SOFTWARE FOUNDATIONS

VOLUME 3: VERIFIED FUNCTIONAL ALGORITHMS

TABLE OF CONTENTS INDEX ROADMAP

PERM

BASIC TECHNIQUES FOR PERMUTATIONS AND ORDERING

Consider these algorithms and data structures:

- sort a sequence of numbers;
- finite maps from numbers to (arbitrary-type) data
- finite maps from any ordered type to (arbitrary-type) data
- priority queues: finding/deleting the highest number in a set

To prove the correctness of such programs, we need to reason about less-than comparisons (for example, on integers) and about "these two sets/sequences have the same contents". In this chapter, we introduce some techniques for reasoning about:

- less-than comparisons on natural numbers
- permutations (rearrangements of lists)

Then, in later chapters, we'll apply these proof techniques to reasoning about algorithms and data structures.

```
Require Export Coq.Bool.Bool.
Require Export Coq.Arith.Arith.
Require Export Coq.Arith.EqNat.
Require Export Coq.omega.Omega.
Require Export Coq.Lists.List.
Export ListNotations.
Require Export Permutation.
```

The Less-Than Order on the Natural Numbers

These `Check` and `Locate` commands remind us about *Propositional* and the *Boolean* less-than operators in the Coq standard library.

```
Check Nat.lt. (* : nat -> nat -> Prop *)
Check lt. (* : nat -> nat -> Prop *)
Goal Nat.lt = lt. Proof. reflexivity. Qed.
(* They are the same *)
Check Nat.ltb. (* : nat -> nat -> bool *)
Locate "_ < _". (* "x < y" := lt x y *)
Locate "<?". (* x <? y  := Nat.ltb x y *)
```

We write `x < y` for the Proposition that `x` *is* less than `y`, and we write `x <? y` for the computable *test* that returns `true` or `false` depending on whether `x<y`. The theorem that `lt` is related in this way to `ltb` is this one:

```
Check Nat.ltb_lt.
(* : forall n m : nat, (n <? m) = true <-> n < m *)
```

For some reason, the Coq library has `<?` and `<=?` notations, but is missing these three:

```
Notation "a >=? b" := (Nat.leb b a)
                        (at level 70, only parsing) :
nat_scope.
Notation "a >? b" := (Nat.ltb b a)
                       (at level 70, only parsing) : nat_scope.
Notation " a =? b" := (beq_nat a b)
                       (at level 70) : nat_scope.
```

Relating `Prop` to `bool`

The `reflect` relation connects a `Proposition` to a `Boolean`.

```
Print reflect.
```

That is, `reflect P b` means that `P`↔`True` if and only if `b=true`. The way to use `reflect` is, for each of your operators, make a lemma like these next three:

```
Lemma beq_reflect : ∀ x y, reflect (x = y) (x =? y).
Proof.
  intros x y.
  apply iff_reflect. symmetry. apply beq_nat_true_iff.
Qed.

Lemma blt_reflect : ∀ x y, reflect (x < y) (x <? y).
Proof.
  intros x y.
  apply iff_reflect. symmetry. apply Nat.ltb_lt.
Qed.

Lemma ble_reflect : ∀ x y, reflect (x ≤ y) (x <=? y).
Proof.
  intros x y.
  apply iff_reflect. symmetry. apply Nat.leb_le.
Qed.
```

Here's an example of how you could use these lemmas. Suppose you have this simple program, `(if a <? 5 then a else 2)`, and you want to prove that it evaluates to a number smaller than 6. You can use `blt_reflect` "by hand":

```
Example reflect_example1: ∀ a, (if a<?5 then a else 2) < 6.
Proof.
  intros.
  destruct (blt_reflect a 5) as [H|H].
  * (* Notice that H above the line has a Propositional
       fact _related_ to a<?5*)
    omega.
(* More explanation of omega later in this chapter. *)
  * (* Notice that H above the line has a a _different_
       Propositional fact. *)
    apply not_lt in H. (* This step is not necessary,
         it just makes the hypothesis H look pretty *)
    omega.
Qed.
```

But there's another way to use `blt_reflect`, etc: read on.

