16 Proving in the Large

It is somewhat unfortunate that the term *theorem proving* looks so much like the word *theory*. Most researchers and practitioners in software assume that mechanized theorem proving is profoundly impractical. Indeed, until recently, most advances in theorem proving for higher-order logics have been largely theoretical. However, starting at the beginning of the twenty-first century, there was a surge in the use of proof assistants in serious verification efforts. That line of work is still quite new, but I believe it is not too soon to distill some lessons on how to work effectively with large formal proofs.

Thus, this chapter discusses structuring and maintaining large Coq developments.

16.1 Ltac Antipatterns

In this book, I follow an unusual style in which proofs are not considered finished until they are fully automated, in a certain sense. Each such theorem is proved by a single tactic. Since Ltac is a Turing-complete programming language, it is not hard to squeeze arbitrary heuristics into single tactics, using operators like the semicolon to combine steps. In contrast, most Ltac proofs "in the wild" consist of many steps, performed by individual tactics followed by periods. Is it really worth drawing a distinction between proof steps terminated by semicolons and steps terminated by periods?

I argue that this is, in fact, a very important distinction, with serious consequences for a majority of important verification domains. The more uninteresting drudge work a proof domain involves, the more important it is to prove theorems with single tactics. From an automation standpoint, single-tactic proofs can be extremely effective, and automation becomes more and more critical as proofs are populated by more uninteresting detail. In this section, I give some examples of the consequences of more common proof styles.

As a running example, consider a basic language of arithmetic expressions, an interpreter for it, and a transformation that scales up every constant in an expression.

```
Inductive exp : Set :=
| Const : nat → exp
| Plus : exp → exp → exp.

Fixpoint eval (e : exp) : nat :=
  match e with
    | Const n ⇒ n
    | Plus e1 e2 ⇒ eval e1 + eval e2
  end.

Fixpoint times (k : nat) (e : exp) : exp :=
  match e with
    | Const n ⇒ Const (k × n)
    | Plus e1 e2 ⇒ Plus (times k e1) (times k e2)
  end.
```

We can write a manual proof that times really implements multiplication.

```
Theorem eval_times : ∀ k e,
  eval (times k e) = k × eval e.
  induction e.

  trivial.

  simpl.
  rewrite IHe1.
  rewrite IHe2.
  rewrite mult_plus_distr_l.
  trivial.
Qed.
```

We use spaces to separate the two inductive cases, but note that these spaces have no real semantic content; Coq does not enforce that spacing matches the real case structure of a proof. The second case mentions automatically generated hypothesis names explicitly. As a result, innocuous changes to the theorem statement can invalidate the proof.

```
Reset eval_times.

Theorem eval_times : ∀ k x,
  eval (times k x) = k × eval x.
  induction x.
  trivial.
  simpl.
  rewrite IHe1.
```

```
Error: The reference IHe1 was not found in the current
environment.
```

The inductive hypotheses are named *IHx1* and *IHx2* now, not *IHe1* and *IHe2*.

```
Abort.
```

We might decide to use a more explicit invocation of `induction` to give explicit binders for all the names that will be referenced later in the proof.

```
Theorem eval_times : ∀ k e,
  eval (times k e) = k × eval e.
  induction e as [ | ? IHe1 ? IHe2 ].

  trivial.

  simpl.
  rewrite IHe1.
  rewrite IHe2.
  rewrite mult_plus_distr_l.
  trivial.
Qed.
```

We pass `induction` an *intro pattern*, using a | character to separate instructions for the different inductive cases. Within a case, we write ? to ask Coq to generate a name automatically, and we write an explicit name to assign that name to the corresponding new variable. It is apparent that in order to use intro patterns to avoid proof brittleness, we need to keep track of the seemingly unimportant facts of the orders in which variables are introduced. Thus, the script keeps working if we replace e by x, but it has become more cluttered. Arguably, neither proof is easy to follow.

That category of complaint has to do with understanding proofs as static artifacts. As with programming in general, with serious projects it is more important to be able to support evolution of proofs as specifications change. Unstructured proofs like the preceding examples can be very hard to update in concert with theorem statements. For instance, consider how the last proof script plays out when we modify `times` to introduce a bug.

```
Reset times.

Fixpoint times (k : nat) (e : exp) : exp :=
  match e with
    | Const n ⇒ Const (1 + k × n)
    | Plus e1 e2 ⇒ Plus (times k e1) (times k e2)
  end.

Theorem eval_times : ∀ k e,
  eval (times k e) = k × eval e.
  induction e as [ | ? IHe1 ? IHe2 ].

  trivial.

  simpl.

  rewrite IHe1.
```

```
Error: The reference IHe1 was not found in the current
environment.
```

```
Abort.
```

What went wrong? The problem is that `trivial` never fails. Originally, `trivial` had been succeeding in proving an equality that follows by reflexivity. The change to times leads to a case where that equality is no longer true. The invocation `trivial` leaves the false equality in place, and we continue on to the span of tactics intended for the second inductive case. Unfortunately, those tactics end up being applied to the *first* case instead.

