

# 2 *Basics*

Our laboratory for this course is the Coq proof assistant. Coq can be seen as a combination of two things:

1. a simple and slightly idiosyncratic (but, in its way, extremely expressive) *programming language*, and
2. a set of tools for stating *logical assertions* (including assertions about the behavior of programs) and assembling evidence of their truth.

We will be investigating both aspects together.

## 2.1 Enumerated Types

In Coq's programming language, almost nothing is built in—not even booleans or numbers! Instead, it provides powerful tools for defining new types of data and functions that process and transform them.

Let's start with a very simple example. The following definition tells Coq that we are defining a new set of data values. The set is called `day` and its members are `monday`, `tuesday`, etc. The lines of the definition can be read "monday is a day, tuesday is a day, etc."

```
Inductive day : Set :=
  | monday : day
  | tuesday : day
  | wednesday : day
  | thursday : day
  | friday : day
  | saturday : day
  | sunday : day.
```

Having defined this set, we can write functions that operate on its members.

```

Definition next_weekday (d:day) : day :=
  match d with
  | monday => tuesday
  | tuesday => wednesday
  | wednesday => thursday
  | thursday => friday
  | friday => monday
  | saturday => monday
  | sunday => monday
end.

```

One thing to note is that the argument and return types of this function are explicitly declared. Like most functional programming languages, Coq can often work out these types even if they are not given explicitly, but we'll always include them to make reading easier.

Having defined a function, we might like to check how it works on some examples. There are actually three different ways to do this in Coq.

1. We can use the command `Eval simpl` to evaluate a compound expression involving `next_weekday`. For example, if we give Coq the following input

```
Eval simpl in (next_weekday (next_weekday saturday)).
```

it will print the simplified result and its type:

```

► = tuesday
   : day

```

The keyword `simpl` (for “simplify”) tells Coq precisely how to evaluate the expression we give it. For the moment, `simpl` is the only one we'll need; later on we'll see some alternatives that are sometimes useful.

If you have a computer handy, now would be an excellent moment to fire up the Coq interpreter under your favorite IDE—either CoqIde or Proof General—and try this for yourself. Load the file `Basics.v` from the book's accompanying Coq sources, find the above example a little ways from the top, send it to Coq, and observe the result.

2. We can record what we *expect* the result to be in the form of a Coq Example:

```

Example test_next_weekday:
  (next_weekday (next_weekday saturday)) = tuesday.

```

This declaration does two things: It makes an assertion (that the second weekday after `saturday` is `tuesday`), and it gives the assertion a name that can be used to refer to it later.

Having made the assertion, we can also ask Coq to verify it, like this:

```
Proof. reflexivity. □
```

The details are not important for now (we'll come back to them in a few chapters), but essentially this can be read "The assertion we've just made can be proved by observing that both sides of the equality are the same after simplification." The symbol  $\square$  is written `Qed.` in ascii `.v` files

3. Third, we can ask Coq to "extract," from a `Definition`, a program in some other, more conventional, programming language (OCaml, Scheme, or Haskell) with a high-performance compiler.

This facility is very interesting, since it gives us a way to construct *fully certified* programs in mainstream languages. This is actually one of the main uses for which Coq was developed. We'll have more to say about this a little later on.

## 2.2 Booleans

Similarly, we can define the type `bool` of booleans, with constants `true` and `false`.

```
Inductive bool : Set :=
  | true : bool
  | false : bool.
```

Although we are rolling our own booleans here for the sake of building up everything from scratch, Coq does, of course, provide a default implementation of the booleans in its standard library, together with a multitude of useful functions and lemmas. (Take a look at `Coq.Init.Datatypes` in the Coq library documentation if you're interested.) Whenever possible, we'll name our own definitions and theorems so that they exactly coincide with the ones in the standard library.

Functions over booleans can be defined in the same way as above:

```
Definition negb (b:bool) :=
  match b with
  | true => false
  | false => true
```

```

end.
Definition ifb (b1 b2 b3:bool) : bool :=
  match b1 with
  | true => b2
  | false => b3
  end.
Definition andb (b1:bool) (b2:bool) : bool := ifb b1 b2 false.
Definition orb (b1:bool) (b2:bool) : bool := ifb b1 true b2.

