SOFTWARE FOUNDATIONS

VOLUME 2: PROGRAMMING LANGUAGE FOUNDATIONS

USETACTICS

TACTIC LIBRARY FOR COQ: A GENTLE INTRODUCTION

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

Coq comes with a set of builtin tactics, such as `reflexivity`, `intros`, `inversion` and so on. While it is possible to conduct proofs using only those tactics, you can significantly increase your productivity by working with a set of more powerful tactics. This chapter describes a number of such useful tactics, which, for various reasons, are not yet available by default in Coq. These tactics are defined in the `LibTactics.v` file.

```
Set Warnings "-notation-overridden,-parsing".

Require Import Coq.Arith.Arith.

Require Import Maps.
Require Import Imp.
Require Import Types.
Require Import Smallstep.
Require Import LibTactics.

Require Stlc.
Require Equiv.
Require Imp.
Require References.
Require Smallstep.
Require Hoare.
Require Sub.
```

Remark: SSReflect is another package providing powerful tactics. The library "LibTactics" differs from "SSReflect" in two respects:

- "SSReflect" was primarily developed for proving mathematical theorems, whereas "LibTactics" was primarily developed for proving theorems on programming languages. In particular, "LibTactics" provides a number of useful tactics that have no counterpart in the "SSReflect" package.

- "SSReflect" entirely rethinks the presentation of tactics, whereas "LibTactics" mostly stick to the traditional presentation of Coq tactics, simply providing a number of additional tactics. For this reason, "LibTactics" is probably easier to get started with than "SSReflect".

This chapter is a tutorial focusing on the most useful features from the "LibTactics" library. It does not aim at presenting all the features of "LibTactics". The detailed specification of tactics can be found in the source file `LibTactics.v`. Further documentation as well as demos can be found at http://www.chargueraud.org/softs/tlc/.

In this tutorial, tactics are presented using examples taken from the core chapters of the "Software Foundations" course. To illustrate the various ways in which a given tactic can be used, we use a tactic that duplicates a given goal. More precisely, `dup` produces two copies of the current goal, and `dup n` produces `n` copies of it.

Tactics for Introduction and Case Analysis

This section presents the following tactics:

- `introv`, for naming hypotheses more efficiently,
- `inverts`, for improving the `inversion` tactic,
- `cases`, for performing a case analysis without losing information,
- `cases_if`, for automating case analysis on the argument of `if`.

The Tactic `introv`

```
Module IntrovExamples.
  Import Stlc.
  Import Imp.
  Import STLC.
```

The tactic `introv` allows to automatically introduce the variables of a theorem and explicitly name the hypotheses involved. In the example shown next, the variables `c`, `st`, st_1 and st_2 involved in the statement of determinism need not be named explicitly, because their name where already given in the statement of the lemma. On the contrary, it is useful to provide names for the two hypotheses, which we name E_1 and E_2, respectively.

```
Theorem ceval_deterministic: ∀ c st st1 st2,
  c / st \\ st1 →
  c / st \\ st2 →
  st1 = st2.
Proof.
```

```
    introv E₁ E₂. (* was intros c st st₁ st₂ E₁ E₂ *)
  Abort.
```

When there is no hypothesis to be named, one can call `introv` without any argument.

```
  Theorem dist_exists_or : ∀ (X:Type) (P Q : X → Prop),
    (∃ x, P x ∨ Q x) ↔ (∃ x, P x) ∨ (∃ x, Q x).
  Proof.
    introv. (* was intros X P Q *)
  Abort.
```

The tactic `introv` also applies to statements in which ∀and → are interleaved.

```
  Theorem ceval_deterministic': ∀ c st st₁,
    (c / st \\ st₁) → ∀ st₂, (c / st \\ st₂) → st₁ = st₂.
  Proof.
    introv E₁ E₂. (* was intros c st st₁ E₁ st₂ E₂ *)
  Abort.
```

Like the arguments of `intros`, the arguments of `introv` can be structured patterns.

```
  Theorem exists_impl: ∀ X (P : X → Prop) (Q : Prop) (R : Prop),
        (∀ x, P x → Q) →
        ((∃ x, P x) → Q).
  Proof.
    introv [x H₂]. eauto.
    (* same as intros X P Q R H₁ [x H₂]., which is itself short
       for intros X P Q R H₁ H₂. destruct H₂ as [x H₂]. *)
  Qed.
```

Remark: the tactic `introv` works even when definitions need to be unfolded in order to reveal hypotheses.

```
  End IntrovExamples.
```

The Tactic `inverts`

```
  Module InvertsExamples.
    Import Stlc.
    Import Equiv.
    Import Imp.
    Import STLC.
```

The `inversion` tactic of Coq is not very satisfying for three reasons. First, it produces a bunch of equalities which one typically wants to substitute away, using `subst`. Second, it introduces meaningless names for hypotheses. Third, a call to `inversion H` does not remove `H` from the context, even though in most cases an hypothesis is no longer needed after being inverted. The tactic `inverts` address all of these three issues. It is intented to be used in place of the tactic `inversion`.

