SOFTWARE FOUNDATIONS

VOLUME 2: PROGRAMMING LANGUAGE FOUNDATIONS

USEAUTO

THEORY AND PRACTICE OF AUTOMATION IN COQ PROOFS

```
(* Chapter written and maintained by Arthur Chargueraud *)
```

In a machine-checked proof, every single detail has to be justified. This can result in huge proof scripts. Fortunately, Coq comes with a proof-search mechanism and with several decision procedures that enable the system to automatically synthesize simple pieces of proof. Automation is very powerful when set up appropriately. The purpose of this chapter is to explain the basics of working of automation.

The chapter is organized in two parts. The first part focuses on a general mechanism called "proof search." In short, proof search consists in naively trying to apply lemmas and assumptions in all possible ways. The second part describes "decision procedures", which are tactics that are very good at solving proof obligations that fall in some particular fragment of the logic of Coq.

Many of the examples used in this chapter consist of small lemmas that have been made up to illustrate particular aspects of automation. These examples are completely independent from the rest of the Software Foundations course. This chapter also contains some bigger examples which are used to explain how to use automation in realistic proofs. These examples are taken from other chapters of the course (mostly from STLC), and the proofs that we present make use of the tactics from the library `LibTactics.v`, which is presented in the chapter `UseTactics`.

```
Require Import Coq.Arith.Arith.

Require Import Maps.
Require Import Smallstep.
Require Import Stlc.
Require Import LibTactics.

Require Imp.

Require Import Coq.Lists.List.
Import ListNotations.
```

Basic Features of Proof Search

The idea of proof search is to replace a sequence of tactics applying lemmas and assumptions with a call to a single tactic, for example `auto`. This form of proof automation saves a lot of effort. It typically leads to much shorter proof scripts, and to scripts that are typically more robust to change. If one makes a little change to a definition, a proof that exploits automation probably won't need to be modified at all. Of course, using too much automation is a bad idea. When a proof script no longer records the main arguments of a proof, it becomes difficult to fix it when it gets broken after a change in a definition. Overall, a reasonable use of automation is generally a big win, as it saves a lot of time both in building proof scripts and in subsequently maintaining those proof scripts.

Strength of Proof Search

We are going to study four proof-search tactics: `auto`, `eauto`, `iauto` and `jauto`. The tactics `auto` and `eauto` are builtin in Coq. The tactic `iauto` is a shorthand for the builtin tactic `try solve [intuition eauto]`. The tactic `jauto` is defined in the library `LibTactics`, and simply performs some preprocessing of the goal before calling `eauto`. The goal of this chapter is to explain the general principles of proof search and to give rule of thumbs for guessing which of the four tactics mentioned above is best suited for solving a given goal.

Proof search is a compromise between efficiency and expressiveness, that is, a tradeoff between how complex goals the tactic can solve and how much time the tactic requires for terminating. The tactic `auto` builds proofs only by using the basic tactics `reflexivity`, `assumption`, and `apply`. The tactic `eauto` can also exploit `eapply`. The tactic `jauto` extends `eauto` by being able to open conjunctions and existentials that occur in the context. The tactic `iauto` is able to deal with conjunctions, disjunctions, and negation in a quite clever way; however it is not able to open existentials from the context. Also, `iauto` usually becomes very slow when the goal involves several disjunctions.

Note that proof search tactics never perform any rewriting step (tactics `rewrite`, `subst`), nor any case analysis on an arbitrary data structure or property (tactics `destruct` and `inversion`), nor any proof by induction (tactic `induction`). So, proof search is really intended to automate the final steps from the various branches of a proof. It is not able to discover the overall structure of a proof.

Basics

The tactic `auto` is able to solve a goal that can be proved using a sequence of `intros`, `apply`, `assumption`, and `reflexivity`. Two examples follow. The first one shows the ability for `auto` to call `reflexivity` at any time. In fact, calling `reflexivity` is always the first thing that `auto` tries to do.

```
Lemma solving_by_reflexivity :
  2 + 3 = 5.
Proof. auto. Qed.
```

The second example illustrates a proof where a sequence of two calls to `apply` are needed. The goal is to prove that if `Q n` implies `P n` for any `n` and if `Q n` holds for any `n`, then `P 2` holds.

```
Lemma solving_by_apply : ∀ (P Q : nat→Prop),
  (∀ n, Q n → P n) →
  (∀ n, Q n) →
  P 2.
Proof. auto. Qed.
```

If we are interested to see which proof `auto` came up with, one possibility is to look at the generated proof-term, using the command:

`Print solving_by_apply.`

The proof term is:

`fun (P Q : nat → Prop) (H : ∀n : nat, Q n → P n) (H0 : ∀n : nat, Q n) ⇒ H 2 (H0 2)`

This essentially means that `auto` applied the hypothesis `H` (the first one), and then applied the hypothesis `H0` (the second one).

The tactic `auto` can invoke `apply` but not `eapply`. So, `auto` cannot exploit lemmas whose instantiation cannot be directly deduced from the proof goal. To exploit such lemmas, one needs to invoke the tactic `eauto`, which is able to call `eapply`.

In the following example, the first hypothesis asserts that `P n` is true when `Q m` is true for some `m`, and the goal is to prove that `Q 1` implies `P 2`. This implication follows direction from the hypothesis by instantiating `m` as the value `1`. The following proof script shows that `eauto` successfully solves the goal, whereas `auto` is not able to do so.

```
Lemma solving_by_eapply : ∀ (P Q : nat→Prop),
  (∀ n m, Q m → P n) →
  Q 1 → P 2.
Proof. auto. eauto. Qed.
```

Conjunctions

So far, we've seen that `eauto` is stronger than `auto` in the sense that it can deal with `eapply`. In the same way, we are going to see how `jauto` and `iauto` are stronger than `auto` and `eauto` in the sense that they provide better support for conjunctions.

The tactics `auto` and `eauto` can prove a goal of the form `F ∧ F'`, where `F` and `F'` are two propositions, as soon as both `F` and `F'` can be proved in the current context. An example follows.

```
Lemma solving_conj_goal : ∀ (P : nat→Prop) (F : Prop),
  (∀ n, P n) → F → F ∧ P 2.
Proof. auto. Qed.
```

However, when an assumption is a conjunction, `auto` and `eauto` are not able to exploit this conjunction. It can be quite surprising at first that `eauto` can prove very complex goals but that it fails to prove that `F ∧ F'` implies `F`. The tactics `iauto` and `jauto` are able to decompose conjunctions from the context. Here is an example.

```
Lemma solving_conj_hyp : ∀ (F F' : Prop),
  F ∧ F' → F.
Proof. auto. eauto. jauto. (* or iauto *) Qed.
```

The tactic `jauto` is implemented by first calling a pre-processing tactic called `jauto_set`, and then calling `eauto`. So, to understand how `jauto` works, one can directly call the tactic `jauto_set`.

```
Lemma solving_conj_hyp' : ∀ (F F' : Prop),
  F ∧ F' → F.
Proof. intros. jauto_set. eauto. Qed.
```

Next is a more involved goal that can be solved by `iauto` and `jauto`.

```
Lemma solving_conj_more : ∀ (P Q R : nat→Prop) (F : Prop),
  (F ∧ (∀ n m, (Q m ∧ R n) → P n)) →
  (F → R 2) →
  Q 1 →
  P 2 ∧ F.
Proof. jauto. (* or iauto *) Qed.
```

The strategy of `iauto` and `jauto` is to run a global analysis of the top-level conjunctions, and then call `eauto`. For this reason, those tactics are not good at dealing with conjunctions that occur as the conclusion of some universally quantified hypothesis. The following example illustrates a general weakness of Coq proof search mechanisms.

```
Lemma solving_conj_hyp_forall : ∀ (P Q : nat→Prop),
  (∀ n, P n ∧ Q n) → P 2.
Proof.
  auto. eauto. iauto. jauto.
  (* Nothing works, so we have to do some of the work by hand *)
  intros. destruct (H 2). auto.
Qed.
```

This situation is slightly disappointing, since automation is able to prove the following goal, which is very similar. The only difference is that the universal quantification has been distributed over the conjunction.

```
Lemma solved_by_jauto : ∀ (P Q : nat→Prop) (F : Prop),
  (∀ n, P n) ∧ (∀ n, Q n) → P 2.
Proof. jauto. (* or iauto *) Qed.
```

