13 Proof Search by Logic Programming

The Curry-Howard correspondence tells us that proving is just programming, but the pragmatics of the two activities are very different. Generally we care about properties of a program besides its type, but the same is not true for proofs. Any proof of a theorem will do just as well. As a result, automated proof search is conceptually simpler than automated programming.

The paradigm of logic programming [21], as embodied in languages like Prolog [42], is a good match for proof search in higher-order logic. This chapter introduces the details, attempting to avoid any dependence on past logic programming experience.

13.1 Introducing Logic Programming

Recall the definition of addition from the standard library.

```
Print plus.
```

```
plus =
fix plus (n m : nat) : nat := match n with
                              | 0 ⇒ m
                              | S p ⇒ S (plus p m)
                              end
```

This is a recursive definition, in the style of functional programming. We might also follow the style of logic programming, which corresponds to the inductive relations defined in previous chapters.

```
Inductive plusR : nat → nat → nat → Prop :=
| PlusO : ∀ m, plusR O m m
| PlusS : ∀ n m r, plusR n m r
  → plusR (S n) m (S r).
```

Intuitively, a fact **plusR** n m r only holds when plus n m $=$ r. It is not hard to prove this correspondence formally.

```
Hint Constructors plusR.

Theorem plus_plusR : ∀ n m,
  plusR n m (n + m).
  induction n; crush.
Qed.

Theorem plusR_plus : ∀ n m r,
  plusR n m r
  → r = n + m.
  induction 1; crush.
Qed.
```

With the functional definition of plus, simple equalities about arithmetic follow by computation.

```
Example four_plus_three : 4 + 3 = 7.
  reflexivity.
Qed.

Print four_plus_three.
```

four_plus_three = eq_refl

With the relational definition, the same equalities take more steps to prove, but the process is completely mechanical. For example, consider this manual proof search strategy. The steps with error messages shown afterward will be omitted from the final script.

```
Example four_plus_three' : plusR 4 3 7.
  apply PlusO.

Error: Impossible to unify "plusR 0 ?24 ?24"
with "plusR 4 3 7".

  apply PlusS.
  apply PlusO.

Error: Impossible to unify "plusR 0 ?25 ?25"
with "plusR 3 3 6".

  apply PlusS.
  apply PlusO.

Error: Impossible to unify "plusR 0 ?26 ?26"
with "plusR 2 3 5".

  apply PlusS.
  apply PlusO.
```

```
Error: Impossible to unify "plusR 0 ?27 ?27"
with "plusR 1 3 4".
```

```
  apply PlusS.
  apply PlusO.
```

At this point the proof is completed. It is no doubt clear that a simple procedure could find all proofs of this kind. We are just exploring all possible proof trees, built from the two candidate steps `apply PlusO` and `apply PlusS`. The built-in tactic `auto` follows exactly this strategy, since we used `Hint Constructors` to register the two candidate proof steps as hints.

```
Restart.
  auto.
Qed.
```

```
Print four_plus_three'.
```

```
four_plus_three' = PlusS (PlusS (PlusS (PlusS (PlusO 3))))
```

Let us try the same approach on a slightly more complex goal.

```
Example five_plus_three : plusR 5 3 8.
  auto.
```

This time, `auto` is not enough to make any progress. Since even a single candidate step may lead to an infinite space of possible proof trees, `auto` is parameterized on the maximum depth of trees to consider. The default depth is 5, and it turns out that we need depth 6 to prove the goal.

```
  auto 6.
```

Sometimes it is useful to see a description of the proof tree that `auto` finds, with the `info` tactical. (This tactical is not available in Coq 8.4, but I hope it reappears soon. The special case `info_auto` tactic is provided as a replacement for `auto`.)

```
Restart.
  info auto 6.
```

```
== apply PlusS; apply PlusS; apply PlusS; apply PlusS;
   apply PlusS; apply PlusO.
```

```
Qed.
```

