4   Inductive Predicates

The Curry-Howard correspondence [10, 14] states a formal connection between functional programs and mathematical proofs. Witness the close similarity between the types **unit** and **True** from the Coq standard library.

Print **unit**.

> Inductive **unit** : Set := tt : **unit**

Print **True**.

> Inductive **True** : Prop := I : **True**

Recall that **unit** is the type with only one value, and **True** is the proposition that always holds. Despite this superficial difference between the two concepts, in both cases we can use the same inductive definition mechanism. The connection goes further than this. We arrive at the definition of **True** by replacing **unit** by **True**, tt by I, and Set by Prop. The first two of these differences are superficial changes of names, but the third difference is the crucial one for separating programs from proofs. A term T of type Set is a type of programs, and a term of type T is a program. A term T of type Prop is a logical proposition, and its proofs are of type T. Chapter 12 goes into more detail about the theoretical differences between Prop and Set. For now, we simply follow common intuitions about what a proof is.

The type **unit** has one value, tt. The type **True** has one proof, I. Why distinguish between these two types? Many people who have read about Curry-Howard in an abstract context but who have not put it to use in proof engineering say that the two types in fact *should not* be distinguished. There is a certain aesthetic appeal to this point of view, but I argue that it is best to treat Curry-Howard very loosely in practical proving. There are Coq-specific reasons for preferring the distinction, involving efficient compilation and avoidance of paradoxes

in the presence of classical math, but there is a more general principle that should lead us to avoid conflating programming and proving.

The essence of the argument is roughly this: to an engineer, not all functions of type $A \rightarrow B$ are created equal, but all proofs of a proposition $P \rightarrow Q$ are. This idea is known as *proof irrelevance*, and its formalizations in logics prevent us from distinguishing between alternative proofs of the same proposition. Proof irrelevance is compatible with, but not derivable in, Gallina. Apart from this theoretical concern, I argue that it is most effective to do engineering with Coq by employing different techniques for programs versus proofs. Most of this book is organized around that distinction, describing how to program by applying standard functional programming techniques in the presence of dependent types, and how to prove by writing custom Ltac decision procedures.

With that perspective in mind, this chapter is a kind of mirror image of the last chapter, introducing how to define predicates with inductive definitions. I point out similarities in places, but much of the effective Coq user's bag of tricks is disjoint for predicates versus datatypes. This chapter is also an implicit introduction to dependent types, which are the foundation on which interesting inductive predicates are built, though here we rely on tactics to build dependently typed proof terms. Chapter 6 begins our study of more manual application of dependent types.

4.1   Propositional Logic

Let us begin with a brief tour through the definitions of the connectives for propositional logic. We will work within a Coq section that provides a set of propositional variables. In Coq parlance, these are just variables of type `Prop`.

```
Section Propositional.
  Variables P Q R : Prop.
```

In Coq, the most basic propositional connective is implication, written $\rightarrow$, which we have already used in almost every proof. Rather than being defined inductively, implication is built into Coq as the function type constructor.

We have also seen the definition of **True**. For a demonstration of a lower-level way of establishing proofs of inductive predicates, we turn to this trivial theorem.

```
  Theorem obvious : True.
```

```
    apply I.
Qed.
```

We may always use the `apply` tactic to take a proof step based on applying a particular constructor of the inductive predicate that we are trying to establish. Sometimes there is only one constructor that could possibly apply, in which case a shortcut is available:

```
Theorem obvious′ : True.
    constructor.
Qed.
```

There is also a predicate **False**, which is the Curry-Howard mirror image of **Empty_set** (see Chapter 3).

```
Print False.
```

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

We can conclude anything from **False**, doing case analysis on a proof of **False** in the same way we might do case analysis on, say, a natural number. Since there are no cases to consider, any such case analysis succeeds immediately in proving the goal.

```
Theorem False_imp : False → 2 + 2 = 5.
    destruct 1.
Qed.
```

In a consistent context, we can never build a proof of **False**. In inconsistent contexts that appear in the courses of proofs, it is usually easiest to proceed by demonstrating the inconsistency with an explicit proof of **False**.

```
Theorem arith_neq : 2 + 2 = 5 → 9 + 9 = 835.
    intro.
```