Disjunctions

The tactics `auto` and `eauto` can handle disjunctions that occur in the goal.

```
Lemma solving_disj_goal : ∀ (F F' : Prop),
  F → F ∨ F'.
Proof. auto. Qed.
```

However, only `iauto` is able to automate reasoning on the disjunctions that appear in the context. For example, `iauto` can prove that `F ∨ F'` entails `F' ∨ F`.

```
Lemma solving_disj_hyp : ∀ (F F' : Prop),
  F ∨ F' → F' ∨ F.
Proof. auto. eauto. jauto. iauto. Qed.
```

More generally, `iauto` can deal with complex combinations of conjunctions, disjunctions, and negations. Here is an example.

```
Lemma solving_tauto : ∀ (F1 F2 F3 : Prop),
  ((¬F1 ∧ F3) ∨ (F2 ∧ ¬F3)) →
  (F2 → F1) →
  (F2 → F3) →
  ¬F2.
Proof. iauto. Qed.
```

However, the ability of `iauto` to automatically perform a case analysis on disjunctions comes with a downside: `iauto` may be very slow. If the context involves several hypotheses with disjunctions, `iauto` typically generates an exponential number of subgoals on which `eauto` is called. One major advantage of `jauto` compared with `iauto` is that it never spends time performing this kind of case analyses.

Existentials

The tactics `eauto`, `iauto`, and `jauto` can prove goals whose conclusion is an existential. For example, if the goal is `∃ x, f x`, the tactic `eauto` introduces an existential variable, say `?25`, in place of `x`. The remaining goal is `f ?25`, and `eauto` tries to solve this goal, allowing itself to instantiate `?25` with any appropriate value. For example, if an assumption `f 2` is available, then the variable `?25` gets instantiated with `2` and the goal is solved, as shown below.

```
Lemma solving_exists_goal : ∀ (f : nat→Prop),
  f 2 → ∃ x, f x.
Proof.
  auto. (* observe that auto does not deal with existentials, *)
  eauto. (* whereas eauto, iauto and jauto solve the goal *)
Qed.
```

A major strength of `jauto` over the other proof search tactics is that it is able to exploit the existentially-quantified hypotheses, i.e., those of the form `∃x, P`.

```
Lemma solving_exists_hyp : ∀ (f g : nat→Prop),
  (∀ x, f x → g x) →
  (∃ a, f a) →
  (∃ a, g a).
```

```
Proof.
  auto. eauto. iauto. (* All of these tactics fail, *)
  jauto. (* whereas jauto succeeds. *)
  (* For the details, run intros. jauto_set. eauto *)
Qed.
```

Negation

The tactics auto and eauto suffer from some limitations with respect to the manipulation of negations, mostly related to the fact that negation, written ¬ P, is defined as P → False but that the unfolding of this definition is not performed automatically. Consider the following example.

```
Lemma negation_study_1 : ∀ (P : nat→Prop),
  P 0 → (∀ x, ¬ P x) → False.
Proof.
  intros P H0 HX.
  eauto. (* It fails to see that HX applies *)
  unfold not in *. eauto.
Qed.
```

For this reason, the tactics iauto and jauto systematically invoke unfold not in * as part of their pre-processing. So, they are able to solve the previous goal right away.

```
Lemma negation_study_2 : ∀ (P : nat→Prop),
  P 0 → (∀ x, ¬ P x) → False.
Proof. jauto. (* or iauto *) Qed.
```

We will come back later on to the behavior of proof search with respect to the unfolding of definitions.

Equalities

Coq's proof-search feature is not good at exploiting equalities. It can do very basic operations, like exploiting reflexivity and symmetry, but that's about it. Here is a simple example that auto can solve, by first calling symmetry and then applying the hypothesis.

```
Lemma equality_by_auto : ∀ (f g : nat→Prop),
  (∀ x, f x = g x) → g 2 = f 2.
Proof. auto. Qed.
```

To automate more advanced reasoning on equalities, one should rather try to use the tactic congruence, which is presented at the end of this chapter in the "Decision Procedures" section.

How Proof Search Works

Search Depth

The tactic auto works as follows. It first tries to call reflexivity and assumption. If one of these calls solves the goal, the job is done. Otherwise auto tries to apply the most recently introduced assumption that can be applied to the goal without producing and error. This application produces subgoals. There are two possible cases. If the sugboals produced can be solved by a recursive call to auto, then the job is done. Otherwise, if this application produces at least one subgoal that auto cannot solve, then auto starts over by trying to apply the second most recently introduced assumption. It continues in a similar fashion until it finds a proof or until no assumption remains to be tried.

It is very important to have a clear idea of the backtracking process involved in the execution of the auto tactic; otherwise its behavior can be quite puzzling. For example, auto is not able to solve the following triviality.

```
Lemma search_depth_0 :
  True ∧ True ∧ True ∧ True ∧ True ∧ True.
Proof.
  auto.
Abort.
```

The reason auto fails to solve the goal is because there are too many conjunctions. If there had been only five of them, auto would have successfully solved the proof, but six is too many. The tactic auto limits the number of lemmas and hypotheses that can be applied in a proof, so as to ensure that the proof search eventually terminates. By default, the maximal number of steps is five. One can specify a different bound, writing for example auto 6 to search for a proof involving at most six steps. For example, auto 6 would solve the previous lemma. (Similarly, one can invoke eauto 6 or intuition eauto 6.) The argument n of auto n is called the "search depth." The tactic auto is simply defined as a shorthand for auto 5.

The behavior of auto n can be summarized as follows. It first tries to solve the goal using reflexivity and assumption. If this fails, it tries to apply a hypothesis (or a lemma that has been registered in the hint database), and this application produces a number of sugoals. The tactic auto (n-1) is then called on each of those subgoals. If all the subgoals are solved, the job is completed, otherwise auto n tries to apply a different hypothesis.

During the process, auto n calls auto (n-1), which in turn might call auto (n-2), and so on. The tactic auto 0 only tries reflexivity and assumption, and does not try to apply any lemma. Overall, this means that when the maximal number of steps allowed has been exceeded, the auto tactic stops searching and backtracks to try and investigate other paths.

The following lemma admits a unique proof that involves exactly three steps. So, auto n proves this goal iff n is greater than three.

```
Lemma search_depth_1 : ∀ (P : nat→Prop),
  P 0 →
  (P 0 → P 1) →
```

```
    (P 1 → P 2) →
    (P 2).
  Proof.
    auto 0. (* does not find the proof *)
    auto 1. (* does not find the proof *)
    auto 2. (* does not find the proof *)
    auto 3. (* finds the proof *)
            (* more generally, auto n solves the goal if n ≥ 3 *)
  Qed.
```

We can generalize the example by introducing an assumption asserting that `P k` is derivable from `P (k-1)` for all `k`, and keep the assumption `P 0`. The tactic `auto`, which is the same as `auto 5`, is able to derive `P k` for all values of `k` less than 5. For example, it can prove `P 4`.

```
  Lemma search_depth_3 : ∀ (P : nat→Prop),
    (* Hypothesis H1: *) (P 0) →
    (* Hypothesis H2: *) (∀ k, P (k-1) → P k) →
    (* Goal:          *) (P 4).
  Proof. auto. Qed.
```

However, to prove `P 5`, one needs to call at least `auto 6`.

```
  Lemma search_depth_4 : ∀ (P : nat→Prop),
    (* Hypothesis H1: *) (P 0) →
    (* Hypothesis H2: *) (∀ k, P (k-1) → P k) →
    (* Goal:          *) (P 5).
  Proof. auto. auto 6. Qed.
```

Because `auto` looks for proofs at a limited depth, there are cases where `auto` can prove a goal `F` and can prove a goal `F'` but cannot prove `F ∧ F'`. In the following example, `auto` can prove `P 4` but it is not able to prove `P 4 ∧ P 4`, because the splitting of the conjunction consumes one proof step. To prove the conjunction, one needs to increase the search depth, using at least `auto 6`.

```
  Lemma search_depth_5 : ∀ (P : nat→Prop),
    (* Hypothesis H1: *) (P 0) →
    (* Hypothesis H2: *) (∀ k, P (k-1) → P k) →
    (* Goal:          *) (P 4 ∧ P 4).
  Proof. auto. auto 6. Qed.
```

Backtracking

In the previous section, we have considered proofs where at each step there was a unique assumption that `auto` could apply. In general, `auto` can have several choices at every step. The strategy of `auto` consists of trying all of the possibilities (using a depth-first search exploration).