The problem with `trivial` could be solved by writing `solve [ trivial ]` instead, for instance, so that an error is signaled early on if something unexpected happens. However, the root problem is that the syntax of a tactic invocation does not imply how many subgoals it produces. Even more confusing instances of this problem are possible. For example, if a lemma L is modified to take an extra hypothesis, then uses of `apply` L will generate more subgoals than before. Old unstructured proof scripts will become hopelessly jumbled, with tactics applied to inappropriate subgoals. Because of the lack of structure, there is usually relatively little to be gleaned from knowledge of the precise point in a proof script where an error is raised.

```
Reset times.

Fixpoint times (k : nat) (e : exp) : exp :=
  match e with
    | Const n ⇒ Const (k × n)
    | Plus e1 e2 ⇒ Plus (times k e1) (times k e2)
  end.
```

Many real developments try to make essentially unstructured proofs look structured by applying careful indentation conventions, idempotent case marker tactics included solely to serve as documentation, and so on. All these strategies suffer from the same kind of failure of abstraction that was just demonstrated. I like to say that if one finds oneself caring about indentation in a proof script, it is a sign that the script is structured poorly.

We can rewrite the current proof with a single tactic.

```
Theorem eval_times : ∀ k e,
  eval (times k e) = k × eval e.
  induction e as [ | ? IHe1 ? IHe2 ]; [
    trivial
    | simpl; rewrite IHe1; rewrite IHe2; rewrite mult_plus_distr_l;
      trivial ].
Qed.
```

We use the form of the semicolon operator that allows a different tactic to be specified for each generated subgoal. This change improves the robustness of the script: we no longer need to worry about tactics from one case being applied to a different case. Still, the proof script is not especially readable. Probably most readers would not find it helpful in explaining why the theorem is true. The same could be said for scripts using the *bullets* or curly braces provided by Coq 8.4, which allow code like this to be stepped through interactively, with periods in place of the semicolons, while representing proof structure in a way that is enforced by Coq. Interactive replay of scripts becomes easier, but readability is not really helped.

The situation gets worse in considering extensions to the theorem we want to prove. Let us add multiplication nodes to the **exp** type and see how the proof fares.

```
Reset exp.

Inductive exp : Set :=
| Const : nat → exp
| Plus : exp → exp → exp
| Mult : exp → exp → exp.

Fixpoint eval (e : exp) : nat :=
  match e with
    | Const n ⇒ n
    | Plus e1 e2 ⇒ eval e1 + eval e2
    | Mult e1 e2 ⇒ eval e1 × eval e2
  end.

Fixpoint times (k : nat) (e : exp) : exp :=
  match e with
    | Const n ⇒ Const (k × n)
    | Plus e1 e2 ⇒ Plus (times k e1) (times k e2)
    | Mult e1 e2 ⇒ Mult (times k e1) e2
  end.

Theorem eval_times : ∀ k e,
  eval (times k e) = k × eval e.
  induction e as [ | ? IHe1 ? IHe2 ]; [
    trivial
    | simpl; rewrite IHe1; rewrite IHe2; rewrite mult_plus_distr_l;
      trivial ].

Error: Expects a disjunctive pattern with 3 branches.

Abort.
```

Unsurprisingly, the old proof fails, because it explicitly says that there are two inductive cases. To update the script, we must, at a minimum, remember the order in which the inductive cases are generated, so that we can insert the new case in the appropriate place. Even then, it will be painful to add the case, because we cannot walk through proof steps interactively when they occur inside an explicit set of cases.

```
Theorem eval_times : ∀ k e,
  eval (times k e) = k × eval e.
  induction e as [ | ? IHe1 ? IHe2 | ? IHe1 ? IHe2 ]; [
    trivial
    | simpl; rewrite IHe1; rewrite IHe2; rewrite mult_plus_distr_l;
      trivial
    | simpl; rewrite IHe1; rewrite mult_assoc; trivial ].
Qed.
```

Now we are in a position to see that the style of proof followed in most of this book is preferable.

```
Reset eval_times.

Hint Rewrite mult_plus_distr_l.

Theorem eval_times : ∀ k e,
  eval (times k e) = k × eval e.
  induction e; crush.
Qed.
```

This style is motivated by a hard truth: one person's manual proof script is almost always inscrutable to everyone else. I claim that step-by-step formal proofs are a poor way of conveying information. Thus, we might as well cut out the steps and automate as much as possible.

What about the illustrative value of proofs? Most informal proofs are read to convey the big ideas of proofs. How can reading `induction` *e*; *crush* convey any big ideas? My position is that any ideas that standard automation can find are not very big, and big ideas should be expressed through lemmas that are added as hints.

