AUTO

MORE AUTOMATION

```
Set Warnings "-notation-overridden,-parsing".
Require Import Coq.omega.Omega.
Require Import Maps.
Require Import Imp.
```

Up to now, we've used the more manual part of Coq's tactic facilities. In this chapter, we'll learn more about some of Coq's powerful automation features: proof search via the `auto` tactic, automated forward reasoning via the `Ltac` hypothesis matching machinery, and deferred instantiation of existential variables using `eapply` and `eauto`. Using these features together with Ltac's scripting facilities will enable us to make our proofs startlingly short! Used properly, they can also make proofs more maintainable and robust to changes in underlying definitions. A deeper treatment of `auto` and `eauto` can be found in the `UseAuto` chapter in *Programming Language Foundations*.

There's another major category of automation we haven't discussed much yet, namely built-in decision procedures for specific kinds of problems: `omega` is one example, but there are others. This topic will be deferred for a while longer.

Our motivating example will be this proof, repeated with just a few small changes from the Imp chapter. We will simplify this proof in several stages.

First, define a little Ltac macro to compress a common pattern into a single command.

```
Ltac inv H := inversion H; subst; clear H.

Theorem ceval_deterministic: ∀ c st st1 st2,
     c / st \\ st1 →
     c / st \\ st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2;
  generalize dependent st2;
  induction E1; intros st2 E2; inv E2.
  - (* E_Skip *) reflexivity.
```

```
  - (* E_Ass *) reflexivity.
  - (* E_Seq *)
    assert (st' = st'0) as EQ1.
    { (* Proof of assertion *) apply IHE1_1; apply H1. }
    subst st'0.
    apply IHE1_2. assumption.
  (* E_IfTrue *)
  - (* b evaluates to true *)
    apply IHE1. assumption.
  - (* b evaluates to false (contradiction) *)
    rewrite H in H5. inversion H5.
  (* E_IfFalse *)
  - (* b evaluates to true (contradiction) *)
    rewrite H in H5. inversion H5.
  - (* b evaluates to false *)
    apply IHE1. assumption.
  (* E_WhileFalse *)
  - (* b evaluates to false *)
    reflexivity.
  - (* b evaluates to true (contradiction) *)
    rewrite H in H2. inversion H2.
  (* E_WhileTrue *)
  - (* b evaluates to false (contradiction) *)
    rewrite H in H4. inversion H4.
  - (* b evaluates to true *)
    assert (st' = st'0) as EQ1.
    { (* Proof of assertion *) apply IHE1_1; assumption. }
    subst st'0.
    apply IHE1_2. assumption. Qed.
```

The auto Tactic

Thus far, our proof scripts mostly apply relevant hypotheses or lemmas by name, and one at a time.

```
Example auto_example_1 : ∀ (P Q R: Prop),
  (P → Q) → (Q → R) → P → R.
Proof.
  intros P Q R H1 H2 H3.
  apply H2. apply H1. assumption.
Qed.
```

The `auto` tactic frees us from this drudgery by *searching* for a sequence of applications that will prove the goal:

```
Example auto_example_1' : ∀ (P Q R: Prop),
  (P → Q) → (Q → R) → P → R.
Proof.
  auto.
Qed.
```

The `auto` tactic solves goals that are solvable by any combination of

- `intros` and
- `apply` (of hypotheses from the local context, by default).

Using `auto` is always "safe" in the sense that it will never fail and will never change the proof state: either it completely solves the current goal, or it does nothing.

Here is a more interesting example showing `auto`'s power:

```
Example auto_example_2 : ∀ P Q R S T U : Prop,
  (P → Q) →
  (P → R) →
  (T → R) →
  (S → T → U) →
  ((P→Q) → (P→S)) →
  T →
  P →
  U.
Proof. auto. Qed.
```

Proof search could, in principle, take an arbitrarily long time, so there are limits to how far `auto` will search by default.

```
Example auto_example_3 : ∀ (P Q R S T U: Prop),
  (P → Q) →
  (Q → R) →
  (R → S) →
  (S → T) →
  (T → U) →
  P →
  U.
Proof.
  (* When it cannot solve the goal, auto does nothing *)
  auto.
  (* Optional argument says how deep to search (default is 5) *)
  auto 6.
Qed.
```

