14 Proof Search in Ltac

We have seen many examples of proof automation so far, some with code snippets from Ltac, Coq's domain-specific language for proof search procedures. This chapter gives a bottom-up presentation of the features of Ltac, focusing in particular on the Ltac `match` construct, which supports a novel approach to backtracking search. First, the chapter runs through some useful automation tactics that are built into Coq. They are described in detail in the Coq manual, so I only outline what is possible.

14.1 Some Built-in Automation Tactics

A number of tactics are called repeatedly by *crush*. The `intuition` tactic simplifies propositional structure of goals. The `congruence` tactic applies the rules of equality and congruence closure, plus properties of constructors of inductive types. The `omega` tactic provides a complete decision procedure for a theory called quantifier-free linear arithmetic or Presburger arithmetic by different communities. That is, `omega` proves any goal that follows from looking only at parts of that goal that can be interpreted as propositional formulas whose atomic formulas are basic comparison operations on natural numbers or integers, with operands built from constants, variables, addition, and subtraction (with multiplication by a constant available as a shorthand for addition or subtraction).

The `ring` tactic solves goals by appealing to the axioms of rings or semirings (as in algebra), depending on the type involved. Coq developments may declare new types to be parts of rings and semirings by proving the associated axioms. There is a similar tactic, `field`, for simplifying values in fields by conversion to fractions over rings. Both `ring` and `field` can only solve goals that are equalities. The `fourier` tactic

uses Fourier's method to prove inequalities over real numbers, which are axiomatized in the Coq standard library.

The *setoid* facility makes it possible to register new equivalence relations to be understood by tactics like `rewrite`. For instance, `Prop` is registered as a setoid with the equivalence relation "if and only if." The ability to register new setoids can be very useful in proofs of a kind common in math, where all reasoning is done after modding out by a relation.

There are several other built-in automation tactics described in the Coq manual. The real promise of Coq, though, is in the coding of problem-specific tactics with Ltac.

14.2 Ltac Programming Basics

We have already seen many examples of Ltac programs. This chapter gives a thorough introduction to the important features and design patterns.

One common use for `match` tactics is identification of subjects for case analysis, as we see in this tactic definition:

```
Ltac find_if :=
  match goal with
    | [ ⊢ if ?X then _ else _ ] ⇒ destruct X
  end.
```

The tactic checks if the conclusion is an `if`, `destruct`ing the test expression if so. Certain classes of theorem are trivial to prove automatically with such a tactic.

```
Theorem hmm : ∀ (a b c : bool),
  if a
    then if b
      then True
      else True
    else if c
      then True
      else True.
  intros; repeat find_if; constructor.
Qed.
```

The `repeat` used here is called a *tactical*, or tactic combinator. The behavior of `repeat` t is to loop through running t, running t on all generated subgoals, running t on *their* generated subgoals, and so on. When t fails at any point in this search tree, that particular subgoal is

left to be handled by later tactics. Thus, it is important never to use `repeat` with a tactic that always succeeds.

Another very useful Ltac building block is *context patterns*.

```
Ltac find_if_inside :=
  match goal with
    | [ ⊢ context[if ?X then _ else _] ] ⇒ destruct X
  end.
```

The behavior of this tactic is to find any subterm of the conclusion that is an `if` and then `destruct` the test expression. This version subsumes *find_if*:

```
Theorem hmm' : ∀ (a b c : bool),
  if a
    then if b
      then True
      else True
    else if c
      then True
      else True.
  intros; repeat find_if_inside; constructor.
Qed.
```

We can also use *find_if_inside* to prove goals that *find_if* does not simplify sufficiently.

```
Theorem hmm2 : ∀ (a b : bool),
  (if a then 42 else 42) = (if b then 42 else 42).
  intros; repeat find_if_inside; reflexivity.
Qed.
```

Many decision procedures can be coded in Ltac via `repeat match` loops. For instance, we can implement a subset of the functionality of `tauto`.

```
Ltac my_tauto :=
  repeat match goal with
           | [ H : ?P ⊢ ?P ] ⇒ exact H

           | [ ⊢ True ] ⇒ constructor
           | [ ⊢ _ ∧ _ ] ⇒ constructor
           | [ ⊢ _ → _ ] ⇒ intro

           | [ H : False ⊢ _ ] ⇒ destruct H
           | [ H : _ ∧ _ ⊢ _ ] ⇒ destruct H
           | [ H : _ ∨ _ ⊢ _ ] ⇒ destruct H
```

```
    | [ H1 : ?P → ?Q, H2 : ?P ⊢ _ ] ⇒ specialize (H1 H2)
  end.
```

Since `match` patterns can share unification variables between hypothesis and conclusion patterns, it is easy to figure out when the conclusion matches a hypothesis. The `exact` tactic solves a goal completely when given a proof term of the proper type.

It is also trivial to implement the introduction rules (in the sense of natural deduction [37]) for a few of the connectives. Implementing elimination rules is only a little more work, since we must give a name for a hypothesis to `destruct`.

The last rule implements modus ponens, using a tactic `specialize`, which will replace a hypothesis with a version that is specialized to a provided set of arguments (for quantified variables or local hypotheses from implications). By convention, when the argument to `specialize` is an application of a hypothesis H to a set of arguments, the result of the specialization replaces H. For other terms, the outcome is the same as with `generalize`.

```
Section propositional.
  Variables P Q R : Prop.

  Theorem propositional : (P ∨ Q ∨ False) ∧ (P → Q) → True ∧ Q.
    my_tauto.
  Qed.
End propositional.
```

It was relatively easy to implement modus ponens because we do not lose information by clearing every implication that we use. If we want to implement a similarly complete procedure for quantifier instantiation, we need a way to ensure that a particular proposition is not already included among the hypotheses. To do that effectively, we first need to know a bit more about the semantics of `match`.