An example should help illustrate what I mean. Consider this function, which rewrites an expression using associativity of addition and multiplication:

```
Fixpoint reassoc (e : exp) : exp :=
  match e with
    | Const _ ⇒ e
    | Plus e1 e2 ⇒
      let e1' := reassoc e1 in
      let e2' := reassoc e2 in
        match e2' with
          | Plus e21 e22 ⇒ Plus (Plus e1' e21) e22
          | _ ⇒ Plus e1' e2'
        end
    | Mult e1 e2 ⇒
      let e1' := reassoc e1 in
      let e2' := reassoc e2 in
        match e2' with
          | Mult e21 e22 ⇒ Mult (Mult e1' e21) e22
          | _ ⇒ Mult e1' e2'
        end
  end.

Theorem reassoc_correct : ∀ e, eval (reassoc e) = eval e.
  induction e; crush;
    match goal with
      | [ ⊢ context[match ?E with Const _ ⇒ _ | _ ⇒ _ end] ] ⇒
        destruct E; crush
    end.
```

One subgoal remains.

```
  IHe2 : eval e3 × eval e4 = eval e2
  ============================
   eval e1 × eval e3 × eval e4 = eval e1 × eval e2
```

The *crush* tactic does not know how to finish this goal. We could finish the proof manually.

```
  rewrite ← IHe2; crush.
```

However, the proof would be easier to understand and maintain if we separated this insight into a separate lemma.

```
Abort.

Lemma rewr : ∀ a b c d, b × c = d → a × b × c = a × d.
  crush.
Qed.

Hint Resolve rewr.

Theorem reassoc_correct : ∀ e, eval (reassoc e) = eval e.
  induction e; crush;
    match goal with
      | [ ⊢ context[match ?E with Const _ ⇒ _ | _ ⇒ _ end] ] ⇒
        destruct E; crush
    end.
Qed.
```

In the limit, a complicated inductive proof might rely on one hint for each inductive case. The lemma for each hint could restate the associated case. Compared to manual proof scripts, we arrive at more readable results. Scripts no longer need to depend on the order in which cases are generated. The lemmas are easier to digest separately than are fragments of tactic code, since lemma statements include complete proof contexts. Such contexts can only be extracted from monolithic manual proofs by stepping through scripts interactively.

The more common situation is that a large induction has several easy cases that automation makes short work of. In the remaining cases, automation performs some standard simplification. Among these cases, some may require quite involved proofs; such a case may deserve a hint lemma of its own, where the lemma statement may copy the simplified version of the case. Alternatively, the proof script for the main theorem may be extended with some automation code targeted at the specific case. Even such targeted scripting is more desirable than manual proving, because it may be read and understood without knowledge of a proof's hierarchical structure, case ordering, or name-binding structure.

A competing alternative to the common style of Coq tactics is the *declarative style*, most frequently associated today with the Isar [47] language. A declarative proof script is very explicit about subgoal structure and introduction of local names, aiming for human readability. The coding of proof automation is taken to be outside the scope of the proof language, an assumption related to the idea that it is not worth building new automation for each serious theorem. I have shown many examples of theorem-specific automation, which I believe is crucial for scaling to significant results. Declarative proof scripts make it easier to read scripts to modify them for theorem statement changes, but the alternative *adaptive style* in this book allows use of the *same* scripts for many versions of a theorem.

Perhaps I am a pessimist for thinking that fully formal proofs will inevitably consist of details that are uninteresting to people, but it is my preference to focus on conveying proof-specific details through choice of lemmas. Additionally, adaptive Ltac scripts contain bits of automation that can be understood in isolation. For instance, in a big `repeat match` loop, each case can generally be digested separately, which is a big contrast from trying to understand the hierarchical structure of a script in a more common style. Adaptive scripts rely on variable binding, but generally only over very small scopes, whereas understanding a traditional script requires tracking the identities of local variables potentially across pages of code.

One might also wonder why it makes sense to prove all theorems automatically (in the sense of adaptive proof scripts) but not construct all programs automatically. My view is that *program synthesis* is a very useful idea that deserves broader application. In practice, there are difficult obstacles in the way of finding a program automatically from its specification. A typical specification is not exhaustive in its description of program properties. For instance, details of performance on particular machine architectures are often omitted. As a result, a synthesized program may be correct in some sense while suffering from deficiencies in other senses. Program synthesis research will continue to come up with ways of dealing with this problem, but the situation for theorem proving is fundamentally different. Following mathematical practice, the only property of a formal proof that we care about is which theorem it proves, and it is trivial to check this property automatically. In other words, with a simple criterion for what makes a proof acceptable, automatic search is straightforward. Of course, in practice we also care about understandability of proofs to facilitate long-term maintenance, which is what motivates the techniques I have just outlined. The next section gives some related advice.

16.2 Debugging and Maintaining Automation

Fully automated proofs are desirable because they open up possibilities for automatic adaptation to changes of specification. A well-engineered script within a narrow domain can survive many changes to the formulation of the problem it solves. Still, when one works with higher-order logic, most theorems fall within no obvious decidable theories. It is inevitable that most long-lived automated proofs will need updating.