The following example illustrates how the tactic `inverts H` behaves mostly like `inversion H` except that it performs some substitutions in order to eliminate the trivial equalities that are being produced by `inversion`.

```
Theorem skip_left: ∀ c,
  cequiv (SKIP;; c) c.
Proof.
  introv. split; intros H.
  dup. (* duplicate the goal for comparison *)
  (* was... *)
  - inversion H. subst. inversion H2. subst. assumption.
  (* now... *)
  - inverts H. inverts H2. assumption.
Abort.
```

A slightly more interesting example appears next.

```
Theorem ceval_deterministic: ∀ c st st1 st2,
  c / st \\ st1 →
  c / st \\ st2 →
  st1 = st2.
Proof.
  introv E1 E2. generalize dependent st2.
  induction E1; intros st2 E2.
  admit. admit. (* skip some basic cases *)
  dup. (* duplicate the goal for comparison *)
  (* was: *)
  - inversion E2. subst. admit.
  (* now: *)
  - inverts E2. admit.
Abort.
```

The tactic `inverts H as.` is like `inverts H` except that the variables and hypotheses being produced are placed in the goal rather than in the context. This strategy allows naming those new variables and hypotheses explicitly, using either `intros` or `introv`.

```
Theorem ceval_deterministic': ∀ c st st1 st2,
  c / st \\ st1 →
  c / st \\ st2 →
  st1 = st2.
Proof.
  introv E1 E2. generalize dependent st2.
  (induction E1); intros st2 E2;
    inverts E2 as.
  - (* E_Skip *) reflexivity.
  - (* E_Ass *)
    (* Observe that the variable n is not automatically
       substituted because, contrary to inversion E2; subst,
       the tactic inverts E2 does not substitute the equalities
       that exist before running the inversion. *)
    (* new: *) subst n.
    reflexivity.
  - (* E_Seq *)
```

```
      (* Here, the newly created variables can be introduced
         using intros, so they can be assigned meaningful names,
         for example st3 instead of st'0. *)
      (* new: *) intros st3 Red1 Red2.
      assert (st' = st3) as EQ1.
      { (* Proof of assertion *) apply IHE1_1; assumption. }
      subst st3.
      apply IHE1_2. assumption.
    (* E_IfTrue *)
    - (* b1 reduces to true *)
      (* In an easy case like this one, there is no need to
         provide meaningful names, so we can just use intros *)
      (* new: *) intros.
      apply IHE1. assumption.
    - (* b1 reduces to false (contradiction) *)
      (* new: *) intros.
      rewrite H in H5. inversion H5.
    (* The other cases are similiar *)
  Abort.
```

In the particular case where a call to `inversion` produces a single subgoal, one can use the syntax `inverts H as H1 H2 H3` for calling `inverts` and naming the new hypotheses `H1`, `H2` and `H3`. In other words, the tactic `inverts H as H1 H2 H3` is equivalent to `inverts H as; introv H1 H2 H3`. An example follows.

```
  Theorem skip_left': ∀ c,
    cequiv (SKIP;; c) c.
  Proof.
    introv. split; intros H.
    inverts H as U V. (* new hypotheses are named U and V *)
    inverts U. assumption.
  Abort.
```

A more involved example appears next. In particular, this example shows that the name of the hypothesis being inverted can be reused.

```
  Example typing_nonexample_1 :
    ¬ ∃ T,
        has_type empty
          (tabs x TBool
              (tabs y TBool
                 (tapp (tvar x) (tvar y))))
          T.
  Proof.
    dup 3.

    (* The old proof: *)
    - intros C. destruct C.
    inversion H. subst. clear H.
    inversion H5. subst. clear H5.
    inversion H4. subst. clear H4.
    inversion H2. subst. clear H2.
```

```
    inversion H5. subst. clear H5.
    inversion H1.

    (* The new proof: *)
    - intros C. destruct C.
    inverts H as H1.
    inverts H1 as H2.
    inverts H2 as H3.
    inverts H3 as H4.
    inverts H4.

    (* The new proof, alternative: *)
    - intros C. destruct C.
    inverts H as H.
    inverts H as H.
    inverts H as H.
    inverts H as H.
    inverts H.
  Qed.

  End InvertsExamples.
```

Note: in the rare cases where one needs to perform an inversion on an hypothesis `H` without clearing `H` from the context, one can use the tactic `inverts keep H`, where the keyword `keep` indicates that the hypothesis should be kept in the context.