At this point, we have an inconsistent hypothesis $2 + 2 = 5$, so the specific conclusion is not important. We use the `elimtype` tactic (for a full description, see the Coq manual). For our purposes, we only need the variant `elimtype` **False**, which lets us replace any conclusion formula with **False**, because any fact follows from an inconsistent context.

```
    elimtype False.
```

$H : 2 + 2 = 5$
============================
    False

For now, we will leave the details of this proof about arithmetic to *crush*.

```
    crush.
Qed.
```

A related notion to **False** is logical negation.

```
Print not.
```

> not = fun A : Prop $\Rightarrow A \to$ **False**
>   : Prop $\to$ Prop

We see that not is just shorthand for implication of **False**. We can use that fact explicitly in proofs. The syntax $\neg P$ (written with a tilde in ASCII) expands to not P.

```
Theorem arith_neq' : ¬ (2 + 2 = 5).
    unfold not.
```

> ============================
> $2 + 2 = 5 \to$ **False**

```
    crush.
Qed.
```

We also have conjunction (see Chapter 3).

```
Print and.
```

> Inductive **and** $(A : \text{Prop})$ $(B : \text{Prop})$: Prop :=
>   conj : $A \to B \to A \land B$

The reader can check that **and** has a Curry-Howard equivalent called **prod**, the type of pairs. However, it is generally most convenient to reason about conjunction using tactics. An explicit proof of commutativity of **and** illustrates how such tasks are usually done. The operator $\land$ is an infix shorthand for **and**.

```
Theorem and_comm : P ∧ Q → Q ∧ P.
```

We start by case analysis on the proof of $P \land Q$.

```
    destruct 1.
```

> $H : P$
> $H0 : Q$
> ============================

$Q \wedge P$

Every proof of a conjunction provides proofs for both conjuncts, so
we get a single subgoal reflecting that. We can split this subgoal into a
case for each conjunct of $Q \wedge P$.

```
split.
```

2 subgoals

$H : P$
$H0 : Q$
==============================
  Q

subgoal 2 is

  P

In each case, the conclusion is among the hypotheses, so the
`assumption` tactic finishes the process.

```
assumption.
assumption.
Qed.
```

Coq disjunction is called **or** and abbreviated with the infix operator
$\vee$.

```
Print or.
```

Inductive **or** $(A : \mathtt{Prop})$ $(B : \mathtt{Prop})$: $\mathtt{Prop} :=$
  or_introl : $A \rightarrow A \vee B$ | or_intror : $B \rightarrow A \vee B$

There are two ways to prove a disjunction: prove the first disjunct or
prove the second. The Curry-Howard analogue of this is the Coq **sum**
type. We can demonstrate the main tactics here with another proof of
commutativity.

Theorem or_comm : $P \vee Q \rightarrow Q \vee P$.

As in the proof for **and**, we begin with case analysis, though this time
we are met by two cases instead of one.

```
destruct 1.
```

2 subgoals

$H : P$
=============================
$\quad Q \lor P$

```
subgoal 2 is
```

$Q \lor P$

In the first subgoal, we want to prove the disjunction by proving its second disjunct. The `right` tactic telegraphs this intent.

```
    right; assumption.
```

The second subgoal has a symmetric proof.

```
1 subgoal
```

$H : Q$
=============================
$\quad Q \lor P$

```
    left; assumption.
Qed.
```

There is no need to plod manually through all proofs about propositional logic. One of the most basic Coq automation tactics is `tauto`, which is a complete decision procedure for constructive propositional logic. We can use `tauto` to dispatch all the purely propositional theorems we have proved so far.

```
Theorem or_comm' : P ∨ Q → Q ∨ P.
    tauto.
Qed.
```

Sometimes propositional reasoning is important for proving a theorem, but we still need to apply some other insights about, say, arithmetic. The tactic `intuition` is a generalization of `tauto` that proves everything it can using propositional reasoning. When some further facts must be established to finish the proof, `intuition` uses propositional laws to simplify them as far as possible. Consider this example, which uses the list concatenation operator `++` from the standard library:

```
Theorem arith_comm : ∀ ls1 ls2 : list nat,
```

```
    length ls1 = length ls2 ∨ length ls1 + length ls2 = 6
    → length (ls1 ++ ls2) = 6 ∨ length ls1 = length ls2.
    intuition.
```

A lot of the proof structure has been generated by `intuition`, but the final proof depends on a fact about lists. The remaining subgoal hints at what is needed.