To illustrate how automation works, we are going to extend the previous example with an additional assumption asserting that `P k` is also derivable from `P (k+1)`. Adding this hypothesis offers a new possibility that `auto` could consider at every step.

There exists a special command that one can use for tracing all the steps that proof-search considers. To view such a trace, one should write `debug eauto`. (For some reason, the command `debug auto` does not exist, so we have to use the command `debug eauto` instead.)

```
Lemma working_of_auto_1 : ∀ (P : nat→Prop),
  (* Hypothesis H1: *) (P 0) →
  (* Hypothesis H2: *) (∀ k, P (k-1) → P k) →
  (* Hypothesis H3: *) (∀ k, P (k+1) → P k) →
  (* Goal:          *) (P 2).
(* Uncomment "debug" in the following line to see the debug trace: *)
Proof. intros P H1 H2 H3. (* debug *) eauto. Qed.
```

The output message produced by `debug eauto` is as follows.

```
depth=5
depth=4 apply H2
depth=3 apply H2
depth=3 exact H1
```

The depth indicates the value of `n` with which `eauto n` is called. The tactics shown in the message indicate that the first thing that `eauto` has tried to do is to apply H_2. The effect of applying H_2 is to replace the goal `P 2` with the goal `P 1`. Then, again, H_2 has been applied, changing the goal `P 1` into `P 0`. At that point, the goal was exactly the hypothesis H_1.

It seems that `eauto` was quite lucky there, as it never even tried to use the hypothesis H_3 at any time. The reason is that `auto` always tried to use the H_2 first. So, let's permute the hypotheses H_2 and H_3 and see what happens.

```
Lemma working_of_auto_2 : ∀ (P : nat→Prop),
  (* Hypothesis H1: *) (P 0) →
  (* Hypothesis H3: *) (∀ k, P (k+1) → P k) →
  (* Hypothesis H2: *) (∀ k, P (k-1) → P k) →
  (* Goal:          *) (P 2).
Proof. intros P H1 H3 H2. (* debug *) eauto. Qed.
```

This time, the output message suggests that the proof search investigates many possibilities. If we print the proof term:

```
Print working_of_auto_2.
```

we observe that the proof term refers to H_3. Thus the proof is not the simplest one, since only H_2 and H_1 are needed.

In turns out that the proof goes through the proof obligation `P 3`, even though it is not required to do so. The following tree drawing describes all the goals that `eauto` has been going through.

```
|5||4||3||2||1||0| -- below, tabulation indicates the depth

[P 2]
-> [P 3]
   -> [P 4]
      -> [P 5]
         -> [P 6]
            -> [P 7]
            -> [P 5]
         -> [P 4]
            -> [P 5]
            -> [P 3]
      --> [P 3]
         -> [P 4]
            -> [P 5]
            -> [P 3]
         -> [P 2]
            -> [P 3]
            -> [P 1]
   -> [P 2]
      -> [P 3]
         -> [P 4]
            -> [P 5]
            -> [P 3]
         -> [P 2]
            -> [P 3]
            -> [P 1]
      -> [P 1]
         -> [P 2]
            -> [P 3]
            -> [P 1]
         -> [P 0]
            -> !! Done !!
```

The first few lines read as follows. To prove `P 2`, `eauto 5` has first tried to apply H_3, producing the subgoal `P 3`. To solve it, `eauto 4` has tried again to apply H_3, producing the goal `P 4`. Similarly, the search goes through `P 5`, `P 6` and `P 7`. When reaching `P 7`, the tactic `eauto 0` is called but as it is not allowed to try and apply any lemma, it fails. So, we come back to the goal `P 6`, and try this time to apply hypothesis H_2, producing the subgoal `P 5`. Here again, `eauto 0` fails to solve this goal.

The process goes on and on, until backtracking to `P 3` and trying to apply H_3 three times in a row, going through `P 2` and `P 1` and `P 0`. This search tree explains why `eauto` came up with a proof term starting with an application of H_3.

Adding Hints

By default, auto (and eauto) only tries to apply the hypotheses that appear in the proof context. There are two possibilities for telling auto to exploit a lemma that have been proved previously: either adding the lemma as an assumption just before calling auto, or adding the lemma as a hint, so that it can be used by every calls to auto.

The first possibility is useful to have auto exploit a lemma that only serves at this particular point. To add the lemma as hypothesis, one can type generalize mylemma; intros, or simply lets: mylemma (the latter requires LibTactics.v).

The second possibility is useful for lemmas that need to be exploited several times. The syntax for adding a lemma as a hint is Hint Resolve mylemma. For example, the lemma asserting than any number is less than or equal to itself, ∀x, x ≤ x, called Le.le_refl in the Coq standard library, can be added as a hint as follows.

```
Hint Resolve Le.le_refl.
```

A convenient shorthand for adding all the constructors of an inductive datatype as hints is the command Hint Constructors mydatatype.

Warning: some lemmas, such as transitivity results, should not be added as hints as they would very badly affect the performance of proof search. The description of this problem and the presentation of a general work-around for transitivity lemmas appear further on.

Integration of Automation in Tactics

The library "LibTactics" introduces a convenient feature for invoking automation after calling a tactic. In short, it suffices to add the symbol star (*) to the name of a tactic. For example, apply* H is equivalent to apply H; auto_star, where auto_star is a tactic that can be defined as needed.

The definition of auto_star, which determines the meaning of the star symbol, can be modified whenever needed. Simply write:

```
Ltac auto_star ::= a_new_definition.
```

Observe the use of ::= instead of :=, which indicates that the tactic is being rebound to a new definition. So, the default definition is as follows.

```
Ltac auto_star ::= try solve [ jauto ].
```

Nearly all standard Coq tactics and all the tactics from "LibTactics" can be called with a star symbol. For example, one can invoke subst*, destruct* H, inverts* H, lets* I: H x, specializes* H x, and so on... There are two notable exceptions. The tactic auto* is just another name for the tactic auto_star. And the tactic apply* H calls eapply H (or the more powerful applys H if needed), and then calls auto_star. Note that there is no eapply* H tactic, use apply* H instead.

In large developments, it can be convenient to use two degrees of automation. Typically, one would use a fast tactic, like `auto`, and a slower but more powerful tactic, like `jauto`. To allow for a smooth coexistence of the two form of automation, `LibTactics.v` also defines a "tilde" version of tactics, like `apply¬ H`, `destruct¬ H`, `subst¬`, `auto¬` and so on. The meaning of the tilde symbol is described by the `auto_tilde` tactic, whose default implementation is `auto`.

```
Ltac auto_tilde ::= auto.
```

In the examples that follow, only `auto_star` is needed.

An alternative, possibly more efficient version of auto_star is the following":

Ltac auto_star ::= try solve `eassumption` | `auto` | `jauto` .

With the above definition, `auto_star` first tries to solve the goal using the assumptions; if it fails, it tries using `auto`, and if this still fails, then it calls `jauto`. Even though `jauto` is strictly stronger than `eassumption` and `auto`, it makes sense to call these tactics first, because, when the succeed, they save a lot of time, and when they fail to prove the goal, they fail very quickly.".

Examples of Use of Automation

Let's see how to use proof search in practice on the main theorems of the "Software Foundations" course, proving in particular results such as determinism, preservation and progress.