Tactics for N-ary Connectives

Because Coq encodes conjunctions and disjunctions using binary constructors ∧ and ∨, working with a conjunction or a disjunction of `N` facts can sometimes be quite cumbursome. For this reason, "LibTactics" provides tactics offering direct support for n-ary conjunctions and disjunctions. It also provides direct support for n-ary existententials.

This section presents the following tactics:

- `splits` for decomposing n-ary conjunctions,
- `branch` for decomposing n-ary disjunctions,
- ∃for proving n-ary existentials.

```
  Module NaryExamples.
    Import References.
    Import Smallstep.
    Import STLCRef.
```

The Tactic `splits`

The tactic `splits` applies to a goal made of a conjunction of `n` propositions and it produces `n` subgoals. For example, it decomposes the goal $G_1 \land G_2 \land G_3$ into the three subgoals G_1, G_2 and G_3.

```
Lemma demo_splits : ∀ n m,
  n > 0 ∧ n < m ∧ m < n+10 ∧ m ≠ 3.
Proof.
  intros. splits.
Abort.
```

The Tactic `branch`

The tactic `branch k` can be used to prove a n-ary disjunction. For example, if the goal takes the form $G_1 \lor G_2 \lor G_3$, the tactic `branch 2` leaves only G_2 as subgoal. The following example illustrates the behavior of the `branch` tactic.

```
Lemma demo_branch : ∀ n m,
  n < m ∨ n = m ∨ m < n.
Proof.
  intros.
  destruct (lt_eq_lt_dec n m) as [[H1|H2]|H3].
  - branch 1. apply H1.
  - branch 2. apply H2.
  - branch 3. apply H3.
Qed.
```

The Tactic ∃

The library "LibTactics" introduces a notation for n-ary existentials. For example, one can write `∃x y z, H` instead of `∃x, ∃y, ∃z, H`. Similarly, the library provides a n-ary tactic `∃a b c`, which is a shorthand for `∃a; ∃b; ∃c`. The following example illustrates both the notation and the tactic for dealing with n-ary existentials.

```
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'.
  (* was: value t ∨ ∃t', ∃st', t / st ==> t' / st' *)
Proof with eauto.
  intros ST t T st Ht HST. remember (@empty ty) as Gamma.
  (induction Ht); subst; try solve_by_invert...
  - (* T_App *)
    right. destruct IHHt1 as [Ht1p | Ht1p]...
    + (* t1 is a value *)
      inversion Ht1p; subst; try solve_by_invert.
      destruct IHHt2 as [Ht2p | Ht2p]...
      (* t2 steps *)
      inversion Ht2p as [t2' [st' Hstep]].
      ∃ (tapp (tabs x T t) t2') st'...
      (* was: ∃(tapp (tabs x T t) t2'). ∃st'... *)
Abort.
```

Remark: a similar facility for n-ary existentials is provided by the module `Coq.Program.Syntax` from the standard library. (`Coq.Program.Syntax` supports existentials up to arity 4; `LibTactics` supports them up to arity 10.

```
End NaryExamples.
```

Tactics for Working with Equality

One of the major weakness of Coq compared with other interactive proof assistants is its relatively poor support for reasoning with equalities. The tactics described next aims at simplifying pieces of proof scripts manipulating equalities.

This section presents the following tactics:

- asserts_rewrite for introducing an equality to rewrite with,
- cuts_rewrite, which is similar except that its subgoals are swapped,
- substs for improving the subst tactic,
- fequals for improving the f_equal tactic,
- applys_eq for proving P x y using an hypothesis P x z, automatically producing an equality y = z as subgoal.

```
Module EqualityExamples.
```

The Tactics asserts_rewrite and cuts_rewrite

The tactic asserts_rewrite ($E_1 = E_2$) replaces E_1 with E_2 in the goal, and produces the goal $E_1 = E_2$.

```
Theorem mult_0_plus : ∀ n m : nat,
  (0 + n) * m = n * m.
Proof.
  dup.
  (* The old proof: *)
  intros n m.
  assert (H: 0 + n = n). reflexivity. rewrite → H.
  reflexivity.

  (* The new proof: *)
  intros n m.
  asserts_rewrite (0 + n = n).
    reflexivity. (* subgoal 0+n = n *)
    reflexivity. (* subgoal n*m = m*n *)
Qed.

(*** Remark: the syntax asserts_rewrite (E1 = E2) in H allows
     rewriting in the hypothesis H rather than in the goal. *)
```