The two key components of logic programming are *backtracking* and *unification*. To see these techniques in action, consider this frivolous example. Here the candidate proof steps are reflexivity and quantifier instantiation.

```
Example seven_minus_three : ∃ x, x + 3 = 7.
```

For explanatory purposes, let us proceed in the manner of a user with minimal understanding of arithmetic. We start by choosing an instantiation for the quantifier. Recall that ex_intro is the constructor for existentially quantified formulas.

```
  apply ex_intro with 0.
  reflexivity.
Error: Impossible to unify "7" with "0 + 3".
```

This seems to be a dead end. Let us backtrack to the point where we ran apply and make a better choice.

```
Restart.
  apply ex_intro with 4.
  reflexivity.
Qed.
```

This is a fairly tame example of backtracking. In general, any node in an under-construction proof tree may be the destination of backtracking an arbitrarily large number of times, as different candidate proof steps are found not to lead to full proof trees, within the depth bound passed to auto.

Next I demonstrate unification, which is easier when we switch to the relational formulation of addition.

```
Example seven_minus_three' : ∃ x, plusR x 3 7.
```

We could attempt to guess the quantifier instantiation manually as before, but there is no need. Instead of apply, we use eapply, which proceeds with placeholder *unification variables* standing in for those parameters we wish to postpone guessing.

```
  eapply ex_intro.

1 subgoal

  ============================
  plusR ?70 3 7
```

Now we can finish the proof with the right applications of plusR's constructors. Note that new unification variables are being generated to stand for new unknowns.

```
  apply PlusS.

  ============================
```

```
  plusR ?71 3 6
apply PlusS. apply PlusS. apply PlusS.

============================
  plusR ?74 3 3
apply PlusO.
```

The `auto` tactic will not perform these sorts of steps that introduce unification variables, but the `eauto` tactic will. It is helpful to work with two separate tactics, because proof search in the `eauto` style can uncover many more potential proof trees and hence take much longer to run.

```
Restart.
  info eauto 6.

== eapply ex_intro; apply PlusS; apply PlusS;
   apply PlusS; apply PlusS; apply PlusO.

Qed.
```

This proof is the first example showing that logic programming simplifies proof search compared to functional programming. In general, functional programs are only meant to be run in a single direction; a function has disjoint sets of inputs and outputs. The last example effectively ran a logic program backwards, deducing an input that gives rise to a certain output. The same works for deducing an unknown value of the other input.

```
Example seven_minus_four' : ∃ x, plusR 4 x 7.
  eauto 6.
Qed.
```

By proving the right auxiliary facts, we can reason about specific functional programs in the same way as for a logic program. Let us prove that the constructors of **plusR** have natural interpretations as lemmas about plus. We can find the first such lemma already proved in the standard library, using the `SearchRewrite` command to find a library theorem proving an equality whose left or right side matches a pattern with wildcards.

```
SearchRewrite (O + _).

plus_O_n: ∀ n : nat, 0 + n = n
```

The command `Hint Immediate` asks `auto` and `eauto` to consider this lemma as a candidate step for any leaf of a proof tree.

```
Hint Immediate plus_O_n.
```

The counterpart to plusS is proved here.

```
Lemma plusS : ∀ n m r,
  n + m = r
  → S n + m = S r.
  crush.
Qed.
```

The command `Hint Resolve` adds a new candidate proof step, to be attempted at any level of a proof tree, not just at leaves.

```
Hint Resolve plusS.
```

Now that we have registered the proper hints, we can replicate the previous examples with the normal, functional addition plus.

```
Example seven_minus_three'' : ∃ x, x + 3 = 7.
  eauto 6.
Qed.

Example seven_minus_four : ∃ x, 4 + x = 7.
  eauto 6.
Qed.
```

This new hint database is far from a complete decision procedure, as we see in a further example that `eauto` does not finish.

```
Example seven_minus_four_zero : ∃ x, 4 + x + 0 = 7.
  eauto 6.
Abort.
```

A further lemma will be helpful.

```
Lemma plusO : ∀ n m,
  n = m
  → n + 0 = m.
  crush.
Qed.

Hint Resolve plusO.
```