$ls1$: **list nat**
$ls2$: **list nat**
$H0$: length $ls1$ + length $ls2$ = 6
==
length $(ls1 ++ ls2) = 6 \lor$ length $ls1 =$ length $ls2$

We need a theorem about lengths of concatenated lists (see Chapter 3 and the standard library).

```
    rewrite app_length.
```

$ls1$: **list nat**
$ls2$: **list nat**
$H0$: length $ls1$ + length $ls2$ = 6
==
length $ls1$ + length $ls2 = 6 \lor$ length $ls1 =$ length $ls2$

Now the subgoal follows by purely propositional reasoning. That is, we could replace length $ls1$ + length $ls2$ = 6 with P and length $ls1$ = length $ls2$ with Q and arrive at a tautology of propositional logic.

```
    tauto.
Qed.
```

The `intuition` tactic is one of the main elements in the implementation of *crush*, so we can get a short automated proof of the theorem.

```
Theorem arith_comm' : ∀ ls1 ls2 : list nat,
    length ls1 = length ls2 ∨ length ls1 + length ls2 = 6
    → length (ls1 ++ ls2) = 6 ∨ length ls1 = length ls2.
    Hint Rewrite app_length.

    crush.
Qed.

End Propositional.
```

Each of the propositional theorems in this section becomes universally quantified over the propositional variables that we used.

4.2   What Does It Mean to Be Constructive?

One potential point of confusion in the presentation so far is the distinction between **bool** and `Prop`. The datatype **bool** is built from two values `true` and `false`, whereas `Prop` is a more primitive type that includes among its members **True** and **False**. Why not collapse these two concepts into one, and why must there be more than two states of mathematical truth, **True** and **False**?

The answer comes from the fact that Coq implements *constructive logic* or *intuitionistic logic*, in contrast to the *classical logic* that we may be more familiar with. In constructive logic, classical tautologies like $\neg \neg P \to P$ and $P \vee \neg P$ do not always hold. In general, we can only prove these tautologies when P is *decidable*, in the sense of computability theory. The Curry-Howard encoding that Coq uses for **or** allows us to extract either a proof of P or a proof of $\neg P$ from any proof of $P \vee \neg P$. Since the proofs are just functional programs that we can run, a general law of the excluded middle would yield a decision procedure for the halting problem, where the instantiations of P would be formulas like "this particular Turing machine halts."

A similar paradoxical situation would result if every proposition evaluated to either **True** or **False**. Evaluation in Coq is decidable, so we would be limiting ourselves to decidable propositions only.

Hence the distinction between **bool** and `Prop`. Programs of type **bool** are computational by construction; we can always run them to determine their results. Many `Prop`s are undecidable, so we can write more expressive formulas with `Prop`s than with **bool**s, but we cannot simply run a `Prop` to determine its truth.

Constructive logic lets us define all the logical connectives in an aesthetically appealing way, with orthogonal inductive definitions. That is, each connective is defined independently using a simple shared mechanism. Constructivity also enables a trick called *program extraction*, where programming tasks are phrased as theorems to be proved. Since the proofs are just functional programs, we can extract executable programs from the final proofs, which we could not do as naturally with classical proofs.

Chapter 6 presents more about Coq's program extraction facility. However, I think it is worth interjecting another warning at this point, following up on the prior warning about taking the Curry-Howard

correspondence too literally. It is possible to write programs by theorem-proving methods in Coq, but hardly anyone does it. It is almost always most useful to maintain the distinction between programs and proofs. If we write a program by proving a theorem, we are likely to run into algorithmic inefficiencies that we introduced into the proof to make it easier to prove. Extracting programs from proofs is mostly limited to theoretical studies.

4.3   First-Order Logic

The $\forall$ connective of first-order logic, used in many earlier examples, is built into Coq. It can be viewed as the dependent function type constructor. In fact, implication and universal quantification are just different syntactic shorthands for the same Coq mechanism. A formula $P \to Q$ is equivalent to $\forall\, x : P,\ Q$, where x does not appear in Q. That is, the real type of the implication says "for every proof of P, there exists a proof of Q."