It is tempting to assume that `match` works as it does in ML. In fact, there are a few critical differences in its behavior. One is that we may include arbitrary expressions in patterns instead of being restricted to variables and constructors. Another is that the same variable may appear multiple times, inducing an implicit equality constraint.

A related pair of two other differences is much more important than the others. The `match` construct has a *backtracking semantics for failure*. In ML, pattern matching works by finding the first pattern to match and then executing its body. If the body raises an exception, then the

overall match raises the same exception. In Coq, failures in case bodies instead trigger continued search through the list of cases.

For instance, this proof script works:

```
Theorem m1 : True.
  match goal with
    | [ ⊢ _ ] ⇒ intro
    | [ ⊢ True ] ⇒ constructor
  end.
Qed.
```

The first case matches trivially, but its body tactic fails, since the conclusion does not begin with a quantifier or implication. In a similar ML match, the whole pattern match would fail. In Coq, we backtrack and try the next pattern, which also matches. Its body tactic succeeds, so the overall tactic succeeds as well.

The example shows how failure can move to a different pattern within a `match`. Failure can also trigger an attempt to find *a different way of matching a single pattern*. Consider another example.

```
Theorem m2 : ∀ P Q R : Prop, P → Q → R → Q.
  intros; match goal with
            | [ H : _ ⊢ _ ] ⇒ idtac H
          end.
```

Coq prints "*H1*". By applying `idtac` with an argument, a convenient debugging tool for leaking information out of `match`es, we see that this `match` first tries binding H to *H1*, which cannot be used to prove Q. Nonetheless, the following variation on the tactic succeeds at proving the goal.

```
  match goal with
    | [ H : _ ⊢ _ ] ⇒ exact H
  end.
Qed.
```

The tactic first unifies H with *H1*, as before, but `exact` H fails in that case, so the tactic engine searches for more possible values of H. Eventually, it arrives at the correct value, so that `exact` H and the overall tactic succeed.

Now we are equipped to implement a tactic for checking that a proposition is not among the hypotheses.

```
Ltac notHyp P :=
  match goal with
    | [ _ : P ⊢ _ ] ⇒ fail 1
    | _ ⇒
```

```
      match P with
        | ?P1 ∧ ?P2 ⇒ first [ notHyp P1 | notHyp P2 | fail 2 ]
        | _ ⇒ idtac
      end
  end.
```

We use the equality checking that is built into pattern matching to see if there is a hypothesis that matches the proposition exactly. If so, we use the `fail` tactic. Without arguments, `fail` signals normal tactic failure, as one might expect. When `fail` is passed an argument n, n is used to count outwards through the enclosing cases of backtracking search. In this case, `fail` 1 says "fail not just in this pattern-matching branch but for the whole `match`." The second case will never be tried when the `fail` 1 is reached.

This second case, used when P matches no hypothesis, checks if P is a conjunction. Other simplifications may have split conjunctions into their component formulas, so we need to check that at least one of those components is also not represented. To achieve this, we apply the `first` tactical, which takes a list of tactics and continues down the list until one of them does not fail. The `fail` 2 at the end says to `fail` both the `first` and the `match` wrapped around it.

The body of the ?*P1* ∧ ?*P2* case guarantees that if it is reached, we either succeed completely or fail completely. Thus, if we reach the wildcard case, P is not a conjunction. We use `idtac`, a tactic that would not be applied on its own, since its effect is to succeed at doing nothing. Nonetheless, `idtac` is a useful placeholder for cases like this.

With the nonpresence check implemented, it is easy to build a tactic that takes as input a proof term and adds its conclusion as a new hypothesis, only if that conclusion is not already present, failing otherwise.

```
Ltac extend pf :=
  let t := type of pf in
    notHyp t; generalize pf; intro.
```

We see the useful `type of` operator of Ltac. This operator could not be implemented in Gallina, but it is easy to support in Ltac. We end up with t bound to the type of *pf*. We check that t is not already present. If so, we use a `generalize`/`intro` combo to add a new hypothesis proved by *pf*. The tactic `generalize` takes as input a term t (for instance, a proof of some proposition) and then changes the conclusion from G to $T \to G$, where T is the type of t (for instance, the proposition proved by the proof t).

With these tactics defined, we can write a tactic *completer* for, among other things, adding to the context all consequences of a set of simple first-order formulas.

```
Ltac completer :=
  repeat match goal with
           | [ ⊢ _ ∧ _ ] ⇒ constructor
           | [ H : _ ∧ _ ⊢ _ ] ⇒ destruct H
           | [ H : ?P → ?Q, H' : ?P ⊢ _ ] ⇒ specialize (H H')
           | [ ⊢ ∀ x, _ ] ⇒ intro

           | [ H : ∀ x, ?P x → _, H' : ?P ?X ⊢ _ ] ⇒
             extend (H X H')
         end.
```

We use the same kind of conjunction and implication handling as previously. Note that since → is the special nondependent case of ∀, the fourth rule handles `intro` for implications, too.

In the fifth rule, when we find a ∀ fact H with a premise matching one of the hypotheses, we add the appropriate instantiation of H's conclusion, if we have not already added it.

We can check that *completer* is working properly, with a theorem that introduces a spurious variable.

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

  Hypothesis H1 : ∀ x, P x → Q x ∧ R x.
  Hypothesis H2 : ∀ x, R x → S x.

  Theorem fo : ∀ (y x : A), P x → S x.
    completer.


y : A
x : A
H : P x
H0 : Q x
H3 : R x
H4 : S x
============================
 S x

    assumption.
  Qed.
```

```
End firstorder.
```

We narrowly avoided a subtle pitfall in the definition of *completer*. Let us try another definition that even seems preferable to the original, to the untrained eye. (We change the second `match` case a bit to make the tactic smart enough to handle some subtleties of Ltac behavior that had not been exercised previously.)