```

The last three illustrate the syntax for multi-argument function definitions.

The following four “unit tests” constitute a complete specification—a truth table—for the `orb` function:

```

Example test_orb1: (orb true false) = true.
Example test_orb2: (orb false false) = false.
Example test_orb3: (orb false true) = true.
Example test_orb4: (orb true true) = true.

```

The proofs of these properties are precisely the same as what we saw above. From now on, proofs will generally be omitted, unless they are particularly relevant to the discussion. They can always be found in full in the accompanying Coq sources.

- 2.2.1 EXERCISE [★]: In the Coq source file `Basics.v`, you will find a comment containing incomplete implementations of two more boolean functions, `nandb` and `and3b`. Uncomment and finish them, making sure that Coq can verify the provided unit tests.

The `Check` command causes Coq to print the type of an expression. For example, when presented with

```
Check (negb true).
```

Coq prints

```

► negb true
   : bool

```

Functions like `negb` itself are also data values, just like `true` and `false`. Their types are called *function types*.

```
Check negb.
```

```

► negb
   : bool → bool

```

The type of `negb` is pronounced “`bool` arrow `bool`” and can be read, “Given an input of type `bool`, this function produces an output of type `bool`.” Similarly, the type of `andb`, written `bool`→`bool`→`bool`, can be read, “Given two inputs, both of type `bool`, this function produces an output of type `bool`.”

## 2.3 Numbers

The types we have defined so far are examples of “enumerated types”: their definitions explicitly enumerate a finite collection of elements. A more interesting way of defining a type is to give a collection of *inductive rules* describing its elements. For example, we can define the natural numbers as follows:

```
Inductive nat : Set :=
  | O : nat
  | S : nat → nat.
```

The clauses of this definition can be read:

1. `O` is a natural number (note that this is the letter “`O`,” not the numeral “`0`”).
2. `S` is a *constructor* that takes a natural number and yields another one—that is, if `n` is a natural number, then `S n` is too.

We can write simple functions that pattern match on natural numbers just as we did above:

```
Definition pred (n : nat) : nat :=
  match n with
  | O => O
  | S n' => n'
  end.
```

```
Definition minustwo (n : nat) : nat :=
  match n with
  | O => O
  | S O => O
  | S (S n') => n'
  end.
```

Because natural numbers are such a pervasive form of data, Coq provides a tiny bit of special built-in magic for parsing and printing them: ordinary arabic numerals can be used as an alternative to the “unary” notation defined by the constructors `S` and `O`. Coq prints numbers in arabic form by default,

```
Check (S (S (S (S O)))).
```

```
► 4
   : nat
```

and expands arabic numerals in its input into appropriate sequences of applications of `S` to `O`:

```
Eval simpl in (minustwo 4).
```

```
► = 2
   : nat
```

The constructor `S` has the same type as functions like `minustwo` and `pred`: both

```
Check minustwo.
Check S.
```

yield

```
► ... : nat → nat
```

signifying that these are things that can be applied to a number to yield a number. However, there is a fundamental difference: functions like `pred` and `minustwo` come with *computation rules*—e.g., the definition of `pred` says that `pred n` can be simplified to `match n with | 0 => 0 | S m' => m'` end—while `S` has no such behavior attached. Although it is a function in the sense that it can be applied to an argument, it does not “do” anything at all!

What’s going on here is that every inductively defined set (`weekday`, `nat`, `bool`, and many others we’ll see below) is actually a set of *expressions*. The definition of `nat` says how expressions in the set `nat` can be constructed:

- the expression `O` belongs to the set `nat`;
- if `n` is an expression belonging to the set `nat`, then `S n` is also an expression belonging to the set `nat`; and
- expressions formed in these two ways are the only ones belonging to the set `nat`.

These three conditions are the precise force of the `Inductive` declaration. They imply that the expression `O`, the expression `S O`, the expression `S (S O)`, the expression `S (S (S O))`, and so on all belong to the set `nat`, while other expressions like `true` and `S (S false)` do not.

