This is a section of doi:10.7551/mitpress/9153.001.0001

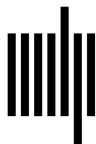
# Certified Programming with Dependent Types

## A Pragmatic Introduction to the Coq Proof Assistant

**By:** Adam Chlipala

## Citation:

*Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*
**By:** Adam Chlipala
**DOI:** 10.7551/mitpress/9153.001.0001
**ISBN (electronic):** 9780262317870
**Publisher:** The MIT Press
**Published:** 2022



The MIT Press

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

# 3 Introducing Inductive Types

The logical foundation of Coq is the Calculus of Inductive Constructions, or CIC. In a sense, CIC is built from just two relatively straightforward features: function types and inductive types. From this modest foundation, we can prove essentially all the theorems of math and carry out effectively all program verifications. This chapter introduces induction and recursion for functional programming in Coq. Most of the examples reproduce functionality from the Coq standard library, and I have tried to copy the standard library's choices of identifiers where possible, so many of the definitions here are available in the default Coq environment.

Chapter 2 presented some of the more advanced Coq features to highlight the unusual approach that I advocate in this book. However, from this point on, we go back to basics and study the relevant features of Coq in a more bottom-up manner. A useful first step is a discussion of the differences and connections between proofs and programs in Coq.

## 3.1 Proof Terms

Mainstream presentations of mathematics treat proofs as objects that exist outside of the universe of mathematical objects. However, for a variety of reasoning tasks, it is convenient to encode proofs, traditional mathematical objects, and programs within a single formal language. Validity checks on mathematical objects are useful in any setting, to catch typos and other trivial errors. The benefits of static typing for programs are widely recognized, and Coq brings those benefits to both mathematical objects and programs via a uniform mechanism. In fact, from here on, we do not distinguish between programs and mathematical objects. Many mathematical formalisms are most easily encoded in terms of programs.

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

Proofs are fundamentally different from programs because any two proofs of a theorem are considered equivalent, from a formal standpoint if not from an engineering standpoint. However, we can use the same type-checking technology to check proofs as we use to validate programs. This is the *Curry-Howard correspondence* [10, 14], an approach for relating proofs and programs. We represent mathematical theorems as types, such that a theorem's proofs are exactly those programs that type-check at the corresponding type.

An example of Curry-Howard was already given in Chapter 2, when the token → was used to stand for both function types and logical implications. Although this might seem like overloading notations, function types and implications are in fact precisely identical according to Curry-Howard. That is, they are just two ways of describing the same computational phenomenon.

A short demonstration should explain this. The identity function over the natural numbers is certainly not a controversial program.

```
Check (fun x : nat ⇒ x).
  : nat → nat
```

Consider this alternative program, which is almost identical to the last one:

```
Check (fun x : True ⇒ x).
  : True → True
```

The identity program is interpreted as a proof that **True**, the always-true proposition, implies itself. Curry-Howard interprets implications as functions, where an input is a proposition being assumed and an output is a proposition being deduced. This intuition is not too far from a common one for informal theorem proving, where we might think of an implication proof as a process for transforming a hypothesis into a conclusion.

More primitive proof forms are also available. For instance, the term I is the single proof of **True**, applicable in any context.

```
Check I.
  : True
```

With I, we can prove another simple propositional theorem.

```
Check (fun _ : False ⇒ I).
  : False → True
```

No proofs of **False** exist in the top-level context, but the implication-as-function analogy gives an easy way to, for example, show that **False** implies itself.

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

```
Check (fun x : False => x).
  : False -> False
```

Every one of these example programs whose type looks like a logical formula is a *proof term*. We use that name for any Gallina term of a logical type.

This chapter introduces different ways of defining types. Every example type can be interpreted alternatively as a type of programs or a type of proofs.

One of the first types introduced is **bool**, with constructors true and false. Newcomers to Coq often wonder about the distinction between **True** and true and the distinction between **False** and false. One answer is that **True** and **False** are types, but true and false are not. A more useful answer is that Coq's metatheory guarantees that any term of type **bool** *evaluates* to either true or false. This means that we have an *algorithm* for answering any question phrased as an expression of type **bool**. Conversely, most propositions do not evaluate to **True** or **False**; the language of inductively defined propositions is much richer than that. We ought to be glad that we have no algorithm for deciding the formalized version of mathematical truth, since otherwise it would be clear that we could not formalize undecidable properties, such as interesting properties of general-purpose programs.

## 3.2 Enumerations

Coq inductive types generalize the algebraic datatypes found in Haskell and ML. Confusingly enough, inductive types also generalize generalized algebraic datatypes (GADTs), by adding the possibility for type dependency. Even so, it is worth backing up from the examples of Chapter 2 and going over basic algebraic datatype uses of inductive datatypes, because the chance to prove things about the values of these types goes beyond usual practice in Haskell and ML.

The singleton type **unit** is an inductive type.

```
Inductive unit : Set :=
  | tt.
```

This vernacular command defines a new inductive type **unit** whose only value is tt. We can verify the types of the two identifiers.

```
Check unit.
  unit : Set

Check tt.
  tt : unit
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

We can prove that **unit** is a genuine singleton type.

```
Theorem unit_singleton : ∀ x : unit, x = tt.
```

The important thing about an inductive type is that we can do induction over its values, and induction is the key to proving this theorem. We ask to proceed by induction on the variable $x$.

```
  induction x.
```

The goal changes to

$$\mathsf{tt} = \mathsf{tt}$$

which we can discharge trivially.

```
  reflexivity.
Qed.
```

It seems odd to write a proof by induction with no inductive hypotheses. We could have arrived at the same result by beginning the proof with

```
  destruct x.
```

which corresponds to proof by case analysis in classical math. For nonrecursive inductive types, the two tactics will always have identical behavior. Often case analysis is sufficient, even in proofs about recursive types, and it is nice to avoid introducing unneeded induction hypotheses.

What exactly *is* the induction principle for **unit**? We can ask Coq.

