SOFTWARE FOUNDATIONS

VOLUME 1: LOGICAL FOUNDATIONS

TABLE OF CONTENTS INDEX ROADMAP

PROOFOBJECTS

THE CURRY-HOWARD CORRESPONDENCE

```
Set Warnings "-notation-overridden,-parsing".
From LF Require Export IndProp.
```

"*Algorithms are the computational content of proofs*." —Robert Harper

We have seen that Coq has mechanisms both for *programming*, using inductive data types like `nat` or `list` and functions over these types, and for *proving* properties of these programs, using inductive propositions (like `ev`), implication, universal quantification, and the like. So far, we have mostly treated these mechanisms as if they were quite separate, and for many purposes this is a good way to think. But we have also seen hints that Coq's programming and proving facilities are closely related. For example, the keyword `Inductive` is used to declare both data types and propositions, and `→` is used both to describe the type of functions on data and logical implication. This is not just a syntactic accident! In fact, programs and proofs in Coq are almost the same thing. In this chapter we will study how this works.

We have already seen the fundamental idea: provability in Coq is represented by concrete *evidence*. When we construct the proof of a basic proposition, we are actually building a tree of evidence, which can be thought of as a data structure.

If the proposition is an implication like `A → B`, then its proof will be an evidence *transformer*: a recipe for converting evidence for A into evidence for B. So at a fundamental level, proofs are simply programs that manipulate evidence.

Question: If evidence is data, what are propositions themselves?

Answer: They are types!

Look again at the formal definition of the ev property.

```
Print ev.
(* ==>
  Inductive ev : nat -> Prop :=
    | ev_0 : ev 0
    | ev_SS : forall n, ev n -> ev (S (S n)).
*)
```

Suppose we introduce an alternative pronunciation of ":". Instead of "has type," we can say "is a proof of." For example, the second line in the definition of ev declares that ev_0 : ev 0. Instead of "ev_0 has type ev 0," we can say that "ev_0 is a proof of ev 0."

This pun between types and propositions — between : as "has type" and : as "is a proof of" or "is evidence for" — is called the *Curry-Howard correspondence*. It proposes a deep connection between the world of logic and the world of computation:

```
                propositions  ~  types
                proofs        ~  data values
```

See [Wadler 2015] for a brief history and up-to-date exposition.

Many useful insights follow from this connection. To begin with, it gives us a natural interpretation of the type of the ev_SS constructor:

```
Check ev_SS.
(* ===> ev_SS : forall n,
                  ev n ->
                  ev (S (S n)) *)
```

This can be read "ev_SS is a constructor that takes two arguments — a number n and evidence for the proposition ev n — and yields evidence for the proposition ev (S (S n))."

Now let's look again at a previous proof involving ev.

```
Theorem ev_4 : ev 4.
Proof.
  apply ev_SS. apply ev_SS. apply ev_0. Qed.
```

As with ordinary data values and functions, we can use the Print command to see the *proof object* that results from this proof script.

```
Print ev_4.
(* ===> ev_4 = ev_SS 2 (ev_SS 0 ev_0)
     : ev 4  *)
```

Indeed, we can also write down this proof object *directly*, without the need for a separate proof script:

```
Check (ev_SS 2 (ev_SS 0 ev_0)).
(* ===> ev 4 *)
```

The expression `ev_SS 2 (ev_SS 0 ev_0)` can be thought of as instantiating the parameterized constructor `ev_SS` with the specific arguments `2` and `0` plus the corresponding proof objects for its premises `ev 2` and `ev 0`. Alternatively, we can think of `ev_SS` as a primitive "evidence constructor" that, when applied to a particular number, wants to be further applied to evidence that that number is even; its type,

```
∀ n, ev n → ev (S (S n)),
```

expresses this functionality, in the same way that the polymorphic type `∀X, list X` expresses the fact that the constructor `nil` can be thought of as a function from types to empty lists with elements of that type.