The tactic cuts_rewrite ($E_1 = E_2$) is like asserts_rewrite ($E_1 = E_2$), except that the equality $E_1 = E_2$ appears as first subgoal.

```
Theorem mult_0_plus' : ∀ n m : nat,
  (0 + n) * m = n * m.
Proof.
  intros n m.
  cuts_rewrite (0 + n = n).
    reflexivity. (* subgoal n*m = m*n *)
```

```
    reflexivity. (* subgoal 0+n = n *)
Qed.
```

More generally, the tactics `asserts_rewrite` and `cuts_rewrite` can be provided a lemma as argument. For example, one can write `asserts_rewrite (∀a b, a*(S b) = a*b+a)`. This formulation is useful when `a` and `b` are big terms, since there is no need to repeat their statements.

```
Theorem mult_0_plus'' : ∀ u v w x y z: nat,
  (u + v) * (S (w * x + y)) = z.
Proof.
  intros. asserts_rewrite (∀ a b, a*(S b) = a*b+a).
    (* first subgoal:  ∀a b, a*(S b) = a*b+a *)
    (* second subgoal: (u + v) * (w * x + y) + (u + v) = z *)
Abort.
```

The Tactic `substs`

The tactic `substs` is similar to `subst` except that it does not fail when the goal contains "circular equalities", such as `x = f x`.

```
Lemma demo_substs : ∀ x y (f:nat→nat),
  x = f x → y = x → y = f x.
Proof.
  intros. substs. (* the tactic subst would fail here *)
  assumption.
Qed.
```

The Tactic `fequals`

The tactic `fequals` is similar to `f_equal` except that it directly discharges all the trivial subgoals produced. Moreover, the tactic `fequals` features an enhanced treatment of equalities between tuples.

```
Lemma demo_fequals : ∀ (a b c d e : nat) (f :
nat→nat→nat→nat→nat),
  a = 1 → b = e → e = 2 →
  f a b c d = f 1 2 c 4.
Proof.
  intros. fequals.
  (* subgoals a = 1, b = 2 and c = c are proved, d =
4 remains *)
Abort.
```

The Tactic `applys_eq`

The tactic `applys_eq` is a variant of `eapply` that introduces equalities for subterms that do not unify. For example, assume the goal is the proposition `P x y` and assume we have the assumption `H` asserting that `P x z` holds. We know that we can prove `y` to be equal to `z`. So, we could call the tactic `assert_rewrite (y = z)` and change the goal to `P x z`, but this would require copy-pasting the values of `y` and `z`. With the tactic `applys_eq`, we can call `applys_eq H 1`, which proves the goal and leaves only the subgoal `y = z`. The value `1` given as argument to `applys_eq` indicates that we want an equality to be introduced for the first argument of `P x y` counting from the right. The three following examples illustrate the behavior of a call to `applys_eq H 1`, a call to `applys_eq H 2`, and a call to `applys_eq H 1 2`.

```
Axiom big_expression_using : nat→nat. (* Used in the example *)

Lemma demo_applys_eq_1 : ∀ (P:nat→nat→Prop) x y z,
  P x (big_expression_using z) →
  P x (big_expression_using y).
Proof.
  introv H. dup.

  (* The old proof: *)
  assert (Eq: big_expression_using y = big_expression_using z).
    admit. (* Assume we can prove this equality somehow. *)
  rewrite Eq. apply H.

  (* The new proof: *)
  applys_eq H 1.
    admit. (* Assume we can prove this equality somehow. *)
Abort.
```

If the mismatch was on the first argument of `P` instead of the second, we would have written `applys_eq H 2`. Recall that the occurences are counted from the right.

```
Lemma demo_applys_eq_2 : ∀ (P:nat→nat→Prop) x y z,
  P (big_expression_using z) x →
  P (big_expression_using y) x.
Proof.
  introv H. applys_eq H 2.
Abort.
```