Determinism

```
Module DeterministicImp.
  Import Imp.
```

Recall the original proof of the determinism lemma for the IMP language, shown below.

```
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; inversion E2; subst.
  - (* E_Skip *) reflexivity.
  - (* E_Ass *) reflexivity.
  - (* E_Seq *)
    assert (st' = st'0) as EQ1.
    { (* Proof of assertion *) apply IHE1_1; assumption. }
    subst st'0.
    apply IHE1_2. assumption.
```

```
    (* E_IfTrue *)
    - (* b1 reduces to true *)
      apply IHE1. assumption.
    - (* b1 reduces to false (contradiction) *)
      rewrite H in H5. inversion H5.
    (* E_IfFalse *)
    - (* b1 reduces to true (contradiction) *)
      rewrite H in H5. inversion H5.
    - (* b1 reduces to false *)
        apply IHE1. assumption.
    (* E_WhileFalse *)
    - (* b1 reduces to true *)
      reflexivity.
    - (* b1 reduces to false (contradiction) *)
      rewrite H in H2. inversion H2.
    (* E_WhileTrue *)
    - (* b1 reduces to true (contradiction) *)
      rewrite H in H4. inversion H4.
    - (* b1 reduces to false *)
      assert (st' = st'0) as EQ1.
      { (* Proof of assertion *) apply IHE1_1; assumption. }
      subst st'0.
      apply IHE1_2. assumption.
  Qed.
```

Exercise: rewrite this proof using `auto` whenever possible. (The solution uses `auto` 9 times.)

```
Theorem ceval_deterministic': ∀ c st st1 st2,
  c / st \\ st1 →
  c / st \\ st2 →
  st1 = st2.
Proof.
  (* FILL IN HERE *) admit.
Admitted.
```

In fact, using automation is not just a matter of calling `auto` in place of one or two other tactics. Using automation is about rethinking the organization of sequences of tactics so as to minimize the effort involved in writing and maintaining the proof. This process is eased by the use of the tactics from `LibTactics.v`. So, before trying to optimize the way automation is used, let's first rewrite the proof of determinism:

- use `introv H` instead of `intros x H`,
- use `gen x` instead of `generalize dependent x`,
- use `inverts H` instead of `inversion H; subst`,
- use `tryfalse` to handle contradictions, and get rid of the cases where `beval st b1 = true` and `beval st b1 = false` both appear in the context,
- stop using `ceval_cases` to label subcases.

```
Theorem ceval_deterministic'': ∀ c st st1 st2,
  c / st \\ st1 →
  c / st \\ st2 →
  st1 = st2.
Proof.
  introv E1 E2. gen st2.
  induction E1; intros; inverts E2; tryfalse.
  - auto.
  - auto.
  - assert (st' = st'0). auto. subst. auto.
  - auto.
  - auto.
  - auto.
  - assert (st' = st'0). auto. subst. auto.
Qed.
```

To obtain a nice clean proof script, we have to remove the calls `assert (st' = st'0)`. Such a tactic invokation is not nice because it refers to some variables whose name has been automatically generated. This kind of tactics tend to be very brittle. The tactic `assert (st' = st'0)` is used to assert the conclusion that we want to derive from the induction hypothesis. So, rather than stating this conclusion explicitly, we are going to ask Coq to instantiate the induction hypothesis, using automation to figure out how to instantiate it. The tactic `forwards`, described in `LibTactics.v` precisely helps with instantiating a fact. So, let's see how it works out on our example.

```
Theorem ceval_deterministic''': ∀ c st st1 st2,
  c / st \\ st1 →
  c / st \\ st2 →
  st1 = st2.
Proof.
  (* Let's replay the proof up to the assert tactic. *)
  introv E1 E2. gen st2.
  induction E1; intros; inverts E2; tryfalse.
  - auto.
  - auto.
  (* We duplicate the goal for comparing different proofs. *)
  - dup 4.

  (* The old proof: *)
  + assert (st' = st'0). apply IHE1_1. apply H1.
    (* produces H: st' = st'0. *) skip.

  (* The new proof, without automation: *)
  + forwards: IHE1_1. apply H1.
    (* produces H: st' = st'0. *) skip.

  (* The new proof, with automation: *)
  + forwards: IHE1_1. eauto.
    (* produces H: st' = st'0. *) skip.

  (* The new proof, with integrated automation: *)
  + forwards*: IHE1_1.
    (* produces H: st' = st'0. *) skip.
```

```
Abort.
```

To polish the proof script, it remains to factorize the calls to `auto`, using the star symbol. The proof of determinism can then be rewritten in only four lines, including no more than 10 tactics.

```
Theorem ceval_deterministic'''': ∀ c st st1 st2,
  c / st \\ st1 →
  c / st \\ st2 →
  st1 = st2.
Proof.
  introv E1 E2. gen st2.
  induction E1; intros; inverts* E2; tryfalse.
  - forwards*: IHE1_1. subst*.
  - forwards*: IHE1_1. subst*.
Qed.

End DeterministicImp.
```

Preservation for STLC

```
Set Warnings "-notation-overridden,-parsing".
Require Import StlcProp.
Module PreservationProgressStlc.
Import STLC.
Import STLCProp.
```

Consider the proof of perservation of STLC, shown below. This proof already uses `eauto` through the triple-dot mechanism.

```
Theorem preservation : ∀ t t' T,
  has_type empty t T →
  t ==> t' →
  has_type empty t' T.
Proof with eauto.
  remember (@empty ty) as Gamma.
  intros t t' T HT. generalize dependent t'.
  (induction HT); intros t' HE; subst Gamma.
  - (* T_Var *)
    inversion HE.
  - (* T_Abs *)
    inversion HE.
  - (* T_App *)
    inversion HE; subst...
    (* The ST_App1 and ST_App2 cases are immediate by induction, and
       auto takes care of them *)
    + (* ST_AppAbs *)
      apply substitution_preserves_typing with T11...
      inversion HT1...
  - (* T_True *)
    inversion HE.
  - (* T_False *)
    inversion HE.
  - (* T_If *)
```

```
      inversion HE; subst...
  Qed.
```

Exercise: rewrite this proof using tactics from `LibTactics` and calling automation using the star symbol rather than the triple-dot notation. More precisely, make use of the tactics `inverts*` and `applys*` to call `auto*` after a call to `inverts` or to `applys`. The solution is three lines long.

```
Theorem preservation' : ∀ t t' T,
  has_type empty t T →
  t ==> t' →
  has_type empty t' T.
Proof.
  (* FILL IN HERE *) admit.
Admitted.
```

Progress for STLC

Consider the proof of the progress theorem.

```
Theorem progress : ∀ t T,
  has_type empty t T →
  value t ∨ ∃ t', t ==> t'.
Proof with eauto.
  intros t T Ht.
  remember (@empty ty) as Gamma.
  (induction Ht); subst Gamma...
  - (* T_Var *)
    inversion H.
  - (* T_App *)
    right. destruct IHHt1...
    + (* t1 is a value *)
      destruct IHHt2...
      * (* t2 is a value *)
        inversion H; subst; try solve_by_invert.
        ∃ ([x0:=t2]t)...
      * (* t2 steps *)
       destruct H0 as [t2' Hstp]. ∃ (tapp t1 t2')...
    + (* t1 steps *)
      destruct H as [t1' Hstp]. ∃ (tapp t1' t2)...
  - (* T_If *)
    right. destruct IHHt1...
    destruct t1; try solve_by_invert...
    inversion H. ∃ (tif x0 t2 t3)...
Qed.
```

Exercise: optimize the above proof. Hint: make use of `destruct*` and `inverts*`. The solution consists of 10 short lines.

```
Theorem progress' : ∀ t T,
  has_type empty t T →
  value t ∨ ∃ t', t ==> t'.
Proof.
```

```
    (* FILL IN HERE *) admit.
  Admitted.

  End PreservationProgressStlc.
```

BigStep and SmallStep

```
  Require Import Smallstep.
  Require Import Program.
  Module Semantics.
```

Consider the proof relating a small-step reduction judgment to a big-step reduction judgment.

```
  Theorem multistep__eval : ∀ t v,
    normal_form_of t v → ∃ n, v = C n ∧ t \\ n.
  Proof.
    intros t v Hnorm.
    unfold normal_form_of in Hnorm.
    inversion Hnorm as [Hs Hnf]; clear Hnorm.
    rewrite nf_same_as_value in Hnf. inversion Hnf. clear Hnf.
    ∃ n. split. reflexivity.
    induction Hs; subst.
    - (* multi_refl *)
      apply E_Const.
    - (* multi_step *)
      eapply step__eval. eassumption. apply IHHs. reflexivity.
  Qed.
```

Our goal is to optimize the above proof. It is generally easier to isolate inductions into separate lemmas. So, we are going to first prove an intermediate result that consists of the judgment over which the induction is being performed.

Exercise: prove the following result, using tactics `introv`, `induction` and `subst`, and `apply*`. The solution is 3 lines long.

```
  Theorem multistep_eval_ind : ∀ t v,
    t ==>* v → ∀ n, C n = v → t \\ n.
  Proof.
    (* FILL IN HERE *) admit.
  Admitted.
```

Exercise: using the lemma above, simplify the proof of the result `multistep__eval`. You should use the tactics `introv`, `inverts`, `split*` and `apply*`. The solution is 2 lines long.

```
  Theorem multistep__eval' : ∀ t v,
    normal_form_of t v → ∃ n, v = C n ∧ t \\ n.
  Proof.
    (* FILL IN HERE *) admit.
  Admitted.
```

If we try to combine the two proofs into a single one, we will likely fail, because of a limitation of the `induction` tactic. Indeed, this tactic looses information when applied to a property whose arguments are not reduced to variables, such as `t ==>* (C n)`. You will thus need to use the more powerful tactic called `dependent induction`. (This tactic is available only after importing the `Program` library, as we did above.)

