SOFTWARE FOUNDATIONS

VOLUME 3: VERIFIED FUNCTIONAL ALGORITHMS

TABLE OF CONTENTS INDEX ROADMAP

MULTISET

INSERTION SORT WITH MULTISETS

We have seen how to specify algorithms on "collections", such as sorting algorithms, using permutations. Instead of using permutations, another way to specify these algorithms is to use multisets. A *set* of values is like a list with no repeats where the order does not matter. A *multiset* is like a list, possibly with repeats, where the order does not matter. One simple representation of a multiset is a function from values to `nat`.

```
Require Import Perm.
Require Import Sort.
Require Export FunctionalExtensionality.
```

In this chapter we will be using natural numbers for two different purposes: the values in the lists that we sort, and the multiplicity (number of times occurring) of those values. To keep things straight, we'll use the `value` type for values, and `nat` for multiplicities.

```
Definition value := nat.

Definition multiset := value → nat.
```

Just like sets, multisets have operators for `union`, for the `empty` multiset, and the multiset with just a single element.

```
Definition empty : multiset :=
   fun x ⇒ 0.

Definition union (a b : multiset) : multiset :=
   fun x ⇒ a x + b x.

Definition singleton (v: value) : multiset :=
   fun x ⇒ if x =? v then 1 else 0.
```

Exercise: 1 star (union_assoc)

Since multisets are represented as functions, to prove that one multiset equals another we must use the axiom of functional extensionality.

```
Lemma union_assoc: ∀ a b c : multiset,
(* assoc stands for "associative" *)
   union a (union b c) = union (union a b) c.
Proof.
  intros.
  extensionality x.
(* FILL IN HERE *) Admitted.
```

□

Exercise: 1 star (union_comm)

```
Lemma union_comm: ∀ a b : multiset,
(* comm stands for "commutative" *)
   union a b = union b a.
Proof.
(* FILL IN HERE *) Admitted.
```

□

Remark on efficiency: These multisets aren't very efficient. If you wrote programs with them, the programs would run slowly. However, we're using them for *specifications*, not for *programs*. Our multisets built with `union` and `singleton` will never really *execute* on any large-scale inputs; they're only used in the proof of correctness of algorithms such as `sort`. Therefore, their inefficiency is not a problem.

Contents of a list, as a multiset:

```
Fixpoint contents (al: list value) : multiset :=
  match al with
  | a :: bl ⇒ union (singleton a) (contents bl)
  | nil ⇒ empty
  end.
```

Recall the insertion-sort program from `Sort.v`. Note that it handles lists with repeated elements just fine.

```
Example sort_pi: sort [3;1;4;1;5;9;2;6;5;3;5] =
[1;1;2;3;3;4;5;5;5;6;9].
+
```

```
Example sort_pi_same_contents:
    contents (sort [3;1;4;1;5;9;2;6;5;3;5]) = contents
[3;1;4;1;5;9;2;6;5;3;5].
Proof.
extensionality x.
do 10 (destruct x; try reflexivity).
  (* Why does this work? Try it step by step, without do 10 *)
Qed.
```

Correctness

A sorting algorithm must rearrange the elements into a list that is totally ordered. But let's say that a different way: the algorithm must produce a list *with the same multiset of values*, and this list must be totally ordered.

```
Definition is_a_sorting_algorithm' (f: list nat → list nat) :=
  ∀ al, contents al = contents (f al) ∧ sorted (f al).
```

Exercise: 3 stars (insert_contents)

First, prove the auxiliary lemma `insert_contents`, which will be useful for proving `sort_contents` below. Your proof will be by induction. You do not need to use `extensionality`.

```
Lemma insert_contents: ∀ x l, contents (x::l) = contents (insert
x l).
Proof.
(* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars (sort_contents)

Now prove that sort preserves contents.

```
Theorem sort_contents: ∀ l, contents l = contents (sort l).
(* FILL IN HERE *) Admitted.
```

□

Now we wrap it all up.