When we have a mismatch on two arguments, we want to produce two equalities. To achieve this, we may call `applys_eq H 1 2`. More generally, the tactic `applys_eq` expects a lemma and a sequence of natural numbers as arguments.

```
Lemma demo_applys_eq_3 : ∀ (P:nat→nat→Prop) x1 x2 y1 y2,
  P (big_expression_using x2) (big_expression_using y2) →
  P (big_expression_using x1) (big_expression_using y1).
Proof.
  introv H. applys_eq H 1 2.
  (* produces two subgoals:
     big_expression_using x1 = big_expression_using x2
     big_expression_using y1 = big_expression_using y2 *)
Abort.

End EqualityExamples.
```

Some Convenient Shorthands

This section of the tutorial introduces a few tactics that help make proof scripts shorter and more readable:

- `unfolds` (without argument) for unfolding the head definition,
- `false` for replacing the goal with `False`,
- `gen` as a shorthand for `dependent generalize`,
- `skip` for skipping a subgoal even if it contains existential variables,
- `sort` for re-ordering the proof context by moving moving all propositions at the bottom.

The Tactic `unfolds`

```
Module UnfoldsExample.
  Import Hoare.
```

The tactic `unfolds` (without any argument) unfolds the head constant of the goal. This tactic saves the need to name the constant explicitly.

```
Lemma bexp_eval_true : ∀ b st,
  beval st b = true → (bassn b) st.
Proof.
  intros b st Hbe. dup.

  (* The old proof: *)
  unfold bassn. assumption.

  (* The new proof: *)
  unfolds. assumption.
Qed.
```

Remark: contrary to the tactic `hnf`, which may unfold several constants, `unfolds` performs only a single step of unfolding.

Remark: the tactic `unfolds in H` can be used to unfold the head definition of the hypothesis `H`.

```
End UnfoldsExample.
```

The Tactics `false` and `tryfalse`

The tactic `false` can be used to replace any goal with `False`. In short, it is a shorthand for `exfalso`. Moreover, `false` proves the goal if it contains an absurd assumption, such as `False` or `0 = S n`, or if it contains contradictory assumptions, such as `x = true` and `x = false`.

```
Lemma demo_false :
  ∀ n, S n = 1 → n = 0.
Proof.
  intros. destruct n. reflexivity. false.
Qed.
```

The tactic `false` can be given an argument: `false H` replace the goals with `False` and then applies `H`.

```
Lemma demo_false_arg :
  (∀ n, n < 0 → False) → (3 < 0) → 4 < 0.
```

```
Proof.
  intros H L. false H. apply L.
Qed.
```

The tactic `tryfalse` is a shorthand for `try solve [false]`: it tries to find a contradiction in the goal. The tactic `tryfalse` is generally called after a case analysis.

```
Lemma demo_tryfalse :
  ∀ n, S n = 1 → n = 0.
Proof.
  intros. destruct n; tryfalse. reflexivity.
Qed.
```

The Tactic `gen`

The tactic `gen` is a shortand for `generalize dependent` that accepts several arguments at once. An invokation of this tactic takes the form `gen x y z`.

```
Module GenExample.
  Import Stlc.
  Import STLC.

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.
  dup.

  (* The old proof: *)
  intros Gamma x U v t S Htypt Htypv.
  generalize dependent S. generalize dependent Gamma.
  induction t; intros; simpl.
  admit. admit. admit. admit. admit. admit.

  (* The new proof: *)
  introv Htypt Htypv. gen S Gamma.
  induction t; intros; simpl.
  admit. admit. admit. admit. admit. admit.
Abort.

End GenExample.
```

The Tactics `skip`, `skip_rewrite` and `skip_goal`

Temporarily admitting a given subgoal is very useful when constructing proofs. It gives the ability to focus first on the most interesting cases of a proof. The tactic `skip` is like `admit` except that it also works when the proof includes existential variables. Recall that existential variables are those whose name starts with a question mark, (e.g., `?24`), and which are typically introduced by `eapply`.

```
Module SkipExample.
  Import Stlc.
  Import STLC.
```

```
Notation " t '/' st '==>a*' t' " := (multi (astep st) t t')
                                    (at level 40, st at level
39).

Example astep_example1 :
  (3 + (3 * 4)) / { --> 0 } ==>a* 15.
Proof.
  eapply multi_step. skip.
(* the tactic admit would not work here *)
  eapply multi_step. skip. skip.
  (* Note that because some unification variables have
     not been instantiated, we still need to write
     Abort instead of Qed at the end of the proof. *)
Abort.
```

The tactic `skip H: P` adds the hypothesis `H: P` to the context, without checking whether the proposition `P` is true. It is useful for exploiting a fact and postponing its proof. Note: `skip H: P` is simply a shorthand for `assert (H:P). skip.`

```
Theorem demo_skipH : True.
Proof.
  skip H: (∀ n m : nat, (0 + n) * m = n * m).
Abort.
```