```
Check unit_ind.
  unit_ind : ∀ P : unit → Prop, P tt → ∀ u : unit, P u
```

Every Inductive command defining a type $T$ also defines an induction principle named $T_ind$. Recall from the last section that the type, operations over it, and principles for reasoning about it all live in the same language and are described by the same type system. The key to telling what is a program and what is a proof lies in the distinction between the type Prop, which appears in the induction principle; and the type Set.

The convention goes like this: Set is the type of normal types used in programming, and the values of such types are programs. Prop is the type of logical propositions, and the values of such types are proofs. Thus, an induction principle has a type that shows it is a function for building proofs.

Specifically, unit_ind quantifies over a predicate $P$ over **unit** values. If we can present a proof that $P$ holds for tt, then we are rewarded with

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

a proof that $P$ holds for any value $u$ of type **unit**. In the last proof, the predicate was (`fun` $u$ : **unit** ⇒ $u$ = `tt`).

The definition of **unit** places the type in `Set`. By replacing `Set` with `Prop`, **unit** with **True**, and `tt` with I, we arrive at precisely the definition of **True** that the Coq standard library employs. The program type **unit** is the Curry-Howard equivalent of the proposition **True**. We might say that while philosophers have expended much ink on the nature of truth, we have now determined that truth is the **unit** type of functional programming.

We can define an inductive type even simpler than **unit**.

```
Inductive Empty_set : Set := .
```

**Empty_set** has no elements. We can prove amusing theorems about it.

```
Theorem the_sky_is_falling : ∀ x : Empty_set, 2 + 2 = 5.
  destruct 1.
Qed.
```

Because **Empty_set** has no elements, the fact of having an element of this type implies anything. We use `destruct 1` instead of `destruct` $x$ in the proof because unused quantified variables are relegated to being referred to by number. (There is a good reason for this, related to the unity of quantifiers and implication. At least within Coq's logical foundation of constructive logic, an implication is just a quantification over a proof, where the quantified variable is never used. It generally makes more sense to refer to implication hypotheses by number than by name, and Coq treats the quantifier over an unused variable as an implication in determining the proper behavior.)

We can see the induction principle that made this proof so easy.

```
Check Empty_set_ind.
  Empty_set_ind : ∀ (P : Empty_set → Prop) (e : Empty_set), P e
```

In other words, any predicate over values from the empty set holds vacuously of every such element. In the last proof, we chose the predicate (`fun _` : **Empty_set** ⇒ 2 + 2 = 5).

We can also apply this get-out-of-jail-free card programmatically. Here is a lazy way of converting values of **Empty_set** to values of **unit**:

```
Definition e2u (e : Empty_set) : unit := match e with end.
```

We employ `match` pattern matching as in Chapter 2. Since we match on a value whose type has no constructors, there is no need to provide any branches. It turns out that **Empty_set** is the Curry-Howard equivalent of **False**. As for why **Empty_set** starts with a capital letter and

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

not a lowercase letter as **unit** does, I must refer readers to the authors of the Coq standard library.

Moving up the ladder of complexity, we can define the Booleans.

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

We can use less vacuous pattern matching to define Boolean negation.

```
Definition negb (b : bool) : bool :=
  match b with
    | true ⇒ false
    | false ⇒ true
  end.
```

An alternative definition desugars to the preceding one, thanks to an `if` notation overloaded to work with any inductive type that has exactly two constructors:

```
Definition negb' (b : bool) : bool :=
  if b then false else true.
```

We might want to prove that negb is its own inverse operation.

```
Theorem negb_inverse : ∀ b : bool, negb (negb b) = b.
  destruct b.
```

After we case-analyze on $b$, we are left with one subgoal for each constructor of **bool**.

```
  2 subgoals

  ============================
   negb (negb true) = true

subgoal 2 is

 negb (negb false) = false
```

The first subgoal follows by Coq's rules of computation, so we can dispatch it easily.

```
  reflexivity.
```

The same holds for the second subgoal, so we can restart the proof and give a very compact justification.

```
Restart.
  destruct b; reflexivity.
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

```
Qed.
```

Another theorem about Booleans illustrates another useful tactic.

```
Theorem negb_ineq : ∀ b : bool, negb b ≠ b.
  destruct b; discriminate.
Qed.
```

The discriminate tactic is used to prove that two values of an inductive type are not equal if the values are formed with different constructors. In this case, the different constructors are true and false.

At this point, it is probably not hard to guess what the underlying induction principle for **bool** is.

```
Check bool_ind.
  bool_ind : ∀ P : bool → Prop, P true → P false → ∀ b : bool, P b
```

That is, to prove that a property describes all **bool**s, prove that it describes both true and false.

There is no interesting Curry-Howard analogue of **bool**. Of course, we can define such a type by replacing Set by Prop, but the resulting proposition is not very useful. It is logically equivalent to **True**, but it provides two indistinguishable primitive proofs, true and false.

## 3.3 Simple Recursive Types

The natural numbers are the simplest common example of an inductive type that actually deserves the name.

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

The constructor O is zero, and S is the successor function, so 0 is syntactic sugar for O, 1 for S O, 2 for S (S O), and so on.

Pattern matching works as demonstrated in Chapter 2:

```
Definition isZero (n : nat) : bool :=
  match n with
    | O ⇒ true
    | S _ ⇒ false
  end.

Definition pred (n : nat) : nat :=
  match n with
    | O ⇒ O
    | S n' ⇒ n'
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

```
  end.
```

We can prove theorems by case analysis with `destruct` as for simpler inductive types, but we can also now get into genuine inductive theorems. First, we need a recursive function.

```
Fixpoint plus (n m : nat) : nat :=
  match n with
    | O ⇒ m
    | S n' ⇒ S (plus n' m)
  end.
```

Recall that `Fixpoint` is Coq's mechanism for recursive function definitions. Some theorems about plus can be proved without induction.

```
Theorem O_plus_n : ∀ n : nat, plus O n = n.
  intro; reflexivity.
Qed.
```

Coq's computation rules automatically simplify the application of plus, because unfolding the definition of plus yields a `match` expression where the branch to be taken is obvious from syntax alone. If we just reverse the order of the arguments, though, this no longer works, and we need induction.

```
Theorem n_plus_O : ∀ n : nat, plus n O = n.
  induction n.
```

The first subgoal is plus O O = O, which is trivial by computation.

```
  reflexivity.
```

The second subgoal requires more work and also gives a first example of an inductive hypothesis.

```
  n : nat
  IHn : plus n O = n
  ============================
   plus (S n) O = S n
```

We can start by using computation to simplify the goal as far as we can.