Before we are ready to update proofs, we need to write them. While fully automated scripts are most robust to changes of specification, it is hard to write every new proof directly in that form. Instead, it is useful to begin a theorem with exploratory proving and then gradually refine it into a suitable automated form.

Consider this theorem from Chapter 8, which we begin by proving in a mostly manual way, invoking *crush* after each step to discharge any simple cases first. The manual effort involves choosing which expressions to case-analyze on.

```
Theorem cfold_correct : ∀ t (e : exp t),
  expDenote e = expDenote (cfold e).
  induction e; crush.

  dep_destruct (cfold e1); crush.
  dep_destruct (cfold e2); crush.

  dep_destruct (cfold e1); crush.
  dep_destruct (cfold e2); crush.

  dep_destruct (cfold e1); crush.
  dep_destruct (cfold e2); crush.

  dep_destruct (cfold e1); crush.
  dep_destruct (expDenote e1); crush.

  dep_destruct (cfold e); crush.

  dep_destruct (cfold e); crush.
Qed.
```

In this complete proof, it is hard to avoid noticing a pattern. We rework the proof, abstracting over the patterns we find.

```
Reset cfold_correct.

Theorem cfold_correct : ∀ t (e : exp t),
  expDenote e = expDenote (cfold e).
  induction e; crush.
```

The expression we want to destruct here turns out to be the discriminee of a `match`, and we can easily write a tactic that destructs all such expressions.

```
Ltac t :=
  repeat (match goal with
            | [ ⊢ context[match ?E with NConst _ ⇒ _
                                    | _ ⇒ _ end] ] ⇒
              dep_destruct E
          end; crush).
```

```
t.
```

This tactic invocation discharges the whole case. It does the same on the next two cases, but it gets stuck on the fourth case.

```
t.
t.
t.
```

The subgoal's conclusion is

```
  ============================
 (if expDenote e1 then expDenote (cfold e2)
   else expDenote (cfold e3)) =
 expDenote (if expDenote e1 then cfold e2 else cfold e3)
```

We need to expand the *t* tactic to handle this case.

```
Ltac t' :=
  repeat (match goal with
            | [ ⊢ context[match ?E with NConst _ ⇒ _
                                    | _ ⇒ _ end] ] ⇒
              dep_destruct E
            | [ ⊢ (if ?E then _ else _) = _ ] ⇒ destruct E
          end; crush).
```

```
t'.
```

Now the goal is discharged, but *t'* has no effect on the next subgoal.

```
t'.
```

A final revision of *t* finishes the proof.

```
Ltac t'' :=
  repeat (match goal with
            | [ ⊢ context[match ?E with NConst _ ⇒ _
                                    | _ ⇒ _ end] ] ⇒
              dep_destruct E
```

```
                | [ ⊢ (if ?E then _ else _) = _ ] ⇒ destruct E
                | [ ⊢ context[match pairOut ?E with Some _ ⇒ _
                                        | None ⇒ _ end] ] ⇒
                  dep_destruct E
              end; crush).

  t''.

  t''.
Qed.
```

We can take the final tactic and move it into the initial part of the proof script, arriving at a nicely automated proof.

```
Reset t.

Theorem cfold_correct : ∀ t (e : exp t),
  expDenote e = expDenote (cfold e).
  induction e; crush;
    repeat (match goal with
                | [ ⊢ context[match ?E with NConst _ ⇒ _
                                        | _ ⇒ _ end] ] ⇒
                  dep_destruct E
                | [ ⊢ (if ?E then _ else _) = _ ] ⇒ destruct E
                | [ ⊢ context[match pairOut ?E with Some _ ⇒ _
                                        | None ⇒ _ end] ] ⇒
                  dep_destruct E
              end; crush).
Qed.
```

Even after we put together automated proofs, we must deal with specification changes that can invalidate them. It is not generally possible to step through single-tactic proofs interactively. There is a command `Debug On` that lets us step through points in tactic execution, but the debugger tends to make counterintuitive choices of which points we would like to stop at, and per-point output is quite verbose, so most Coq users do not find this debugging mode very helpful. How are we to understand what has broken in a script that used to work?

An example helps demonstrate a useful approach. Consider what would have happened in the proof of reassoc_correct if we had first added a misleading rewrite hint.

```
Reset reassoc_correct.

Theorem confounder : ∀ e1 e2 e3,
  eval e1 × eval e2 × eval e3 = eval e1 × (eval e2 + 1 - 1) × eval e3.
  crush.
```

```
Qed.

Hint Rewrite confounder.

Theorem reassoc_correct : ∀ e, eval (reassoc e) = eval e.
  induction e; crush;
    match goal with
      | [ ⊢ context[match ?E with Const _ ⇒ _ | _ ⇒ _ end] ] ⇒
        destruct E; crush
    end.
```

One subgoal remains.

```
  ============================
   eval e1 × (eval e3 + 1 - 1) × eval e4 = eval e1 × eval e2
```