For most function definitions over numbers, pure pattern matching is not enough: we need recursion. For example, to check that a number `n` is even, we may need to recursively check whether `n-2` is even. To write such functions, we use the keyword `Fixpoint`.

```

Fixpoint evenb (n:nat) {struct n} : bool :=
  match n with
  | 0      => true
  | S 0    => false
  | S (S n') => evenb n'
  end.

```

The most important thing to note about this definition is the annotation `{struct n}` on the first line. This instructs Coq to check that we are performing a “structural recursion” over the argument `n`—i.e., that we make recursive calls only on strictly smaller values of `n`. This implies that all calls to `evenb` will eventually terminate.

We can define `oddb` by a similar `Fixpoint` declaration, but here is a simpler definition that will be easier to work with later:

```

Definition oddb (n:nat) : bool := negb (evenb n).

```

Naturally, we can also define multi-argument functions by recursion.

```

Fixpoint plus (n : nat) (m : nat) {struct n} : nat :=
  match n with
  | 0 => m
  | S n' => S (plus n' m)
  end.

```

Adding three to two now gives us five, as we’d expect.

```

Eval simpl in (plus (S (S (S O))) (S (S O))).

► = 5
   : nat

```

The simplification that Coq performs to reach this conclusion can be visualized as follows:

```

      plus (S (S (S O))) (S (S O))
= S (plus (S (S O)) (S (S O)))   by the second clause of the match
= S (S (plus (S O) (S (S O))))   by the second clause of the match
= S (S (S (plus O (S (S O))))))  by the second clause of the match
= S (S (S (S (S O))))           by the first clause of the match.

```

As a notational convenience, if two or more arguments have the same type, they can be written together. In the following definition, `(n m : nat)` means just the same as if we had written `(n : nat) (m : nat)`.

```

Fixpoint mult (n m : nat) {struct n} : nat :=
  match n with
  | 0 => 0
  | S n' => plus m (mult n' m)
  end.

```

Other arithmetic functions like `minus` and `exp` can be defined similarly (see `Basics.v`).

- 2.3.1 EXERCISE [★]: Recall that the factorial function is defined like this in conventional mathematical notation:

$$\begin{aligned}
 \mathit{factorial}(0) &= 1 \\
 \mathit{factorial}(n) &= n * (\mathit{factorial}(n - 1)) \quad \text{if } n > 0
 \end{aligned}$$

Translate this into Coq's notation. (An incomplete definition can be found in a comment in `Basics.v`.)

When we say that Coq comes with nothing built-in, we really mean it: even equality testing for numbers is a user-defined operation!

```

Fixpoint beq_nat (n m : nat) {struct n} : bool :=
  match n with
  | 0 => match m with
        | 0 => true
        | S m' => false
        end
  | S n' => match m with
            | 0 => false
            | S m' => beq_nat n' m'
            end
  end.

```

- 2.3.2 EXERCISE [★]: Complete the definition (in `Basics.v`) of the comparison function `blt_nat`.

## 2.4 Proof by Simplification

Now that we've defined a few datatypes and functions, let's turn to the question of how to state and prove properties of their behavior.

Actually, in a sense, we've already started doing this: each `Example` in the previous sections makes a precise claim about the behavior of some function on some particular inputs. The proofs of these claims were always the same:

use the function’s definition to simplify the expressions on both sides of the `=` and notice that they become identical.

The same sort of “proof by simplification” can be used to prove more interesting properties as well. For example, the fact that `0` is a “neutral element” for `plus` on the left can be proved just by observing that `plus 0 n` reduces to `n` no matter what `n` is, since the definition of `plus` is recursive in its first argument.

```
Theorem plus_0_l : forall n:nat, plus 0 n = n.
```

```
Proof.
```

```
  simpl. reflexivity. □
```

The form of this theorem and proof are almost exactly the same as the examples above: the only differences are that we’ve added the quantifier `forall n:nat` and that we’ve used the keyword `Theorem` instead of `Example`. Indeed, the latter difference is purely a matter of style; the keywords `Example` and `Theorem` (and a few others, including `Lemma`, `Fact`, and `Remark`) mean exactly the same thing to `Coq`.