We saw in the Logic chapter that we can use function application syntax to instantiate universally quantified variables in lemmas, as well as to supply evidence for assumptions that these lemmas impose. For instance:

```
Theorem ev_4': ev 4.
Proof.
  apply (ev_SS 2 (ev_SS 0 ev_0)).
Qed.
```

Proof Scripts

The *proof objects* we've been discussing lie at the core of how Coq operates. When Coq is following a proof script, what is happening internally is that it is gradually constructing a proof object — a term whose type is the proposition being proved. The tactics between `Proof` and `Qed` tell it how to build up a term of the required type. To see this process in action, let's use the `Show Proof` command to display the current state of the proof tree at various points in the following tactic proof.

```
Theorem ev_4'' : ev 4.
Proof.
  Show Proof.
  apply ev_SS.
  Show Proof.
  apply ev_SS.
  Show Proof.
  apply ev_0.
  Show Proof.
Qed.
```

At any given moment, Coq has constructed a term with a "hole" (indicated by `?Goal` here, and so on), and it knows what type of evidence is needed to fill this hole.

Each hole corresponds to a subgoal, and the proof is finished when there are no more subgoals. At this point, the evidence we've built stored in the global context under the name given in the `Theorem` command.

Tactic proofs are useful and convenient, but they are not essential: in principle, we can always construct the required evidence by hand, as shown above. Then we can use `Definition` (rather than `Theorem`) to give a global name directly to this evidence.

```
Definition ev_4''' : ev 4 :=
  ev_SS 2 (ev_SS 0 ev_0).
```

All these different ways of building the proof lead to exactly the same evidence being saved in the global environment.

```
Print ev_4.
(* ===> ev_4    =   ev_SS 2 (ev_SS 0 ev_0) : ev 4 *)
Print ev_4'.
(* ===> ev_4'   =   ev_SS 2 (ev_SS 0 ev_0) : ev 4 *)
Print ev_4''.
(* ===> ev_4''  =   ev_SS 2 (ev_SS 0 ev_0) : ev 4 *)
Print ev_4'''.
(* ===> ev_4''' =   ev_SS 2 (ev_SS 0 ev_0) : ev 4 *)
```

Exercise: 2 stars (eight_is_even)

Give a tactic proof and a proof object showing that `ev 8`.

```
Theorem ev_8 : ev 8.
Proof.
  (* FILL IN HERE *) Admitted.

Definition ev_8' : ev 8
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.
```

□

Quantifiers, Implications, Functions

In Coq's computational universe (where data structures and programs live), there are two sorts of values with arrows in their types: *constructors* introduced by `Inductive`ly defined data types, and

functions.

Similarly, in Coq's logical universe (where we carry out proofs), there are two ways of giving evidence for an implication: constructors introduced by `Inductive`ly defined propositions, and... functions!

For example, consider this statement:

```
Theorem ev_plus4 : ∀ n, ev n → ev (4 + n).
Proof.
  intros n H. simpl.
  apply ev_SS.
  apply ev_SS.
  apply H.
Qed.
```

What is the proof object corresponding to `ev_plus4`?

We're looking for an expression whose *type* is `∀n, ev n → ev (4 + n)` — that is, a *function* that takes two arguments (one number and a piece of evidence) and returns a piece of evidence!

Here it is:

```
Definition ev_plus4' : ∀ n, ev n → ev (4 + n) :=
  fun (n : nat) ⇒ fun (H : ev n) ⇒
    ev_SS (S (S n)) (ev_SS n H).
```

Recall that `fun n ⇒ blah` means "the function that, given `n`, yields `blah`," and that Coq treats `4 + n` and `S (S (S (S n)))` as synonyms. Another equivalent way to write this definition is:

```
Definition ev_plus4'' (n : nat) (H : ev n)
                      : ev (4 + n) :=
  ev_SS (S (S n)) (ev_SS n H).

Check ev_plus4''.
(* ===>
     : forall n : nat, ev n -> ev (4 + n) *)
```

When we view the proposition being proved by `ev_plus4` as a function type, one interesting point becomes apparent: The second argument's type, `ev n`, mentions the *value* of the first argument, `n`.