Note that if we consider the inputs to plus as the inputs of a corresponding logic program, the new rule plusO introduces an ambiguity. For instance, a sum $0 + 0$ would match both of plus_O_n and plusO, depending on which operand we focus on. This ambiguity may increase the number of potential search trees, slowing proof search, but semantically it presents no problems, and in fact it leads to an automated proof of the present example.

```
Example seven_minus_four_zero : ∃ x, 4 + x + 0 = 7.
  eauto 7.
Qed.
```

Just how much damage can be done by adding hints that increase the space of possible proof trees? A classic example comes from unrestricted use of transitivity, as embodied in this library theorem about equality:

```
Check eq_trans.
```

```
eq_trans
     : ∀ (A : Type) (x y z : A), x = y → y = z → x = z
```

Hints are scoped over sections, so let us enter a section to contain the effects of an unfortunate hint choice.

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

The following fact is false, but that does not stop `eauto` from taking a very long time to search for proofs of it. We use the handy `Time` command to measure how long a proof step takes to run. None of the following steps makes any progress.

```
  Example zero_minus_one : ∃ x, 1 + x = 0.
    Time eauto 1.
Finished transaction in 0. secs (0.u,0.s)

    Time eauto 2.
Finished transaction in 0. secs (0.u,0.s)

    Time eauto 3.
Finished transaction in 0. secs (0.008u,0.s)

    Time eauto 4.
Finished transaction in 0. secs (0.068005u,0.004s)

    Time eauto 5.
Finished transaction in 2. secs (1.92012u,0.044003s)
```

We see worrying exponential growth in running time, and the `debug` tactical helps us see where `eauto` is wasting its time, outputting a trace of every proof step that is attempted. The rule `eq_trans` applies at every node of a proof tree, and `eauto` tries all such positions.

```
    debug eauto 3.
```

```
1 depth=3
1.1 depth=2 eapply ex_intro
1.1.1 depth=1 apply plusO
1.1.1.1 depth=0 eapply eq_trans
1.1.2 depth=1 eapply eq_trans
1.1.2.1 depth=1 apply plus_n_O
1.1.2.1.1 depth=0 apply plusO
1.1.2.1.2 depth=0 eapply eq_trans
1.1.2.2 depth=1 apply @eq_refl
1.1.2.2.1 depth=0 apply plusO
1.1.2.2.2 depth=0 eapply eq_trans
1.1.2.3 depth=1 apply eq_add_S ; trivial
1.1.2.3.1 depth=0 apply plusO
1.1.2.3.2 depth=0 eapply eq_trans
1.1.2.4 depth=1 apply eq_sym ; trivial
1.1.2.4.1 depth=0 eapply eq_trans
1.1.2.5 depth=0 apply plusO
1.1.2.6 depth=0 apply plusS
1.1.2.7 depth=0 apply f_equal (A:=nat)
1.1.2.8 depth=0 apply f_equal2 (A1:=nat) (A2:=nat)
1.1.2.9 depth=0 eapply eq_trans
```

```
  Abort.
End slow.
```

Sometimes, though, transitivity is just what is needed to get a proof to go through automatically with `eauto`. For those cases, we can use named *hint databases* to segregate hints into different groups that may be called on as needed. Here we put `eq_trans` into the database `slow`.

```
Hint Resolve eq_trans : slow.

Example three_minus_four_zero : ∃ x, 1 + x = 0.
  Time eauto.
```

```
Finished transaction in 0. secs (0.004u,0.s)
```

This `eauto` fails to prove the goal, but at least it takes substantially less than the 2 seconds required previously.

```
Abort.
```

One simple example from before runs in the same amount of time, avoiding pollution by transitivity.

```
Example seven_minus_three_again : ∃ x, x + 3 = 7.
```

```
  Time eauto 6.
Finished transaction in 0. secs (0.004001u,0.s)
Qed.
```