The keywords `simpl` and `reflexivity` are examples of *tactics*. A tactic is a command that is used between `Proof` and `Qed` to tell `Coq` how it should check the correctness of some claim we are making. We will see several more tactics in the rest of this lecture, and yet more in future lectures.

Actually, `reflexivity` implicitly simplifies both sides of the equality before testing to see if they are the same. So we can omit the explicit use of `simpl` just before any use of `reflexivity`, and we’ll generally do so from now on.

Quite a few simple facts about addition and multiplication can be proved in this way. Here are a couple more examples:

```
Theorem plus_1_l : forall n:nat, plus 1 n = S n.
```

```
Theorem mult_0_l : forall n:nat, mult 0 n = 0.
```

(The `_l` suffix in the names of these theorems is pronounced “on the left.”)

## 2.5 The intros Tactic

Aside from unit tests, which apply functions to particular arguments, most of the properties we will be interested in proving about programs will begin with some quantifiers (e.g., “for all numbers `n`, ...”) and/or hypothesis (“assuming `m=n`, ...”). In such situations, we will need to be able to reason by *assuming the hypothesis*—i.e., we start by saying “OK, suppose `n` is some arbitrary number,” or “OK, suppose `m=n`.”

The `intros` tactic permits us to do this, by moving one or more quantifiers or hypotheses from the goal to a *context* of current assumptions.

For example, here is a similar theorem with a slightly different proof.

```
Theorem plus_1_1 : forall n:nat, plus 1 n = S n.
```

```
Proof.
```

```
  intros n. reflexivity. □
```

Step through this proof in Coq and notice how the goal and context change at each point.

## 2.6 Proof by Rewriting

Here is a slightly more interesting theorem:

```
Theorem plus_id_example : forall n m:nat,
```

```
  n = m → plus n n = plus m m.
```

Instead of making a completely universal claim about all numbers  $n$  and  $m$ , this theorem talks about a more specialized property that only holds when  $n = m$ . The arrow symbol, written  $\rightarrow$  in typeset code and `->` in ascii `.v` files, is pronounced *implies*.

Since  $n$  and  $m$  are arbitrary numbers, we can't just use simplification to prove this theorem. Instead, we prove it by observing that, if we are assuming  $n = m$ , then we can replace  $n$  with  $m$  in the goal statement and obtain an equality with the same expression on both sides. The tactic that tells Coq to perform this replacement is called `rewrite`.

```
Proof.
```

```
  intros n m.      (* move both quantifiers into the context *)
```

```
  intros H.        (* move the hypothesis into the context *)
```

```
  rewrite → H.    (* Rewrite the goal using the hypothesis *)
```

```
  reflexivity.    □
```

The first line of the proof moves the universally quantified variables  $n$  and  $m$  into the context. The second moves the hypothesis  $n = m$  into the context and gives it the name `H`. The third tells Coq to rewrite the current goal (`plus n n = plus m m`) by replacing the left side of the equality hypothesis `H` with the right side.

(The arrow symbol in the `rewrite` has nothing to do with implication: it tells Coq to apply the rewrite from left to right. To rewrite from right to left, you can use `rewrite ←`. Try making this change in the above proof and see what difference it makes in Coq's behavior.)

- 2.6.1 EXERCISE [★]: Prove `plus_id_exercise`, found in `Basics.v`. Notice that in the file, the current proof consists of the keyword `Admitted`. The `Admitted` command tells Coq that we want to give up trying to prove this theorem and just accept it as a given. This can be useful for developing longer proofs, since we can state subsidiary facts that we believe will be useful for making some larger argument, use `Admitted` to accept them on faith for the moment, and continue thinking about the larger argument until we are sure it makes sense; then we can go back and fill in the proofs we skipped. Be careful, though: every time you say `Admitted` you are leaving a door open for total nonsense to enter Coq's nice, rigorous, formally checked world!

We can also use the `rewrite` tactic with a previously proved theorem instead of a hypothesis from the context.

```
Theorem mult_0_plus : forall n m : nat,
  mult (plus 0 n) m = mult n m.
Proof.
  intros n m.
  rewrite → plus_0_1.
  reflexivity. □
```

- 2.6.2 EXERCISE [★]: Prove the theorem `mult_1_plus` in `Basics.v`.