```
  simpl.
```

Now the conclusion is S (plus $n$ O) = S $n$. Using the inductive hypothesis

```
  rewrite IHn.
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

we get a trivial conclusion S $n$ = S $n$.

```
  reflexivity.
```

Not much really went on in this proof, so the *crush* tactic from the CpdtTactics module can prove this theorem automatically.

```
Restart.
  induction n; crush.
Qed.
```

We can check out the induction principle at work here.

```
Check nat_ind.
```

```
  nat_ind : ∀ P : nat → Prop,
              P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

Each of the two cases of the last proof came from the type of one of the arguments to nat_ind. We chose $P$ to be (fun $n$ : **nat** ⇒ plus $n$ O = $n$). The first proof case corresponded to $P$ O and the second case to (∀ $n$ : **nat**, $P$ $n$ → $P$ (S $n$)). The free variable $n$ and inductive hypothesis *IHn* came from the argument types given here.

Since **nat** has a constructor that takes an argument, we may sometimes need to know that the constructor is injective.

```
Theorem S_inj : ∀ n m : nat, S n = S m → n = m.
  injection 1; trivial.
Qed.
```

The `injection` tactic refers to a premise by number, adding new equalities between the corresponding arguments of equated terms that are formed with the same constructor. We need to prove $n = m \rightarrow n = m$, so it is unsurprising that a tactic named `trivial` is able to finish the proof. This tactic attempts a variety of single proof steps, drawn from a user-specified database that can be extended.

There is also a very useful tactic called `congruence` that can prove this theorem immediately. The `congruence` tactic generalizes `discriminate` and `injection`, and it also adds reasoning about the general properties of equality, such as that a function returns equal results on equal arguments. That is, `congruence` is a *complete decision procedure for the theory of equality and uninterpreted functions* plus some reasoning about inductive types.

We can define a type of lists of natural numbers.

```
Inductive nat_list : Set :=
| NNil : nat_list
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

```
| NCons : nat → nat_list → nat_list.
```

Recursive definitions over **nat_list** are straightforward extensions of what we have seen before.

```
Fixpoint nlength (ls : nat_list) : nat :=
  match ls with
    | NNil ⇒ O
    | NCons _ ls' ⇒ S (nlength ls')
  end.

Fixpoint napp (ls1 ls2 : nat_list) : nat_list :=
  match ls1 with
    | NNil ⇒ ls2
    | NCons n ls1' ⇒ NCons n (napp ls1' ls2)
  end.
```

Inductive theorem proving can again be automated quite effectively.

```
Theorem nlength_napp : ∀ ls1 ls2 : nat_list, nlength (napp ls1 ls2)
  = plus (nlength ls1) (nlength ls2).
  induction ls1; crush.
Qed.

Check nat_list_ind.

  nat_list_ind
     : ∀ P : nat_list → Prop,
       P NNil →
       (∀ (n : nat) (n0 : nat_list), P n0 → P (NCons n n0)) →
       ∀ n : nat_list, P n
```

In general, we can implement any tree type as an inductive type. For example, here are binary trees of naturals:

```
Inductive nat_btree : Set :=
| NLeaf : nat_btree
| NNode : nat_btree → nat → nat_btree → nat_btree.
```

Here are two functions whose intuitive explanations are not so important. The first one computes the size of a tree, and the second performs some sort of splicing of one tree into the leftmost available leaf node of another.

```
Fixpoint nsize (tr : nat_btree) : nat :=
  match tr with
    | NLeaf ⇒ S O
    | NNode tr1 _ tr2 ⇒ plus (nsize tr1) (nsize tr2)
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

```
  end.

Fixpoint nsplice (tr1 tr2 : nat_btree) : nat_btree :=
  match tr1 with
    | NLeaf ⇒ NNode tr2 O NLeaf
    | NNode tr1' n tr2' ⇒ NNode (nsplice tr1' tr2) n tr2'
  end.

Theorem plus_assoc : ∀ n1 n2 n3 : nat, plus (plus n1 n2) n3
  = plus n1 (plus n2 n3).
  induction n1; crush.
Qed.

Theorem nsize_nsplice : ∀ tr1 tr2 : nat_btree, nsize (nsplice tr1 tr2)
  = plus (nsize tr2) (nsize tr1).
  Hint Rewrite n_plus_O plus_assoc.

  induction tr1; crush.
Qed.
```

It is convenient that these proofs go through so easily, but it is still useful to look into the details of what happened, by checking the statement of the tree induction principle.

```
Check nat_btree_ind.

  nat_btree_ind
     : ∀ P : nat_btree → Prop,
        P NLeaf →
        (∀ n : nat_btree,
         P n → ∀ (n0 : nat) (n1 : nat_btree), P n1
           → P (NNode n n0 n1)) →
        ∀ n : nat_btree, P n
```

We have the usual two cases, one for each constructor of **nat_btree**.

## 3.4 Parameterized Types

We can also define polymorphic inductive types, as with algebraic datatypes in Haskell and ML.

```
Inductive list (T : Set) : Set :=
| nil : list T
| cons : T → list T → list T.

Fixpoint length T (ls : list T) : nat :=
  match ls with
    | nil ⇒ O
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

```
    | cons _ ls' ⇒ S (length ls')
  end.

Fixpoint app T (ls1 ls2 : list T) : list T :=
  match ls1 with
    | nil ⇒ ls2
    | cons x ls1' ⇒ cons x (app ls1' ls2)
  end.

Theorem length_app : ∀ T (ls1 ls2 : list T), length (app ls1 ls2)
  = plus (length ls1) (length ls2).
  induction ls1; crush.
Qed.
```

There is a useful shorthand for writing many definitions that share the same parameter, based on Coq's *section* mechanism. The following block of code is equivalent to the previous one.

```
Section list.
  Variable T : Set.

  Inductive list : Set :=
  | nil : list
  | cons : T → list → list.

  Fixpoint length (ls : list) : nat :=
    match ls with
      | nil ⇒ O
      | cons _ ls' ⇒ S (length ls')
    end.

  Fixpoint app (ls1 ls2 : list) : list :=
    match ls1 with
      | nil ⇒ ls2
      | cons x ls1' ⇒ cons x (app ls1' ls2)
    end.

  Theorem length_app : ∀ ls1 ls2 : list, length (app ls1 ls2)
    = plus (length ls1) (length ls2).
    induction ls1; crush.
  Qed.
End list.

Implicit Arguments nil [T].
```