When searching for potential proofs of the current goal, `auto` considers the hypotheses in the current context together with a *hint database* of other lemmas and constructors. Some common lemmas about equality and logical operators are installed in this hint database by default.

```
Example auto_example_4 : ∀ P Q R : Prop,
  Q →
  (Q → R) →
  P ∨ (Q ∧ R).
Proof. auto. Qed.
```

We can extend the hint database just for the purposes of one application of `auto` by writing "`auto using ...`".

```
Lemma le_antisym : ∀ n m: nat, (n ≤ m ∧ m ≤ n) → n = m.
Proof. intros. omega. Qed.

Example auto_example_6 : ∀ n m p : nat,
  (n ≤ p → (n ≤ m ∧ m ≤ n)) →
```

```
  n ≤ p →
  n = m.
Proof.
  intros.
  auto using le_antisym.
Qed.
```

Of course, in any given development there will probably be some specific constructors and lemmas that are used very often in proofs. We can add these to the global hint database by writing

```
Hint Resolve T.
```

at the top level, where `T` is a top-level theorem or a constructor of an inductively defined proposition (i.e., anything whose type is an implication). As a shorthand, we can write

```
Hint Constructors c.
```

to tell Coq to do a `Hint Resolve` for *all* of the constructors from the inductive definition of `c`.

It is also sometimes necessary to add

```
Hint Unfold d.
```

where `d` is a defined symbol, so that `auto` knows to expand uses of `d`, thus enabling further possibilities for applying lemmas that it knows about.

It is also possible to define specialized hint databases that can be activated only when needed. See the Coq reference manual for more.

```
Hint Resolve le_antisym.

Example auto_example_6' : ∀ n m p : nat,
  (n≤ p → (n ≤ m ∧ m ≤ n)) →
  n ≤ p →
  n = m.
Proof.
  intros.
  auto. (* picks up hint from database *)
Qed.

Definition is_fortytwo x := (x = 42).

Example auto_example_7: ∀ x,
  (x ≤ 42 ∧ 42 ≤ x) → is_fortytwo x.
Proof.
  auto. (* does nothing *)
Abort.

Hint Unfold is_fortytwo.

Example auto_example_7' : ∀ x,
  (x ≤ 42 ∧ 42 ≤ x) → is_fortytwo x.
Proof. auto. Qed.
```

Let's take a first pass over `ceval_deterministic` to simplify the proof script.

```
Theorem ceval_deterministic': ∀ c st st₁ st₂,
     c / st \\ st₁ →
     c / st \\ st₂ →
     st₁ = st₂.
Proof.
  intros c st st₁ st₂ E₁ E₂.
  generalize dependent st₂;
       induction E₁; intros st₂ E₂; inv E₂; auto.
  - (* E_Seq *)
    assert (st' = st'0) as EQ₁ by auto.
    subst st'0.
    auto.
  - (* E_IfTrue *)
    + (* b evaluates to false (contradiction) *)
      rewrite H in H₅. inversion H₅.
  - (* E_IfFalse *)
    + (* b evaluates to true (contradiction) *)
      rewrite H in H₅. inversion H₅.
  - (* E_WhileFalse *)
    + (* b evaluates to true (contradiction) *)
      rewrite H in H₂. inversion H₂.
  (* E_WhileTrue *)
  - (* b evaluates to false (contradiction) *)
    rewrite H in H₄. inversion H₄.
  - (* b evaluates to true *)
    assert (st' = st'0) as EQ₁ by auto.
    subst st'0.
    auto.
Qed.
```

When we are using a particular tactic many times in a proof, we can use a variant of the `Proof` command to make that tactic into a default within the proof. Saying `Proof with t` (where `t` is an arbitrary tactic) allows us to use t_1`...` as a shorthand for t_1`;t` within the proof. As an illustration, here is an alternate version of the previous proof, using `Proof with auto`.

```
Theorem ceval_deterministic'_alt: ∀ c st st₁ st₂,
     c / st \\ st₁ →
     c / st \\ st₂ →
     st₁ = st₂.
+
```

Searching For Hypotheses

The proof has become simpler, but there is still an annoying amount of repetition. Let's start by tackling the contradiction cases. Each of them occurs in a situation where

we have both

```
H1: beval st b = false
```

and

```
H2: beval st b = true
```

as hypotheses. The contradiction is evident, but demonstrating it is a little complicated: we have to locate the two hypotheses `H1` and `H2` and do a `rewrite` following by an `inversion`. We'd like to automate this process.

