SOFTWARE FOUNDATIONS

VOLUME 3: VERIFIED FUNCTIONAL ALGORITHMS

SELECTION

SELECTION SORT, WITH SPECIFICATION AND PROOF OF CORRECTNESS

This sorting algorithm works by choosing (and deleting) the smallest element, then doing it again, and so on. It takes O(N^2) time.

You should never* use a selection sort. If you want a simple quadratic-time sorting algorithm (for small input sizes) you should use insertion sort. Insertion sort is simpler to implement, runs faster, and is simpler to prove correct. We use selection sort here only to illustrate the proof techniques.

*Well, hardly ever. If the cost of "moving" an element is *much* larger than the cost of comparing two keys, then selection sort is better than insertion sort. But this consideration does not apply in our setting, where the elements are represented as pointers into the heap, and only the pointers need to be moved.

What you should really never use is bubble sort. Bubble sort would be the wrong way to go. Everybody knows that! https://www.youtube.com/watch?v=k4RRi_ntQc8

The Selection-Sort Program

```
Require Import Perm.
```

Find (and delete) the smallest element in a list.

```
Fixpoint select (x: nat) (l: list nat) : nat * list nat :=
match l with
| nil ⇒ (x, nil)
| h::t ⇒ if x <=? h
               then let (j, l') := select x t in (j, h::l')
               else let (j,l') := select h t in (j, x::l')
end.
```

Now, selection-sort works by repeatedly extracting the smallest element, and making a list of the results.

```
(* Uncomment this function, and try it.
Fixpoint selsort l :=
match l with
| i::r => let (j,r') := select i r
               in j :: selsort r'
| nil => nil
end.
*)
```

Error: Recursive call to selsort has principal argument equal to `r'` *instead of* `r`. That is, the recursion is not *structural*, since the list r' is not a structural sublist of (i::r). One way to fix the problem is to use Coq's `Function` feature, and prove that `length(r') <length(i::r)`. Later in this chapter, we'll show that approach.

Instead, here we solve this problem is by providing "fuel", an additional argument that has no use in the algorithm except to bound the amount of recursion. The `n` argument, below, is the fuel.

```
Fixpoint selsort l n {struct n} :=
match l, n with
| x::r, S n' ⇒ let (y,r') := select x r
               in y :: selsort r' n'
| nil, _ ⇒ nil
| _::_, O ⇒ nil (* Oops!  Ran out of fuel! *)
end.
```

What happens if we run out of fuel before we reach the end of the list? Then WE GET THE WRONG ANSWER.

```
Example out_of_gas: selsort [3;1;4;1;5] 3 ≠ [1;1;3;4;5].
Proof.
simpl.
intro. inversion H.
Qed.
```

What happens if we have have too much fuel? No problem.

```
Example too_much_gas: selsort [3;1;4;1;5] 10 = [1;1;3;4;5].
Proof.
simpl.
auto.
Qed.
```

The selection_sort algorithm provides just enough fuel.

```
Definition selection_sort l := selsort l (length l).

Example sort_pi: selection_sort [3;1;4;1;5;9;2;6;5;3;5] =
[1;1;2;3;3;4;5;5;5;6;9].
Proof.
unfold selection_sort.
simpl.
```

```
reflexivity.
Qed.
```

Specification of correctness of a sorting algorithm: it rearranges the elements into a list that is totally ordered.

```
Inductive sorted: list nat → Prop :=
 | sorted_nil: sorted nil
 | sorted_1: ∀ i, sorted (i::nil)
 | sorted_cons: ∀ i j l, i ≤ j → sorted (j::l) → sorted
(i::j::l).

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

Proof of Correctness of Selection sort

Here's what we want to prove.

```
Definition selection_sort_correct : Prop :=
    is_a_sorting_algorithm selection_sort.
```

We'll start by working on part 1, permutations.

Exercise: 3 stars (select_perm)

```
Lemma select_perm: ∀ x l,
  let (y,r) := select x l in
   Permutation (x::l) (y::r).
Proof.
```

NOTE: If you wish, you may `Require Import Multiset` and use the multiset method, along with the theorem `contents_perm`. If you do, you'll still leave the statement of this theorem unchanged.