Exercise: prove the lemma `multistep__eval` without invoking the lemma `multistep_eval_ind`, that is, by inlining the proof by induction involved in `multistep_eval_ind`, using the tactic `dependent induction` instead of `induction`. The solution is 5 lines long.

```
Theorem multistep__eval'' : ∀ t v,
  normal_form_of t v → ∃ n, v = C n ∧ t \\ n.
Proof.
  (* FILL IN HERE *) admit.
Admitted.

End Semantics.
```

Preservation for STLCRef

```
Require Import Coq.omega.Omega.
Require Import References.
Import STLCRef.
Require Import Program.
Module PreservationProgressReferences.
Hint Resolve store_weakening extends_refl.
```

The proof of preservation for `STLCRef` can be found in chapter `References`. The optimized proof script is more than twice shorter. The following material explains how to build the optimized proof script. The resulting optimized proof script for the preservation theorem appears afterwards.

```
Theorem preservation : ∀ ST t t' T st st',
  has_type empty ST t T →
  store_well_typed ST st →
  t / st ==> t' / st' →
  ∃ ST',
    (extends ST' ST ∧
     has_type empty ST' t' T ∧
     store_well_typed ST' st').
Proof.
  (* old: Proof. with eauto using store_weakening, extends_refl.
     new: Proof., and the two lemmas are registered as hints
     before the proof of the lemma, possibly inside a section in
     order to restrict the scope of the hints. *)

  remember (@empty ty) as Gamma. introv Ht. gen t'.
  (induction Ht); introv HST Hstep;
    (* old: subst; try solve_by_invert; inversion Hstep; subst;
try (eauto using store_weakening, extends_refl)
      new: subst Gamma; inverts Hstep; eauto.
      We want to be more precise on what exactly we substitute,
      and we do not want to call try solve_by_invert which
      is way to slow. *)
   subst Gamma; inverts Hstep; eauto.
```

```
  (* T_App *)
  - (* ST_AppAbs *)
  (* old:
      exists ST. inversion Ht1; subst.
      split; try split... eapply substitution_preserves_typing... *)
  (* new: we use inverts in place of inversion and splits to
     split the conjunction, and applys* in place of eapply... *)
  ∃ ST. inverts Ht1. splits*. applys*
substitution_preserves_typing.

  - (* ST_App1 *)
  (* old:
      eapply IHHt1 in H0...
      inversion H0 as ST' [Hext [Hty Hsty]].
      exists ST'... *)
  (* new: The tactic eapply IHHt1 in H0... applies IHHt1 to H0.
     But H0 is only thing that IHHt1 could be applied to, so
     there eauto can figure this out on its own. The tactic
     forwards is used to instantiate all the arguments of IHHt1,
     producing existential variables and subgoals when needed. *)
  forwards: IHHt1. eauto. eauto. eauto.
  (* At this point, we need to decompose the hypothesis H that has
     just been created by forwards. This is done by the first part
     of the preprocessing phase of jauto. *)
  jauto_set_hyps; intros.
  (* It remains to decompose the goal, which is done by the second
     part of the preprocessing phase of jauto. *)
  jauto_set_goal; intros.
  (* All the subgoals produced can then be solved by eauto. *)
  eauto. eauto. eauto.

  -(* ST_App2 *)
  (* old:
      eapply IHHt2 in H5...
      inversion H5 as ST' [Hext [Hty Hsty]].
      exists ST'... *)
  (* new: this time, we need to call forwards on IHHt2,
     and we call jauto right away, by writing forwards*,
     proving the goal in a single tactic! *)
  forwards*: IHHt2.

  (* The same trick works for many of the other subgoals. *)
  - forwards*: IHHt.
  - forwards*: IHHt.
  - forwards*: IHHt1.
  - forwards*: IHHt2.
  - forwards*: IHHt1.

  - (* T_Ref *)
  + (* ST_RefValue *)
    (* old:
        exists (ST ++ T1::nil).
        inversion HST; subst.
        split.
          apply extends_app.
        split.
```

```
          replace (TRef T1)
            with (TRef (store_Tlookup (length st) (ST ++ T1::nil))).
          apply T_Loc.
          rewrite <- H. rewrite app_length, plus_comm. simpl. omega.
          unfold store_Tlookup. rewrite <- H. rewrite app_nth2; try omega.
          rewrite minus_diag. simpl. reflexivity.
          apply store_well_typed_app; assumption. *)
    (* new: In this proof case, we need to perform an inversion
       without removing the hypothesis. The tactic inverts keep
       serves exactly this purpose. *)
    ∃ (ST ++ T1::nil). inverts keep HST. splits.
    (* The proof of the first subgoal needs no change *)
      apply extends_app.
    (* For the second subgoal, we use the tactic applys_eq to avoid
       a manual replace before T_loc can be applied. *)
      applys_eq T_Loc 1.
    (* To justify the inequality, there is no need to call rewrite
<- H,
       because the tactic omega is able to exploit H on its own.
       So, only the rewriting of app_length and the call to the
       tactic omega remain, with a call to simpl to unfold the
       definition of app. *)
        rewrite app_length. simpl. omega.
    (* The next proof case is hard to polish because it relies on the
       lemma app_nth1 whose statement is not automation-friendly.
       We'll come back to this proof case further on. *)
      unfold store_Tlookup. rewrite <- H. rewrite* app_nth2.
    (* Last, we replace apply ..; assumption with apply* .. *)
    rewrite minus_diag. simpl. reflexivity.
    apply* store_well_typed_app.

  - forwards*: IHHt.

  - (* T_Deref *)
  + (* ST_DerefLoc *)
  (* old:
      exists ST. split; try split...
      destruct HST as _ Hsty.
      replace T11 with (store_Tlookup l ST).
      apply Hsty...
      inversion Ht; subst... *)
  (* new: we start by calling ∃ST and splits*. *)
  ∃ ST. splits*.
  (* new: we replace destruct HST as [_ Hsty] by the following *)
  lets [_ Hsty]: HST.
  (* new: then we use the tactic applys_eq to avoid the need to
     perform a manual replace before applying Hsty. *)
  applys_eq* Hsty 1.
  (* new: we then can call inverts in place of inversion;subst *)
  inverts* Ht.

  - forwards*: IHHt.

  - (* T_Assign *)
  + (* ST_Assign *)
  (* old:
      exists ST. split; try split...
```

```
      eapply assign_pres_store_typing...
      inversion Ht1; subst... *)
  (* new: simply using nicer tactics *)
  ∃ ST. splits*. applys* assign_pres_store_typing. inverts* Ht1.

  - forwards*: IHHt1.
  - forwards*: IHHt2.
Qed.
```

Let's come back to the proof case that was hard to optimize. The difficulty comes from the statement of `nth_eq_last`, which takes the form `nth (length l) (l ++ x::nil) d = x`. This lemma is hard to exploit because its first argument, `length l`, mentions a list `l` that has to be exactly the same as the `l` occuring in `snoc l x`. In practice, the first argument is often a natural number `n` that is provably equal to `length l` yet that is not syntactically equal to `length l`. There is a simple fix for making `nth_eq_last` easy to apply: introduce the intermediate variable `n` explicitly, so that the goal becomes `nth n (snoc l x) d = x`, with a premise asserting `n = length l`.

```
Lemma nth_eq_last' : ∀ (A : Type) (l : list A) (x d : A) (n : nat),
  n = length l → nth n (l ++ x::nil) d = x.
Proof. intros. subst. apply nth_eq_last. Qed.
```

The proof case for `ref` from the preservation theorem then becomes much easier to prove, because `rewrite nth_eq_last'` now succeeds.

```
Lemma preservation_ref : ∀ (st:store) (ST : store_ty) T1,
  length ST = length st →
  TRef T1 = TRef (store_Tlookup (length st) (ST ++ T1::nil)).
Proof.
  intros. dup.

  (* A first proof, with an explicit unfold *)
  unfold store_Tlookup. rewrite* nth_eq_last'.

  (* A second proof, with a call to fequal *)
  fequal. symmetry. apply* nth_eq_last'.
Qed.
```