The tactic `skip_rewrite` ($E_1 = E_2$) replaces E_1 with E_2 in the goal, without checking that E_1 is actually equal to E_2.

```
Theorem mult_0_plus : ∀ n m : nat,
  (0 + n) * m = n * m.
Proof.
  dup.

  (* The old proof: *)
  intros n m.
  assert (H: 0 + n = n). skip. rewrite → H.
  reflexivity.

  (* The new proof: *)
  intros n m.
  skip_rewrite (0 + n = n).
  reflexivity.
Qed.
```

Remark: the tactic `skip_rewrite` can in fact be given a lemma statement as argument, in the same way as `asserts_rewrite`.

The tactic `skip_goal` adds the current goal as hypothesis. This cheat is useful to set up the structure of a proof by induction without having to worry about the induction hypothesis being applied only to smaller arguments. Using `skip_goal`, one can construct a proof in two steps: first, check that the main arguments go through without waisting time on fixing the details of the induction hypotheses; then, focus on fixing the invokations of the induction hypothesis.

```
Theorem ceval_deterministic: ∀ c st st1 st2,
  c / st \\ st1 →
  c / st \\ st2 →
  st1 = st2.
Proof.
  (* The tactic skip_goal creates an hypothesis called IH
     asserting that the statment of ceval_deterministic is true. *)
  skip_goal.
  (* Of course, if we call assumption here, then the goal is solved
     right away, but the point is to do the proof and use IH
     only at the places where we need an induction hypothesis. *)
  introv E1 E2. gen st2.
  (induction E1); introv E2; inverts E2 as.
  - (* E_Skip *) reflexivity.
  - (* E_Ass *)
    subst n.
    reflexivity.
  - (* E_Seq *)
    intros st3 Red1 Red2.
    assert (st' = st3) as EQ1.
    { (* Proof of assertion *)
      (* was: apply IHE1_1; assumption. *)
      (* new: *) eapply IH. eapply E1_1. eapply Red1. }
    subst st3.
    (* was: apply IHE1_2. assumption.] *)
    (* new: *) apply IH. eapply E1_2. eapply Red2.
  (* The other cases are similiar. *)
Abort.

End SkipExample.
```

The Tactic `sort`

```
Module SortExamples.
  Import Imp.
```

The tactic `sort` reorganizes the proof context by placing all the variables at the top and all the hypotheses at the bottom, thereby making the proof context more readable.

```
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; inverts E2.
  admit. admit. (* Skipping some trivial cases *)
  sort. (* Observe how the context is reorganized *)
Abort.

End SortExamples.
```

Tactics for Advanced Lemma Instantiation

This last section describes a mechanism for instantiating a lemma by providing some of its arguments and leaving other implicit. Variables whose instantiation is not provided are turned into existentential variables, and facts whose instantiation is not provided are turned into subgoals.

Remark: this instantion mechanism goes far beyond the abilities of the "Implicit Arguments" mechanism. The point of the instantiation mechanism described in this section is that you will no longer need to spend time figuring out how many underscore symbols you need to write.

In this section, we'll use a useful feature of Coq for decomposing conjunctions and existentials. In short, a tactic like `intros` or `destruct` can be provided with a pattern `(H1 & H2 & H3 & H4 & H5)`, which is a shorthand for `[H1 [H2 [H3 [H4 H5]]]]`. For example, `destruct (H _ _ _ Htypt) as [T [Hctx Hsub]].` can be rewritten in the form `destruct (H _ _ _ Htypt) as (T & Hctx & Hsub).`

Working of `lets`

When we have a lemma (or an assumption) that we want to exploit, we often need to explicitly provide arguments to this lemma, writing something like: `destruct (typing_inversion_var _ _ _ Htypt) as (T & Hctx & Hsub).` The need to write several times the "underscore" symbol is tedious. Not only we need to figure out how many of them to write down, but it also makes the proof scripts look prettly ugly. With the tactic `lets`, one can simply write: `lets (T & Hctx & Hsub): typing_inversion_var Htypt.`

In short, this tactic `lets` allows to specialize a lemma on a bunch of variables and hypotheses. The syntax is `lets I: E0 E1 .. EN`, for building an hypothesis named `I` by applying the fact `E0` to the arguments `E1` to `EN`. Not all the arguments need to be provided, however the arguments that are provided need to be provided in the correct order. The tactic relies on a first-match algorithm based on types in order to figure out how the to instantiate the lemma with the arguments provided.

```
Module ExamplesLets.
  Import Sub.

(* To illustrate the working of lets, assume that we want to
   exploit the following lemma. *)

Axiom typing_inversion_var : ∀ (G:context) (x:string) (T:ty),
  has_type G (tvar x) T →
  ∃ S, G x = Some S ∧ subtype S T.
```