## 2.7 Case Analysis

Of course, not everything can be proved by simple calculation: In general, unknown, hypothetical values (arbitrary numbers, booleans, lists, etc.) can show up in the "head position" of functions that we want to reason about, blocking simplification. For example, if we try to prove the following fact using the `simpl` tactic as above, we get stuck.

```
Theorem plus_1_neq_0_firsttry : forall n,
  beq_nat (plus n 1) 0 = false.
```

The reason for this is that the definitions of both `beq_nat` and `plus` begin by performing a `match` on their first argument. But here, the first argument to `plus` is the unknown number `n` and the argument to `beq_nat` is the compound expression `plus n 1`; neither can be simplified.

What we need is to be able to consider the possible forms of `n` separately. If `n` is `0`, then we can calculate the final result of `beq_nat (plus n 1) 0` and check that it is, indeed, `false`. And if `n = S n'` for some `n'`, then, although we don't know exactly what number `plus n 1` yields, we can calculate that,

at least, it will begin with one  $S$ , and this is enough to calculate that, again, `beq_nat (plus n 1) 0` will yield `false`.

The tactic that tells Coq to consider the cases where  $n = 0$  and where  $n = S\ n'$  separately is called `destruct`.

```
Theorem plus_1_neq_0 : forall n,
  beq_nat (plus n 1) 0 = false.
```

Proof.

```
  intros n. destruct n as [| n'].
    reflexivity.
    reflexivity.  □
```

The `destruct` generates *two* subgoals, which we must then prove, separately, in order to get Coq to accept the theorem as proved. (No special command is needed for moving from one subgoal to the other. When the first subgoal has been proved, it just disappears and we are left with the other “in focus.”) In this case, each of the subgoals is easily proved by a single use of `reflexivity`.

The annotation “`as [| n']`” is called an *intro pattern*. It tells Coq what variable names to introduce in each subgoal. In general, what goes between the square brackets is a *list* of lists of names, separated by `|`. Here, the first component is empty, since the `0` constructor is nullary (it doesn’t carry any data). The second component gives a single name,  $n'$ , since  $S$  is a unary constructor.

The `destruct` tactic can be used with any inductively defined datatype. For example, we use it here to prove that boolean negation is “involutive”—i.e., that negation is its own inverse.

```
Theorem negb_involutive : forall b : bool,
  negb (negb b) = b.
```

Proof.

```
  intros b. destruct b.
    reflexivity.
    reflexivity.  □
```

Note that the `destruct` here has no `as` clause because none of the subcases of the `destruct` need to bind any variables, so there is no need to specify any names. (We could also have written “`as []`”.) In fact, we can omit the `as` clause from *any* `destruct` and Coq will fill in variable names automatically. However, although this is convenient, it is arguably bad style, since Coq often makes confusing choices of names when left to its own devices.

2.7.1 EXERCISE [★]: Prove the theorem `zero_nbeq_plus_1` in `Basics.v`.

2.7.2 EXERCISE [★]: Recall the definition of `plus`:

```

Fixpoint plus (n : nat) (m : nat) {struct n} : nat :=
  match n with
  | 0 => m
  | S n' => S (plus n' m)
  end.

```

What will Coq print in response to this query?

```
Eval simpl in (forall n, plus n 0 = n).
```

What will Coq print in response to this query?

```
Eval simpl in (forall n, plus 0 n = n).
```

Briefly explain the difference.

## 2.8 Naming Cases

The fact that there is no explicit command for moving from one branch of a case analysis to the next can make proof scripts rather hard to read. In larger proofs, with nested case analyses, it can even become hard to stay oriented when you're sitting with Coq and stepping through the proof. (Imagine trying to remember that the first five subgoals belong to the inner case analysis and the remaining seven are the cases that are left of the outer one...) Disciplined use of indentation and comments can help, but a better way is to use the `Case` tactic.

```

Theorem andb_true_l : forall b c,
  andb b c = true → b = true.