```
intros x l; revert x.
induction l; intros; simpl in *.
(* FILL IN HERE *) Admitted.
```

☐

Exercise: 3 stars (selection_sort_perm)

```
Lemma selsort_perm:
  ∀ n,
  ∀ l, length l = n → Permutation l (selsort l n).
Proof.
```

NOTE: If you wish, you may `Require Import Multiset` and use the multiset method, along with the theorem `same_contents_iff_perm`.

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

Theorem selection_sort_perm:
  ∀ l, Permutation l (selection_sort l).
```

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

☐

Exercise: 3 stars (select_smallest)

```
Lemma select_smallest_aux:
  ∀ x al y bl,
    Forall (fun z ⇒ y ≤ z) bl →
    select x al = (y,bl) →
    y ≤ x.
Proof.
(* Hint: no induction needed in this lemma.
   Just use existing lemmas about select, along with Forall_perm *)
(* FILL IN HERE *) Admitted.

Theorem select_smallest:
  ∀ x al y bl, select x al = (y,bl) →
     Forall (fun z ⇒ y ≤ z) bl.
Proof.
intros x al; revert x; induction al; intros; simpl in *.
 (* FILL IN HERE *) admit.
bdestruct (x <=? a).
*
destruct (select x al) eqn:?H.
 (* FILL IN HERE *) Admitted.
```

☐

Exercise: 3 stars (selection_sort_sorted)

```
Lemma selection_sort_sorted_aux:
  ∀ y bl,
   sorted (selsort bl (length bl)) →
   Forall (fun z : nat ⇒ y ≤ z) bl →
   sorted (y :: selsort bl (length bl)).
Proof.
 (* Hint: no induction needed.  Use lemmas selsort_perm and Forall_perm.*)
 (* FILL IN HERE *) Admitted.

Theorem selection_sort_sorted: ∀ al, sorted (selection_sort al).
Proof.
intros.
unfold selection_sort.
(* Hint: do induction on the length of al.
    In the inductive case, use select_smallest, select_perm,
    and selection_sort_sorted_aux. *)
 (* FILL IN HERE *) Admitted.
```

☐

Now we wrap it all up.

```
Theorem selection_sort_is_correct: selection_sort_correct.
Proof.
split. apply selection_sort_perm. apply selection_sort_sorted.
Qed.
```

Recursive Functions That are Not Structurally Recursive

`Fixpoint` in Coq allows for recursive functions where some parameter is structurally recursive: in every call, the argument passed at that parameter position is an immediate substructure of the corresponding formal parameter. For recursive functions where that is not the case — but for which you can still prove that they terminate — you can use a more advanced feature of Coq, called `Function`.

```
Require Import Recdef. (* needed for Function feature *)

Function selsort' l {measure length l} :=
match l with
| x::r ⇒ let (y,r') := select x r
               in y :: selsort' r'
| nil ⇒ nil
end.
```

When you use `Function` with `measure`, it's your obligation to prove that the measure actually decreases, before you can use the function.

```
Proof.
intros.
pose proof (select_perm x r).
rewrite teq0 in H.
apply Permutation_length in H.
simpl in *; omega.
Defined. (* Use Defined instead of Qed, otherwise you
  can't compute with the function in Coq. *)
```

Exercise: 3 stars (selsort'_perm)

```
Lemma selsort'_perm:
  ∀ n,
  ∀ l, length l = n → Permutation l (selsort' l).
Proof.
```

NOTE: If you wish, you may `Require Import Multiset` and use the multiset method, along with the theorem `same_contents_iff_perm`.

Important! Don't unfold `selsort'`, or in general, never unfold anything defined with `Function`. Instead, use the recursion equation `selsort'_equation` that is automatically defined by the `Function` command.

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

☐

```
Eval compute in selsort' [3;1;4;1;5;9;2;6;5].
```