Some Advanced Tactical Hacking

You may skip ahead to "Inversion/clear/subst". Right here, we build some machinery that you'll want to *use*, but you won't need to know how to *build* it.

Let's put several of these `reflect` lemmas into a Hint database, called `bdestruct` because we'll use it in our boolean-destruction tactic:

```
Hint Resolve blt_reflect ble_reflect beq_reflect : bdestruct.
```

Our high-tech *boolean destruction* tactic:

```
Ltac bdestruct X :=
 +
```

Here's a brief example of how to use `bdestruct`. There are more examples later.

```
Example reflect_example2: ∀ a, (if a<?5 then a else 2) < 6.
Proof.
  intros.
  bdestruct (a<?5). (* instead of: destruct (blt_reflect a 5) as
[H|H]. *)
  * (* Notice that H above the line has a Propositional
       fact _related_ to a<?5*)
    omega.
(* More explanation of omega later in this chapter. *)
  * (* Notice that H above the line has a a _different_
       Propositional fact. We don't need to apply not_lt,
       as bdestruct has already done it. *)
    omega.
Qed.
```

`inversion` / `clear` / `subst`

Coq's `inversion H` tactic is so good at extracting information from the hypothesis `H` that `H` becomes completely redundant, and one might as well `clear` it from the goal. Then, since the `inversion` typically creates some equality facts, why not then `subst` ? This motivates the following useful tactic, `inv`:

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

Linear Integer Inequalities

In our proofs about searching and sorting algorithms, we sometimes have to reason about the consequences of less-than and greater-than. Here's a contrived example.

```
Module Exploration1.

Theorem omega_example1:
 ∀ i j k,
    i < j →
    ¬ (k - 3 ≤ j) →
   k > i.
Proof.
  intros.
```

Now, there's a hard way to prove this, and an easy way. Here's the hard way.

```
  (* try to remember the name of the lemma about negation and ≤ *)
  Search (¬ _ ≤ _ → _).
  apply not_le in H0.
  (* try to remember the name of the transitivity lemma about > *)
  Search (_ > _ → _ > _ → _ > _).
  apply gt_trans with j.
  apply gt_trans with (k-3).
  (* _OBVIOUSLY_, k is greater than k-3.  But _OOPS_,
     this is not actually true, because we are talking about
     natural numbers with "bogus subtraction." *)
Abort.

Theorem bogus_subtraction: ¬ (∀ k:nat, k > k - 3).
Proof.
  (* intro introduces exactly one thing, like intros ? *)
  intro.
  (* specialize applies a hypothesis to an argument *)
  specialize (H O).
  simpl in H. inversion H.
Qed.
```

With bogus subtraction, this omega_example1 theorem even True? Yes it is; let's try again, the hard way, to find the proof.

```
Theorem omega_example1:
 ∀ i j k,
    i < j →
    ¬ (k - 3 ≤ j) →
   k > i.
Proof. (* try again! *)
  intros.
  apply not_le in H0.
  unfold gt in H0.
  unfold gt.
  (* try to remember the name ... *)
  Search (_ < _ → _ ≤ _ → _ < _).
  apply lt_le_trans with j.
  apply H.
  apply le_trans with (k-3).
```

```
  Search (_ < _ → _ ≤ _).
  apply lt_le_weak.
  auto.
  apply le_minus.
Qed. (* Oof!  That was exhausting and tedious.  *)
```

And here's the easy way.

```
Theorem omega_example2:
 ∀ i j k,
    i < j →
    ¬ (k - 3 ≤ j) →
   k > i.
Proof.
  intros.
  omega.
Qed.
```

Here we have used the `omega` tactic, made available by importing `Coq.omega.Omega` as we have done above. Omega is an algorithm for integer linear programming, invented in 1991 by William Pugh. Because ILP is NP-complete, we might expect that this algorithm is exponential-time in the worst case, and indeed that's true: if you have `N` equations, it could take `2^N` time. But in the typical cases that result from reasoning about programs, omega is much faster than that. Coq's `omega` tactic is an implementation of this algorithm that generates a machine-checkable Coq proof. It "understands" the types Z and nat, and these operators: < = > ≤ ≥ + - ¬, as well as multiplication by small integer literals (such as 0,1,2,3...) and some uses of ∨ and ∧.