Existential quantification is defined in the standard library.

```
Print ex.
```

```
Inductive ex (A : Type) (P : A → Prop) : Prop :=
    ex_intro : ∀ x : A, P x → ex P
```

Note that here, as always, each $\forall$ quantifier has the largest possible scope; the type of ex_intro could also be written $\forall\, x : A,\ (P\ x \to \mathbf{ex}\ P)$.

The family **ex** is parameterized by the type A that we quantify over, and by a predicate P over As. We prove an existential by exhibiting some x of type A, along with a proof of $P\ x$. As usual, there are tactics that take care of low-level details most of the time.

Here is an example of a theorem statement with existential quantification. We use the equality operator $=$, which, depending on the settings in which they learned logic, different people will say either is or is not part of first-order logic. For our purposes, it is.

```
Theorem exist1 : ∃ x : nat, x + 1 = 2.
```

We can start this proof with a tactic `exists`, which should not be confused with the formula constructor shorthand of the same name. The reverse E in formulas stands for the ASCII token `exists`.

```
exists 1.
```

The conclusion is replaced with a version using the existential witness that we announced.

```
===============================
```
$$1 + 1 = 2$$

```
reflexivity.
Qed.
```

We can also use tactics to reason about existential hypotheses.

Theorem exist2 : $\forall$ n m : **nat**, $(\exists$ x : **nat,** n + x = $m)$ $\rightarrow$ $n \leq m$.
   We start by case analysis on the proof of the existential fact.

```
destruct 1.
```

n : **nat**
m : **nat**
x : **nat**
H : $n + x = m$
```
===============================
```
$$n \leq m$$

The goal has been replaced by a form with a new free variable x and a new hypothesis that the body of the existential holds with x substituted for the old bound variable. From here, the proof is just about arithmetic and is easy to automate.

   crush.
```
Qed.
```

The tactic `intuition` has a first-order cousin called `firstorder`, which proves many formulas when only first-order reasoning is needed, and it tries to perform first-order simplifications in any case. First-order reasoning is much harder than propositional reasoning, so `firstorder` is much more likely than `intuition` to get stuck and run long enough to be useless.

4.4   Predicates with Implicit Equality

We start our exploration of a more complicated class of predicates with a simple example: an alternative way of characterizing when a natural number is zero.

```
Inductive isZero : nat → Prop :=
| IsZero : isZero 0.
```

```
Theorem isZero_zero : isZero 0.
  constructor.
Qed.
```

We can call **isZero** a *judgment*, in the sense often used in the semantics of programming languages. Judgments are typically defined in the style of *natural deduction*, where we write a number of *inference rules* with premises appearing above a solid line and a conclusion appearing below the line. In this example, the sole constructor IsZero of **isZero** can be thought of as the single inference rule for deducing **isZero**, with nothing above the line and **isZero** 0 below it. The proof of isZero_zero demonstrates how we can apply an inference rule. (Readers not familiar with formal semantics should not worry about not following this paragraph.)

The definition of **isZero** differs in an important way from all the earlier inductive definitions. Instead of writing just Set or Prop after the colon, here we write **nat** → Prop. We saw examples of parameterized types like **list**, but there the parameters appeared with names *before* the colon. Every constructor of a parameterized inductive type must have a range type that uses the same parameter, whereas the form we use here enables us to choose different arguments to the type for different constructors.

For instance, the definition **isZero** makes the predicate provable only when the argument is 0. We can see that the concept of equality is somehow implicit in the inductive definition mechanism. The way this is accomplished is similar to the way that logic variables are used in Prolog, and it is a very powerful mechanism that forms a foundation for formalizing all of mathematics. In fact, though it is natural to think of inductive types as folding in the functionality of equality, in Coq the true situation is reversed, with equality defined as just another inductive type.

```
Print eq.
```

> Inductive **eq** $(A : \texttt{Type})\ (x : A) : A \rightarrow \texttt{Prop} := \texttt{eq\_refl} : x = x$

Behind the scenes, uses of infix $=$ are expanded to instances of **eq**. We see that **eq** has both a parameter x that is fixed and an extra unnamed argument of the same type. The type of **eq** allows us to state any equalities, even those that are provably false. However, examining the type of equality's sole constructor **eq_refl**, we see that we can only *prove* equality when its two arguments are syntactically equal. This definition turns out to capture all the basic properties of equality, and the equality-manipulating tactics that we have seen so far, like `reflexivity` and

rewrite, are implemented treating **eq** as just another inductive type with a well-chosen definition. Another way of stating that definition is, equality is defined as the least reflexive relation.