```
Ltac completer' :=
  repeat match goal with
           | [ ⊢ _ ∧ _ ] ⇒ constructor
           | [ H : ?P ∧ ?Q ⊢ _ ] ⇒ destruct H;
             repeat match goal with
                      | [ H' : P ∧ Q ⊢ _ ] ⇒ clear H'
                    end
           | [ H : ?P → _, H' : ?P ⊢ _ ] ⇒ specialize (H H')
           | [ ⊢ ∀ x, _ ] ⇒ intro

           | [ H : ∀ x, ?P x → _, H' : ?P ?X ⊢ _ ] ⇒
             extend (H X H')
         end.
```

The only other difference is in the modus ponens rule, where we have replaced an unused unification variable $?Q$ with a wildcard. Let us try our example again with this version:

```
Section firstorder'.
  Variable A : Set.
  Variables P Q R S : A → Prop.

  Hypothesis H1 : ∀ x, P x → Q x ∧ R x.
  Hypothesis H2 : ∀ x, R x → S x.

  Theorem fo' : ∀ (y x : A), P x → S x.
    completer'.


  y : A
  H1 : P y → Q y ∧ R y
  H2 : R y → S y
  x : A
  H : P x
  ============================
   S x
```

The quantified theorems have been instantiated with y instead of x, reducing a provable goal to one that is unprovable. The code in the last

match case for *completer'* is careful only to instantiate quantifiers along with suitable hypotheses, so why were incorrect choices made?

```
  Abort.
End firstorder'.
```

A few examples should illustrate the issue. This match-based proof works fine:

```
Theorem t1 : ∀ x : nat, x = x.
  match goal with
    | [ ⊢ ∀ x, _ ] ⇒ trivial
  end.
Qed.
```

This one fails:

```
Theorem t1' : ∀ x : nat, x = x.
  match goal with
    | [ ⊢ ∀ x, ?P ] ⇒ trivial
  end.

User error: No matching clauses for match goal

Abort.
```

The problem is that unification variables may not contain locally bound variables. In this case, $?P$ would need to be bound to $x = x$, which contains the local quantified variable x. By using a wildcard in the earlier version, we avoided this restriction. To understand why this restriction affects the behavior of the *completer* tactic, recall that, in Coq, implication is shorthand for degenerate universal quantification where the quantified variable is not used. Nonetheless, in an Ltac pattern, Coq matches a wildcard implication against a universal quantification.

The Coq 8.2 release includes a special pattern form for a unification variable with an explicit set of free variables. That unification variable is then bound to a function from the free variables to the real value. In Coq 8.1 and earlier, there is no such work-around. Section 15.5 shows an example of this binding form.

No matter which Coq version is used, it is important to be aware of this restriction. As mentioned, the restriction is the culprit behind the surprising behavior of *completer'*. We unintentionally match quantified facts with the modus ponens rule, circumventing the check that a suitably matching hypothesis is available and leading to different behavior, where wrong quantifier instantiations are chosen. The earlier *completer*

tactic uses a modus ponens rule that matches the implication conclusion with a variable, which blocks matching against nontrivial universal quantifiers.

Actually, the behavior demonstrated here applies to Coq version 8.4, but not 8.4pl1. The latter version will allow regular Ltac pattern variables to match terms that contain locally bound variables, but a tactic failure occurs if the variable is later used as a Gallina term.

14.3 Functional Programming in Ltac

Ltac supports quite convenient functional programming, with a Lisp-with-syntax kind of flavor. However, there are a few syntactic conventions involved in getting programs to be accepted. The Ltac syntax is optimized for tactic writing, so one has to deal with some inconveniences in writing more standard functional programs.

To illustrate, let us try to write a simple list length function. We start out writing it just as in Gallina, simply replacing `Fixpoint` (and its annotations) with `Ltac`.

```
Ltac length ls :=
  match ls with
    | nil ⇒ O
    | _ :: ls' ⇒ S (length ls')
  end.
```

```
Error: The reference ls' was not found in the current environment
```

At this point, recall that pattern variable names must be prefixed by question marks in Ltac.

```
Ltac length ls :=
  match ls with
    | nil ⇒ O
    | _ :: ?ls' ⇒ S (length ls')
  end.
```

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

The problem is that Ltac treats the expression S (length *ls'*) as an invocation of a tactic S with argument length *ls'*. We need to use a special annotation to escape into the Gallina parsing nonterminal.

```
Ltac length ls :=
  match ls with
```

```
    | nil ⇒ O
    | _ :: ?ls' ⇒ constr:(S (length ls'))
  end.
```

This definition is accepted. It can be a little awkward to test Ltac definitions like this one. Here is one method:

```
Goal False.
  let n := length (1 :: 2 :: 3 :: nil) in
    pose n.

  n := S (length (2 :: 3 :: nil)) : nat
  ============================
   False
```

We use the `pose` tactic, which extends the proof context with a new variable that is set equal to a particular term. We could also have used `idtac` *n* in place of `pose` *n*, which would have printed the result without changing the context.

The value of *n* only has the length calculation unrolled one step. What has happened here is that by escaping into the `constr` nonterminal, we referred to the length function of Gallina rather than to the length Ltac function that we are defining.

```
Abort.

Reset length.
```

Gallina terms built by tactics must be bound explicitly via `let` or a similar technique rather than inserting Ltac calls directly in other Gallina terms.

```
Ltac length ls :=
  match ls with
    | nil ⇒ O
    | _ :: ?ls' ⇒
      let ls'' := length ls' in
        constr:(S ls'')
  end.