While such *dependent types* are not found in conventional programming languages, they can be useful in programming too, as the recent flurry of activity in the functional programming community demonstrates.

Notice that both implication (`→`) and quantification (`∀`) correspond to functions on evidence. In fact, they are really the same thing: `→` is just a shorthand for a degenerate use of `∀`where there is no dependency, i.e., no need to give a name to the type on the left-hand side of the

arrow:

```
           ∀ (x:nat), nat
      =    ∀ (_:nat), nat
      =    nat → nat
```

For example, consider this proposition:

```
Definition ev_plus2 : Prop :=
  ∀ n, ∀ (E : ev n), ev (n + 2).
```

A proof term inhabiting this proposition would be a function with two arguments: a number `n` and some evidence `E` that `n` is even. But the name `E` for this evidence is not used in the rest of the statement of `ev_plus2`, so it's a bit silly to bother making up a name for it. We could write it like this instead, using the dummy identifier `_` in place of a real name:

```
Definition ev_plus2' : Prop :=
  ∀ n, ∀ (_ : ev n), ev (n + 2).
```

Or, equivalently, we can write it in more familiar notation:

```
Definition ev_plus2'' : Prop :=
  ∀ n, ev n → ev (n + 2).
```

In general, "`P → Q`" is just syntactic sugar for "`∀(_:P), Q`".

Programming with Tactics

If we can build proofs by giving explicit terms rather than executing tactic scripts, you may be wondering whether we can build *programs* using *tactics* rather than explicit terms. Naturally, the answer is yes!

```
Definition add1 : nat → nat.
intro n.
Show Proof.
apply S.
Show Proof.
apply n. Defined.

Print add1.
(* ==>
    add1 = fun n : nat => S n
         : nat -> nat
*)

Compute add1 2.
(* ==> 3 : nat *)
```

Notice that we terminate the `Definition` with a `.` rather than with `:=`

followed by a term. This tells Coq to enter *proof scripting mode* to build an object of type `nat → nat`. Also, we terminate the proof with `Defined` rather than `Qed`; this makes the definition *transparent* so that it can be used in computation like a normally-defined function. (`Qed`-defined objects are opaque during computation.)

This feature is mainly useful for writing functions with dependent types, which we won't explore much further in this book. But it does illustrate the uniformity and orthogonality of the basic ideas in Coq.

Logical Connectives as Inductive Types

Inductive definitions are powerful enough to express most of the connectives and quantifiers we have seen so far. Indeed, only universal quantification (and thus implication) is built into Coq; all the others are defined inductively. We'll see these definitions in this section.

```
Module Props.
```

Conjunction

To prove that `P ∧ Q` holds, we must present evidence for both `P` and `Q`. Thus, it makes sense to define a proof object for `P ∧ Q` as consisting of a pair of two proofs: one for `P` and another one for `Q`. This leads to the following definition.

```
Module And.

Inductive and (P Q : Prop) : Prop :=
| conj : P → Q → and P Q.

End And.
```

Notice the similarity with the definition of the `prod` type, given in chapter Poly; the only difference is that `prod` takes `Type` arguments, whereas `and` takes `Prop` arguments.

```
Print prod.
(* ===>
   Inductive prod (X Y : Type) : Type :=
   | pair : X -> Y -> X * Y. *)
```

