SOFTWARE FOUNDATIONS

VOLUME 3: VERIFIED FUNCTIONAL ALGORITHMS

TABLE OF CONTENTS INDEX ROADMAP

ADT

ABSTRACT DATA TYPES

```
Require Import Omega.
```

Let's consider the concept of lookup tables, indexed by keys that are numbers, mapping those keys to values of arbitrary (parametric) type. We can express this in Coq as follows:

```
Module Type TABLE.
 Parameter V: Type.
 Parameter default: V.
 Parameter table: Type.
 Definition key := nat.
 Parameter empty: table.
 Parameter get: key → table → V.
 Parameter set: key → V → table → table.
 Axiom gempty: ∀ k, (* get-empty *)
       get k empty = default.
 Axiom gss: ∀ k v t, (* get-set-same *)
      get k (set k v t) = v.
 Axiom gso: ∀ j k v t, (* get-set-other *)
      j ≠ k → get j (set k v t) = get j t.
End TABLE.
```

This means: in any `Module` that satisfies this `Module Type`, there's a type `table` of lookup-tables, a type `V` of values, and operators `empty`, `get`, `set` that satisfy the axioms `gempty`, `gss`, and `gso`.

It's easy to make an implementation of `TABLE`, using `Maps`. Just for example, let's choose `V` to be `Type`.

```
Require Import Maps.

Module MapsTable <: TABLE.
 Definition V := Type.
 Definition default: V := Prop.
 Definition table := total_map V.
 Definition key := nat.
 Definition empty : table := t_empty default.
 Definition get (k: key) (m: table) : V := m k.
 Definition set (k: key) (v: V) (m: table) : table :=
    t_update m k v.
 Theorem gempty: ∀ k, get k empty = default.
   Proof. intros. reflexivity. Qed.
 Theorem gss: ∀ k v t, get k (set k v t) = v.
   Proof. intros. unfold get, set. apply t_update_eq. Qed.
```

```
  Theorem gso: ∀ j k v t, j≠k → get j (set k v t) = get j t.
    Proof. intros. unfold get, set. apply t_update_neq.
        congruence.
    Qed.
End MapsTable.
```

In summary: to make a `Module` that implements a `Module Type`, you need to provide a `Definition` or `Theorem` in the `Module`, whose type matches the corresponding `Parameter` or `Axiom` in the `Module Type`.

Now, let's calculate: put 1 and then 3 into a map, then lookup 1.

```
Eval compute in MapsTable.get 1 (MapsTable.set 3 unit (MapsTable.set 1
bool MapsTable.empty)).
 (* = bool *)
```

An *Abstract Data Type* comprises:

- A *type* with a hidden representation (in this case, `t`).
- Interface functions that operate on that type (`empty`, `get`, `set`).
- Axioms about the interaction of those functions (`gempty`, `gss`, `gso`).

So, `MapsTable` is an implementation of the `TABLE` abstract type.

The problem with `MapsTable` is that the `Maps` implementation is very inefficient: linear time per `get` operation. If you do a sequence of `N` `get` and `set` operations, it can take time quadratic in `N`. For a more efficient implementation, let's use our search trees.

```
Require Import SearchTree.

Module TreeTable <: TABLE.
 Definition V := Type.
 Definition default : V := Prop.
 Definition table := tree V.
 Definition key := nat.
 Definition empty : table := empty_tree V.
 Definition get (k: key) (m: table) : V := lookup V default k m.
 Definition set (k: key) (v: V) (m: table) : table :=
     insert V k v m.
 Theorem gempty: ∀ k, get k empty = default.
   Proof. intros. reflexivity. Qed.

 Theorem gss: ∀ k v t, get k (set k v t) = v.
   Proof. intros. unfold get, set.
     destruct (unrealistically_strong_can_relate V default t)
        as [cts H].
     assert (H0 := insert_relate V default k v t cts H).
     assert (H1 := lookup_relate V default k _ _ H0).
    rewrite H1. apply t_update_eq.
   Qed.
```

Exercise: 3 stars (TreeTable_gso)

Prove this using techniques similar to the proof of `gss` just above.

```
 Theorem gso: ∀ j k v t, j≠k → get j (set k v t) = get j t.
   Proof.
 (* FILL IN HERE *) Admitted.
```

□

```
End TreeTable.
```

But suppose we don't have an unrealistically strong can-relate theorem? Remember the type of the "ordinary" can_relate:

```
Check can_relate.
  (*  : forall (V : Type) (default : V) (t : tree V),
       SearchTree V t ->
       exists cts : total_map V, Abs V default t cts *)
```

This requires that `t` have the `SearchTree` property, or in general, any value of type `table` should be well-formed, that is, should satisfy the representation invariant. We must ensure that the client of an ADT cannot "forge" values, that is, cannot coerce the representation type into the abstract type; especially ill-formed values of the representation type. This "unforgeability" is enforced in some real programming languages: ML (Standard ML or Ocaml) with its module system; Java, whose Classes have "private variables" that the client cannot see.