Goal False.
  let n := length (1 :: 2 :: 3 :: nil) in
    pose n.

  n := 3 : nat
  ============================
   False
```

```
Abort.
```

We can also use anonymous function expressions and local function definitions in Ltac, as this example of a standard list map function shows:

```
Ltac map T f :=
  let rec map' ls :=
    match ls with
      | nil ⇒ constr:(@nil T)
      | ?x :: ?ls' ⇒
        let x' := f x in
          let ls'' := map' ls' in
            constr:(x' :: ls'')
    end in
  map'.
```

Ltac functions can have no implicit arguments. It may seem surprising that we need to pass T, the carried type of the output list, explicitly. We cannot just use `type of` f, because f is an Ltac term, not a Gallina term, and Ltac programs are dynamically typed. The function f could use very syntactic methods to decide to return differently typed terms for different inputs. We also could not replace `constr:(@nil T)` with `constr:nil`, because we have no strongly typed context to use to infer the parameter to nil. Luckily, we do have sufficient context within `constr:(x' :: ls'')`.

Sometimes we need to employ the opposite direction of nonterminal escape, when we want to pass a complicated tactic expression as an argument to another tactic, as we might want to do in invoking *map*.

```
Goal False.
  let ls := map (nat × nat)%type ltac:(fun x ⇒ constr:(x, x))
    (1 :: 2 :: 3 :: nil) in
    pose ls.

  l := (1, 1) :: (2, 2) :: (3, 3) :: nil : list (nat × nat)
  ============================
   False

Abort.
```

Each position within an Ltac script has a default applicable nonterminal, where `constr` and `ltac` are the main options, standing respectively for terms of Gallina and Ltac. The explicit colon notation can always be used to override the default nonterminal choice, though code being

parsed as Gallina can no longer use such overrides. Within the `ltac` nonterminal, top-level function applications are treated as applications in Ltac, not Gallina; but the *arguments* to such functions are parsed with `constr` by default. This choice may seem strange, until we realize that we have been relying on it all along in proof scripts. For instance, the `apply` tactic is an Ltac function, and it is natural to interpret its argument as a term of Gallina, not Ltac. We use an `ltac` prefix to parse Ltac function arguments as Ltac terms themselves, as in the call to *map* in the previous example. For some simple cases, Ltac terms may be passed without an extra prefix. For instance, an identifier that has an Ltac meaning but no Gallina meaning will be interpreted in Ltac automatically.

One other problem shows up when we want to debug Ltac functional programs. We might expect the following code to work, to yield a version of *length* that prints a debug trace of the arguments it is called with.

```
Reset length.

Ltac length ls :=
  idtac ls;
  match ls with
    | nil ⇒ O
    | _ :: ?ls' ⇒
      let ls'' := length ls' in
        constr:(S ls'')
  end.
```

Coq accepts the tactic definition, but the code is fatally flawed and will always lead to dynamic type errors.

```
Goal False.

  let n := length (1 :: 2 :: 3 :: nil) in
    pose n.

Error: variable n should be bound to a term.

Abort.
```

What is going wrong here? The problem has to do with the dual status of Ltac as both a purely functional and an imperative programming language. The basic programming language is purely functional, but tactic scripts are one "datatype" that can be returned by such programs, and Coq will run such a script using an imperative semantics that mutates proof states. Readers familiar with monadic programming in Haskell [44, 34] may recognize a similarity. Haskell programs with side

effects can be thought of as pure programs that return *the code of programs in an imperative language*, where some out-of-band mechanism takes responsibility for running these derived programs. In this way, Haskell remains pure while supporting usual input-output side effects and more. Ltac uses the same basic mechanism, but in a dynamically typed setting. Here the embedded imperative language includes all the tactics we have applied so far.

Even basic `idtac` is an embedded imperative program, so we may not automatically mix it with purely functional code. In fact, a semicolon operator alone marks a span of Ltac code as an embedded tactic script. This makes some sense, since pure functional languages have no need for sequencing. Because they lack side effects, there is no reason to run an expression and then just throw away its value and move on to another expression.

Another explanation that avoids an analogy to Haskell monads is that an Ltac tactic program returns a function that, when run later, will perform the desired proof modification. These functions are distinct from other types of data, like numbers or Gallina terms. The prior, correctly working version of length computed solely with Gallina terms, but the new one implicitly returns a tactic function, as indicated by the use of `idtac` and semicolon. However, the new version's recursive call to length is structured to expect a Gallina term, not a tactic function, as output. As a result, we have a basic dynamic type error, perhaps obscured by the involvement of first-class tactic scripts.

The solution is like the one in Haskell: we must monadify the pure program to give it access to side effects. The trouble is that the embedded tactic language has no `return` construct. Proof scripts are about proving theorems, not calculating results. We can apply a somewhat awkward work-around that requires translating the program into *continuation-passing style* [39], a program-structuring idea popular in functional programming.

```
Reset length.

Ltac length ls k :=
  idtac ls;
  match ls with
    | nil ⇒ k O
    | _ :: ?ls' ⇒ length ls' ltac:(fun n ⇒ k (S n))
  end.
```

The new length takes a new input: a *continuation* k, which is a function to be called to continue whatever proving process we were in the

middle of when we called *length*. The argument passed to k may be thought of as the return value of *length*.

```
Goal False.
  length (1 :: 2 :: 3 :: nil) ltac:(fun n ⇒ pose n).