(In fact, Coq has a built-in tactic `congruence` that will do the job in this case. But we'll ignore the existence of this tactic for now, in order to demonstrate how to build forward search tactics by hand.)

As a first step, we can abstract out the piece of script in question by writing a little function in Ltac.

```
Ltac rwinv H1 H2 := rewrite H1 in H2; inv H2.

Theorem ceval_deterministic'': ∀ c st st1 st2,
     c / st \\ st1 →
     c / st \\ st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2;
  induction E1; intros st2 E2; inv E2; auto.
  - (* E_Seq *)
    assert (st' = st'0) as EQ1 by auto.
    subst st'0.
    auto.
  - (* E_IfTrue *)
    + (* b evaluates to false (contradiction) *)
      rwinv H H5.
  - (* E_IfFalse *)
    + (* b evaluates to true (contradiction) *)
      rwinv H H5.
  - (* E_WhileFalse *)
    + (* b evaluates to true (contradiction) *)
      rwinv H H2.
  (* E_WhileTrue *)
  - (* b evaluates to false (contradiction) *)
    rwinv H H4.
  - (* b evaluates to true *)
    assert (st' = st'0) as EQ1 by auto.
    subst st'0.
    auto. Qed.
```

That was is a bit better, but we really want Coq to discover the relevant hypotheses for us. We can do this by using the `match goal` facility of Ltac.

```
Ltac find_rwinv :=
  match goal with
    H1: ?E = true,
    H2: ?E = false
    |- _ ⇒ rwinv H1 H2
  end.
```

This `match goal` looks for two distinct hypotheses that have the form of equalities, with the same arbitrary expression `E` on the left and with conflicting boolean values on the right. If such hypotheses are found, it binds H_1 and H_2 to their names and applies the `rwinv` tactic to H_1 and H_2.

Adding this tactic to the ones that we invoke in each case of the induction handles all of the contradictory cases.

```
Theorem ceval_deterministic''': ∀ c st st1 st2,
     c / st \\ st1 →
     c / st \\ st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2;
  induction E1; intros st2 E2; inv E2; try find_rwinv; auto.
  - (* E_Seq *)
    assert (st' = st'0) as EQ1 by auto.
    subst st'0.
    auto.
  - (* E_WhileTrue *)
    + (* b evaluates to true *)
      assert (st' = st'0) as EQ1 by auto.
      subst st'0.
      auto. Qed.
```

Let's see about the remaining cases. Each of them involves applying a conditional hypothesis to extract an equality. Currently we have phrased these as assertions, so that we have to predict what the resulting equality will be (although we can then use `auto` to prove it). An alternative is to pick the relevant hypotheses to use and then `rewrite` with them, as follows:

```
Theorem ceval_deterministic'''': ∀ c st st1 st2,
     c / st \\ st1 →
     c / st \\ st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2;
  induction E1; intros st2 E2; inv E2; try find_rwinv; auto.
  - (* E_Seq *)
    rewrite (IHE1_1 st'0 H1) in *. auto.
  - (* E_WhileTrue *)
```

```
    + (* b evaluates to true *)
      rewrite (IHE1_1 st'0 H3) in *. auto. Qed.
```

Now we can automate the task of finding the relevant hypotheses to rewrite with.

```
Ltac find_eqn :=
  match goal with
    H1: ∀ x, ?P x → ?L = ?R,
    H2: ?P ?X
    |- _ ⇒ rewrite (H1 X H2) in *
  end.
```

The pattern `∀x, ?P x → ?L = ?R` matches any hypothesis of the form "for all `x`, *some property of `x`* implies *some equality*." The property of `x` is bound to the pattern variable `P`, and the left- and right-hand sides of the equality are bound to `L` and `R`. The name of this hypothesis is bound to H_1. Then the pattern `?P ?X` matches any hypothesis that provides evidence that `P` holds for some concrete `X`. If both patterns succeed, we apply the `rewrite` tactic (instantiating the quantified `x` with `X` and providing H_2 as the required evidence for `P X`) in all hypotheses and the goal.