Returning to the example of **isZero**, we can see how to work with hypotheses that use this predicate.

Theorem isZero_plus : $\forall$ n m : **nat**, **isZero** $m \rightarrow n + m = n$.

We want to proceed by cases on the proof of the assumption about **isZero**.

    destruct 1.

    n : **nat**
    ============================
    $n + 0 = n$

Since **isZero** has only one constructor, we are presented with only one subgoal. The argument m to **isZero** is replaced with that type's argument from the single constructor IsZero. From this point, the proof is trivial.

    crush.
Qed.

Another example seems at first like it should admit an analogous proof, but in fact it provides a demonstration of one of the most common mistakes in basic Coq proving.

Theorem isZero_contra : **isZero** $1 \rightarrow$ **False**.

Let us try a proof by cases on the assumption, as in the last proof.

    destruct 1.

    ============================
    False

It seems that case analysis has not helped. The sole hypothesis disappears, leaving us worse off than before. What went wrong? We have met an important restriction in tactics like destruct and induction when applied to types with arguments. If the arguments are not already free variables, they will be replaced by new free variables internally before the case analysis or induction is done. Since the argument 1 to **isZero** is replaced by a fresh variable, we lose the crucial fact that it is not equal to 0.

Why does Coq use this restriction? Chapter 8 discusses the issue in detail, describing the dependently typed programming techniques to write this proof term manually. For now, I just say that the algorithmic problem of "logically complete case analysis" is undecidable when phrased in Coq's logic. A few tactics and design patterns presented later in this chapter suffice in almost all cases. For the current example, what we want is a tactic called `inversion`, which corresponds to the concept of inversion that is frequently used with natural deduction proof systems.

```
  Undo.
  inversion 1.
Qed.
```

What does `inversion` do? Think of it as a version of `destruct` that takes advantage of the structure of arguments to inductive types. In this case, `inversion` completed the proof immediately because it was able to detect that we were using **isZero** with an impossible argument.

Sometimes using `destruct` when we should have used `inversion` can lead to confusing results. To illustrate, consider another proof attempt for the last theorem with a different choice of contradictory conclusion.

```
Theorem isZero_contra' : isZero 1 → 2 + 2 = 5.
  destruct 1.
```

```
  ============================
   1 + 1 = 4
```

What happened here? Internally, `destruct` replaced 1 with a fresh variable, and trying to be helpful, it also replaced the occurrence of 1 within the unary representation of each number in the goal. Then, within the `O` case of the proof, the fresh variable was replaced with `O`. This had the net effect of decrementing each of these numbers.

```
Abort.
```

To see more clearly what happened, consider the type of **isZero**'s induction principle.

```
Check isZero_ind.
```

```
isZero_ind
     : ∀ P : nat → Prop, P 0 → ∀ n : nat, isZero n → P n
```

In the last proof script, `destruct` chose to instantiate P as `fun n ⇒` `S n + S n = S (S (S (S n)))`. Readers can verify that this specialization of the principle applies to the goal and that the hypothesis $P\ 0$

then matches the subgoal that was generated. A strange transmutation like this while doing a proof likely indicates that `destruct` should be replaced with `inversion`.

4.5 Recursive Predicates

We have already seen all the ingredients needed to build interesting recursive predicates, like this predicate capturing evenness.

```
Inductive even : nat → Prop :=
| EvenO : even O
| EvenSS : ∀ n, even n → even (S (S n)).
```

Think of **even** as another judgment defined by natural deduction rules. The rule EvenO has nothing above the line and **even** O below the line, and EvenSS is a rule with **even** n above the line and **even** (S (S n)) below.