Omega does *not* understand other operators. It treats things like `a*b` and `f x y` as if they were variables. That is, it can prove `f x y > a*b → f x y + 3 ≥ a*b`, in the same way it would prove `u > v → u+3 ≥ v`.

Now let's consider a silly little program: swap the first two elements of a list, if they are out of order.

```
Definition maybe_swap (al: list nat) : list nat :=
  match al with
  | a :: b :: ar ⇒ if a >? b then b::a::ar else a::b::ar
  | _ ⇒ al
  end.

Example maybe_swap_123:
  maybe_swap [1; 2; 3] = [1; 2; 3].
Proof. reflexivity. Qed.

Example maybe_swap_321:
  maybe_swap [3; 2; 1] = [2; 3; 1].
Proof. reflexivity. Qed.
```

In this program, we wrote `a>?b` instead of `a>b`. Why is that?

```
Check (1>2). (* : Prop *)
Check (1>?2). (* : bool *)
```

We cannot compute with elements of `Prop`: we need some kind of constructible (and pattern-matchable) value. For that we use `bool`.

```
Locate ">?". (* a >? b :=  ltb b a *)
```

The name `ltb` stands for "less-than boolean."

```
Print Nat.ltb.
(* =  fun n m : nat => S n <=? m : nat -> nat -> bool  *)
Locate ">=?".
```

Instead of defining an operator `Nat.geb`, the standard library just defines the notation for greater-or-equal-boolean as a less-or-equal-boolean with the arguments swapped.

```
Locate leb.
Print leb.
Print Nat.leb. (* The computation to compare natural numbers. *)
```

Here's a theorem: `maybe_swap` is idempotent — that is, applying it twice gives the same result as applying it once.

```
Theorem maybe_swap_idempotent:
  ∀ al, maybe_swap (maybe_swap al) = maybe_swap al.
Proof.
  intros.
  destruct al as [ | a al].
  simpl.
  reflexivity.
  destruct al as [ | b al].
  simpl.
  reflexivity.
  simpl.
```

What do we do here? We must proceed by case analysis on whether a>b.

```
  destruct (b <? a) eqn:H.
  simpl.
  destruct (a <? b) eqn:H0.
```

Now what? Look at the hypotheses `H: b<a` and `H0: a<b` above the line. They can't both be true. In fact, `omega` "knows" how to prove that kind of thing. Let's try it:

```
  try omega.
```

`omega` didn't work, because it operates on comparisons in `Prop`, such as `a>b`; not upon comparisons yielding bool, such as `a>?b`. We need to convert these comparisons to `Prop`, so that we can use `omega`.

Actually, we don't "need" to. Instead, we could reason directly about these operations in `bool`. But that would be even more tedious than the `omega_example1` proof. Therefore: let's set up some machinery so that we can use `omega` on boolean tests.

```
Abort.
```

Let's try again, a new way:

```
Theorem maybe_swap_idempotent:
  ∀ al, maybe_swap (maybe_swap al) = maybe_swap al.
Proof.
  intros.
  destruct al as [ | a al].
  simpl.
  reflexivity.
  destruct al as [ | b al].
  simpl.
  reflexivity.
  simpl.
```

This is where we left off before. Now, watch:

```
    destruct (blt_reflect b a). (* THIS LINE *)
    (* Notice that b<a is above the line as a Prop, not a bool.
       Now, comment out THIS LINE, and uncomment THAT LINE.  *)
    (* bdestruct (b <? a).    (* THAT LINE *) *)
    (* THAT LINE, with bdestruct, does the same thing as THIS LINE. *)
  * (* case b<a *)
    simpl.
    bdestruct (a <? b).
    omega.
```