```

Proof.

```

  intros b c H.
  destruct b.
  Case "b = true".
    reflexivity.
  Case "b = false".
    rewrite ← H. reflexivity. □

```

`Case` does something very trivial: It simply adds a string that we choose (tagged with the identifier “Case”) to the context for the current goal. When subgoals are generated, this string is carried over into their contexts. When the last of these subgoals is finally proved and the next top-level goal (a sibling of the current one) becomes active, this string will no longer appear in the context and we will be able to see that the case where we introduced it is complete. Also, as a sanity check, if we try to execute a new `Case` tactic

while the string left by the previous one is still in the context, we get a nice clear error message.

For nested case analyses (i.e., when we want to use a `destruct` to solve a goal that has itself been generated by a `destruct`), there is an `SCase` (“sub-case”) tactic. For deeper nesting there are `SSCase`, `SSSCase`, etc.

`Case` and its friends are not actually built-in facilities of Coq: they can be programmed using “`Ltac`,” Coq’s language for writing user-defined tactics. You can see the actual definitions at this point in the `Basics.v` file, if you’re curious, but there’s no need to understand any of the details of how they work.

- 2.8.1 EXERCISE [★]: Prove the theorem `andb_true_r` in `Basics.v`, using `Case` (or `SCase`) to mark the branches of each `destruct`.

There are no hard and fast rules for how proofs should be formatted in Coq—in particular, where lines should be broken and how sections of the proof should be indented to indicate their nested structure. However, if the places where multiple subgoals are generated are marked with explicit `Case` tactics placed at the beginning of lines, then the proof will be readable almost no matter what choices are made about other aspects of layout.

This is a good place to mention one other piece of (possibly obvious) advice about line lengths. Beginning Coq users sometimes tend to the extremes, either writing each tactic on its own line or entire proofs on one line. Good style lies somewhere in the middle. In particular, one convention (not just for Coq proofs, but for all programming) is to limit yourself to 80 character lines. Lines longer than this are hard to read and can be inconvenient to display and print. Many editors have features that help enforce this. Of course, there is no need to make every line this long, but it’s a simple and reasonable upper bound.

## 2.9 Induction

We proved above that `0` is a neutral element for `plus` on the left using a simple partial evaluation argument. The fact that it is also a neutral element on the *right*...

```
Theorem plus_0_r : forall n:nat, plus n 0 = n.
```

... cannot be proved in the same simple way. Just applying `reflexivity` doesn’t work: the `n` in `plus 0 n` is an arbitrary unknown number, so the `match` in the definition of `plus` can’t be simplified. And reasoning by cases using `destruct n` doesn’t get us much further: the branch of the case analysis where we assume `n = 0` goes through, but in the branch where `n = S n'`

for some  $n'$  we get stuck in exactly the same way. We could use `destruct n'` to get one step further, but since  $n$  can be arbitrarily large, if we continue this way we'll never be done.

To prove such facts—indeed, to prove most interesting facts about numbers, lists, and other inductively defined sets—we need a more powerful reasoning principle: *induction*.

Recall (from high school) the principle of induction over natural numbers: If  $P(n)$  is some proposition involving a natural number  $n$  and we want to show that  $P$  holds for *all* numbers  $n$ , we can reason like this:

1. show that  $P(0)$  holds;
2. show that, for any  $n'$ , if  $P(n')$  holds, then so does  $P(S n')$ ;
3. conclude that  $P(n)$  holds for all  $n$ .

In Coq, the steps are the same but the order is backwards: we begin with the goal of proving  $P(n)$  for all  $n$  and break it down (by applying the `induction` tactic) into two separate subgoals: first showing  $P(0)$  and then showing  $P(n') \rightarrow P(S n')$ . Here's how this works for the theorem we are trying to prove at the moment:

Proof.

```
intros n. induction n as [| n'].
Case "n = 0".      reflexivity.
Case "n = S n'".  simpl. rewrite → IHn'. reflexivity.  □
```

Like `destruct`, the `induction` tactic takes an `as...` clause that specifies the names of the variables to be introduced in the subgoals. In the first branch,  $n$  is replaced by  $0$  and the goal becomes `plus 0 0 = 0`, which follows by simplification. In the second,  $n$  is replaced by  $S n'$  and the assumption `plus n' 0 = n'` is added to the context (with the name `IHn'`, i.e., the Induction Hypothesis for  $n'$ ). The goal in this case becomes `plus (S n') 0 = S n'`, which simplifies to `S (plus n' 0) = S n'`, which in turn follows from the induction hypothesis.