The poorly chosen rewrite rule fired, changing the goal to a form where another hint no longer applies. Imagine that we are in the middle of a large development with many hints. How would we diagnose the problem? We might not be sure which case of the inductive proof has gone wrong. It is useful to separate out the automation procedure and apply it manually.

```
  Restart.

  Ltac t := crush; match goal with
                      | [ ⊢ context[match ?E with Const _ ⇒ _
                                                  | _ ⇒ _ end] ] ⇒
                        destruct E; crush
                    end.

  induction e.
```

Since we see the subgoals before any simplification occurs, it is clear that we are looking at the case for constants. Then we can make short work of it.

```
  t.
```

The next subgoal, for addition, is also discharged without trouble.

```
  t.
```

The final subgoal is for multiplication, and it is here that the proof got stuck.

```
  t.
```

What is *t* doing? The `info` command can help us find out. (In the most recent Coq release, `info` no longer functions, but I hope it returns.)

```
  Undo.
  info t.

== simpl in *; intuition; subst; autorewrite with core in *;
    simpl in *; intuition; subst; autorewrite with core in *;
    simpl in *; intuition; subst; destruct (reassoc e2).
    simpl in *; intuition.

    simpl in *; intuition.

    simpl in *; intuition; subst; autorewrite with core in *;
       refine (eq_ind_r
                  (fun n : nat ⇒
                   n × (eval e3 + 1 - 1) × eval e4
                   = eval e1 × eval e2) _ IHe1);
       autorewrite with core in *; simpl in *; intuition;
    subst; autorewrite with core in *; simpl in *;
    intuition; subst.
```

A detailed trace of *t*'s execution appears. Since we are using the very general *crush* tactic, many of these steps have no effect and only occur as instances of a more general strategy. We can copy-and-paste the details to see where things go wrong.

```
  Undo.
```

We arbitrarily split the script into chunks. The first few seem not to do any harm.

```
  simpl in *; intuition; subst; autorewrite with core in *.
  simpl in *; intuition; subst; autorewrite with core in *.
  simpl in *; intuition; subst; destruct (reassoc e2).
  simpl in *; intuition.
  simpl in *; intuition.
```

The next step is revealed as the culprit, bringing us to the final unproved subgoal.

```
  simpl in *; intuition; subst; autorewrite with core in *.
```

We can split the steps further to determine the problem.

```
  Undo.
```

```
  simpl in *.
  intuition.
  subst.
  autorewrite with core in *.
```

It is the final of these four tactics that made the rewrite. We can find out exactly what happened. The `info` command presents hierarchical views of proof steps, and we can zoom down to a lower level of detail by applying `info` to one of the steps in the original trace.

```
  Undo.

  info autorewrite with core in *.

 == refine (eq_ind_r (fun n : nat ⇒ n = eval e1 × eval e2) _
                (confounder (reassoc e1) e3 e4)).
```

We can see that theorem confounder is the final culprit. At this point, we could remove that hint, prove an alternative version of the key lemma rewr, or find some other remedy. Fixing this kind of problem tends to be relatively easy once the problem is revealed.

```
Abort.
```

Sometimes a change to a development has undesirable performance consequences, even if it does not prevent any proof scripts from completing. If the performance consequences are severe enough, the proof scripts can be considered broken for practical purposes.

Here is one example of a performance surprise:

```
Section slow.
  Hint Resolve eq_trans.
```

The central element of the problem is the addition of transitivity as a hint. With transitivity available, it is easy for proof search to wind up exploring exponential search spaces. We also add a few other arbitrary variables and hypotheses that will lead to trouble later.

```
  Variable A : Set.
  Variables P Q R S : A → A → Prop.
  Variable f : A → A.

  Hypothesis H1 : ∀ x y, P x y → Q x y → R x y → f x = f y.
  Hypothesis H2 : ∀ x y, S x y → R x y.
```

We prove a simple lemma very quickly, using the `Time` command to measure exactly how quickly.

```
  Lemma slow : ∀ x y, P x y → Q x y → S x y → f x = f y.
    Time eauto 6.

Finished transaction in 0. secs (0.068004u,0.s)
```

```
Qed.
```

Now we add a different hypothesis, which is innocent enough; in fact, it is even provable as a theorem.

```
Hypothesis H3 : ∀ x y, x = y → f x = f y.
Lemma slow' : ∀ x y, P x y → Q x y → S x y → f x = f y.
  Time eauto 6.
```

```
Finished transaction in 2. secs (1.264079u,0.s)
```

Why has the search time gone up so much? The `info` command is not much help, since it only shows the result of search, not all the paths that turned out to be worthless.

```
  Restart.
  info eauto 6.
```

```
 == intro x; intro y; intro H; intro H0; intro H4;
       simple eapply eq_trans.
    simple apply eq_refl.

    simple eapply eq_trans.
    simple apply eq_refl.

    simple eapply eq_trans.
    simple apply eq_refl.

    simple apply H1.
    eexact H.

    eexact H0.

    simple apply H2; eexact H4.
```