This should clarify why `destruct` and `intros` patterns can be used on a conjunctive hypothesis. Case analysis allows us to consider all possible ways in which `P ∧ Q` was proved — here just one (the `conj`

constructor). Similarly, the `split` tactic actually works for any inductively defined proposition with only one constructor. In particular, it works for `and`:

```
Lemma and_comm : ∀ P Q : Prop, P ∧ Q ↔ Q ∧ P.
Proof.
  intros P Q. split.
  - intros [HP HQ]. split.
    + apply HQ.
    + apply HP.
  - intros [HP HQ]. split.
    + apply HQ.
    + apply HP.
Qed.
```

This shows why the inductive definition of `and` can be manipulated by tactics as we've been doing. We can also use it to build proofs directly, using pattern-matching. For instance:

```
Definition and_comm'_aux P Q (H : P ∧ Q) :=
  match H with
  | conj HP HQ ⇒ conj HQ HP
  end.

Definition and_comm' P Q : P ∧ Q ↔ Q ∧ P :=
  conj (and_comm'_aux P Q) (and_comm'_aux Q P).
```

Exercise: 2 stars, optional (conj_fact)

Construct a proof object demonstrating the following proposition.

```
Definition conj_fact : ∀ P Q R, P ∧ Q → Q ∧ R → P ∧
R
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.
```

□

Disjunction

The inductive definition of disjunction uses two constructors, one for each side of the disjunct:

```
Module Or.

Inductive or (P Q : Prop) : Prop :=
| or_introl : P → or P Q
| or_intror : Q → or P Q.

End Or.
```

This declaration explains the behavior of the `destruct` tactic on a disjunctive hypothesis, since the generated subgoals match the shape of the `or_introl` and `or_intror` constructors.

Once again, we can also directly write proof objects for theorems

involving `or`, without resorting to tactics.

Exercise: 2 stars, optional (or_commut'')

Try to write down an explicit proof object for `or_commut` (without using `Print` to peek at the ones we already defined!).

```
Definition or_comm : ∀ P Q, P ∨ Q → Q ∨ P
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.
```

□

Existential Quantification

To give evidence for an existential quantifier, we package a witness `x` together with a proof that `x` satisfies the property `P`:

```
Module Ex.

Inductive ex {A : Type} (P : A → Prop) : Prop :=
| ex_intro : ∀ x : A, P x → ex P.

End Ex.
```

This may benefit from a little unpacking. The core definition is for a type former `ex` that can be used to build propositions of the form `ex P`, where `P` itself is a *function* from witness values in the type `A` to propositions. The `ex_intro` constructor then offers a way of constructing evidence for `ex P`, given a witness `x` and a proof of `P x`.

The more familiar form `∃x, P x` desugars to an expression involving `ex`:

```
Check ex (fun n ⇒ ev n).
(* ===> exists n : nat, ev n
        : Prop *)
```

Here's how to define an explicit proof object involving `ex`:

```
Definition some_nat_is_even : ∃ n, ev n :=
  ex_intro ev 4 (ev_SS 2 (ev_SS 0 ev_0)).
```

Exercise: 2 stars, optional (ex_ev_Sn)

Complete the definition of the following proof object:

```
Definition ex_ev_Sn : ex (fun n ⇒ ev (S n))
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.
```

□

`True` and `False`

The inductive definition of the `True` proposition is simple:

```
Inductive True : Prop :=
  | I : True.
```

It has one constructor (so every proof of `True` is the same, so being given a proof of `True` is not informative.)

`False` is equally simple — indeed, so simple it may look syntactically wrong at first glance!

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

That is, `False` is an inductive type with *no* constructors — i.e., no way to build evidence for it.

```
End Props.
```

Equality

Even Coq's equality relation is not built in. It has the following inductive definition. (Actually, the definition in the standard library is a small variant of this, which gives an induction principle that is slightly easier to use.)

```
Module MyEquality.

Inductive eq {X:Type} : X → X → Prop :=
| eq_refl : ∀ x, eq x x.

Notation "x = y" := (eq x y)
                    (at level 70, no associativity)
                    : type_scope.
```

The way to think about this definition is that, given a set `X`, it defines a *family* of propositions "`x` is equal to `y`," indexed by pairs of values (`x` and `y`) from `X`. There is just one way of constructing evidence for each member of this family: applying the constructor `eq_refl` to a type `X` and a value `x : X` yields evidence that `x` is equal to `x`.

