Briefly (1 sentence) explain your answer.

10.1.4 EXERCISE [★★★★, CHALLENGE PROBLEM]: One of the main purposes of Coq is to prove that programs match their specifications. To this end, let's prove that our definition of `filter` matches a specification. Here is the specification, written out informally in English.

> Suppose we have a set `X`, a function `test: X→bool`, and a list `l` of type `list X`. Suppose further that `l` is an in-order merge of two lists, `l1` and `l2`, such that every item in `l1` satisfies `test` and no item in `l2` satisfies test. Then `filter test l = l1`.

Your job is to translate this specification into a Coq theorem and prove it. (Hint: You'll need to begin by defining what it means for one list to be a merge of two others. Do this with an inductive relation, not a `Fixpoint`.)

10.2 Relations, in General

See `Logic.v` *for the text of this section.*

10.2.1 EXERCISE [★★]: Prove that the total relation is not a partial function, but that the empty relation is.

10.2.2 EXERCISE [★★]: Show the transitivity of `lt` directly, by induction on the derivation.

10.2.3 EXERCISE [★★]: Show the transitiveity of `lt` directly, by induction on `o` (the greatest variable).

10.2.4 EXERCISE [★]: Prove the following theorem about `le`:

```
Theorem le_S_n : forall n m,
  (S n <= S m) → (n <= m).
```

10.2.5 EXERCISE [★★]: Write an informal proof of `le_Sn_n`: $\forall n, \neg(n + 1 <= n)$.

10.2.6 EXERCISE [★]: Using your informal proof, construct a formal proof of `le_Sn_n`.

10.2.7 EXERCISE [★★]: Show that `le` is not `symmetric`.

10.2.8 EXERCISE [★★]: Show that `le` is `antisymmetric`.

10.3 Some Facts about Orderings

Let's pause briefly to record several facts about the <= and < relations and the `ble_nat` function that we are going to need later in the course.

```
Theorem O_le_n : forall n,
  0 <= n.

Theorem le_plus : forall a b,
  a <= a + b.

Theorem plus_lt : forall n1 n2 m,
  plus n1 n2 < m →
  n1 < m ∧ n2 < m.

Theorem n_le_m__Sn_le_Sm : forall n m,
  n <= m → S n <= S m.

Theorem lt_S : forall n m,
  n < m →
  n < S m.

Theorem le_step : forall n m p,
  n < m →
  m <= S p →
  n <= p.

Theorem ble_nat_true : forall n m,
  ble_nat n m = true → n <= m.
```

```
Theorem ble_nat_n_Sn_false : forall n m,
  ble_nat n (S m) = false →
  ble_nat n m = false.

Theorem ble_nat_false : forall n m,
  ble_nat n m = false → ~(n <= m).
```

10.3.1 EXERCISE [★★, OPTIONAL]: Prove some or all of these.