One problem remains: in general, there may be several pairs of hypotheses that have the right general form, and it seems tricky to pick out the ones we actually need. A key trick is to realize that we can *try them all*! Here's how this works:

- each execution of `match goal` will keep trying to find a valid pair of hypotheses until the tactic on the RHS of the match succeeds; if there are no such pairs, it fails;
- `rewrite` will fail given a trivial equation of the form `X = X`;
- we can wrap the whole thing in a `repeat`, which will keep doing useful rewrites until only trivial ones are left.

```
Theorem ceval_deterministic''''': ∀ c st st1 st2,
     c / st \\ st1 →
     c / st \\ st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2;
  induction E1; intros st2 E2; inv E2; try find_rwinv;
    repeat find_eqn; auto.
Qed.
```

The big payoff in this approach is that our proof script should be more robust in the face of modest changes to our language. To test whether it really is, let's try adding a `REPEAT` command to the language.

```
Module Repeat.

Inductive com : Type :=
  | CSkip : com
```

```
  | CAsgn : string → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com
  | CRepeat : com → bexp → com.
```

`REPEAT` behaves like `WHILE`, except that the loop guard is checked *after* each execution of the body, with the loop repeating as long as the guard stays *false*. Because of this, the body will always execute at least once.

```
Notation "'SKIP'" :=
  CSkip.
Notation "c₁ ; c₂" :=
  (CSeq c₁ c₂) (at level 80, right associativity).
Notation "X '::=' a" :=
  (CAsgn X a) (at level 60).
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity).
Notation "'IFB' e₁ 'THEN' e₂ 'ELSE' e₃ 'FI'" :=
  (CIf e₁ e₂ e₃) (at level 80, right associativity).
Notation "'REPEAT' e₁ 'UNTIL' b₂ 'END'" :=
  (CRepeat e₁ b₂) (at level 80, right associativity).

Inductive ceval : state → com → state → Prop :=
  | E_Skip : ∀ st,
      ceval st SKIP st
  | E_Ass : ∀ st a₁ n X,
      aeval st a₁ = n →
      ceval st (X ::= a₁) (t_update st X n)
  | E_Seq : ∀ c₁ c₂ st st' st'',
      ceval st c₁ st' →
      ceval st' c₂ st'' →
      ceval st (c₁ ; c₂) st''
  | E_IfTrue : ∀ st st' b₁ c₁ c₂,
      beval st b₁ = true →
      ceval st c₁ st' →
      ceval st (IFB b₁ THEN c₁ ELSE c₂ FI) st'
  | E_IfFalse : ∀ st st' b₁ c₁ c₂,
      beval st b₁ = false →
      ceval st c₂ st' →
      ceval st (IFB b₁ THEN c₁ ELSE c₂ FI) st'
  | E_WhileFalse : ∀ b₁ st c₁,
      beval st b₁ = false →
      ceval st (WHILE b₁ DO c₁ END) st
  | E_WhileTrue : ∀ st st' st'' b₁ c₁,
      beval st b₁ = true →
      ceval st c₁ st' →
      ceval st' (WHILE b₁ DO c₁ END) st'' →
      ceval st (WHILE b₁ DO c₁ END) st''
```

```
  | E_RepeatEnd : ∀ st st' b1 c1,
      ceval st c1 st' →
      beval st' b1 = true →
      ceval st (CRepeat c1 b1) st'
  | E_RepeatLoop : ∀ st st' st'' b1 c1,
      ceval st c1 st' →
      beval st' b1 = false →
      ceval st' (CRepeat c1 b1) st'' →
      ceval st (CRepeat c1 b1) st''.

Notation "c1 '/' st '\\' st'" := (ceval st c1 st')
                                 (at level 40, st at level 39).
```

Our first attempt at the determinacy proof does not quite succeed: the `E_RepeatEnd` and `E_RepeatLoop` cases are not handled by our previous automation.

```
Theorem ceval_deterministic: ∀ c st st1 st2,
     c / st \\ st1 →
     c / st \\ st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2;
  induction E1;
    intros st2 E2; inv E2; try find_rwinv; repeat find_eqn; auto.
  - (* E_RepeatEnd *)
    + (* b evaluates to false (contradiction) *)
      find_rwinv.
      (* oops: why didn't find_rwinv solve this for us already?
         answer: we did things in the wrong order. *)
  - (* E_RepeatLoop *)
     + (* b evaluates to true (contradiction) *)
       find_rwinv.
Qed.
```

Fortunately, to fix this, we just have to swap the invocations of `find_eqn` and `find_rwinv`.

```
Theorem ceval_deterministic': ∀ c st st1 st2,
     c / st \\ st1 →
     c / st \\ st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2;
  induction E1;
    intros st2 E2; inv E2; repeat find_eqn; try find_rwinv; auto.
Qed.

End Repeat.
```

These examples just give a flavor of what "hyper-automation" can achieve in Coq. The details of `match goal` are a bit tricky (and debugging scripts using it is, frankly, not very pleasant). But it is well worth adding at least simple uses to your proofs, both to avoid tedium and to "future proof" them.