After we end the section, the `Variable`s we used are added as extra function parameters for each defined identifier, as needed. With an `Implicit Arguments` command, we ask that $T$ be inferred when we

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

use nil; Coq's heuristics already decided to apply a similar policy to cons, because of the Set Implicit Arguments command elided at the beginning of this chapter. We verify that our definitions have been saved properly using the Print command, a cousin of Check that shows the definition of a symbol rather than just its type.

```
Print list.

  Inductive list (T : Set) : Set :=
    nil : list T | cons : T → list T → list T
```

The final definition is the same as what we wrote manually before. The other elements of the section are altered similarly, turning out exactly as they were before, though we wrote their definitions more succinctly.

```
Check length.

  length
     : ∀ T : Set, list T → nat
```

The parameter $T$ is treated as a new argument to the induction principle, too.

```
Check list_ind.

  list_ind
     : ∀ (T : Set) (P : list T → Prop),
       P (nil T) →
       (∀ (t : T) (l : list T), P l → P (cons t l)) →
       ∀ l : list T, P l
```

Thus, despite a very real sense in which the type $T$ is an argument to the constructor cons, the inductive case in the type of list_ind (the third line of the type) includes no quantifier for $T$, even though all the other arguments are quantified explicitly. Parameters in other inductive definitions are treated similarly in stating induction principles.

## 3.5 Mutually Inductive Types

We can define inductive types that refer to each other:

```
Inductive even_list : Set :=
| ENil : even_list
| ECons : nat → odd_list → even_list

with odd_list : Set :=
| OCons : nat → even_list → odd_list.
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

```
Fixpoint elength (el : even_list) : nat :=
  match el with
    | ENil ⇒ O
    | ECons _ ol ⇒ S (olength ol)
  end

with olength (ol : odd_list) : nat :=
  match ol with
    | OCons _ el ⇒ S (elength el)
  end.

Fixpoint eapp (el1 el2 : even_list) : even_list :=
  match el1 with
    | ENil ⇒ el2
    | ECons n ol ⇒ ECons n (oapp ol el2)
  end

with oapp (ol : odd_list) (el : even_list) : odd_list :=
  match ol with
    | OCons n el' ⇒ OCons n (eapp el' el)
  end.
```

Everything is roughly the same as in earlier examples until we try to prove a theorem similar to those that came before.

```
Theorem elength_eapp : ∀ el1 el2 : even_list,
  elength (eapp el1 el2) = plus (elength el1) (elength el2).
  induction el1; crush.
```

One goal remains:

```
  n : nat
  o : odd_list
  el2 : even_list
  ============================
   S (olength (oapp o el2)) = S (plus (olength o) (elength el2))
```

We have no induction hypothesis, so we cannot prove this goal without starting another induction, which would reach a similar point, resulting in a futile infinite chain of inductions. The problem is that Coq's generation of *T_ind* principles is incomplete. We only get nonmutual induction principles generated by default.

```
Abort.
Check even_list_ind.
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

```
even_list_ind
    : ∀ P : even_list → Prop,
      P ENil →
      (∀ (n : nat) (o : odd_list), P (ECons n o)) →
      ∀ e : even_list, P e
```

We see that no inductive hypotheses are included anywhere in the type. To get them, we must ask for mutual principles as we need them, using the Scheme command.

```
Scheme even_list_mut := Induction for even_list Sort Prop
with odd_list_mut := Induction for odd_list Sort Prop.
```

This invocation of Scheme asks for the creation of induction principles even_list_mut for the type **even_list** and odd_list_mut for the type **odd_list**. The Induction keyword says we want standard induction schemes, since Scheme supports more exotic choices. Finally, Sort Prop establishes that we really want induction schemes, not recursion schemes, which are the same according to Curry-Howard, save for the Prop/Set distinction.

```
Check even_list_mut.
```

```
even_list_mut
    : ∀ (P : even_list → Prop) (P0 : odd_list → Prop),
      P ENil →
      (∀ (n : nat) (o : odd_list), P0 o → P (ECons n o)) →
      (∀ (n : nat) (e : even_list), P e → P0 (OCons n e)) →
      ∀ e : even_list, P e
```

This is the principle we wanted in the first place.

The Scheme command is for asking Coq to generate particular induction schemes that are mutual among a set of inductive types (possibly only one such type, in which case we get a normal induction principle). In a sense, it generalizes the induction scheme generation that goes on automatically for each inductive definition. Future Coq versions might make that automatic generation smarter so that Scheme will be needed in fewer places. Section 3.7 explains how induction principles are derived theorems in Coq so that there is no need to build in *any* automatic scheme generation.

There is one more consideration in using the even_list_mut induction principle: the induction tactic will not apply it automatically. It is helpful to look at how to prove one of the past examples without using induction so that we can then generalize the technique to mutual inductive types.

```
Theorem n_plus_O' : ∀ n : nat, plus n O = n.
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

```
  apply nat_ind.
```

Here we use apply, which is one of the most essential basic tactics. When we are trying to prove fact $P$, and when *thm* is a theorem whose conclusion can be made to match $P$ by proper choice of quantified variable values, the invocation apply *thm* will replace the current goal with one new goal for each premise of *thm*.

This use of apply may seem a bit too magical. To better see what is going on, we use a variant where we partially apply the theorem nat_ind to give an explicit value for the predicate that gives the induction hypothesis.

```
  Undo.
  apply (nat_ind (fun n ⇒ plus n O = n)); crush.
Qed.
```

From this example, we see that induction is not magic. It only does some bookkeeping for us to make it easy to apply a theorem, which we can do directly with the apply tactic.

This technique generalizes to the mutual example.