When we *do* need transitivity, we ask for it explicitly.

```
Example needs_trans : ∀ x y, 1 + x = y
  → y = 2
  → ∃ z, z + x = 3.
  info eauto with slow.
 == intro x; intro y; intro H; intro H0; simple eapply ex_intro;
    apply plusS; simple eapply eq_trans.
    exact H.

    exact H0.
```

Alternatively, we can invoke `eauto` with a `using` clause, to specify extra hints without adding them to a database ahead of time.

```
  Restart.

  eauto using eq_trans.
Qed.
```

The `info` trace shows that `eq_trans` was used in just the position where it is needed to complete the proof. We also see that `auto` and `eauto` always perform `intro` steps without counting them toward the bound on proof tree depth.

13.2 Searching for Underconstrained Values

Recall the definition of the list length function.

```
Print length.

length =
fun A : Type ⇒
fix length (l : list A) : nat :=
  match l with
  | nil ⇒ 0
  | _ :: l' ⇒ S (length l')
  end
```

This function is easy to reason about in the forward direction, computing output from input.

```
Example length_1_2 : length (1 :: 2 :: nil) = 2.
```

```
  auto.
Qed.

Print length_1_2.

length_1_2 = eq_refl
```

As in Section 13.1, we will prove some lemmas to recast length in logic programming style, to help us compute inputs from outputs.

```
Theorem length_O : ∀ A, length (nil (A := A)) = O.
  crush.
Qed.

Theorem length_S : ∀ A (h : A) t n,
  length t = n
  → length (h :: t) = S n.
  crush.
Qed.

Hint Resolve length_O length_S.
```

Let us apply these hints to prove that a **list nat** of length 2 exists. (Here we register length_O with `Hint Resolve` instead of `Hint Immediate` merely as a convenience to use the same command as for length_S; `Resolve` and `Immediate` have the same meaning for a premise-free hint.)

```
Example length_is_2 : ∃ ls : list nat, length ls = 2.
  eauto.

No more subgoals but non-instantiated existential variables:
Existential 1 = ?20249 : [ |- nat]
Existential 2 = ?20252 : [ |- nat]
```

Coq complains that we finished the proof without determining the values of some unification variables created during proof search. The error message may seem a bit silly, since *any* value of type **nat** (for instance, 0) can be plugged in for either variable. However, for more complex types, finding their inhabitants may be as complex as theorem proving in general.

The `Show Proof` command shows exactly which proof term `eauto` has found, with the undetermined unification variables appearing explicitly where they are used.

```
  Show Proof.

Proof: ex_intro (fun ls : list nat => length ls = 2)
         (?20249 :: ?20252 :: nil)
```

```
          (length_S ?20249 (?20252 :: nil)
             (length_S ?20252 nil (length_O nat)))
Abort.
```

We see that the two unification variables stand for the two elements of the list. Indeed, list length is independent of data values. Paradoxically, we can make the proof search process easier by constraining the list further, so that proof search naturally locates appropriate data elements by unification. The library predicate **Forall** will be helpful.

```
Print Forall.

Inductive Forall (A : Type) (P : A → Prop) : list A → Prop :=
    Forall_nil : Forall P nil
  | Forall_cons : ∀ (x : A) (l : list A),
                    P x → Forall P l → Forall P (x :: l)

Example length_is_2 : ∃ ls : list nat, length ls = 2
  ∧ Forall (fun n ⇒ n ≥ 1) ls.
  eauto 9.
Qed.
```

We can see which list `eauto` found by printing the proof term.

```
Print length_is_2.

length_is_2 =
ex_intro
  (fun ls : list nat ⇒ length ls = 2 ∧ Forall (fun n : nat ⇒ n ≥ 1) ls)
  (1 :: 1 :: nil)
  (conj (length_S 1 (1 :: nil) (length_S 1 nil (length_O nat)))
     (Forall_cons 1 (le_n 1)
        (Forall_cons 1 (le_n 1) (Forall_nil (fun n : nat ⇒ n ≥ 1)))))
```