(1 :: 2 :: 3 :: nil)
(2 :: 3 :: nil)
(3 :: nil)
nil

Abort.
```

We see exactly the trace of function arguments that we expected initially, and an examination of the proof state afterward would show that variable n has been added with value 3.

Considering the comparison with Haskell's IO monad, there is an important subtlety that deserves to be mentioned. A Haskell IO computation represents (theoretically, at least) a transformer from one state of the real world to another, plus a pure value to return. Some of the state can be very specific to the program, as in the case of heap-allocated mutable references, but some can be along the lines of the example "launch missile," where the program has a side effect on the real world that is not possible to undo.

In contrast, Ltac scripts can be thought of as controlling just two simple kinds of mutable state. First, there is the current sequence of proof subgoals. Second, there is a partial assignment of discovered values to unification variables introduced by proof search (for instance, by `eauto`, as shown in Chapter 13). Crucially, *every mutation of this state can be undone* during backtracking introduced by `match`, `auto`, and other built-in Ltac constructs. Ltac proof scripts have state, but it is purely local, and all changes to it are reversible, which is a very useful semantics for proof search.

14.4 Recursive Proof Search

Deciding how to instantiate quantifiers is one of the hardest parts of automated first-order theorem proving. For a given problem, we can consider all possible bounded-length sequences of quantifier instantiations, applying only propositional reasoning at the end. This is probably a bad idea for almost all goals, but it makes for a nice example of recursive proof search procedures in Ltac.

We can consider the maximum dependency chain length for a first-order proof. We define the chain length for a hypothesis to be 0, and the chain length for an instantiation of a quantified fact to be one greater than the length for that fact. The tactic *inster* n is meant to try all possible proofs with chain length at most n.

```
Ltac inster n :=
  intuition;
    match n with
      | S ?n' ⇒
        match goal with
          | [ H : ∀ x : ?T, _, y : ?T ⊢ _ ] ⇒
            generalize (H y); inster n'
        end
    end.
```

The tactic begins by applying propositional simplification. Next, it checks if any chain length remains, failing if not. Otherwise, it tries all possible ways of instantiating quantified hypotheses with properly typed local variables. It is critical to realize that if the recursive call *inster n'* fails, then the `match goal` just seeks out another way of unifying its pattern against proof state. Thus, this small amount of code provides an elegant demonstration of how backtracking `match` enables exhaustive search.

We can verify the efficacy of *inster* with two short examples. The built-in `firstorder` tactic (with no extra arguments) is able to prove the first but not the second.

```
Section test_inster.
  Variable A : Set.
  Variables P Q : A → Prop.
  Variable f : A → A.
  Variable g : A → A → A.

  Hypothesis H1 : ∀ x y, P (g x y) → Q (f x).

  Theorem test_inster : ∀ x, P (g x x) → Q (f x).
    inster 2.
  Qed.

  Hypothesis H3 : ∀ u v, P u ∧ P v ∧ u ≠ v → P (g u v).
  Hypothesis H4 : ∀ u, Q (f u) → P u ∧ P (f u).

  Theorem test_inster2 : ∀ x y, x ≠ y → P x → Q (f y) → Q (f x).
    inster 3.
  Qed.
End test_inster.
```

The style employed in the definition of *inster* can seem counterintuitive to functional programmers. Usually, functional programs accumulate state changes in explicit arguments to recursive functions. In Ltac, the state of the current subgoal is always implicit. Nonetheless, in accord with the discussion at the end of Section 14.3, in contrast to general imperative programming, it is easy to undo any changes to this state, and indeed such undoing happens automatically at failures within `match`es. In this way, Ltac programming is similar to programming in Haskell with a stateful failure monad that supports a composition operator along the lines of the `first` tactical.

Functional programming purists may reject programming in this way. Nonetheless, as with other kinds of monadic programming, many problems are much simpler to solve with Ltac than they would be with explicit, pure proof manipulation in ML or Haskell. To demonstrate, we write a basic simplification procedure for logical implications.

This procedure is inspired by one for separation logic [40], where conjuncts in formulas are thought of as resources, such that we lose no completeness by crossing out equal conjuncts on the two sides of an implication. This process is complicated by the fact that, for reasons of modularity, the formulas can have arbitrary nested tree structure (branching at conjunctions) and may include existential quantifiers. It is helpful for the matching process to "go under" quantifiers and in fact decide how to instantiate existential quantifiers in the conclusion.

To distinguish the implications that the tactic handles from the implications that show up as plumbing in various lemmas, we define a wrapper definition, a notation, and a tactic.

```
Definition imp (P1 P2 : Prop) := P1 → P2.
Infix "->" := imp (no associativity, at level 95).
Ltac imp := unfold imp; firstorder.
```

The following lemmas about `imp` will be useful in writing the tactic.

```
Theorem and_True_prem : ∀ P Q,
  (P ∧ True -> Q)
  → (P -> Q).
  imp.
Qed.

Theorem and_True_conc : ∀ P Q,
  (P -> Q ∧ True)
  → (P -> Q).
  imp.
Qed.
```

```
Theorem pick_prem1 : ∀ P Q R S,
  (P ∧ (Q ∧ R) -> S)
  → ((P ∧ Q) ∧ R -> S).
  imp.
Qed.

Theorem pick_prem2 : ∀ P Q R S,
  (Q ∧ (P ∧ R) -> S)
  → ((P ∧ Q) ∧ R -> S).
  imp.
Qed.

Theorem comm_prem : ∀ P Q R,
  (P ∧ Q -> R)
  → (Q ∧ P -> R).
  imp.
Qed.

Theorem pick_conc1 : ∀ P Q R S,
  (S -> P ∧ (Q ∧ R))
  → (S -> (P ∧ Q) ∧ R).
  imp.
Qed.

Theorem pick_conc2 : ∀ P Q R S,
  (S -> Q ∧ (P ∧ R))
  → (S -> (P ∧ Q) ∧ R).
  imp.
Qed.

Theorem comm_conc : ∀ P Q R,
  (R -> P ∧ Q)
  → (R -> Q ∧ P).
  imp.
Qed.
```

The first order of business in crafting the *matcher* tactic is to include auxiliary support for searching through formula trees. The *search_prem* tactic implements running its tactic argument *tac* on every subformula of an imp premise. As it traverses a tree, *search_prem* applies some of the preceding lemmas to rewrite the goal to bring different subformulas to the head of the goal. That is, for every subformula P of the implication premise, we want P to have a turn, where the premise is rearranged into the form $P \wedge Q$ for some Q. The tactic *tac* should expect to see a goal in this form and focus its attention on the first conjunct of the premise.