```
Theorem elength_eapp : ∀ el1 el2 : even_list,
  elength (eapp el1 el2) = plus (elength el1) (elength el2).

  apply (even_list_mut
    (fun el1 : even_list ⇒ ∀ el2 : even_list,
      elength (eapp el1 el2) = plus (elength el1) (elength el2))
    (fun ol : odd_list ⇒ ∀ el : even_list,
      olength (oapp ol el) = plus (olength ol) (elength el))); crush.
Qed.
```

We simply need to specify two predicates, one for each of the mutually inductive types. In general, it is not a good idea to assume that a proof assistant can infer extra predicates, so this way of applying mutual induction is about as straightforward as we may hope for.

## 3.6 Reflexive Types

A kind of inductive type called a *reflexive type* includes at least one constructor that takes as an argument *a function returning the same type we are defining*. One very useful class of examples is in modeling variable binders. The example will be an encoding of the syntax of first-order logic. Since the idea of syntactic encodings of logic may require a bit of acclimation, let us first consider a simpler formula type for a subset of propositional logic. We are not yet using a reflexive type, but later we will extend the example reflexively.

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

```
Inductive pformula : Set :=
| Truth : pformula
| Falsehood : pformula
| Conjunction : pformula → pformula → pformula.
```

A key distinction here is between, for instance, the *syntax* Truth and its *semantics* **True**. We can make the semantics explicit with a recursive function. This function uses the infix operator ∧, which desugars to instances of the type family **and** from the standard library. The family **and** implements conjunction, the Prop Curry-Howard analogue of the usual pair type from functional programming (which is the type family **prod** in Coq's standard library).

```
Fixpoint pformulaDenote (f : pformula) : Prop :=
  match f with
    | Truth ⇒ True
    | Falsehood ⇒ False
    | Conjunction f1 f2 ⇒ pformulaDenote f1 ∧ pformulaDenote f2
  end.
```

This example does not use reflexive types, the new feature to be introduced. When we set our sights on first-order logic instead, it becomes very handy to give constructors recursive arguments that are functions.

```
Inductive formula : Set :=
| Eq : nat → nat → formula
| And : formula → formula → formula
| Forall : (nat → formula) → formula.
```

Formulas are equalities between naturals, conjunction, and universal quantification over natural numbers. We avoid needing to include a notion of variables in the type by using Coq functions to encode the syntax of quantification. For instance, here is the encoding of $\forall\ x : \textbf{nat},\ x = x$:

```
Example forall_refl : formula := Forall (fun x ⇒ Eq x x).
```

We can write recursive functions over reflexive types quite naturally. Here is one translating the formulas into native Coq propositions:

```
Fixpoint formulaDenote (f : formula) : Prop :=
  match f with
    | Eq n1 n2 ⇒ n1 = n2
    | And f1 f2 ⇒ formulaDenote f1 ∧ formulaDenote f2
    | Forall f' ⇒ ∀ n : nat, formulaDenote (f' n)
  end.
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

We can also encode a trivial formula transformation that swaps the order of equality and conjunction operands.

```
Fixpoint swapper (f : formula) : formula :=
  match f with
    | Eq n1 n2 ⇒ Eq n2 n1
    | And f1 f2 ⇒ And (swapper f2) (swapper f1)
    | Forall f' ⇒ Forall (fun n ⇒ swapper (f' n))
  end.
```

It is helpful to prove that this transformation does not make true formulas false.

```
Theorem swapper_preserves_truth : ∀ f, formulaDenote f
  → formulaDenote (swapper f).
  induction f; crush.
Qed.
```

We can take a look at the induction principle behind this proof.

```
Check formula_ind.

  formula_ind
     : ∀ P : formula → Prop,
       (∀ n n0 : nat, P (Eq n n0)) →
       (∀ f0 : formula,
        P f0 → ∀ f1 : formula, P f1 → P (And f0 f1)) →
       (∀ f1 : nat → formula,
        (∀ n : nat, P (f1 n)) → P (Forall f1)) →
       ∀ f2 : formula, P f2
```

Focusing on the Forall case, which comes third, we see that we are allowed to assume that the theorem holds *for any application of the argument function f1*. That is, Coq induction principles do not follow a simple rule that the textual representations of induction variables must get shorter in appeals to induction hypotheses. Luckily, the people behind the metatheory of Coq have verified that this flexibility does not introduce unsoundness.

Up to this point, we have seen how to encode in Coq more and more of what is possible with algebraic datatypes in Haskell and ML. This may have given the inaccurate impression that inductive types are a strict extension of algebraic datatypes. In fact, Coq must rule out some types allowed by Haskell and ML, for reasons of soundness. Reflexive types provide the first good example of such a case; only some of them are legal.

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

Given the last example of an inductive type, readers may be eager to try encoding the syntax of lambda calculus. Indeed, the function-based representation technique just used, called *higher-order abstract syntax* (HOAS) [35], is the representation of choice for lambda calculi in Twelf and in many applications implemented in Haskell and ML. Let us try to import that choice to Coq.

```
Inductive term : Set :=
| App : term → term → term
| Abs : (term → term) → term.
```

```
Error: Non strictly positive occurrence of "term" in
"(term -> term) -> term"
```

We have run afoul of the *strict positivity requirement* for inductive definitions, which says that the type being defined may not occur to the left of an arrow in the type of a constructor argument. It is important that the type of a constructor is viewed in terms of a series of arguments and a result, since clearly we need recursive occurrences on the left sides of the outermost arrows if we are to have recursive occurrences at all. The candidate definition violates the positivity requirement because it involves an argument of type **term** → **term**, where the type **term** that we are defining appears to the left of an arrow. The candidate type of App is fine, however, since every occurrence of **term** is either a constructor argument or the final result type.

Why must Coq enforce this restriction? Imagine that the last definition had been accepted, allowing us to write this function:

```
Definition uhoh (t : term) : term :=
  match t with
    | Abs f ⇒ f t
    | _ ⇒ t
  end.
```

Using an informal idea of Coq's semantics, it is easy to verify that the application uhoh (Abs uhoh) will run forever. This would be a mere curiosity in OCaml and Haskell, where nontermination is commonplace, though the fact that we have a nonterminating program without explicit recursive function definitions is unusual.

For Coq, however, this would be a disaster. The possibility of writing such a function would destroy all confidence that proving a theorem means anything. Since Coq combines programs and proofs in one language, we would be able to prove every theorem with an infinite loop.

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

Nonetheless, the basic insight of HOAS is a very useful one, and there are ways to realize most benefits of HOAS in Coq. Chapter 17 describes a particular technique of this kind.

## 3.7 An Interlude on Induction Principles

As mentioned, Coq proofs are actually programs, written in the same language used in the examples so far. We can get a sense of what this means by looking at the definitions of some of the induction principles. Studying the details will help us construct induction principles manually, which is necessary for some more advanced inductive definitions.

```
Print nat_ind.
```

```
nat_ind =
fun P : nat → Prop ⇒ nat_rect P
      : ∀ P : nat → Prop,
        P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

This induction principle is defined in terms of a more general principle, `nat_rect`. The `rec` stands for *recursion principle*, and the `t` at the end stands for `Type`.