The proof techniques of the last section are easily adapted.

```
Theorem even_0 : even 0.
  constructor.
Qed.
```

```
Theorem even_4 : even 4.
  constructor; constructor; constructor.
Qed.
```

It is not hard to see that such sequences of constructor applications can get tedious. We can avoid them using Coq's hint facility, with a new `Hint` variant that asks to consider all constructors of an inductive type during proof search. The tactic `auto` performs exhaustive proof search up to a fixed depth, considering only the proof steps registered as hints.

```
Hint Constructors even.
```

```
Theorem even_4' : even 4.
  auto.
Qed.
```

We may also use `inversion` with **even**.

```
Theorem even_1_contra : even 1 → False.
  inversion 1.
Qed.
```

```
Theorem even_3_contra : even 3 → False.
```

```
inversion 1.
```

H : **even** 3
n : **nat**
H1 : **even** 1
H0 : $n = 1$
============================
  False

The `inversion` tactic can be overzealous at times, as here with the introduction of the unused variable n and an equality hypothesis about it. For more complicated predicates, though, adding such assumptions is critical to dealing with the undecidability of general inversion. More complex inductive definitions and theorems can cause `inversion` to generate equalities where neither side is a variable.

  `inversion` *H1.*
`Qed.`

We can also do inductive proofs about **even**.

`Theorem` even_plus : $\forall\ n\ m$, **even** $n \rightarrow$ **even** $m \rightarrow$ **even** $(n + m)$.

It seems a reasonable first choice to proceed by induction on n.

  `induction` n; *crush.*

n : **nat**
IHn : $\forall\ m$: **nat**, **even** $n \rightarrow$ **even** $m \rightarrow$ **even** $(n + m)$
m : **nat**
H : **even** $(\mathsf{S}\ n)$
H0 : **even** m
============================
  even $(\mathsf{S}\ (n + m))$

We need to use the hypotheses H and *H0* somehow. The most natural choice is to invert H.

  `inversion` *H.*

n : **nat**
IHn : $\forall\ m$: **nat**, **even** $n \rightarrow$ **even** $m \rightarrow$ **even** $(n + m)$
m : **nat**
H : **even** $(\mathsf{S}\ n)$
H0 : **even** m

n0 : **nat**
H2 : **even** *n0*
H1 : S *n0* = *n*
===============================
  even (S (S *n0* + *m*))

Simplifying the conclusion brings us to a point where we can apply a constructor.

```
simpl.
```

===============================
  even (S (S (*n0* + *m*)))

```
constructor.
```

===============================
  even (*n0* + *m*)

At this point, we would like to apply the inductive hypothesis, which is

IHn : ∀ *m* : **nat**, **even** *n* → **even** *m* → **even** (*n* + *m*)

Unfortunately, the goal mentions *n0* where it would need to mention *n* to match *IHn*. We could keep looking for a way to finish this proof from here, but it is easier to change the basic strategy. Instead of inducting on the structure of *n*, we should induct *on the structure of one of the* **even** *proofs*. This technique is commonly called *rule induction* in programming language semantics. In the setting of Coq, we have already seen how predicates are defined using the same inductive type mechanism as datatypes, so the fundamental unity of rule induction with normal induction is apparent.

Recall that tactics like `induction` and `destruct` may be passed numbers to refer to unnamed left sides of implications in the conclusion, where the argument *n* refers to the *n*th such hypothesis.

```
Restart.
```

```
induction 1.
```

m : **nat**

```
        ============================
      even m → even (0 + m)
subgoal 2 is:
   even m → even (S (S n) + m)
```

The first case is easily discharged by *crush*, based on the hint to try the constructors of **even**.

crush.

Now we focus on the second case.

`intro`.

```
   m : nat
   n : nat
   H : even n
   IHeven : even m → even (n + m)
   H0 : even m
        ============================
      even (S (S n) + m)
```

We simplify and apply a constructor, as in the last proof attempt.

`simpl`; `constructor`.

```
        ============================
      even (n + m)
```

Now we have an exact match with the inductive hypothesis, and the remainder of the proof is trivial.

`apply` *IHeven*; `assumption`.

In fact, *crush* can handle all the details of the proof once we declare the induction strategy.

`Restart`.

   `induction 1`; *crush*.
`Qed`.

Induction on recursive predicates has similar pitfalls to those encountered with inversion in the last section.

`Theorem even_contra : ∀ n,` **even** `(S (n + n)) →` **False**.

```
induction 1.
```

n : **nat**
===============================
  False

```
subgoal 2 is:
```
  False

We cannot prove the first subgoal, since the argument to **even** was replaced by a fresh variable internally. This time, it is easier to prove this theorem by way of a lemma. Instead of trusting `induction` to replace expressions with fresh variables, we do it ourselves, explicitly adding the appropriate equalities as new assumptions.

```
Abort.
```

Lemma even_contra' : $\forall$ n', **even** $n' \to \forall$ n, n' = S $(n + n) \to$ **False**.
  induction 1; *crush*.