This output does not reveal why proof search takes so long, but it does provide a clue that is useful if we have forgotten that we added transitivity as a hint. The `eauto` tactic is applying depth-first search, and the relevant proof script is buried inside a chain of pointless invocations of transitivity, where each invocation uses reflexivity to discharge one subgoal. Each increment to the depth argument to `eauto` adds another unnecessary call to transitivity. This wasted proof effort only adds linear time overhead, as long as proof search never makes false steps. No false steps were made before we added the new hypothesis, but somehow the addition made possible a new faulty path. To understand which paths we enabled, we can use the `debug` command.

```
Restart.
debug eauto 6.
```

The output is a large proof tree. The beginning of the tree is enough to reveal what is happening:

```
1 depth=6
1.1 depth=6 intro
1.1.1 depth=6 intro
1.1.1.1 depth=6 intro
1.1.1.1.1 depth=6 intro
1.1.1.1.1.1 depth=6 intro
1.1.1.1.1.1.1 depth=5 apply H3
1.1.1.1.1.1.1.1 depth=4 eapply eq_trans
1.1.1.1.1.1.1.1.1 depth=4 apply eq_refl
1.1.1.1.1.1.1.1.1.1 depth=3 eapply eq_trans
1.1.1.1.1.1.1.1.1.1.1 depth=3 apply eq_refl
1.1.1.1.1.1.1.1.1.1.1.1 depth=2 eapply eq_trans
1.1.1.1.1.1.1.1.1.1.1.1.1 depth=2 apply eq_refl
1.1.1.1.1.1.1.1.1.1.1.1.1.1 depth=1 eapply eq_trans
1.1.1.1.1.1.1.1.1.1.1.1.1.1.1 depth=1 apply eq_refl
1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1 depth=0 eapply eq_trans
1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.2 depth=1 apply sym_eq ; trivial
1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.2.1 depth=0 eapply eq_trans
1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.3 depth=0 eapply eq_trans
1.1.1.1.1.1.1.1.1.1.1.1.1.2 depth=2 apply sym_eq ; trivial
1.1.1.1.1.1.1.1.1.1.1.1.1.2.1 depth=1 eapply eq_trans
1.1.1.1.1.1.1.1.1.1.1.1.1.2.1.1 depth=1 apply eq_refl
1.1.1.1.1.1.1.1.1.1.1.1.1.2.1.1.1 depth=0 eapply eq_trans
1.1.1.1.1.1.1.1.1.1.1.1.1.2.1.2 depth=1 apply sym_eq ; trivial
1.1.1.1.1.1.1.1.1.1.1.1.1.2.1.2.1 depth=0 eapply eq_trans
1.1.1.1.1.1.1.1.1.1.1.1.1.2.1.3 depth=0 eapply eq_trans
```

The first choice `eauto` makes is to apply *H3*, since *H3* has the fewest hypotheses of all the hypotheses and hints that match. However, it turns out that the single hypothesis generated is unprovable. That does not stop `eauto` from trying to prove it with an exponentially sized tree of applications of transitivity, reflexivity, and symmetry of equality. It is the children of the initial `apply` *H3* that account for all the additional time in proof execution. In a more realistic development, we might conclude from this output of `debug` that adding transitivity as a hint was a bad idea.

```
  Qed.
End slow.
```

Even greater problems can result from importing library modules with commands like `Require Import`. Such a command imports not just the Gallina terms from a module but also all the hints for `auto`, `eauto`, and `autorewrite`. Some very recent versions of Coq include mechanisms for removing hints from databases, but the proper solution is to be very conservative in exporting hints from modules. Consider putting hints in named databases, so that they may be used only when called upon explicitly, as demonstrated in Chapter 13.

It is also easy to end up with a proof script that uses too much memory. As tactics run, they avoid generating proof terms, since serious proof search will consider many possible avenues, and we do not want to build proof terms for subproofs that end up unused. Instead, tactic execution maintains *thunks* (suspended computations, represented with closures), such that a tactic's proof-producing thunk is only executed when we run `Qed`. These thunks can use up large amounts of space, such that a proof script exhausts available memory, even when we know that we could have used much less memory by forcing some thunks earlier.

The `abstract` tactical helps us force thunks by proving some subgoals as their own lemmas. For instance, a proof `induction` x; *crush* can in many cases be made to use significantly less peak memory by changing it to `induction` x; `abstract` *crush*. The main limitation of `abstract` is that it can only be applied to subgoals that are proved completely, with no undetermined unification variables in their initial states. Still, many large automated proofs can realize vast memory savings via `abstract`.

16.3 Modules

The examples in Chapter 15 of proof by reflection demonstrate opportunities for implementing abstract proof strategies with stronger formal guarantees than can be had with Ltac scripting. Coq's *module system* provides another tool for more rigorous development of generic theorems. This feature is inspired by the module systems found in Standard ML [22] and OCaml, and the discussion that follows assumes familiarity with the basics of one of those systems.