Let us try one more example. First, we use a standard higher-order function to define a function for summing all data elements of a list.

```
Definition sum := fold_right plus O.
```

Another basic lemma is helpful to guide proof search.

```
Lemma plusO' : ∀ n m,
  n = m
  → 0 + n = m.
  crush.
Qed.

Hint Resolve plusO'.
```

Finally, we meet `Hint Extern`, the command to register a custom hint. That is, we provide a pattern to match against goals during proof search. Whenever the pattern matches, a tactic (given to the right of an arrow ⇒) is attempted. In the following, the number 1 gives a priority for this step. Lower priorities are tried before higher priorities, which can have a significant effect on proof search time.

```
Hint Extern 1 (sum _ = _) ⇒ simpl.
```

Now we can find a length 2 list whose sum is 0.

```
Example length_and_sum : ∃ ls : list nat, length ls = 2
  ∧ sum ls = O.
  eauto 7.
Qed.
```

Printing the proof term shows the unsurprising list that is found. Here is an example where it is less obvious which list will be used. Can you guess which list `eauto` will choose?

```
Example length_and_sum' : ∃ ls : list nat, length ls = 5
  ∧ sum ls = 42.
  eauto 15.
Qed.
```

I give away part of the answer and say that this list is less interesting than one would like, because it contains too many zeroes. A further constraint forces a different solution for a smaller instance of the problem.

```
Example length_and_sum'' : ∃ ls : list nat, length ls = 2
  ∧ sum ls = 3
  ∧ Forall (fun n ⇒ n ≠ 0) ls.
  eauto 11.
Qed.
```

We could continue through exercises of this kind, but even more interesting than finding lists automatically is finding *programs* automatically.

13.3 Synthesizing Programs

Here is a simple syntax type for arithmetic expressions, similar to those used earlier. In this case, we allow expressions to mention exactly one distinguished variable.

```
Inductive exp : Set :=
```

```
| Const : nat → exp
| Var : exp
| Plus : exp → exp → exp.
```

An inductive relation specifies the semantics of an expression, relating a variable value and an expression to the expression value.

```
Inductive eval (var : nat) : exp → nat → Prop :=
| EvalConst : ∀ n, eval var (Const n) n
| EvalVar : eval var Var var
| EvalPlus : ∀ e1 e2 n1 n2, eval var e1 n1
  → eval var e2 n2
  → eval var (Plus e1 e2) (n1 + n2).

Hint Constructors eval.
```

We can use `auto` to execute the semantics for specific expressions.

```
Example eval1 : ∀ var,
  eval var (Plus Var (Plus (Const 8) Var)) (var + (8 + var)).
  auto.
Qed.
```

Unfortunately, just the constructors of `eval` are not enough to prove theorems like the following, which depends on an arithmetic identity.

```
Example eval1' : ∀ var,
  eval var (Plus Var (Plus (Const 8) Var)) (2 × var + 8).
  eauto.
Abort.
```

To help prove eval1', we prove an alternative version of EvalPlus that inserts an extra equality premise. This sort of staging is helpful to get around limitations of `eauto`'s unification: EvalPlus as a direct hint will only match goals whose results are already expressed as additions rather than as constants. In the following version, to prove the first two premises, `eauto` is given free reign in deciding the values of *n1* and *n2*; the third premise can then be proved by `reflexivity`, no matter how each of its sides is decomposed as a tree of additions.

```
Theorem EvalPlus' : ∀ var e1 e2 n1 n2 n, eval var e1 n1
  → eval var e2 n2
  → n1 + n2 = n
  → eval var (Plus e1 e2) n.
  crush.
Qed.

Hint Resolve EvalPlus'.
```

Further, we instruct `eauto` to apply `omega`, a standard tactic that provides a complete decision procedure for quantifier-free linear arithmetic. Via `Hint Extern`, we ask for use of `omega` on any equality goal. The `abstract` tactical generates a new lemma for every such successful proof, so that in the final proof term, the lemma may be referenced in place of dropping in the full proof of the arithmetic equality.

```
Hint Extern 1 (_ = _) ⇒ abstract omega.
```