The `eapply` and `eauto` variants

To close the chapter, we'll introduce one more convenient feature of Coq: its ability to delay instantiation of quantifiers. To motivate this feature, recall this example from the Imp chapter:

```
Example ceval_example1:
    (X ::= 2;;
     IFB X ≤ 1
       THEN Y ::= 3
       ELSE Z ::= 4
     FI)
   / { --> 0 }
   \\ { X --> 2 ; Z --> 4 }.
Proof.
  (* We supply the intermediate state st'... *)
  apply E_Seq with { X --> 2 }.
  - apply E_Ass. reflexivity.
  - apply E_IfFalse. reflexivity. apply E_Ass. reflexivity.
Qed.
```

In the first step of the proof, we had to explicitly provide a longish expression to help Coq instantiate a "hidden" argument to the `E_Seq` constructor. This was needed because the definition of `E_Seq`...

```
E_Seq : ∀ c1 c2 st st' st'',
  c1 / st  \\ st' →
  c2 / st' \\ st'' →
  (c1 ;; c2) / st \\ st''
```

is quantified over a variable, `st'`, that does not appear in its conclusion, so unifying its conclusion with the goal state doesn't help Coq find a suitable value for this variable. If we leave out the `with`, this step fails ("Error: Unable to find an instance for the variable `st'`").

What's silly about this error is that the appropriate value for `st'` will actually become obvious in the very next step, where we apply `E_Ass`. If Coq could just wait until we get to this step, there would be no need to give the value explicitly. This is exactly what the `eapply` tactic gives us:

```
Example ceval'_example1:
    (X ::= 2;;
     IFB X ≤ 1
       THEN Y ::= 3
       ELSE Z ::= 4
     FI)
```

```
    / { --> 0 }
    \\ { X --> 2 ; Z --> 4 }.
Proof.
  eapply E_Seq. (* 1 *)
  - apply E_Ass. (* 2 *)
    reflexivity. (* 3 *)
  - (* 4 *) apply E_IfFalse. reflexivity. apply E_Ass.
reflexivity.
Qed.
```

The `eapply H` tactic behaves just like `apply H` except that, after it finishes unifying the goal state with the conclusion of `H`, it does not bother to check whether all the variables that were introduced in the process have been given concrete values during unification.

If you step through the proof above, you'll see that the goal state at position `1` mentions the *existential variable* `?st'` in both of the generated subgoals. The next step (which gets us to position `2`) replaces `?st'` with a concrete value. This new value contains a new existential variable `?n`, which is instantiated in its turn by the following `reflexivity` step, position `3`. When we start working on the second subgoal (position `4`), we observe that the occurrence of `?st'` in this subgoal has been replaced by the value that it was given during the first subgoal.

Several of the tactics that we've seen so far, including `∃`, `constructor`, and `auto`, have `e...` variants. For example, here's a proof using `eauto`:

```
Hint Constructors ceval.
Hint Transparent state.
Hint Transparent total_map.

Definition st12 := { X --> 1 ; Y --> 2 }.
Definition st21 := { X --> 2 ; Y --> 1 }.

Example eauto_example : ∃ s',
  (IFB X ≤ Y
    THEN Z ::= Y - X
    ELSE Y ::= X + Z
  FI) / st21 \\ s'.
Proof. eauto. Qed.
```

The `eauto` tactic works just like `auto`, except that it uses `eapply` instead of `apply`.

Pro tip: One might think that, since `eapply` and `eauto` are more powerful than `apply` and `auto`, it would be a good idea to use them all the time. Unfortunately, they are also significantly slower — especially `eauto`. Coq experts tend to use `apply` and `auto` most of the time, only switching to the `e` variants when the ordinary variants don't do the job.