### 2.9.1 EXERCISE [★]: Use induction to prove the following theorems:

```
Theorem mult_0_r : forall n:nat,
  mult n 0 = 0.
```

```
Theorem plus_assoc : forall n m p : nat,
  plus n (plus m p) = plus (plus n m) p.
```

```
Theorem plus_n_Sm : forall n m : nat,
  S (plus n m) = plus n (S m).
```

```
Theorem plus_comm : forall n m : nat,
  plus n m = plus m n.
```

## 2.10 Formal vs. Informal Proofs

The question of what, exactly, constitutes a *proof* of a mathematical claim has challenged philosophers throughout the ages. A rough and ready definition, though, could be this: a proof of a mathematical proposition  $P$  is a written (or, sometimes, spoken) text that instills in the reader or hearer the certainty that  $P$  is true. That is, a proof is an act of communication.

Now, acts of communication may involve different sorts of readers. On one hand, the “reader” can be a program like Coq, in which case the “belief” that is instilled is a simple mechanical check that  $P$  can be derived from a certain set of formal logical rules, and the proof is a recipe that guides the program in performing this check. Such recipes are called *formal proofs*.

Alternatively, the reader can be a human being, in which case the proof will be written in English or some other natural language, thus necessarily *informal*. Here, the criteria for success are less clearly specified. A “good” proof is one that makes the reader believe  $P$ . But the same proof may be read by many different readers, some of whom may be convinced by a particular way of phrasing the argument, while others may not be. One reader may be particularly pedantic, inexperienced, or just plain thick-headed; the only way to convince them will be to make the argument in painstaking detail. But another reader, more familiar in the area, may find all this detail so overwhelming that they lose the overall thread. All they want is to be told the main ideas, because it is easier to fill in the details for themselves. Ultimately, there is no universal standard, because there is no single way of writing an informal proof that is guaranteed to convince every conceivable reader. In practice, however, mathematicians have developed a rich set of conventions and idioms for writing about complex mathematical objects that, within a certain community, make communication fairly reliable. The conventions of this stylized form of communication give a fairly clear standard for judging proofs good or bad.

Because we will be using Coq in this course, we will be working heavily with formal proofs. But this doesn’t mean we can ignore the informal ones! Formal proofs are useful in many ways, but they are *not* very efficient ways of communicating ideas between human beings.

For example, consider this statement:

```
Theorem plus_assoc : forall n m p : nat,
  plus n (plus m p) = plus (plus n m) p.
```

Coq is perfectly happy with this as a proof:

Proof.

```
intros n m p. induction n as [| n'].
Case "n = 0".
  reflexivity.
Case "n = S n'".
  simpl. rewrite → IHn'. reflexivity.   □
```

For a human, however, it is difficult to make much sense of this. If you're used to Coq you can probably step through the tactics one after the other in your mind and imagine the state of the context and goal stack at each point, but if the proof were even a little bit more complicated this would be next to impossible. Instead, a mathematician would write it as in Figure 2-1. The overall form of the proof is basically similar. (This is no accident, of course: Coq has been designed so that its `induction` tactic generates the same sub-goals, in the same order, as the bullet points that a mathematician would write.) But there are significant differences of detail: the formal proof is much more explicit in some ways (e.g., the use of `reflexivity`) but much less explicit in others; in particular, the “proof state” at any given point in the Coq proof is completely implicit, whereas the informal proof reminds the reader several times where things stand.

- 2.10.1 EXERCISE [★★]: Write an informal proof corresponding to your formal proof of `plus_comm` (from exercise 2.9.1). Use the informal proof of `plus_assoc` as a model. Don't just paraphrase the Coq tactics into English!
- 2.10.2 EXERCISE [★★]: In `Basics.v`, you will find a careful informal proof of a theorem called `beq_nat_refl`. Write a corresponding Coq proof.