```
Ltac search_prem tac :=
  let rec search P :=
    tac
    || (apply and_True_prem; tac)
    || match P with
         | ?P1 ∧ ?P2 ⇒
           (apply pick_prem1; search P1)
           || (apply pick_prem2; search P2)
       end
  in match goal with
       | [ ⊢ ?P ∧ _ -> _ ] ⇒ search P
       | [ ⊢ _ ∧ ?P -> _ ] ⇒ apply comm_prem; search P
       | [ ⊢ _ -> _ ] ⇒ progress (tac || (apply and_True_prem; tac))
     end.
```

To understand how *search_prem* works, we turn first to the final `match`. If the premise begins with a conjunction, we call the *search* procedure on each of the conjuncts, or only the first conjunct if that already yields a case where *tac* does not fail. The call *search P* expects and maintains the invariant that the premise is of the form $P \wedge Q$ for some Q. We pass P explicitly as a kind of decreasing induction measure, to avoid looping forever when *tac* always fails. The second `match` case calls a commutativity lemma to realize this invariant, before passing control to *search*. The final `match` case tries applying *tac* directly and then, if that fails, changes the form of the goal by adding an extraneous **True** conjunct and calls *tac* again. The `progress` tactical fails when its argument tactic succeeds without changing the current subgoal.

The *search* function itself tries the same routine as in the last case of the final `match`, using the || operator as a shorthand for trying one tactic and then, if the first fails, trying another. Additionally, if neither works, it checks if P is a conjunction. If so, it calls itself recursively on each conjunct, first applying associativity/commutativity lemmas to maintain the goal-form invariant.

We also want a dual function *search_conc*, which does tree search through an imp conclusion.

```
Ltac search_conc tac :=
  let rec search P :=
    tac
    || (apply and_True_conc; tac)
    || match P with
         | ?P1 ∧ ?P2 ⇒
           (apply pick_conc1; search P1)
```

```
            || (apply pick_conc2; search P2)
      end
  in match goal with
       | [ ⊢ _ -> ?P ∧ _ ] ⇒ search P
       | [ ⊢ _ -> _ ∧ ?P ] ⇒ apply comm_conc; search P
       | [ ⊢ _ -> _ ] ⇒ progress (tac || (apply and_True_conc; tac))
     end.
```

Now we can prove a number of lemmas that are suitable for application by the search tactics. A lemma that is meant to handle a premise should have the form $P \wedge Q \dashrightarrow R$ for some interesting P, and a lemma that is meant to handle a conclusion should have the form $P \dashrightarrow Q \wedge R$ for some interesting Q.

```
Theorem False_prem : ∀ P Q,
  False ∧ P -> Q.
  imp.
Qed.

Theorem True_conc : ∀ P Q : Prop,
  (P -> Q)
  → (P -> True ∧ Q).
  imp.
Qed.

Theorem Match : ∀ P Q R : Prop,
  (Q -> R)
  → (P ∧ Q -> P ∧ R).
  imp.
Qed.

Theorem ex_prem : ∀ (T : Type) (P : T → Prop) (Q R : Prop),
  (∀ x, P x ∧ Q -> R)
  → (ex P ∧ Q -> R).
  imp.
Qed.

Theorem ex_conc : ∀ (T : Type) (P : T → Prop) (Q R : Prop) x,
  (Q -> P x ∧ R)
  → (Q -> ex P ∧ R).
  imp.
Qed.
```

We also want a base case lemma for finishing proofs where cancellation has removed every constituent of the conclusion.

```
Theorem imp_True : ∀ P,
```

```
  P -> True.
  imp.
Qed.
```

The final *matcher* tactic is now straightforward. First, we `intros` all variables into scope. Then we attempt simple premise simplifications, finishing the proof upon finding **False** and eliminating any existential quantifiers that we find. After that, we search through the conclusion. We remove **True** conjuncts, remove existential quantifiers by introducing unification variables for their bound variables, and search for matching premises to cancel. Finally, when no more progress is made, we see if the goal has become trivial and can be solved by imp_True. In each case, we use the tactic `simple apply` in place of `apply` to use a simpler, less expensive unification algorithm.

```
Ltac matcher :=
  intros;
    repeat search_prem ltac:(simple apply False_prem
      || (simple apply ex_prem; intro));
    repeat search_conc ltac:(simple apply True_conc
      || simple eapply ex_conc
      || search_prem ltac:(simple apply Match));
    try simple apply imp_True.
```

The tactic succeeds at proving a simple example.

```
Theorem t2 : ∀ P Q : Prop,
  Q ∧ (P ∧ False) ∧ P -> P ∧ Q.
  matcher.
Qed.
```

In the generated proof, we find a trace of the workings of the search tactics.

```
Print t2.
```

```
t2 =
fun P Q : Prop ⇒
comm_prem (pick_prem1 (pick_prem2
  (False_prem (P:=P ∧ P ∧ Q) (P ∧ Q))))
     : ∀ P Q : Prop, Q ∧ (P ∧ False) ∧ P -> P ∧ Q
```

We can also see that *matcher* is well-suited for cases where some human intervention is needed after the automation finishes.

```
Theorem t3 : ∀ P Q R : Prop,
  P ∧ Q -> Q ∧ R ∧ P.
```

```
  matcher.

  ============================
   True -> R
```

The tactic canceled those conjuncts that it was able to cancel, leaving a simplified subgoal, much as `intuition` does.

```
Abort.
```

The *matcher* tactic even succeeds at guessing quantifier instantiations. It is the unification that occurs in uses of the Match lemma that does the real work here.

```
Theorem t4 : ∀ (P : nat → Prop) Q, (∃ x, P x ∧ Q)
  -> Q ∧ (∃ x, P x).
  matcher.
Qed.