First, assume we have an assumption `H` with the type of the form `has_type G (tvar x) T`. We can obtain the conclusion of the lemma `typing_inversion_var` by invoking the tactics `lets K: typing_inversion_var H`, as shown next.

```
Lemma demo_lets_1 : ∀ (G:context) (x:string) (T:ty),
  has_type G (tvar x) T → True.
Proof.
  intros G x T H. dup.

  (* step-by-step: *)
  lets K: typing_inversion_var H.
  destruct K as (S & Eq & Sub).
  admit.

  (* all-at-once: *)
  lets (S & Eq & Sub): typing_inversion_var H.
  admit.
Abort.
```

Assume now that we know the values of `G`, `x` and `T` and we want to obtain `S`, and have `has_type G (tvar x) T` be produced as a subgoal. To indicate that we want all the remaining arguments of `typing_inversion_var` to be produced as subgoals, we use a triple-underscore symbol `___`. (We'll later introduce a shorthand tactic called `forwards` to avoid writing triple underscores.)

```
Lemma demo_lets_2 : ∀ (G:context) (x:string) (T:ty), True.
Proof.
  intros G x T.
  lets (S & Eq & Sub): typing_inversion_var G x T ___.
Abort.
```

Usually, there is only one context `G` and one type `T` that are going to be suitable for proving `has_type G (tvar x) T`, so we don't really need to bother giving `G` and `T` explicitly. It suffices to call `lets (S & Eq & Sub): typing_inversion_var x`. The variables `G` and `T` are then instantiated using existential variables.

```
Lemma demo_lets_3 : ∀ (x:string), True.
Proof.
  intros x.
  lets (S & Eq & Sub): typing_inversion_var x ___.
Abort.
```

We may go even further by not giving any argument to instantiate `typing_inversion_var`. In this case, three unification variables are introduced.

```
Lemma demo_lets_4 : True.
Proof.
  lets (S & Eq & Sub): typing_inversion_var ___.
Abort.
```

Note: if we provide `lets` with only the name of the lemma as argument, it simply adds this lemma in the proof context, without trying to instantiate any of its arguments.

```
Lemma demo_lets_5 : True.
Proof.
  lets H: typing_inversion_var.
Abort.
```

A last useful feature of `lets` is the double-underscore symbol, which allows skipping an argument when several arguments have the same type. In the following example, our assumption quantifies over two variables `n` and `m`, both of type `nat`. We would like `m` to be instantiated as the value 3, but without specifying a value for `n`. This can be achieved by writting `lets K: H __ 3`.

```
Lemma demo_lets_underscore :
  (∀ n m, n ≤ m → n < m+1) → True.
Proof.
  intros H.

  (* If we do not use a double underscore, the first argument,
     which is n, gets instantiated as 3. *)
  lets K: H 3. (* gives K of type ∀m, 3 ≤ m → 3 < m+1 *)
    clear K.

  (* The double underscore preceeding 3 indicates that we want
     to skip a value that has the type nat (because 3 has
     the type nat). So, the variable m gets instiated as 3. *)
  lets K: H __ 3. (* gives K of type ?X ≤ 3 → ?X < 3+1 *)
    clear K.
Abort.
```

Note: one can write `lets:` E_0 E_1 E_2 in place of `lets H:` E_0 E_1 E_2. In this case, the name `H` is chosen arbitrarily.

Note: the tactics `lets` accepts up to five arguments. Another syntax is available for providing more than five arguments. It consists in using a list introduced with the special symbol >>, for example `lets H: (>>` E_0 E_1 E_2 E_3 E_4 E_5 E_6 E_7 E_8 E_9 `10)`.

```
End ExamplesLets.
```

Working of `applys`, `forwards` and `specializes`

The tactics `applys`, `forwards` and `specializes` are shorthand that may be used in place of `lets` to perform specific tasks.

- `forwards` is a shorthand for instantiating all the arguments

of a lemma. More precisely, `forwards H:` E_0 E_1 E_2 E_3 is the same as `lets H:` E_0 E_1 E_2 E_3 `___`, where the triple-underscore has the same meaning as explained earlier on.