The optimized proof of preservation is summarized next.

```
Theorem preservation' : ∀ ST t t' T st st',
  has_type empty ST t T →
  store_well_typed ST st →
  t / st ==> t' / st' →
  ∃ ST',
    (extends ST' ST ∧
     has_type empty ST' t' T ∧
     store_well_typed ST' st').
Proof.
  remember (@empty ty) as Gamma. introv Ht. gen t'.
  induction Ht; introv HST Hstep; subst Gamma; inverts Hstep;
eauto.
```

```
    - ∃ ST. inverts Ht₁. splits*. applys*
  substitution_preserves_typing.
    - forwards*: IHHt1.
    - forwards*: IHHt2.
    - forwards*: IHHt.
    - forwards*: IHHt.
    - forwards*: IHHt1.
    - forwards*: IHHt2.
    - forwards*: IHHt1.
    - ∃ (ST ++ T₁::nil). inverts keep HST. splits.
      apply extends_app.
      applys_eq T_Loc 1.
        rewrite app_length. simpl. omega.
        unfold store_Tlookup. rewrite* nth_eq_last'.
      apply* store_well_typed_app.
    - forwards*: IHHt.
    - ∃ ST. splits*. lets [_ Hsty]: HST.
      applys_eq* Hsty 1. inverts* Ht.
    - forwards*: IHHt.
    - ∃ ST. splits*. applys* assign_pres_store_typing. inverts*
  Ht₁.
    - forwards*: IHHt1.
    - forwards*: IHHt2.
  Qed.
```

Progress for STLCRef

The proof of progress for `STLCRef` can be found in chapter `References`. The optimized proof script is, here again, about half the length.

```
  Theorem progress : ∀ ST t T st,
    has_type empty ST t T →
    store_well_typed ST st →
    (value t ∨ ∃ t', ∃ st', t / st ==> t' / st').
  Proof.
    introv Ht HST. remember (@empty ty) as Gamma.
    induction Ht; subst Gamma; tryfalse; try solve [left*].
    - right. destruct* IHHt1 as [K|].
      inverts K; inverts Ht₁.
       destruct* IHHt2.
    - right. destruct* IHHt as [K|].
      inverts K; try solve [inverts Ht]. eauto.
    - right. destruct* IHHt as [K|].
      inverts K; try solve [inverts Ht]. eauto.
    - right. destruct* IHHt1 as [K|].
      inverts K; try solve [inverts Ht₁].
       destruct* IHHt2 as [M|].
        inverts M; try solve [inverts Ht₂]. eauto.
    - right. destruct* IHHt1 as [K|].
      inverts K; try solve [inverts Ht₁]. destruct* n.
    - right. destruct* IHHt.
    - right. destruct* IHHt as [K|].
      inverts K; inverts Ht as M.
        inverts HST as N. rewrite* N in M.
    - right. destruct* IHHt1 as [K|].
```

```
    destruct* IHHt2.
     inverts K; inverts Ht1 as M.
     inverts HST as N. rewrite* N in M.
Qed.

End PreservationProgressReferences.
```

Subtyping

```
Require Sub.
Module SubtypingInversion.
Import Sub.
```

Consider the inversion lemma for typing judgment of abstractions in a type system with subtyping.

```
Lemma abs_arrow : ∀ x S1 s2 T1 T2,
  has_type empty (tabs x S1 s2) (TArrow T1 T2) →
     subtype T1 S1
  ∧ has_type (update empty x S1) s2 T2.
Proof with eauto.
  intros x S1 s2 T1 T2 Hty.
  apply typing_inversion_abs in Hty.
  destruct Hty as [S2 [Hsub Hty]].
  apply sub_inversion_arrow in Hsub.
  destruct Hsub as [U1 [U2 [Heq [Hsub1 Hsub2]]]].
  inversion Heq; subst...
Qed.
```

Exercise: optimize the proof script, using `introv`, `lets` and `inverts*`. In particular, you will find it useful to replace the pattern `apply K in H. destruct H as I` with `lets I: K H`. The solution is 4 lines.

```
Lemma abs_arrow' : ∀ x S1 s2 T1 T2,
  has_type empty (tabs x S1 s2) (TArrow T1 T2) →
     subtype T1 S1
  ∧ has_type (update empty x S1) s2 T2.
Proof.
  (* FILL IN HERE *) admit.
Admitted.
```

The lemma `substitution_preserves_typing` has already been used to illustrate the working of `lets` and `applys` in chapter `UseTactics`. Optimize further this proof using automation (with the star symbol), and using the tactic `cases_if'`. The solution is 33 lines).

```
Lemma substitution_preserves_typing : ∀ Gamma x U v t S,
  has_type (update Gamma x U) t S →
  has_type empty v U →
  has_type Gamma ([x:=v]t) S.
Proof.
  (* FILL IN HERE *) admit.
Admitted.
```

```
End SubtypingInversion.
```

Advanced Topics in Proof Search

Stating Lemmas in the Right Way

Due to its depth-first strategy, `eauto` can get exponentially slower as the depth search increases, even when a short proof exists. In general, to make proof search run reasonably fast, one should avoid using a depth search greater than 5 or 6. Moreover, one should try to minimize the number of applicable lemmas, and usually put first the hypotheses whose proof usefully instantiates the existential variables.

In fact, the ability for `eauto` to solve certain goals actually depends on the order in which the hypotheses are stated. This point is illustrated through the following example, in which `P` is a property of natural numbers. This property is such that `P n` holds for any `n` as soon as `P m` holds for at least one `m` different from zero. The goal is to prove that `P 2` implies `P 1`. When the hypothesis about `P` is stated in the form `∀n m, P m → m ≠ 0 → P n`, then `eauto` works. However, with `∀n m, m ≠ 0 → P m → P n`, the tactic `eauto` fails.

```
Lemma order_matters_1 : ∀ (P : nat→Prop),
  (∀ n m, P m → m ≠ 0 → P n) → P 2 → P 1.
Proof.
  eauto. (* Success *)
  (* The proof: intros P H K. eapply H. apply K. auto. *)
Qed.

Lemma order_matters_2 : ∀ (P : nat→Prop),
  (∀ n m, m ≠ 0 → P m → P n) → P 5 → P 1.
Proof.
  eauto. (* Failure *)

  (* To understand why, let us replay the previous proof *)
  intros P H K.
  eapply H.
  (* The application of eapply has left two subgoals,
     ?X ≠ 0 and P ?X, where ?X is an existential variable. *)
  (* Solving the first subgoal is easy for eauto: it suffices
     to instantiate ?X as the value 1, which is the simplest
     value that satisfies ?X ≠ 0. *)
  eauto.
  (* But then the second goal becomes P 1, which is where we
     started from. So, eauto gets stuck at this point. *)
Abort.
```

It is very important to understand that the hypothesis `∀n m, P m → m ≠ 0 → P n` is eauto-friendly, whereas `∀n m, m ≠ 0 → P m → P n` really isn't. Guessing a value of `m` for which `P m` holds and then checking that `m ≠ 0` holds works well because there are few values of `m` for which `P m` holds. So, it is likely that `eauto` comes up with the right one. On the other hand, guessing a value of `m` for which `m ≠ 0` and then checking that `P m` holds does not work well, because there are many values of `m` that satisfy `m ≠ 0` but not `P m`.

Unfolding of Definitions During Proof-Search

The use of intermediate definitions is generally encouraged in a formal development as it usually leads to more concise and more readable statements. Yet, definitions can make it a little harder to automate proofs. The problem is that it is not obvious for a proof search mechanism to know when definitions need to be unfolded. Note that a naive strategy that consists in unfolding all definitions before calling proof search does not scale up to large proofs, so we avoid it. This section introduces a few techniques for avoiding to manually unfold definitions before calling proof search.