```
Theorem insertion_sort_correct:
  is_a_sorting_algorithm' sort.
Proof.
split. apply sort_contents. apply sort_sorted.
Qed.
```

Exercise: 1 star (permutations_vs_multiset)

Compare your proofs of `insert_perm`, `sort_perm` with your proofs of `insert_contents`, `sort_contents`. Which proofs are simpler?

- easier with permutations,
- easier with multisets
- about the same.

Regardless of "difficulty", which do you prefer / find easier to think about?

- permutations or
- multisets

Put an X in one box in each list. □

Permutations and Multisets

The two specifications of insertion sort are equivalent. One reason is that permutations and multisets are closely related. We're going to prove:

`Permutation al bl ↔ contents al = contents bl.`

Exercise: 3 stars (perm_contents)

The forward direction is easy, by induction on the evidence for `Permutation`:

```
Lemma perm_contents:
  ∀ al bl : list nat,
   Permutation al bl → contents al = contents bl.
(* FILL IN HERE *) Admitted.
```

□

The other direction, `contents al = contents bl → Permutation al bl`, is surprisingly difficult. (Or maybe there's an easy way that I didn't find.)

```
Fixpoint list_delete (al: list value) (v: value) :=
  match al with
  | x::bl ⇒ if x =? v then bl else x :: list_delete bl v
  | nil ⇒ nil
  end.

Definition multiset_delete (m: multiset) (v: value) :=
   fun x ⇒ if x =? v then pred(m x) else m x.
```

Exercise: 3 stars (delete_contents)

```
Lemma delete_contents:
  ∀ v al,
   contents (list_delete al v) = multiset_delete (contents al)
v.
Proof.
  intros.
  extensionality x.
  induction al.
  simpl. unfold empty, multiset_delete.
  bdestruct (x =? v); auto.
  simpl.
  bdestruct (a =? v).
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 2 stars (contents_perm_aux)

```
Lemma contents_perm_aux:
 ∀ v b, empty = union (singleton v) b → False.
Proof.
(* FILL IN HERE *) Admitted.
```

□

Exercise: 2 stars (contents_in)

```
Lemma contents_in:
  ∀ (a: value) (bl: list value) , contents bl a > 0 → In a bl.
Proof.
(* FILL IN HERE *) Admitted.
```

□

Exercise: 2 stars (in_perm_delete)

```
Lemma in_perm_delete:
  ∀ a bl,
  In a bl → Permutation (a :: list_delete bl a) bl.
Proof.
(* FILL IN HERE *) Admitted.
```
□

Exercise: 4 stars (contents_perm)

```
Lemma contents_perm:
 ∀ al bl, contents al = contents bl → Permutation al bl.
Proof.
  induction al; destruct bl; intro.
  auto.
  simpl in H.
  contradiction (contents_perm_aux _ _ H).
  simpl in H. symmetry in H.
  contradiction (contents_perm_aux _ _ H).
  specialize (IHal (list_delete (v :: bl) a)).
  remember (v::bl) as cl.
  clear v bl Heqcl.
```

From this point on, you don't need induction. Use the lemmas `perm_trans`, `delete_contents`, `in_perm_delete`, `contents_in`. At *certain points* you'll need to unfold the definitions of `multiset_delete`, `union`, `singleton`.

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

The Main Theorem: Equivalence of Multisets and Permutations

```
Theorem same_contents_iff_perm:
  ∀ al bl, contents al = contents bl ↔ Permutation al bl.
Proof.
  intros. split. apply contents_perm. apply perm_contents.
Qed.
```

Therefore, it doesn't matter whether you prove your sorting algorithm using the Permutations method or the multiset method.

```
Corollary sort_specifications_equivalent:
    ∀ sort, is_a_sorting_algorithm sort ↔
is_a_sorting_algorithm' sort.
Proof.
  unfold is_a_sorting_algorithm, is_a_sorting_algorithm'.
  split; intros;
  destruct (H al); split; auto;
  apply same_contents_iff_perm; auto.
Qed.
```