- `applys` allows building a lemma using the advanced instantion

mode of `lets`, and then apply that lemma right away. So, `applys` E_0 E_1 E_2 E_3 is the same as `lets H:` E_0 E_1 E_2 E_3 followed with `eapply H` and then `clear H`.

- `specializes` is a shorthand for instantiating in-place an assumption from the context with particular arguments. More precisely, `specializes H E0 E1` is the same as `lets H' : H E0 E1` followed with `clear H` and `rename H' into H`.

Examples of use of `applys` appear further on. Several examples of use of `forwards` can be found in the tutorial chapter UseAuto.

Example of Instantiations

```
Module ExamplesInstantiations.
  Import Sub.
```

The following proof shows several examples where `lets` is used instead of `destruct`, as well as examples where `applys` is used instead of `apply`. The proof also contains some holes that you need to fill in as an exercise.

```
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 with eauto.
  intros Gamma x U v t S Htypt Htypv.
  generalize dependent S. generalize dependent Gamma.
  (induction t); intros; simpl.
  - (* tvar *)
    rename s into y.

    (* An example where destruct is replaced with lets. *)
    (* old: destruct (typing_inversion_var _ _ _ Htypt) as T
[Hctx Hsub].*)
    (* new: *) lets (T&Hctx&Hsub): typing_inversion_var Htypt.
    unfold update, t_update in Hctx.
    destruct (beq_stringP x y)...
    + (* x=y *)
      subst.
      inversion Hctx; subst. clear Hctx.
      apply context_invariance with empty...
      intros x Hcontra.

       (* A more involved example. *)
       (* old: destruct (free_in_context _ _ S empty Hcontra)
                 as T' HT'... *)
       (* new: *)
        lets [T' HT']: free_in_context S (@empty ty) Hcontra...
        inversion HT'.
  - (* tapp *)

    (* Exercise: replace the following destruct with a lets. *)
    (* old: destruct (typing_inversion_app _ _ _ _ Htypt)
              as T1 [Htypt1 Htypt2]. eapply T_App... *)
    (* FILL IN HERE *) admit.

  - (* tabs *)
    rename s into y. rename t into T1.
```

```
    (* Here is another example of using lets. *)
    (* old: destruct (typing_inversion_abs _ _ _ _ _ Htypt). *)
    (* new: *) lets (T2&Hsub&Htypt2): typing_inversion_abs Htypt.

    (* An example of where apply
with can be replaced with applys. *)
    (* old: apply T_Sub with (TArrow T1 T2)... *)
    (* new: *) applys T_Sub (TArrow T1 T2)...
     apply T_Abs...
    destruct (beq_stringP x y).
    + (* x=y *)
      eapply context_invariance...
      subst.
      intros x Hafi. unfold update, t_update.
      destruct (beq_stringP y x)...
    + (* x<>y *)
      apply IHt. eapply context_invariance...
      intros z Hafi. unfold update, t_update.
      destruct (beq_stringP y z)...
      subst. rewrite false_beq_string...
  - (* ttrue *)
    lets: typing_inversion_true Htypt...
  - (* tfalse *)
    lets: typing_inversion_false Htypt...
  - (* tif *)
    lets (Htyp1&Htyp2&Htyp3): typing_inversion_if Htypt...
  - (* tunit *)
    (* An example where assert can be replaced with lets. *)
    (* old: assert (subtype TUnit S)
             by apply (typing_inversion_unit _ _ Htypt)... *)
    (* new: *) lets: typing_inversion_unit Htypt...

Admitted.

End ExamplesInstantiations.
```

Summary

In this chapter we have presented a number of tactics that help make proof script more concise and more robust on change.

- `introv` and `inverts` improve naming and inversions.
- `false` and `tryfalse` help discarding absurd goals.
- `unfolds` automatically calls `unfold` on the head definition.
- `gen` helps setting up goals for induction.
- `cases` and `cases_if` help with case analysis.
- `splits`, `branch` and ∃to deal with n-ary constructs.
- `asserts_rewrite`, `cuts_rewrite`, `substs` and `fequals` help working with equalities.

- `lets`, `forwards`, `specializes` and `applys` provide means of very conveniently instantiating lemmas.
- `applys_eq` can save the need to perform manual rewriting steps before being able to apply lemma.
- `skip`, `skip_rewrite` and `skip_goal` give the flexibility to choose which subgoals to try and discharge first.

Making use of these tactics can boost one's productivity in Coq proofs.

If you are interested in using `LibTactics.v` in your own developments, make sure you get the lastest version from: http://www.chargueraud.org/softs/tlc/.