```
Check nat_rect.
```

```
nat_rect
      : ∀ P : nat → Type,
        P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

The principle `nat_rect` gives $P$ type **nat** → `Type` instead of **nat** → `Prop`. `Type` is another universe, like `Set` and `Prop`. In fact, it is a common supertype of both (see Chapter 12). `Type` can be used as a sort of meta-universe that may turn out to be either `Set` or `Prop`. We can see the symmetry inherent in the subtyping relation by printing the definition of another principle that was generated for **nat** automatically:

```
Print nat_rec.
```

```
nat_rec =
fun P : nat → Set ⇒ nat_rect P
      : ∀ P : nat → Set,
        P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

This is identical to the definition for `nat_ind` except that `Set` is substituted for `Prop`. For most inductive types $T$, then, we get not just

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

induction principles $T_ind$ but also recursion principles $T_rec$. We can use $T_rec$ to write recursive definitions without explicit Fixpoint recursion. For instance, the following two definitions are equivalent.

```
Fixpoint plus_recursive (n : nat) : nat → nat :=
  match n with
    | O ⇒ fun m ⇒ m
    | S n' ⇒ fun m ⇒ S (plus_recursive n' m)
  end.

Definition plus_rec : nat → nat → nat :=
  nat_rec (fun _ : nat ⇒ nat → nat) (fun m ⇒ m)
  (fun _ r m ⇒ S (r m)).

Theorem plus_equivalent : plus_recursive = plus_rec.
  reflexivity.
Qed.
```

Finally, nat_rect itself is not even a primitive. It is a functional program that we can write manually.

```
Print nat_rect.

nat_rect =
fun (P : nat → Type) (f : P O) (f0 : ∀ n : nat, P n → P (S n)) ⇒
fix F (n : nat) : P n :=
  match n as n0 return (P n0) with
  | O ⇒ f
  | S n0 ⇒ f0 n0 (F n0)
  end
     : ∀ P : nat → Type,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

The only new items here are first, an anonymous recursive function definition using the fix keyword of Gallina (which is like fun with recursion supported), and second, the annotations on the match expression. This is a *dependently typed* pattern match, because the *type* of the expression depends on the *value* being matched on. (More complex examples are given in Part II.)

Type inference for dependent pattern matching is undecidable, which can be proved by reduction from higher-order unification [15]. Thus, we often need to annotate programs in a way that explains dependencies to the type checker. In the example of nat_rect, we have an as clause, which binds a name for the discriminee, and a return clause, which gives a way to compute the match result type as a function of the discriminee.

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

To prove that nat_rect is not extraordinary, we can reimplement it manually.

```
Fixpoint nat_rect' (P : nat → Type)
  (HO : P O)
  (HS : ∀ n, P n → P (S n)) (n : nat) :=
  match n return P n with
    | O ⇒ HO
    | S n' ⇒ HS n' (nat_rect' P HO HS n')
  end.
```

We can understand the definition of nat_rect better by reimplementing nat_ind using sections.

```
Section nat_ind'.
```

First, we have the property of natural numbers that we aim to prove.

```
  Variable P : nat → Prop.
```

Then we require a proof of the O case, which we declare with the command Hypothesis, which is a synonym for Variable that, by convention, is used for variables whose types are propositions.

```
  Hypothesis O_case : P O.
```

Next is a proof of the S case, which may assume an inductive hypothesis.

```
  Hypothesis S_case : ∀ n : nat, P n → P (S n).
```

Finally, we define a recursive function to tie the pieces together.

```
  Fixpoint nat_ind' (n : nat) : P n :=
    match n with
      | O ⇒ O_case
      | S n' ⇒ S_case (nat_ind' n')
    end.
End nat_ind'.
```

Closing the section adds the variables declared with Variable and Hypothesis as new fun-bound arguments to nat_ind', and, modulo the use of Prop instead of Type, we end up with the exact definition that was generated automatically for nat_rect.

We can also examine the definition of even_list_mut, which we generated with Scheme for a mutually recursive type.

```
Print even_list_mut.
```

```
  even_list_mut =
  fun (P : even_list → Prop) (P0 : odd_list → Prop)
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

```
    (f : P ENil)
    (f0 : ∀ (n : nat) (o : odd_list), P0 o → P (ECons n o))
    (f1 : ∀ (n : nat) (e : even_list), P e → P0 (OCons n e)) ⇒
  fix F (e : even_list) : P e :=
    match e as e0 return (P e0) with
    | ENil ⇒ f
    | ECons n o ⇒ f0 n o (F0 o)
    end
  with F0 (o : odd_list) : P0 o :=
    match o as o0 return (P0 o0) with
    | OCons n e ⇒ f1 n e (F e)
    end
  for F
     : ∀ (P : even_list → Prop) (P0 : odd_list → Prop),
       P ENil →
       (∀ (n : nat) (o : odd_list), P0 o → P (ECons n o)) →
       (∀ (n : nat) (e : even_list), P e → P0 (OCons n e)) →
       ∀ e : even_list, P e
```

We see a mutually recursive `fix`, with the different functions separated by `with` in the same way that they would be separated by `and` in ML. A final `for` clause identifies which of the mutually recursive functions should be the final value of the `fix` expression. Using this definition as a template, we can reimplement even_list_mut directly.

```
Section even_list_mut'.
```

First, we need the properties that we are proving.

```
  Variable Peven : even_list → Prop.
  Variable Podd : odd_list → Prop.
```

Next, we need proofs of the three cases.

```
  Hypothesis ENil_case : Peven ENil.
  Hypothesis ECons_case : ∀ (n : nat) (o : odd_list),
    Podd o → Peven (ECons n o).
  Hypothesis OCons_case : ∀ (n : nat) (e : even_list),
    Peven e → Podd (OCons n e).
```

Finally, we define the recursive functions.