Now we can return to eval1' and prove it automatically.

```
Example eval1' : ∀ var,
  eval var (Plus Var (Plus (Const 8) Var)) (2 × var + 8).
  eauto.
Qed.

Print eval1'.

eval1' =
fun var : nat ⇒
EvalPlus' (EvalVar var) (EvalPlus (EvalConst var 8) (EvalVar var))
  (eval1'_subproof var)
    : ∀ var : nat,
      eval var (Plus Var (Plus (Const 8) Var)) (2 × var + 8)
```

The lemma eval1'_subproof was generated by `abstract omega`.

Now we are ready to take advantage of logic programming's flexibility by searching for a program (arithmetic expression) that always evaluates to a particular symbolic value.

```
Example synthesize1 : ∃ e, ∀ var, eval var e (var + 7).
  eauto.
Qed.

Print synthesize1.

synthesize1 =
ex_intro (fun e : exp ⇒ ∀ var : nat, eval var e (var + 7))
  (Plus Var (Const 7))
  (fun var : nat ⇒ EvalPlus (EvalVar var) (EvalConst var 7))
```

Here are two more examples demonstrating program synthesis:

```
Example synthesize2 : ∃ e, ∀ var, eval var e (2 × var + 8).
  eauto.
Qed.

Example synthesize3 : ∃ e, ∀ var, eval var e (3 × var + 42).
  eauto.
```

```
Qed.
```

These examples show linear expressions over the variable *var*. Any such expression is equivalent to $k \times var + n$ for some k and n. We can prove that any expression's semantics is equivalent to some such linear expression, but it is tedious to prove such a fact manually. We use `eauto` to complete the proof, finding k and n values automatically.

We prove a series of lemmas and add them as hints. We have alternative `eval` constructor lemmas and some facts about arithmetic.

```
Theorem EvalConst' : ∀ var n m, n = m
  → eval var (Const n) m.
  crush.
Qed.

Hint Resolve EvalConst'.

Theorem zero_times : ∀ n m r,
  r = m
  → r = 0 × n + m.
  crush.
Qed.

Hint Resolve zero_times.

Theorem EvalVar' : ∀ var n,
  var = n
  → eval var Var n.
  crush.
Qed.

Hint Resolve EvalVar'.

Theorem plus_0 : ∀ n r,
  r = n
  → r = n + 0.
  crush.
Qed.

Theorem times_1 : ∀ n, n = 1 × n.
  crush.
Qed.

Hint Resolve plus_0 times_1.
```

We finish with one more arithmetic lemma that is particularly specialized to this theorem. This fact follows by the axioms of the *ring* algebraic structure; since the naturals form a semiring, we can use the built-in tactic `ring`.

```
Require Import Arith Ring.

Theorem combine : ∀ x k1 k2 n1 n2,
  (k1 × x + n1) + (k2 × x + n2) = (k1 + k2) × x + (n1 + n2).
  intros; ring.
Qed.

Hint Resolve combine.
```

This choice of hints is cheating, to an extent, by telegraphing the procedure for choosing values of k and n. Nonetheless, with these lemmas in place, we achieve an automated proof without explicitly orchestrating the lemmas' composition.

```
Theorem linear : ∀ e, ∃ k, ∃ n,
  ∀ var, eval var e (k × var + n).
  induction e; crush; eauto.
Qed.
```

By printing the proof term, it is possible to see the procedure that is used to choose the constants for each input term.

13.4 More on `auto` Hints

Let us take stock of the possibilities for `auto` and `eauto` hints. Hints are contained within *hint databases*, which we have seen extended in many examples so far. When no hint database is specified, a default database is used. Hints in the default database are always used by `auto` or `eauto`. The chance to extend hint databases imperatively is important, because, in Ltac programming, we cannot create global variables whose values can be extended seamlessly by different modules in different source files. We have seen the advantages of hints so far, where *crush* can be defined once and for all, while still automatically applying the hints added throughout developments. In fact, *crush* is defined in terms of `auto`, which explains how we achieve this extensibility. Other user-defined tactics can take similar advantage of `auto` and `eauto`.