ML modules facilitate the grouping of abstract types with operations over those types. Moreover, there is support for *functors*, which are functions from modules to modules. A canonical example of a functor is one that builds a data structure implementation from a module that describes a domain of keys and its associated comparison operations.

When we add modules to a base language with dependent types, it becomes possible to use modules and functors to formalize kinds of reasoning that are common in algebra. For instance, the following module signature captures the essence of the algebraic structure known as a group. A group consists of a carrier set G, an associative binary operation f, a left identity element id for f, and an operation i that is a left inverse for f.

```
Module Type GROUP.
  Parameter G : Set.
  Parameter f : G → G → G.
  Parameter id : G.
  Parameter i : G → G.
  Axiom assoc : ∀ a b c, f (f a b) c = f a (f b c).
  Axiom ident : ∀ a, f id a = a.
  Axiom inverse : ∀ a, f (i a) a = id.
End GROUP.
```

Many useful theorems hold of arbitrary groups. We capture some such theorem statements in another module signature.

```
Module Type GROUP_THEOREMS.
  Declare Module M : GROUP.
  Axiom ident' : ∀ a, M.f a M.id = a.
  Axiom inverse' : ∀ a, M.f a (M.i a) = M.id.
  Axiom unique_ident : ∀ id', (∀ a, M.f id' a = a) → id' = M.id.
End GROUP_THEOREMS.
```

We implement generic proofs of these theorems with a functor, whose input is an arbitrary group *M*.

```
Module GroupProofs (M : GROUP) : GROUP_THEOREMS
  with Module M := M.
```

As in ML, Coq provides multiple options for ascribing signatures to modules. Here we use just the colon operator, which implements *opaque ascription*, hiding all details of the module not exposed by the signature. Another option is *transparent ascription* via the <: operator, which checks for signature compatibility without hiding implementation details. Here we stick with opaque ascription but employ the `with` operation to add more detail to a signature, exposing just those implementation details that we need to. For instance, here we expose the underlying group representation set and operator definitions. Without such a refinement, we would get an output module proving theorems about some unknown group, which is not very useful. Also note that opaque ascription can in Coq have some undesirable consequences without analogues in ML, since not just the types but also the *definitions* of identifiers have significance in type checking and theorem proving.

```
Module M := M.
```

To ensure that the module we are building meets its signature, we add an extra local name for *M*, the functor argument.

```
Import M.
```

It would be inconvenient to repeat the prefix *M*. everywhere in theorem statements and proofs, so we bring all the identifiers of *M* into the local scope unqualified.

Now we are ready to prove the three theorems. The proofs are completely manual, which may seem ironic given the content of the previous sections. Nonetheless, short proof scripts that change infrequently may be worth leaving unautomated. It would take some effort to build suitable generic automation for these theorems about groups, so I stick with manual proof scripts to avoid distracting from the main message here. We take the proofs from the Wikipedia page on elementary group theory.

```
Theorem inverse' : ∀ a, f a (i a) = id.
  intro.
  rewrite ← (ident (f a (i a))).
  rewrite ← (inverse (f a (i a))) at 1.
  rewrite assoc.
  rewrite assoc.
  rewrite ← (assoc (i a) a (i a)).
  rewrite inverse.
  rewrite ident.
  apply inverse.
Qed.

Theorem ident' : ∀ a, f a id = a.
  intro.
  rewrite ← (inverse a).
  rewrite ← assoc.
  rewrite inverse'.
  apply ident.
Qed.

Theorem unique_ident : ∀ id', (∀ a, f id' a = a) → id' = id.
  intros.
```

```
    rewrite ← (H id).
    symmetry.
    apply ident'.
  Qed.
End GroupProofs.
```

We can show that the integers with + form a group.

```
Require Import ZArith.
Open Scope Z_scope.

Module Int.
  Definition G := Z.
  Definition f x y := x + y.
  Definition id := 0.
  Definition i x := -x.

  Theorem assoc : ∀ a b c, f (f a b) c = f a (f b c).
    unfold f; crush.
  Qed.
  Theorem ident : ∀ a, f id a = a.
    unfold f, id; crush.
  Qed.
  Theorem inverse : ∀ a, f (i a) a = id.
    unfold f, i, id; crush.
  Qed.
End Int.
```

Next, we can produce integer-specific versions of the generic group theorems.

```
Module IntProofs := GroupProofs(Int).

Check IntProofs.unique_ident.

  IntProofs.unique_ident
      : ∀ e' : Int.G, (∀ a : Int.G, Int.f e' a = a) → e' = Int.e
```

Projections like *Int.G* are known to be definitionally equal to the concrete values we have assigned to them, so this theorem yields as a trivial corollary the following more natural restatement.

```
Theorem unique_ident : ∀ id', (∀ a, id' + a = a) → id' = 0.
  exact IntProofs.unique_ident.
Qed.
```

As in ML, the module system provides an effective way to structure large developments. Unlike in ML, Coq modules add no expressiveness; we can implement any module as an inhabitant of a dependent record type. It is the second-class nature of modules that makes them easier to use than dependent records in many cases. Because modules may only be used in quite restricted ways, it is easier to support convenient module coding through special commands and editing modes, as the preceding example demonstrates. An isomorphic implementation with records would have suffered from lack of such conveniences as module subtyping and importation of the fields of a module. On the other hand, all module values must be determined statically, so modules may not be computed, for instance, within the definitions of normal functions, based on particular function parameters.