## 2.11 Proofs Within Proofs

In Coq, as in informal mathematics, large proofs are very often broken into sequence of theorems, with later proofs referring to earlier theorems. Occasionally, however, a proof will need some miscellaneous fact that is too trivial (and of too little general interest) to bother giving it its own top-level name. In such cases, it is convenient to be able to simply state and prove the needed “sub-theorem” right at the point where it is used. The `assert` tactic allows

**Proof:** By induction on  $n$ .

- First, suppose  $n = 0$ . We must show

$$\text{plus } 0 \text{ (plus } m \text{ } p) = \text{plus (plus } 0 \text{ } m) \text{ } p.$$

This follows directly from the definition of `plus`.

- Next, suppose  $n = S \ n'$ , with

$$\text{plus } n' \text{ (plus } m \text{ } p) = \text{plus (plus } n' \text{ } m) \text{ } p.$$

We must show

$$\text{plus (S } n') \text{ (plus } m \text{ } p) = \text{plus (plus (S } n') \text{ } m) \text{ } p.$$

By the definition of `plus`, this follows from

$$S \ (\text{plus } n' \text{ (plus } m \text{ } p)) = S \ (\text{plus (plus } n' \text{ } m) \text{ } p),$$

which is immediate from the induction hypothesis.

**Figure 2-1** A mathematician's inductive proof

us to do this. For example, our earlier proof of the `mult_0_plus` theorem referred to a previous theorem named `plus_0_1`. We can also use `assert` to state and prove `plus_0_1` in-line:

```
Proof.
  intros n m.
  assert (plus 0 n = n).
    Case "Proof of assertion". reflexivity.
  rewrite → H.
  reflexivity. □
```

The `assert` tactic introduces two sub-goals. The first is the assertion itself. (We mark this with a `Case`, both for readability and so that, when using Coq interactively, we can see when we're finished proving the assertion by observing when the "Proof of assertion" string disappears from the context.) The second goal is the same as the one at the point where we invoke `assert`, except that, in the context, we have an assumption, called `H`, that `plus 0 n = n`. That is, `assert` generates one subgoal where we must prove

the asserted fact and a second subgoal where we can use the asserted fact to make progress on whatever we were trying to prove in the first place.

Actually, `assert` will turn out to be handy in many sorts of situations. For example, suppose we want to prove that

$$\text{plus } (\text{plus } n\ m) \ (\text{plus } p\ q) = \text{plus } (\text{plus } m\ n) \ (\text{plus } p\ q).$$

The only difference between the two sides of the `=` is that the arguments `m` and `n` to the first inner `plus` are swapped, so it seems we should be able to use the commutativity of addition (`plus_comm`) to rewrite one into the other. However, the `rewrite` tactic is a little stupid about where it applies the rewrite. There are three uses of `plus` here, and it turns out that doing `rewrite → plus_comm` will affect only the *outer* one. (Try it!) To get `plus_comm` to apply at the point where we want it, we can introduce a local lemma stating that `plus n m = plus m n` (for the particular `m` and `n` that we are talking about here), prove this lemma using `plus_comm`, and then use this lemma to do the desired rewrite.

Proof.

```
intros n m p q.
assert (plus n m = plus m n).
  Case "Proof of assertion".
  rewrite → plus_comm. reflexivity.
rewrite → H. reflexivity. □
```

- 2.11.1 EXERCISE [★★]: Use `assert` to help prove the theorem `plus_swap` in `Basics.v`. Then use `plus_swap` to prove `mult_comm`.
- 2.11.2 EXERCISE [★★]: The theorem `evenb_n__oddb_Sn` in `Basics.v` states that, if `n` is even, then its successor is odd. Prove it.
- 2.11.3 EXERCISE [★★]: Take a piece of paper. Find the section marked “More exercises” in `Basics.v`. For each of the theorems there, first *think* about whether (a) it can be proved using only simplification and rewriting, (b) it also requires case analysis (`destruct`), or (c) it also requires induction. Write down your prediction. Then fill in the proof. (There is no need to turn in your piece of paper; this is just to encourage you to think before hacking!)