The basic hints for `auto` and `eauto` are `Hint Immediate` *lemma*, asking to try solving a goal immediately by applying a lemma and discharging any hypotheses with a single proof step each; `Resolve` *lemma*, which does the same but may add new premises that are themselves to be subjects of nested proof search; `Constructors` *typename*, which acts like `Resolve` applied to every constructor of an inductive type; and `Unfold` *ident*, which tries unfolding *ident* when it appears at the head of a proof goal. Each of these `Hint` commands may be used with a suffix, as in `Hint Resolve` *lemma* : *my_db*, to add the hint only to the

specified database, so that it would only be used by, for instance, `auto with` *my_db*. An additional argument to `auto` specifies the maximum depth of proof trees to search in depth-first order, as in `auto` 8 or `auto` 8 `with` *my_db*. The default depth is 5.

All these `Hint` commands can be expressed with a more primitive hint kind, `Extern`. A few more examples of `Hint Extern` illustrate the possibilities.

```
Theorem bool_neq : true ≠ false.
  auto.
```

A call to *crush* would have discharged this goal, but the default hint database for `auto` contains no hint that applies.

```
Abort.
```

It is hard to come up with a **bool**-specific hint that is not just a restatement of the theorem we mean to prove. Luckily, a simpler form suffices, by appealing to the built-in tactic `congruence`, a complete procedure for the theory of equality, uninterpreted functions, and datatype constructors.

```
Hint Extern 1 (_ ≠ _) ⇒ congruence.

Theorem bool_neq : true ≠ false.
  auto.
Qed.
```

A `Hint Extern` may be implemented with the full Ltac language. This example shows a case where a hint uses a `match`.

```
Section forall_and.
  Variable A : Set.
  Variables P Q : A → Prop.

  Hypothesis both : ∀ x, P x ∧ Q x.

  Theorem forall_and : ∀ z, P z.
    crush.
```

The *crush* invocation makes no progress beyond what `intros` would have accomplished. An `auto` invocation will not apply the hypothesis *both* to prove the goal, because the conclusion of *both* does not unify with the conclusion of the goal. However, we can teach `auto` to handle this kind of goal.

```
    Hint Extern 1 (P ?X) ⇒
      match goal with
        | [ H : ∀ x, P x ∧ _ ⊢ _ ] ⇒ apply (proj1 (H X))
```

```
      end.

    auto.
  Qed.
```

We see that an `Extern` pattern may bind unification variables used in the associated tactic. The function `proj1` is from the standard library, for extracting a proof of U from a proof of $U \wedge V$.

```
End forall_and.
```

We might get more ambitious and seek to generalize the hint to all possible predicates P.

```
Hint Extern 1 (?P ?X) ⇒
  match goal with
    | [ H : ∀ x, P x ∧ _ ⊢ _ ] ⇒ apply (proj1 (H X))
  end.
```

```
User error: Bound head variable
```

Coq's `auto` hint databases work as tables mapping *head symbols* to lists of tactics to try. Because of this, the constant head of an `Extern` pattern must be determinable statically. In the first `Extern` hint, the head symbol was `not`, since $x \neq y$ desugars to `not` (`eq` x y); and in the second example, the head symbol was P.

Fortunately, a more basic form of `Hint Extern` also applies. We may simply leave out the pattern to the left of the ⇒, incorporating the corresponding logic into the Ltac script.

```
Hint Extern 1 ⇒
  match goal with
    | [ H : ∀ x, ?P x ∧ _ ⊢ ?P ?X ] ⇒ apply (proj1 (H X))
  end.
```

Be forewarned that a `Hint Extern` of this kind will be applied at *every* node of a proof tree, so an extensive Ltac script may slow proof search significantly.