We can use `eq_refl` to construct evidence that, for example, `2 = 2`. Can we also use it to construct evidence that `1 + 1 = 2`? Yes, we can. Indeed, it is the very same piece of evidence!

The reason is that Coq treats as "the same" any two terms that are *convertible* according to a simple set of computation rules.

These rules, which are similar to those used by `Compute`, include evaluation of function application, inlining of definitions, and simplification of `match`es.

```
Lemma four: 2 + 2 = 1 + 3.
```

```
  Proof.
    apply eq_refl.
  Qed.
```

The `reflexivity` tactic that we have used to prove equalities up to now is essentially just short-hand for `apply eq_refl`.

In tactic-based proofs of equality, the conversion rules are normally hidden in uses of `simpl` (either explicit or implicit in other tactics such as `reflexivity`).

But you can see them directly at work in the following explicit proof objects:

```
  Definition four' : 2 + 2 = 1 + 3 :=
    eq_refl 4.

  Definition singleton : ∀ (X:Type) (x:X), []++[x] =
  x::[] :=
    fun (X:Type) (x:X) ⇒ eq_refl [x].

  End MyEquality.
```

Exercise: 2 stars (equality__leibniz_equality)

The inductive definition of equality implies *Leibniz equality*: what we mean when we say "x and y are equal" is that every property on P that is true of x is also true of y.

```
  Lemma equality__leibniz_equality : ∀ (X : Type) (x
  y: X),
    x = y → ∀ P:X→Prop, P x → P y.
  Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 5 stars, optional (leibniz_equality__equality)

Show that, in fact, the inductive definition of equality is *equivalent* to Leibniz equality:

```
  Lemma leibniz_equality__equality : ∀ (X : Type) (x
  y: X),
    (∀ P:X→Prop, P x → P y) → x = y.
  Proof.
  (* FILL IN HERE *) Admitted.
```

□

Inversion, Again

We've seen `inversion` used with both equality hypotheses and hypotheses about inductively defined propositions. Now that we've seen that these are actually the same thing, we're in a position to take

a closer look at how `inversion` behaves.

In general, the `inversion` tactic...

- takes a hypothesis `H` whose type `P` is inductively defined, and
- for each constructor `C` in `P`'s definition,
    - generates a new subgoal in which we assume `H` was built with `C`,
    - adds the arguments (premises) of `C` to the context of the subgoal as extra hypotheses,
    - matches the conclusion (result type) of `C` against the current goal and calculates a set of equalities that must hold in order for `C` to be applicable,
    - adds these equalities to the context (and, for convenience, rewrites them in the goal), and
    - if the equalities are not satisfiable (e.g., they involve things like `S n = O`), immediately solves the subgoal.

Example: If we invert a hypothesis built with `or`, there are two constructors, so two subgoals get generated. The conclusion (result type) of the constructor (`P ∨ Q`) doesn't place any restrictions on the form of `P` or `Q`, so we don't get any extra equalities in the context of the subgoal.

Example: If we invert a hypothesis built with `and`, there is only one constructor, so only one subgoal gets generated. Again, the conclusion (result type) of the constructor (`P ∧ Q`) doesn't place any restrictions on the form of `P` or `Q`, so we don't get any extra equalities in the context of the subgoal. The constructor does have two arguments, though, and these can be seen in the context in the subgoal.

Example: If we invert a hypothesis built with `eq`, there is again only one constructor, so only one subgoal gets generated. Now, though, the form of the `refl_equal` constructor does give us some extra information: it tells us that the two arguments to `eq` must be the same! The `inversion` tactic adds this fact to the context.