To illustrate the treatment of definitions, let `P` be an abstract property on natural numbers, and let `myFact` be a definition denoting the proposition `P x` holds for any `x` less than or equal to 3.

```
Axiom P : nat → Prop.

Definition myFact := ∀ x, x ≤ 3 → P x.
```

Proving that `myFact` under the assumption that `P x` holds for any `x` should be trivial. Yet, `auto` fails to prove it unless we unfold the definition of `myFact` explicitly.

```
Lemma demo_hint_unfold_goal_1 :
  (∀ x, P x) → myFact.
Proof.
  auto. (* Proof search doesn't know what to do, *)
  unfold myFact. auto. (* unless we unfold the definition. *)
Qed.
```

To automate the unfolding of definitions that appear as proof obligation, one can use the command `Hint Unfold myFact` to tell Coq that it should always try to unfold `myFact` when `myFact` appears in the goal.

```
Hint Unfold myFact.
```

This time, automation is able to see through the definition of `myFact`.

```
Lemma demo_hint_unfold_goal_2 :
  (∀ x, P x) → myFact.
Proof. auto. Qed.
```

However, the `Hint Unfold` mechanism only works for unfolding definitions that appear in the goal. In general, proof search does not unfold definitions from the context. For example, assume we want to prove that `P 3` holds under the assumption that `True → myFact`.

```
Lemma demo_hint_unfold_context_1 :
  (True → myFact) → P 3.
Proof.
  intros.
  auto. (* fails *)
```

```
  unfold myFact in *. auto. (* succeeds *)
Qed.
```

There is actually one exception to the previous rule: a constant occuring in an hypothesis is automatically unfolded if the hypothesis can be directly applied to the current goal. For example, `auto` can prove `myFact → P 3`, as illustrated below.

```
Lemma demo_hint_unfold_context_2 :
  myFact → P 3.
Proof. auto. Qed.
```

Automation for Proving Absurd Goals

In this section, we'll see that lemmas concluding on a negation are generally not useful as hints, and that lemmas whose conclusion is `False` can be useful hints but having too many of them makes proof search inefficient. We'll also see a practical work-around to the efficiency issue.

Consider the following lemma, which asserts that a number less than or equal to 3 is not greater than 3.

```
Parameter le_not_gt : ∀ x,
  (x ≤ 3) → ¬ (x > 3).
```

Equivalently, one could state that a number greater than three is not less than or equal to 3.

```
Parameter gt_not_le : ∀ x,
  (x > 3) → ¬ (x ≤ 3).
```

In fact, both statements are equivalent to a third one stating that `x ≤ 3` and `x > 3` are contradictory, in the sense that they imply `False`.

```
Parameter le_gt_false : ∀ x,
  (x ≤ 3) → (x > 3) → False.
```

The following investigation aim at figuring out which of the three statments is the most convenient with respect to proof automation. The following material is enclosed inside a `Section`, so as to restrict the scope of the hints that we are adding. In other words, after the end of the section, the hints added within the section will no longer be active.

```
Section DemoAbsurd1.
```

Let's try to add the first lemma, `le_not_gt`, as hint, and see whether we can prove that the proposition `∃x, x ≤ 3 ∧ x > 3` is absurd.

```
Hint Resolve le_not_gt.

Lemma demo_auto_absurd_1 :
  (∃ x, x ≤ 3 ∧ x > 3) → False.
Proof.
  intros. jauto_set. (* decomposes the assumption *)
  (* debug *) eauto.
(* does not see that le_not_gt could apply *)
```

```
    eapply le_not_gt. eauto. eauto.
  Qed.
```

The lemma `gt_not_le` is symmetric to `le_not_gt`, so it will not be any better. The third lemma, `le_gt_false`, is a more useful hint, because it concludes on `False`, so proof search will try to apply it when the current goal is `False`.

```
Hint Resolve le_gt_false.

Lemma demo_auto_absurd_2 :
  (∃ x, x ≤ 3 ∧ x > 3) → False.
Proof.
  dup.

  (* detailed version: *)
  intros. jauto_set. (* debug *) eauto.

  (* short version: *)
  jauto.
Qed.
```

In summary, a lemma of the form `H1 → H2 → False` is a much more effective hint than `H1 → ¬ H2`, even though the two statments are equivalent up to the definition of the negation symbol `¬`.

That said, one should be careful with adding lemmas whose conclusion is `False` as hint. The reason is that whenever reaching the goal `False`, the proof search mechanism will potentially try to apply all the hints whose conclusion is `False` before applying the appropriate one.

```
End DemoAbsurd1.
```

Adding lemmas whose conclusion is `False` as hint can be, locally, a very effective solution. However, this approach does not scale up for global hints. For most practical applications, it is reasonable to give the name of the lemmas to be exploited for deriving a contradiction. The tactic `false H`, provided by `LibTactics` serves that purpose: `false H` replaces the goal with `False` and calls `eapply H`. Its behavior is described next. Observe that any of the three statements `le_not_gt`, `gt_not_le` or `le_gt_false` can be used.

```
Lemma demo_false : ∀ x,
  (x ≤ 3) → (x > 3) → 4 = 5.
Proof.
  intros. dup 4.

  (* A failed proof: *)
  - false. eapply le_gt_false.
    + auto. (* here, auto does not prove ?x ≤ 3 by using H but
             by using the lemma le_refl : ∀x, x ≤ x. *)
    (* The second subgoal becomes 3 >
3, which is not provable. *)
    + skip.
```

```
  (* A correct proof: *)
  - false. eapply le_gt_false.
    + eauto. (* here, eauto uses H, as expected, to prove ?x ≤
3 *)
    + eauto. (* so the second subgoal becomes x > 3 *)

  (* The same proof using false: *)
  - false le_gt_false. eauto. eauto.

  (* The lemmas le_not_gt and gt_not_le work as well *)
  - false le_not_gt. eauto. eauto.
Qed.
```

In the above example, `false le_gt_false; eauto` proves the goal, but `false le_gt_false; auto` does not, because `auto` does not correctly instantiate the existential variable. Note that `false* le_gt_false` would not work either, because the star symbol tries to call `auto` first. So, there are two possibilities for completing the proof: either call `false le_gt_false; eauto`, or call `false* (le_gt_false 3)`.

Automation for Transitivity Lemmas

Some lemmas should never be added as hints, because they would very badly slow down proof search. The typical example is that of transitivity results. This section describes the problem and presents a general workaround.

Consider a subtyping relation, written `subtype S T`, that relates two object `S` and `T` of type `typ`. Assume that this relation has been proved reflexive and transitive. The corresponding lemmas are named `subtype_refl` and `subtype_trans`.

```
Parameter typ : Type.

Parameter subtype : typ → typ → Prop.

Parameter subtype_refl : ∀ T,
  subtype T T.

Parameter subtype_trans : ∀ S T U,
  subtype S T → subtype T U → subtype S U.
```

Adding reflexivity as hint is generally a good idea, so let's add reflexivity of subtyping as hint.

```
Hint Resolve subtype_refl.
```

Adding transitivity as hint is generally a bad idea. To understand why, let's add it as hint and see what happens. Because we cannot remove hints once we've added them, we are going to open a "Section," so as to restrict the scope of the transitivity hint to that section.

```
Section HintsTransitivity.

Hint Resolve subtype_trans.
```

Now, consider the goal `∀S T, subtype S T`, which clearly has no hope of being solved. Let's call `eauto` on this goal.

```
Lemma transitivity_bad_hint_1 : ∀ S T,
  subtype S T.
Proof.
  intros. (* debug *) eauto.
(* Investigates 106 applications... *)
Abort.
```

Note that after closing the section, the hint `subtype_trans` is no longer active.

```
End HintsTransitivity.
```

In the previous example, the proof search has spent a lot of time trying to apply transitivity and reflexivity in every possible way. Its process can be summarized as follows. The first goal is `subtype S T`. Since reflexivity does not apply, `eauto` invokes transitivity, which produces two subgoals, `subtype S ?X` and `subtype ?X T`. Solving the first subgoal, `subtype S ?X`, is straightforward, it suffices to apply reflexivity. This unifies `?X` with `S`. So, the second sugoal, `subtype ?X T`, becomes `subtype S T`, which is exactly what we started from...

The problem with the transitivity lemma is that it is applicable to any goal concluding on a subtyping relation. Because of this, `eauto` keeps trying to apply it even though it most often doesn't help to solve the goal. So, one should never add a transitivity lemma as a hint for proof search.

There is a general workaround for having automation to exploit transitivity lemmas without giving up on efficiency. This workaround relies on a powerful mechanism called "external hint." This mechanism allows to manually describe the condition under which a particular lemma should be tried out during proof search.