```
  Fixpoint even_list_mut' (e : even_list) : Peven e :=
    match e with
      | ENil ⇒ ENil_case
      | ECons n o ⇒ ECons_case n (odd_list_mut' o)
    end
  with odd_list_mut' (o : odd_list) : Podd o :=
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

```
    match o with
      | OCons n e ⇒ OCons_case n (even_list_mut' e)
    end.
End even_list_mut'.
```

Even induction principles for reflexive types are easy to implement directly. For the **formula** type, we can use a recursive definition much like the earlier ones.

```
Section formula_ind'.
  Variable P : formula → Prop.
  Hypothesis Eq_case : ∀ n1 n2 : nat, P (Eq n1 n2).
  Hypothesis And_case : ∀ f1 f2 : formula,
    P f1 → P f2 → P (And f1 f2).
  Hypothesis Forall_case : ∀ f : nat → formula,
    (∀ n : nat, P (f n)) → P (Forall f).

  Fixpoint formula_ind' (f : formula) : P f :=
    match f with
      | Eq n1 n2 ⇒ Eq_case n1 n2
      | And f1 f2 ⇒ And_case (formula_ind' f1) (formula_ind' f2)
      | Forall f' ⇒ Forall_case f' (fun n ⇒ formula_ind' (f' n))
    end.
End formula_ind'.
```

It is apparent that induction principle implementations involve some tedium but not much creativity.

## 3.8 Nested Inductive Types

Suppose we want to extend the earlier type of binary trees to trees with arbitrary finite branching. We can use lists to give a simple definition.

```
Inductive nat_tree : Set :=
| NNode' : nat → list nat_tree → nat_tree.
```

This is an example of a *nested* inductive type definition, because we use the type we are defining as an argument to a parameterized type family. Coq will not allow all such definitions; it effectively pretends that we are defining **nat_tree** mutually with a version of **list** specialized to **nat_tree**, checking that the resulting expanded definition satisfies the usual rules. For instance, if we replaced **list** with a type family that used its parameter as a function argument, then the definition would be rejected as violating the positivity restriction.

As with mutual inductive types, we find that the automatically generated induction principle for **nat_tree** is too weak.

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

```
Check nat_tree_ind.

  nat_tree_ind
     : ∀ P : nat_tree → Prop,
       (∀ (n : nat) (l : list nat_tree), P (NNode' n l)) →
       ∀ n : nat_tree, P n
```

There is no command like `Scheme` that will implement an improved principle. In general, it takes creativity to figure out good ways to incorporate nested uses of different type families. Now that we know how to implement induction principles manually, it is possible to apply just such creativity to this problem.

Many induction principles for types with nested used of **list** could benefit from a unified predicate capturing the idea that some property holds of every element in a list. By defining this generic predicate once, we facilitate reuse of library theorems about it. (Here, we are actually duplicating the standard library's Forall predicate, with a different implementation, for didactic purposes.)

```
Section All.
  Variable T : Set.
  Variable P : T → Prop.

  Fixpoint All (ls : list T) : Prop :=
    match ls with
      | nil ⇒ True
      | cons h t ⇒ P h ∧ All t
    end.
End All.
```

It will be useful to review the definitions of **True** and ∧, since we will want to write manual proofs of them.

```
Print True.

  Inductive True : Prop := I : True
```

That is, **True** is a proposition with exactly one proof, I, which we may always supply trivially.

Finding the definition of ∧ takes a little more work. Coq supports user registration of arbitrary parsing rules, and it is such a rule that lets us write ∧ instead of an application of some inductive type family. We can find the underlying inductive type with the `Locate` command, whose argument may be a parsing token.

```
Locate "/\".

  "A /\ B" := and A B : type_scope (default interpretation)
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

```
Print and.

  Inductive and (A : Prop) (B : Prop) : Prop :=
    conj : A → B → A ∧ B

  For conj: Arguments A, B are implicit
```

In addition to the definition of **and** itself, we get information on implicit arguments (and some other information, omitted here). The implicit argument information tells us that we build a proof of a conjunction by calling the constructor conj on proofs of the conjuncts, with no need to include the types of those proofs as explicit arguments.

Now we create a section for the induction principle, following the same basic plan as earlier.

```
Section nat_tree_ind'.
  Variable P : nat_tree → Prop.

  Hypothesis NNode'_case : ∀ (n : nat) (ls : list nat_tree),
    All P ls → P (NNode' n ls).
```

A first attempt at writing the induction principle itself follows the intuition that nested inductive type definitions are expanded into mutual inductive definitions.

```
  Fixpoint nat_tree_ind' (tr : nat_tree) : P tr :=
    match tr with
      | NNode' n ls ⇒ NNode'_case n ls (list_nat_tree_ind ls)
    end

  with list_nat_tree_ind (ls : list nat_tree) : All P ls :=
    match ls with
      | nil ⇒ I
      | cons tr rest ⇒ conj (nat_tree_ind' tr) (list_nat_tree_ind rest)
    end.
```

Coq rejects this definition, saying

```
Recursive call to nat_tree_ind' has principal argument
equal to "tr" instead of rest.
```

There is no theoretical reason why this program should be rejected; Coq applies incomplete termination-checking heuristics, and it is necessary to learn a few of the most important rules. The term *nested inductive type* hints at the solution to this particular problem.

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

Just as mutually inductive types require mutually recursive induction principles, nested types require nested recursion.

```
Fixpoint nat_tree_ind' (tr : nat_tree) : P tr :=
  match tr with
    | NNode' n ls => NNode'_case n ls
      ((fix list_nat_tree_ind (ls : list nat_tree) : All P ls :=
        match ls with
          | nil => I
          | cons tr' rest =>
            conj (nat_tree_ind' tr') (list_nat_tree_ind rest)
        end) ls)
  end.
```

We include an anonymous `fix` version of list_nat_tree_ind that is literally nested inside the definition of the recursive function corresponding to the inductive definition that had the nested use of **list**.

```
End nat_tree_ind'.
```

We can test the induction principle by defining some recursive functions on **nat_tree** and proving a theorem about them. First, we define some helper functions that operate on lists.

```
Section map.
  Variables T T' : Set.
  Variable F : T -> T'.

  Fixpoint map (ls : list T) : list T' :=
    match ls with
      | nil => nil
      | cons h t => cons (F h) (map t)
    end.
End map.

Fixpoint sum (ls : list nat) : nat :=
  match ls with
    | nil => O
    | cons h t => plus h (sum t)
  end.
```

Now we can define a size function over the trees.