The `omega` tactic noticed that above the line we have an arithmetic contradiction. Perhaps it seems wasteful to bring out the "big gun" to shoot this flea, but really, it's easier than remembering the names of all those lemmas about arithmetic!

```
    reflexivity.
  * (* case a >= b *)
    simpl.
    bdestruct (b <? a).
    omega.
    reflexivity.
Qed.
```

Moral of this story: When proving things about a program that uses boolean comparisons `(a <? b)`, use `bdestruct`. Then use `omega`. Let's review that proof without all the comments.

```
Theorem maybe_swap_idempotent':
  ∀ al, maybe_swap (maybe_swap al) = maybe_swap al.
Proof.
  intros.
  destruct al as [ | a al].
  simpl.
  reflexivity.
  destruct al as [ | b al].
  simpl.
  reflexivity.
  simpl.
  bdestruct (b <? a).
  *
```

```
  simpl.
  bdestruct (a <? b).
  omega.
  reflexivity.
*
  simpl.
  bdestruct (b <? a).
  omega.
  reflexivity.
Qed.
```

Permutations

Another useful fact about `maybe_swap` is that it doesn't add or remove elements from the list: it only reorders them. We can say that the output list is a *permutation* of the input. The Coq `Permutation` library has an inductive definition of permutations, along with some lemmas about them.

```
Locate Permutation.
(* Inductive Coq.Sorting.Permutation.Permutation *)
Check Permutation. (*  : forall {A : Type}, list A -> list A -
> Prop *)
```

We say "list `al` is a permutation of list `bl`", written `Permutation al bl`, if the elements of `al` can be reordered (without insertions or deletions) to get the list `bl`.

```
Print Permutation.
(*
 Inductive Permutation {A : Type} : list A -> list A -> Prop :=
    perm_nil : Permutation
  | perm_skip : forall (x : A) (l l' : list A),
                Permutation l l' ->
                Permutation (x :: l) (x :: l')
  | perm_swap : forall (x y : A) (l : list A),
                Permutation (y :: x :: l) (x :: y :: l)
  | perm_trans : forall l l' l'' : list A,
                 Permutation l l' ->
                 Permutation l' l'' ->
                 Permutation l l''.
*)
```

You might wonder, "is that really the right definition?" And indeed, it's important that we get a right definition, because `Permutation` is going to be used in the specification of correctness of our searching and sorting algorithms. If we have the wrong specification, then all our proofs of "correctness" will be useless.

It's not obvious that this is indeed the right specification of permutations. (It happens to be true, but it's not obvious!) In order to gain confidence that we have the right specification, we should use this specification to prove some properties that we think permutations ought to have.

Exercise: 2 stars (Permutation_properties)

Think of some properties of the `Permutation` relation and write them down informally in English, or a mix of Coq and English. Here are four to get you started:

- 1. If `Permutation al bl`, then `length al = length bl`.
- 2. If `Permutation al bl`, then `Permutation bl al`.
- 3. `[1;1]` is NOT a permutation of `[1;2]`.
- 4. `[1;2;3;4]` IS a permutation of `[3;4;2;1]`.

YOUR ASSIGNMENT: Add three more properties. Write them here:

Now, let's examine all the theorems in the Coq library about permutations:

```
Search Permutation.
(* Browse through the results of this query! *)
```

Which of the properties that you wrote down above have already been proved as theorems by the Coq library developers? Answer here:

□

Let's use the permutation rules in the library to prove the following theorem.

```
Example butterfly: ∀ b u t e r f l y : nat,
  Permutation ([b;u;t;t;e;r]++[f;l;y]) ([f;l;u;t;t;e;r]++[b;y]).
Proof.
 intros.
 (* Just to illustrate a method, let's group u;t;t;e;r together: *)
 change [b;u;t;t;e;r] with ([b]++[u;t;t;e;r]).
 change [f;l;u;t;t;e;r] with ([f;l]++[u;t;t;e;r]).
 remember [u;t;t;e;r] as utter.
 clear Hequtter.
 (* Next, let's cancel utter from both sides.  In order to do that,
    we need to bring utter to the beginning of each list. *)
Check app_assoc.
  rewrite <- app_assoc.
  rewrite <- app_assoc.
Check perm_trans.
  apply perm_trans with (utter ++ [f;l;y] ++ [b]).
  rewrite (app_assoc utter [f;l;y]).
Check Permutation_app_comm.
  apply Permutation_app_comm.
 eapply perm_trans.
 2: apply Permutation_app_comm.
  rewrite <- app_assoc.
Search (Permutation (_++_) (_++_)).
 apply Permutation_app_head.
 (* Now that utter is utterly removed from the goal, let's cancel f;l. *)
 eapply perm_trans.
 2: apply Permutation_app_comm.
 simpl.
Check perm_skip.
 apply perm_skip.
 apply perm_skip.
Search (Permutation (_::_) (_::_)).
 apply perm_swap.
Qed.
```