At this point, it is useful to consider all cases of n and $n0$ as being zero or nonzero. Only one of these cases has any trickiness to it.

  destruct n; destruct $n0$; *crush*.

n : **nat**
H : **even** (S n)
IHeven : $\forall$ $n0$: **nat**, S n = S $(n0 + n0) \to$ **False**
$n0$: **nat**
$H0$: S n = $n0$ + S $n0$
===============================
  False

Now it is useful to use a theorem from the standard library, which we also proved with a different name in the last chapter. We can search for a theorem that allows us to rewrite terms of the form x + S y.

  SearchRewrite (_ + S _).

  plus_n_Sm : $\forall$ n m : **nat**, S $(n + m)$ = n + S m

  rewrite $\leftarrow$ plus_n_Sm in *H0*.

The induction hypothesis lets us complete the proof if we use a variant of `apply` that has a `with` clause to give instantiations of quantified variables.

apply *IHeven* with *n0*; `assumption`.

As usual, we can rewrite the proof to avoid referencing any locally generated names, which makes the proof script more readable and more robust to changes in the theorem statement. We use the notation ← to request a hint that does right-to-left rewriting, just as with the `rewrite` tactic.

```
Restart.
```

```
Hint Rewrite ← plus_n_Sm.
```

```
induction 1; crush;
  match goal with
    | [ H : S ?N = ?N0 + ?N0 ⊢ _ ] ⇒ destruct N; destruct N0
  end; crush.
```
```
Qed.
```

We write the proof in a way that avoids the use of local variable or hypothesis names, using the `match` tactic form to do pattern matching on the goal. We use unification variables prefixed by question marks in the pattern and take advantage of the possibility to mention a unification variable twice in one pattern, to enforce equality between occurrences. The hint to rewrite with plus_n_Sm in a particular direction saves us from having to figure out the right place to apply that theorem.

The original theorem now follows trivially from the lemma, using a new tactic `eauto`, a fancier version of `auto` (see Chapter 13).

```
Theorem even_contra : ∀ n, even (S (n + n)) → False.
  intros; eapply even_contra'; eauto.
Qed.
```

We use a variant `eapply` of `apply`, which has the same relation to `apply` as `eauto` has to `auto`. An invocation of `apply` only succeeds if all arguments to the rule being used can be determined from the form of the goal, whereas `eapply` introduces unification variables for undetermined arguments. In this case, `eauto` is able to determine the right values for those unification variables, using a variant of the classic algorithm for *unification* [41].

By considering an alternative way to prove the lemma, we see another common pitfall of inductive proofs in Coq. Imagine that we had tried to prove even_contra' with all the ∀ quantifiers moved to the front of the lemma statement.

```
Lemma even_contra'' : ∀ n' n, even n' → n' = S (n + n) → False.
  induction 1; crush;
```

```
    match goal with
      | [ H : S ?N = ?N0 + ?N0 ⊢ _ ] ⇒ destruct N; destruct N0
    end; crush.
```

One subgoal remains.

n : **nat**
H : **even** (S $(n + n)$)
IHeven : S $(n + n)$ = S (S (S $(n + n)$)) → **False**
================================
  False

We are out of luck here. The inductive hypothesis is trivially true, since its assumption is false. In the version of this proof that succeeded, *IHeven* had an explicit quantification over n. This is because the quantification of n *appeared after the thing we are inducting on* in the theorem statement. In general, quantified variables and hypotheses that appear before the induction object in the theorem statement stay fixed throughout the inductive proof. Variables and hypotheses that are quantified after the induction object may be varied explicitly in uses of inductive hypotheses.

```
Abort.
```

Why should Coq implement `induction` this way? One answer is that it avoids burdening this basic tactic with additional heuristics, but that is not the whole picture. Imagine that `induction` analyzed dependencies among variables and reordered quantifiers to preserve as much freedom as possible in later uses of inductive hypotheses. That could make the inductive hypotheses more complex, which could in turn cause particular automation machinery to fail when it would have succeeded before. In general, we want to avoid quantifiers in proofs whenever we can, and that goal is furthered by the refactoring that the `induction` tactic forces us to do.