16.4 Build Processes

As in software development, large Coq projects are much more manageable when split across multiple files and when decomposed into libraries. Coq and Proof General provide very good support for these activities.

Consider a library that I name Lib, housed in directory `LIB` and split between files `A.v`, `B.v`, and `C.v`. A simple Makefile will compile the library, relying on the standard Coq tool `coq_makefile` to do the hard work.

```
MODULES := A B C
VS      := $(MODULES:%=%.v)

.PHONY: coq clean

coq: Makefile.coq
        $(MAKE) -f Makefile.coq

Makefile.coq: Makefile $(VS)
        coq_makefile -R . Lib $(VS) -o Makefile.coq

clean:: Makefile.coq
        $(MAKE) -f Makefile.coq clean
        rm -f Makefile.coq
```

The Makefile begins by defining a variable `VS` holding the list of filenames to be included in the project. The primary target is `coq`, which depends on the construction of an auxiliary Makefile called `Makefile.coq`. Another rule explains how to build that file. We call `coq_makefile`, using the `-R` flag to specify that files in the current directory should be considered to belong to the library Lib. This Makefile will build a compiled version of each module, such that `X.v` is compiled into `X.vo`.

Now code in `B.v` may refer to definitions in `A.v` after running

`Require Import` LIB.A.

Library LIB is presented as a module, containing a submodule *A*, which contains the definitions from `A.v`. These are genuine modules in the sense of Coq's module system, and they may be passed to functors, and so on.

The command `Require Import` is a convenient combination of two more primitive commands. The `Require` command finds the `.vo` file containing the named module, ensuring that the module is loaded into memory. The `Import` command loads all top-level definitions of the named module into the current namespace, and it may be used with local modules that do not have corresponding `.vo` files. Another command, `Load`, is for inserting the contents of a named file verbatim. It is generally better to use the module-based commands, since they avoid rerunning proof scripts, and they facilitate reorganization of directory structure without the need to change code.

Now we would like to use the library from a different development, called CLIENT and found in directory `CLIENT`, which has its own Makefile.

```
MODULES := D E
VS      := $(MODULES:%=%.v)

.PHONY: coq clean

coq: Makefile.coq
        $(MAKE) -f Makefile.coq

Makefile.coq: Makefile $(VS)
        coq_makefile -R LIB Lib -R . Client $(VS) \
                -o Makefile.coq

clean:: Makefile.coq
        $(MAKE) -f Makefile.coq clean
        rm -f Makefile.coq
```

We change the `coq_makefile` call to indicate where the library LIB is found. Now `D.v` and `E.v` can refer to definitions from LIB module *A* after running

`Require Import` LIB.A.

and `E.v` can refer to definitions from `D.v` by running

`Require Import` CLIENT.D.

It can be useful to split a library into several files, but it is also inconvenient for client code to import library modules individually. We can get the best of both worlds by, for example, adding an extra source file `Lib.v` to LIB's directory and Makefile, where that file contains just this line:

`Require Export` LIB.A LIB.B LIB.C.

Now client code can import all definitions from all of LIB's modules simply by running

`Require Import` LIB.

The two Makefiles above share a lot of code, so, in practice, it is useful to define a common Makefile that is included by multiple library-specific Makefiles.

The remaining ingredient is the proper way of editing library code files in Proof General. Recall this snippet of `.emacs` code from Chapter 1, which tells Proof General where to find the library associated with this book:

```
(custom-set-variables
  ...
  '(coq-prog-args '("-I" "/path/to/cpdt/src"))
  ...
)
```

To do interactive editing of the current example, we just need to change the flags to point to the right places.

```
(custom-set-variables
  ...
; '(coq-prog-args '("-I" "/path/to/cpdt/src"))
  '(coq-prog-args '("-R" "LIB" "Lib" "-R" "CLIENT" "Client"))
  ...
)
```

When working on multiple projects, it is useful to leave multiple versions of this setting in the `.emacs` file, commenting out all but one of them at any moment. To switch between projects, change the commenting structure and restart Emacs.

Alternatively, we can revisit the directory-local settings approach and write the following into a file .dir-locals.el in CLIENT.

```
((coq-mode . ((coq-prog-args .
  ("-emacs-U" "-R" "LIB" "Lib" "-R" "CLIENT" "Client")))))
```

A downside of this approach is that users of the code may not want to trust the arbitrary Emacs Lisp programs that are allowed to be placed in such files and prefer to add mappings manually.