Print t4.

t4 =
fun (P : nat → Prop) (Q : Prop) ⇒
and_True_prem
  (ex_prem (P:=fun x : nat ⇒ P x ∧ Q)
     (fun x : nat ⇒
      pick_prem2
        (Match (P:=Q)
           (and_True_conc
              (ex_conc (fun x0 : nat ⇒ P x0) x
                 (Match (P:=P x) (imp_True (P:=True))))))))
     : ∀ (P : nat → Prop) (Q : Prop),
       (∃ x : nat, P x ∧ Q) -> Q ∧ (∃ x : nat, P x)
```

We can be glad that we did not have to build this proof term manually.

14.5 Creating Unification Variables

A final useful ingredient in tactic crafting is the ability to allocate new unification variables explicitly. Tactics like `eauto` introduce unification variables internally to support flexible proof search. While `eauto` and its relatives do *backward* reasoning, we often want to do similar *forward* reasoning, where unification variables can be useful for similar reasons.

For example, we can write a tactic that instantiates the quantifiers of a universally quantified hypothesis. The tactic should not need to know

what the appropriate instantiations are; rather, we want these choices filled with placeholders. We hope that when we apply the specialized hypothesis later, syntactic unification will determine concrete values.

Before we are ready to write a tactic, we can try out its ingredients one at a time.

```
Theorem t5 : (∀ x : nat, S x > x) → 2 > 1.
  intros.
```

```
H : ∀ x : nat, S x > x
============================
 2 > 1
```

To instantiate H generically, we first need to name the value to be used for x.

```
  evar (y : nat).
```

```
H : ∀ x : nat, S x > x
y := ?279 : nat
============================
 2 > 1
```

The proof context is extended with a new variable y, which has been assigned to be equal to a fresh unification variable ?279. We want to instantiate H with ?279. To get hold of the new unification variable rather than just its alias y, we perform a trivial unfolding in the expression y, using the `eval` Ltac construct, which works with the same reduction strategies seen in tactics (e.g., `simpl`, `compute`, etc.).

```
  let y' := eval unfold y in y in
    clear y; specialize (H y').
```

```
H : S ?279 > ?279
============================
  2 > 1
```

The instantiation was successful. We can finish the proof by using `apply`'s unification to figure out the proper value of ?279.

```
  apply H.
```

```
Qed.
```

Now we can write a tactic that encapsulates the pattern we just employed, instantiating all quantifiers of a particular hypothesis.

```
Ltac insterU H :=
  repeat match type of H with
           | ∀ x : ?T, _ ⇒
             let x := fresh "x" in
               evar (x : T);
               let x' := eval unfold x in x in
                 clear x; specialize (H x')
         end.

Theorem t5' : (∀ x : nat, S x > x) → 2 > 1.
  intro H; insterU H; apply H.
Qed.
```

This particular example is somewhat trivial, since `apply` by itself would have solved the goal originally. Separate forward reasoning is more useful on hypotheses that end in existential quantifications. Before we go through an example, it is useful to define a variant of *insterU* that does not clear the base hypothesis we pass to it. We use the Ltac construct `fresh` to generate a hypothesis name that is not already used, based on a string suggesting a good name.

```
Ltac insterKeep H :=
  let H' := fresh "H'" in
    generalize H; intro H'; insterU H'.

Section t6.
  Variables A B : Type.
  Variable P : A → B → Prop.
  Variable f : A → A → A.
  Variable g : B → B → B.

  Hypothesis H1 : ∀ v, ∃ u, P v u.
  Hypothesis H2 : ∀ v1 u1 v2 u2,
    P v1 u1
    → P v2 u2
    → P (f v1 v2) (g u1 u2).

  Theorem t6 : ∀ v1 v2, ∃ u1, ∃ u2, P (f v1 v2) (g u1 u2).
    intros.
```

Neither `eauto` nor `firstorder` is clever enough to prove this goal. We can help out by doing some of the work with quantifiers, abbreviating

the proof with the `do` tactical for repetition of a tactic a set number of times.

```
do 2 insterKeep H1.
```

The proof state is extended with two generic instances of *H1*.

```
H' : ∃ u : B, P ?4289 u
H'0 : ∃ u : B, P ?4288 u
============================
 ∃ u1 : B, ∃ u2 : B, P (f v1 v2) (g u1 u2)
```

Normal `eauto` still cannot prove the goal, so we eliminate the two new existential quantifiers. (Recall that **ex** is the underlying type family to which uses of the ∃ syntax are compiled.)

```
repeat match goal with
         | [ H : ex _ ⊢ _ ] ⇒ destruct H
       end.
```

Now the goal is simple enough to solve by logic programming.

```
  eauto.
Qed.
End t6.
```

The *insterU* tactic does not fare so well with quantified hypotheses that also contain implications. We can see the problem in a slight modification of the last example. We introduce a new unary predicate *Q* and use it to state an additional requirement of the hypothesis *H1*.

```
Section t7.
  Variables A B : Type.
  Variable Q : A → Prop.
  Variable P : A → B → Prop.
  Variable f : A → A → A.
  Variable g : B → B → B.

  Hypothesis H1 : ∀ v, Q v → ∃ u, P v u.
  Hypothesis H2 : ∀ v1 u1 v2 u2,
    P v1 u1
    → P v2 u2
    → P (f v1 v2) (g u1 u2).