For the case of transitivity of subtyping, we are going to tell Coq to try and apply the transitivity lemma on a goal of the form `subtype S U` only when the proof context already contains an assumption either of the form `subtype S T` or of the form `subtype T U`. In other words, we only apply the transitivity lemma when there is some evidence that this application might help. To set up this "external hint," one has to write the following.

```
Hint Extern 1 (subtype ?S ?U) ⇒
  match goal with
  | H: subtype S ?T |- _ ⇒ apply (@subtype_trans S T U)
  | H: subtype ?T U |- _ ⇒ apply (@subtype_trans S T U)
  end.
```

This hint declaration can be understood as follows.

- "Hint Extern" introduces the hint.
- The number "1" corresponds to a priority for proof search. It doesn't matter so much what priority is used in practice.

- The pattern `subtype ?S ?U` describes the kind of goal on which the pattern should apply. The question marks are used to indicate that the variables `?S` and `?U` should be bound to some value in the rest of the hint description.
- The construction `match goal with ... end` tries to recognize patterns in the goal, or in the proof context, or both.
- The first pattern is `H: subtype S ?T |- _`. It indices that the context should contain an hypothesis `H` of type `subtype S ?T`, where `S` has to be the same as in the goal, and where `?T` can have any value.
- The symbol `|- _` at the end of `H: subtype S ?T |- _` indicates that we do not impose further condition on how the proof obligation has to look like.
- The branch `⇒ apply (@subtype_trans S T U)` that follows indicates that if the goal has the form `subtype S U` and if there exists an hypothesis of the form `subtype S T`, then we should try and apply transitivity lemma instantiated on the arguments `S`, `T` and `U`. (Note: the symbol `@` in front of `subtype_trans` is only actually needed when the "Implicit Arguments" feature is activated.)
- The other branch, which corresponds to an hypothesis of the form `H: subtype ?T U` is symmetrical.

Note: the same external hint can be reused for any other transitive relation, simply by renaming `subtype` into the name of that relation.

Let us see an example illustrating how the hint works.

```
Lemma transitivity_workaround_1 : ∀ T1 T2 T3 T4,
  subtype T1 T2 → subtype T2 T3 → subtype T3 T4 → subtype T1 T4.
Proof.
  intros. (* debug *) eauto.
(* The trace shows the external hint being used *)
Qed.
```

We may also check that the new external hint does not suffer from the complexity blow up.

```
Lemma transitivity_workaround_2 : ∀ S T,
  subtype S T.
Proof.
  intros. (* debug *) eauto. (* Investigates 0 applications *)
Abort.
```

Decision Procedures

A decision procedure is able to solve proof obligations whose statement admits a particular form. This section describes three useful decision procedures. The tactic `omega` handles goals involving arithmetic and inequalities, but not general multiplications. The tactic `ring` handles goals involving arithmetic, including multiplications, but does not support inequalities. The tactic `congruence` is able to prove equalities and inequalities by exploiting equalities available in the proof context.

Omega

The tactic `omega` supports natural numbers (type `nat`) as well as integers (type `Z`, available by including the module `ZArith`). It supports addition, substraction, equalities and inequalities. Before using `omega`, one needs to import the module `Omega`, as follows.

```
Require Import Omega.
```

Here is an example. Let `x` and `y` be two natural numbers (they cannot be negative). Assume `y` is less than 4, assume `x+x+1` is less than `y`, and assume `x` is not zero. Then, it must be the case that `x` is equal to one.

```
Lemma omega_demo_1 : ∀ (x y : nat),
  (y ≤ 4) → (x + x + 1 ≤ y) → (x ≠ 0) → (x = 1).
Proof. intros. omega. Qed.
```

Another example: if `z` is the mean of `x` and `y`, and if the difference between `x` and `y` is at most `4`, then the difference between `x` and `z` is at most 2.

```
Lemma omega_demo_2 : ∀ (x y z : nat),
  (x + y = z + z) → (x - y ≤ 4) → (x - z ≤ 2).
Proof. intros. omega. Qed.
```

One can proof `False` using `omega` if the mathematical facts from the context are contradictory. In the following example, the constraints on the values `x` and `y` cannot be all satisfied in the same time.

```
Lemma omega_demo_3 : ∀ (x y : nat),
  (x + 5 ≤ y) → (y - x < 3) → False.
Proof. intros. omega. Qed.
```

Note: `omega` can prove a goal by contradiction only if its conclusion reduces to `False`. The tactic `omega` always fails when the conclusion is an arbitrary proposition `P`, even though `False` implies any proposition `P` (by `ex_falso_quodlibet`).

```
Lemma omega_demo_4 : ∀ (x y : nat) (P : Prop),
  (x + 5 ≤ y) → (y - x < 3) → P.
Proof.
  intros.
  (* Calling omega at this point fails with the message:
    "Omega: Can't solve a goal with proposition variables" *)
  (* So, one needs to replace the goal by False first. *)
  false. omega.
Qed.
```

Ring

Compared with `omega`, the tactic `ring` adds support for multiplications, however it gives up the ability to reason on inequations. Moreover, it supports only integers (type `Z`) and not natural numbers (type `nat`). Here is an example showing how to use `ring`.

```
Require Import ZArith.
Module RingDemo.
```

```
    Open Scope Z_scope.
    (* Arithmetic symbols are now interpreted in Z *)

  Lemma ring_demo : ∀ (x y z : Z),
      x * (y + z) - z * 3 * x
    = x * y - 2 * x * z.
  Proof. intros. ring. Qed.

  End RingDemo.
```

Congruence

The tactic `congruence` is able to exploit equalities from the proof context in order to automatically perform the rewriting operations necessary to establish a goal. It is slightly more powerful than the tactic `subst`, which can only handle equalities of the form `x = e` where `x` is a variable and `e` an expression.

```
Lemma congruence_demo_1 :
   ∀ (f : nat→nat→nat) (g h : nat→nat) (x y z : nat),
   f (g x) (g y) = z →
   2 = g x →
   g y = h z →
   f 2 (h z) = z.
Proof. intros. congruence. Qed.
```

Moreover, `congruence` is able to exploit universally quantified equalities, for example `∀a, g a = h a`.

```
Lemma congruence_demo_2 :
   ∀ (f : nat→nat→nat) (g h : nat→nat) (x y z : nat),
   (∀ a, g a = h a) →
   f (g x) (g y) = z →
   g x = 2 →
   f 2 (h y) = z.
Proof. congruence. Qed.
```

Next is an example where `congruence` is very useful.

```
Lemma congruence_demo_4 : ∀ (f g : nat→nat),
  (∀ a, f a = g a) →
  f (g (g 2)) = g (f (f 2)).
Proof. congruence. Qed.
```

The tactic `congruence` is able to prove a contradiction if the goal entails an equality that contradicts an inequality available in the proof context.

```
Lemma congruence_demo_3 :
   ∀ (f g h : nat→nat) (x : nat),
   (∀ a, f a = h a) →
   g x = f x →
   g x ≠ h x →
   False.
Proof. congruence. Qed.
```

One of the strengths of `congruence` is that it is a very fast tactic. So, one should not hesitate to invoke it wherever it might help.

Summary

Let us summarize the main automation tactics available.

- `auto` automatically applies `reflexivity`, `assumption`, and `apply`.
- `eauto` moreover tries `eapply`, and in particular can instantiate existentials in the conclusion.
- `iauto` extends `eauto` with support for negation, conjunctions, and disjunctions. However, its support for disjunction can make it exponentially slow.
- `jauto` extends `eauto` with support for negation, conjunctions, and existential at the head of hypothesis.
- `congruence` helps reasoning about equalities and inequalities.
- `omega` proves arithmetic goals with equalities and inequalities, but it does not support multiplication.
- `ring` proves arithmetic goals with multiplications, but does not support inequalities.

In order to set up automation appropriately, keep in mind the following rule of thumbs:

- automation is all about balance: not enough automation makes proofs not very robust on change, whereas too much automation makes proofs very hard to fix when they break.
- if a lemma is not goal directed (i.e., some of its variables do not occur in its conclusion), then the premises need to be ordered in such a way that proving the first premises maximizes the chances of correctly instantiating the variables that do not occur in the conclusion.
- a lemma whose conclusion is `False` should only be added as a local hint, i.e., as a hint within the current section.
- a transitivity lemma should never be considered as hint; if automation of transitivity reasoning is really necessary, an `Extern Hint` needs to be set up.
- a definition usually needs to be accompanied with a `Hint Unfold`.

Becoming a master in the black art of automation certainly requires some investment, however this investment will pay off very quickly.