That example illustrates a general method for proving permutations involving cons `::` and append `++`. You identify some portion appearing in both sides; you bring that portion to the front on each side using lemmas such as `Permutation_app_comm` and `perm_swap`, with generous use of `perm_trans`. Then, you use `perm_skip` to cancel a single element, or `Permutation_app_head` to cancel an append-chunk.

Exercise: 3 stars (permut_example)

Use the permutation rules in the library (see the `Search`, above) to prove the following theorem. These `Check` commands are a hint about what lemmas you'll need.

```
Check perm_skip.
Check Permutation_refl.
Check Permutation_app_comm.
Check app_assoc.

Example permut_example: ∀ (a b: list nat),
  Permutation (5::6::a++b) ((5::b)++(6::a++[])).
Proof.
 (* After you cancel the 5, then bring the 6 to the front... *)
(* FILL IN HERE *) Admitted.
```

□

Exercise: 1 star (not_a_permutation)

Prove that `[1;1]` is not a permutation of `[1;2]`. Hints are given as `Check` commands.

```
Check Permutation_cons_inv.
Check Permutation_length_1_inv.

Example not_a_permutation:
  ¬ Permutation [1;1] [1;2].
Proof.
(* FILL IN HERE *) Admitted.
```

□

Back to `maybe_swap`. We prove that it doesn't lose or gain any elements, only reorders them.

```
Theorem maybe_swap_perm: ∀ al,
  Permutation al (maybe_swap al).
Proof.
  (* WORKED IN CLASS *)
  intros.
  destruct al as [ | a al].
  simpl. apply Permutation_refl.
  destruct al as [ | b al].
  simpl. apply Permutation_refl.
  simpl.
  bdestruct (a>?b).
  apply perm_swap.
  apply Permutation_refl.
Qed.
```

Now let us specify functional correctness of `maybe_swap`: it rearranges the elements in such a way that the first is less-or-equal than the second.

```
Definition first_le_second (al: list nat) : Prop :=
  match al with
  | a::b::_ ⇒ a ≤ b
  | _ ⇒ True
  end.

Theorem maybe_swap_correct: ∀ al,
    Permutation al (maybe_swap al)
    ∧ first_le_second (maybe_swap al).
Proof.
  intros.
  split.
  apply maybe_swap_perm.
  (* WORKED IN CLASS *)
  destruct al as [ | a al].
  simpl. auto.
  destruct al as [ | b al].
  simpl. auto.
  simpl.
  bdestruct (b <? a).
  simpl.
  omega.
  simpl.
  omega.
Qed.

End Exploration1.
```

Summary: Comparisons and Permutations

To prove correctness of algorithms for sorting and searching, we'll reason about comparisons and permutations using the tools developed in this chapter. The `maybe_swap` program is a tiny little example of a sorting program. The proof style in `maybe_swap_correct` will be applied (at a larger scale) in the next few chapters.

Exercise: 2 stars (Forall_perm)

To close, a useful utility lemma. Prove this by induction; but is it induction on `al`, or on `bl`, or on `Permutation al bl`, or on `Forall f al`?

```
Theorem Forall_perm: ∀ {A} (f: A → Prop) al bl,
  Permutation al bl →
  Forall f al → Forall f bl.
Proof.
(* FILL IN HERE *) Admitted.
```

□