13.5 Rewrite Hints

Another dimension of extensibility with hints is rewriting with quantified equalities. We have used the associated command `Hint Rewrite` in many examples so far. The *crush* tactic uses these hints by calling the built-in tactic `autorewrite`. The rewrite hints have taken the form `Hint Rewrite` *lemma*, which by default adds them to the default hint

database *core*; but other hint databases may also be specified, just as with `Hint Resolve`, for instance.

The next example shows a direct use of `autorewrite`. Note that whereas `Hint Rewrite` uses a default database, `autorewrite` requires that a database be named.

```
Section autorewrite.
  Variable A : Set.
  Variable f : A → A.

  Hypothesis f_f : ∀ x, f (f x) = f x.

  Hint Rewrite f_f.

  Lemma f_f_f : ∀ x, f (f (f x)) = f x.
    intros; autorewrite with core; reflexivity.
  Qed.
```

There are a few ways in which `autorewrite` can lead to trouble when insufficient care is taken in choosing hints. First, the set of hints may define a nonterminating rewrite system, in which case invocations to `autorewrite` may not terminate. Second, we may add hints that lead `autorewrite` down the wrong path. For instance,

```
  Section garden_path.
    Variable g : A → A.
    Hypothesis f_g : ∀ x, f x = g x.
    Hint Rewrite f_g.

    Lemma f_f_f' : ∀ x, f (f (f x)) = f x.
      intros; autorewrite with core.

============================
 g (g (g x)) = g x

    Abort.
```

The new hint was used to rewrite the goal into a form where the old hint could no longer be applied. This nonmonotonicity of rewrite hints contrasts with the situation for `auto`, where new hints may slow down proof search but can never break old proofs. The key difference is that `auto` either solves a goal or makes no changes to it, whereas `autorewrite` may change goals without solving them. The situation for `eauto` is slightly more complicated, as changes to hint databases may change the proof found for a particular goal, and that proof may influence the settings of unification variables that appear elsewhere in the proof state.

```
Reset garden_path.
```

The `autorewrite` tactic also works with quantified equalities that include additional premises, but we must be careful to avoid similar incorrect rewritings.

```
Section garden_path.
  Variable P : A → Prop.
  Variable g : A → A.
  Hypothesis f_g : ∀ x, P x → f x = g x.
  Hint Rewrite f_g.

  Lemma f_f_f' : ∀ x, f (f (f x)) = f x.
    intros; autorewrite with core.
```

```
============================
 g (g (g x)) = g x

subgoal 2 is:
 P x
subgoal 3 is:
 P (f x)
subgoal 4 is:
 P (f x)
```

```
  Abort.
```

The inappropriate rule fired the same three times as before, even though we know we will not be able to prove the premises.

```
Reset garden_path.
```

The final, successful, attempt uses an extra argument to `Hint Rewrite` that specifies a tactic to apply to generated premises. Such a hint is only used when the tactic succeeds for all premises, possibly leaving further subgoals for some premises.

```
Section garden_path.
  Variable P : A → Prop.
  Variable g : A → A.
  Hypothesis f_g : ∀ x, P x → f x = g x.
  Hint Rewrite f_g using assumption.

  Lemma f_f_f' : ∀ x, f (f (f x)) = f x.
    intros; autorewrite with core; reflexivity.
  Qed.
```

We may still use `autorewrite` to apply *f_g* when the generated premise is among the assumptions.

```
  Lemma f_f_f_g : ∀ x, P x → f (f x) = g x.
    intros; autorewrite with core; reflexivity.
  Qed.
End garden_path.
```

It can also be useful to apply the `autorewrite with` *db* `in` * form, which does rewriting in hypotheses as well as in the conclusion.

```
Lemma in_star : ∀ x y, f (f (f (f x))) = f (f y)
  → f x = f (f (f y)).
  intros; autorewrite with core in *; assumption.
Qed.

End autorewrite.
```

Many proofs can be automated in modular ways with deft combinations of `auto` and `autorewrite`.