  Theorem t7 : ∀ v1 v2, Q v1 → Q v2 → ∃ u1, ∃ u2,
    P (f v1 v2) (g u1 u2).
    intros; do 2 insterKeep H1;
```

```
repeat match goal with
         | [ H : ex _ ⊢ _ ] ⇒ destruct H
       end; eauto.
```

This proof script does not hit any errors until the very end, when an error message like this one is displayed:

```
No more subgoals but non-instantiated existential variables :
Existential 1 =
?4384 : [A : Type
         B : Type
         Q : A → Prop
         P : A → B → Prop
         f : A → A → A
         g : B → B → B
         H1 : ∀ v : A, Q v → ∃ u : B, P v u
         H2 : ∀ (v1 : A) (u1 : B) (v2 : A) (u2 : B),
              P v1 u1 → P v2 u2 → P (f v1 v2) (g u1 u2)
         v1 : A
         v2 : A
         H : Q v1
         H0 : Q v2
         H' : Q v2 → ∃ u : B, P v2 u ⊢ Q v2]
```

There is another similar line about a different existential variable. Here, *existential variable* means what we have also called unification variable. In the course of the proof, some unification variable ?4384 was introduced but never unified. Unification variables are just a device to structure proof search; the language of Gallina proof terms does not include them. Thus, we cannot produce a proof term without instantiating the variable.

The error message shows that ?4384 is meant to be a proof of *Q v2* in a particular proof state, whose variables and hypotheses are displayed. It turns out that ?4384 was created by *insterU*, as the value of a proof to pass to *H1*. Recall that, in Gallina, implication is just a degenerate case of ∀ quantification, so the *insterU* code to match against ∀ also matched the implication. Since any proof of *Q v2* is as good as any other in this context, there was never any opportunity to use unification to determine exactly which proof is appropriate. We expect similar problems with any implications in arguments to *insterU*.

```
  Abort.
End t7.

Reset insterU.
```

We can redefine *insterU* to treat implications differently. In particular, we pattern-match on the type of the type T in $\forall\ x : ?T$, ... If T has type `Prop`, then x's instantiation should be thought of as a proof. Thus, instead of picking a new unification variable for it, we apply a user-supplied tactic *tac*. It is important that we end this special `Prop` case with || `fail` 1, so that, if *tac* fails to prove T, we abort the instantiation rather than continuing on to the default quantifier handling. Also recall that the tactic form `solve` [t] fails if t does not completely solve the goal.

```
Ltac insterU tac H :=
  repeat match type of H with
           | ∀ x : ?T, _ ⇒
             match type of T with
               | Prop ⇒
                 (let H' := fresh "H'" in
                   assert (H' : T) by solve [ tac ];
                     specialize (H H'); clear H')
                 || fail 1
               | _ ⇒
                 let x := fresh "x" in
                   evar (x : T);
                   let x' := eval unfold x in x in
                     clear x; specialize (H x')
             end
         end.

Ltac insterKeep tac H :=
  let H' := fresh "H'" in
    generalize H; intro H'; insterU tac H'.

Section t7.
  Variables A B : Type.
  Variable Q : A → Prop.
  Variable P : A → B → Prop.
  Variable f : A → A → A.
  Variable g : B → B → B.

  Hypothesis H1 : ∀ v, Q v → ∃ u, P v u.
  Hypothesis H2 : ∀ v1 u1 v2 u2,
    P v1 u1
    → P v2 u2
    → P (f v1 v2) (g u1 u2).

  Theorem t7 : ∀ v1 v2, Q v1 → Q v2 → ∃ u1, ∃ u2,
```

```
    P (f v1 v2) (g u1 u2).
```

We can prove the goal by calling *insterKeep* with a tactic that tries to find and apply a Q hypothesis over a variable about which we do not yet know any P facts. We need to begin this tactic code with `idtac`; to get around a strange limitation in Coq's proof engine, where a first-class tactic argument may not begin with a `match`.

```
  intros; do 2 insterKeep
    ltac:(idtac; match goal with
                   | [ H : Q ?v ⊢ _ ] ⇒
                     match goal with
                       | [ _ : context[P v _] ⊢ _ ] ⇒ fail 1
                       | _ ⇒ apply H
                     end
                 end) H1;
  repeat match goal with
           | [ H : ex _ ⊢ _ ] ⇒ destruct H
         end; eauto.
  Qed.
End t7.
```

It is often useful to instantiate existential variables explicitly. A built-in tactic provides one way of doing so.

```
Theorem t8 : ∃ p : nat × nat, fst p = 3.
  econstructor; instantiate (1 := (3, 2)); reflexivity.
Qed.
```

The 1 identifies an existential variable appearing in the current goal, with the last existential assigned number 1, the second-last assigned number 2, and so on. The named existential is replaced everywhere by the term to the right of the :=.

The `instantiate` tactic can be convenient for exploratory proving, but it leads to very brittle proof scripts that are unlikely to adapt to changing theorem statements. It is often more helpful to have a tactic that can be used to assign a value to a term that is known to be an existential. By employing a roundabout implementation technique, we can build a tactic that generalizes this functionality. In particular, the tactic *equate* will assert that two terms are equal. If one of the terms happens to be an existential, then it will be replaced everywhere with the other term.

```
Ltac equate x y :=
  let dummy := constr:(eq_refl : x = y) in idtac.
```

This tactic fails if it is not possible to prove $x = y$ by eq_refl. We check the proof only for its unification side effects, ignoring the associated variable *dummy*. With *equate*, we can build a less brittle version of the prior example.

```
Theorem t9 : ∃ p : nat × nat, fst p = 3.
  econstructor; match goal with
                  | [ ⊢ fst ?x = 3 ] ⇒ equate x (3, 2)
                end; reflexivity.
Qed.
```

This technique is even more useful within recursive and iterative tactics that are meant to solve broad classes of goals.

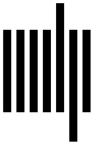
Certified Programming with Dependent Types

A Pragmatic Introduction to the Coq Proof Assistant

By: Adam Chlipala

Citation:

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



The MIT Press

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

Library of Congress Cataloging-in-Publication Data

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

2013012837

10 9 8 7 6 5 4 3 2 1