```
Fixpoint ntsize (tr : nat_tree) : nat :=
  match tr with
    | NNode' _ trs => S (sum (map ntsize trs))
  end.
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

Notice that Coq expanded the definition of map to verify that we are using proper nested recursion, even through a use of a higher-order function.

```
Fixpoint ntsplice (tr1 tr2 : nat_tree) : nat_tree :=
  match tr1 with
    | NNode' n nil ⇒ NNode' n (cons tr2 nil)
    | NNode' n (cons tr trs) ⇒ NNode' n (cons (ntsplice tr tr2) trs)
  end.
```

We have defined another arbitrary notion of tree splicing, similar to before, and we can prove an analogous theorem about its relation to tree size. We start with a useful lemma about addition.

```
Lemma plus_S : ∀ n1 n2 : nat,
  plus n1 (S n2) = S (plus n1 n2).
  induction n1; crush.
Qed.
```

Now we begin the proof of the theorem, adding the lemma plus_S as a hint.

```
Theorem ntsize_ntsplice : ∀ tr1 tr2 : nat_tree, ntsize (ntsplice tr1 tr2)
  = plus (ntsize tr2) (ntsize tr1).
  Hint Rewrite plus_S.
```

We know that the standard induction principle is insufficient for the task, so we need to provide a using clause for the induction tactic to specify the alternative principle.

```
  induction tr1 using nat_tree_ind'; crush.
```

One subgoal remains.

```
  n : nat
  ls : list nat_tree
  H : All
        (fun tr1 : nat_tree ⇒
         ∀ tr2 : nat_tree,
         ntsize (ntsplice tr1 tr2) = plus (ntsize tr2) (ntsize tr1)) ls
  tr2 : nat_tree
  ============================
   ntsize
     match ls with
     | nil ⇒ NNode' n (cons tr2 nil)
     | cons tr trs ⇒ NNode' n (cons (ntsplice tr tr2) trs)
     end = S (plus (ntsize tr2) (sum (map ntsize ls)))
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

Now we need to do a case analysis on the structure of *ls*. The rest is routine.

```
destruct ls; crush.
```

We can go further in automating the proof by exploiting the hint mechanism.

```
Restart.

Hint Extern 1 (ntsize (match ?LS with nil ⇒ _
                                   | cons _ _ ⇒ _ end) = _) ⇒
  destruct LS; crush.

induction tr1 using nat_tree_ind'; crush.
Qed.
```

Note that with the hint we register a pattern that describes a conclusion we expect to encounter during the proof. The pattern may contain unification variables, whose names are prefixed with question marks, and we may refer to those bound variables in a tactic that we ask to have run whenever the pattern matches.

The advantage of using the hint is not very clear here because the original proof was so short. However, the hint has fundamentally improved the readability of the proof. Before, the proof referred to the local variable *ls*, which has an automatically generated name. To a human reading the proof script without stepping through it interactively, it was not clear where *ls* came from. The hint explains to the reader the process for choosing which variables to case-analyze, and the hint can continue working even if the rest of the proof structure changes significantly.

## 3.9 Manual Proofs about Constructors

It can be useful to understand how tactics like `discriminate` and `injection` work, so it is worth stepping through a manual proof of each kind. We start with a proof fit for `discriminate`.

```
Theorem true_neq_false : true ≠ false.
```

We begin with the tactic `red`, which is short for one step of reduction, to unfold the definition of logical negation.

```
red.
```

```
============================
 true = false → False
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

The negation is replaced with an implication of falsehood. We use the tactic `intro` $H$ to change the assumption of the implication into a hypothesis named $H$.

```
intro H.
```

```
H : true = false
================================
 False
```

This is the point in the proof where we apply some creativity. We define a function whose utility will become clear soon.

```
Definition toProp (b : bool) := if b then True else False.
```

It is worth recalling the difference between the lowercase and uppercase versions of truth and falsehood: **True** and **False** are logical propositions, whereas true and false are Boolean values that we can case-analyze. We have defined toProp such that the conclusion of **False** is computationally equivalent to toProp false. Thus, the `change` tactic lets us change the conclusion to toProp false. The general form `change` $e$ replaces the conclusion with $e$ whenever Coq's built-in computation rules suffice to establish the equivalence of $e$ with the original conclusion.

```
change (toProp false).
```

```
H : true = false
================================
 toProp false
```

Now the right side of $H$'s equality appears in the conclusion, so we can rewrite, using the notation $\leftarrow$ to request to replace the right side of the equality with the left side.

```
rewrite ← H.
```

```
H : true = false
================================
 toProp true
```

Some computational simplification reveals that we are almost done.

```
simpl.
```

```
H : true = false
================================
 True
```

```
trivial.
```

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

```
Qed.
```

I have no trivial automated version of this proof to suggest, beyond using `discriminate` or `congruence` in the first place.

We can perform a similar manual proof of injectivity of the constructor S. I leave the details to readers who want to run the proof script interactively.

```
Theorem S_inj' : ∀ n m : nat, S n = S m → n = m.
  intros n m H.
  change (pred (S n) = pred (S m)).
  rewrite H.
  reflexivity.
Qed.
```

The key in this theorem comes in using the natural number predecessor function pred. Embodied in the implementation of `injection` is a generic recipe for writing such type-specific functions.

The examples in this section illustrate an important aspect of the design philosophy behind Coq. We could certainly design a Gallina replacement with built-in rules for constructor discrimination and injectivity, but a simpler alternative is to include a few carefully chosen rules that enable the desired reasoning patterns and many others. A key benefit of this philosophy is that the complexity of proof checking is minimized, which bolsters confidence that proved theorems are really true.

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023

MIT Press Direct

© 2013 Massachusetts Institute of Technology

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

MIT Press books may be purchased at special quantity discounts for business or sales promotional use. For information, please email special_sales@mitpress.mit.edu or write to Special Sales Department, The MIT Press, 55 Hayward Street, Cambridge, MA 02142.

This book was set in 10/13 Lucida Bright by the author using LaTeX $2_\varepsilon$. Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Chlipala, Adam, 1981–
Certified programming with dependent types : a pragmatic introduction to the Coq proof assistant / Adam Chlipala.
p. cm
Includes bibliographical references and index.
ISBN 978-0-262-02665-9 (hardcover : alk. paper)
1. Automatic theorem proving–Computer programs. 2. Computer programming. 3. Coq (Electronic resource) I. Title.
QA76.9.A96C45 2013
005.1—dc23

2013012837

10 9 8 7 6 5 4 3 2 1

Downloaded from http://direct.mit.edu/books/oa-monograph/chapter-pdf/2091155/9780262317870_cac.pdf by guest on 05 October 2023