A Brief Excursion into Dependent Types

We can enforce the representation invariant in Coq using dependent types. Suppose `P` is a predicate on type `A`, that is, `P: A → Prop`. Suppose `x` is a value of type `A`, and `proof: P x` is the name of the theorem that `x` satisfies `P`. Then `(exist x, proof)` is a "package" of two things: `x`, along with the proof of `P(x)`. The type of `(∃x, proof)` is written as `{x | P x}`.

```
Check exist. (* forall {A : Type} (P : A -> Prop) (x : A),
                P x -> {x | P x}  *)
Check proj1_sig. (* forall {A : Type} {P : A -> Prop},
                 {x | P x} -> A *)
Check proj2_sig. (* forall (A : Type) {P : A -> Prop}
                 (e : {x | P x}),
                 P (proj1_sig e) *)
```

We'll apply that idea to search trees. The type `A` will be `tree V`. The predicate `P(x)` will be `SearchTree(x)`.

```
Module TreeTable2 <: TABLE.
 Definition V := Type.
 Definition default : V := Prop.
 Definition table := {x | SearchTree V x}.
 Definition key := nat.
 Definition empty : table :=
   exist (SearchTree V) (empty_tree V) (empty_tree_SearchTree V).
 Definition get (k: key) (m: table) : V :=
          (lookup V default k (proj1_sig m)).
 Definition set (k: key) (v: V) (m: table) : table :=
   exist (SearchTree V) (insert V k v (proj1_sig m))
          (insert_SearchTree _ _ _ _ (proj2_sig m)).

 Theorem gempty: ∀ k, get k empty = default.
   Proof. intros. reflexivity. Qed.

 Theorem gss: ∀ k v t, get k (set k v t) = v.
  Proof. intros. unfold get, set.
    unfold table in t.
```

Now: `t` is a package with two components: The first component is a tree, and the second component is a proof that the first component has the SearchTree property. We can destruct `t` to see that more clearly.

```
    destruct t as [a Ha].
    (* Watch what this simpl does: *)
    simpl.
    (* Now we can use can_relate instead of unrealistically_strong_can_relate: *)
    destruct (can_relate V default a Ha) as [cts H].
    pose proof (insert_relate V default k v a cts H).
    pose proof (lookup_relate V default k _ _ H0).
    rewrite H1. apply t_update_eq.
  Qed.
```

Exercise: 3 stars (TreeTable_gso)

Prove this using techniques similar to the proof of `gss` just above; don't use `unrealistically_strong_can_relate`.

```
  Theorem gso: ∀ j k v t, j≠k → get j (set k v t) = get j t.
    Proof.
  (* FILL IN HERE *) Admitted.
```

☐

```
End TreeTable2.
```

(End of the brief excursion into dependent types.)

Summary of Abstract Data Type Proofs

```
Section ADT_SUMMARY.
Variable V: Type.
Variable default: V.
```

Step 1. Define a *representation invariant*. (In the case of search trees, the representation invariant is the `SearchTree` predicate.) Prove that each operation on the data type *preserves* the representation invariant. For example:

```
Check (empty_tree_SearchTree V).
  (*  SearchTree V (empty_tree V) *)
Check (insert_SearchTree V).
  (* forall (k : key) (v : V) (t : tree V),
       SearchTree V t -> SearchTree V (insert V k v t) *)
```

Notice two things: Any operator (such as `insert`) that takes a `tree` *parameter* can *assume* that the parameter satisfies the representation invariant. That is, the `insert_SearchTree` theorem takes a premise, `SearchTree V t`.

Any operator that produces a `tree` *result* must prove that the result satisfies the representation invariant. Thus, the conclusions, `SearchTree V (empty_tree V)` and `SearchTree V (empty_tree V)` of the two theorems above.

Finally, any operator that produces a result of "base type", has no obligation to prove that the result satisfies the representation invariant; that wouldn't make any sense anyway, because the types wouldn't match. That is, there's no "lookup_SearchTree" theorem, because `lookup` doesn't return a result that's a `tree`.

Step 2. Define an *abstraction relation*. (In the case of search trees, it's the `Abs` relation. This relates the data structure to some mathematical value that is (presumably) simpler to reason about.

```
Check (Abs V default). (* tree V -> total_map V -> Prop *)
```

For each operator, prove that: assuming each `tree` argument satisfies the representation invariant *and* the abstraction relation, prove that the results also satisfy the appropriate abstraction relation.

```
Check (empty_tree_relate V default). (*
       Abs V default (empty_tree V) (t_empty default)    *)
Check (lookup_relate' V default). (* forall k t cts,
       SearchTree V t ->
       Abs V default t cts ->
       lookup V default k t = cts (Id k)  *)
Check (insert_relate' V default). (*     : forall k v t cts,
       SearchTree V t ->
       Abs V default t cts ->
       Abs V default (insert V k v t) (t_update cts (Id k) v) *)
```

Step 3. Using the representation invariant and the abstraction relation, prove that all the axioms of your ADT are valid. For example...

```
Check TreeTable2.gso. (*
      : forall (j k : TreeTable2.key) (v : TreeTable2.V)
          (t : TreeTable2.table),
       j <> k ->
       TreeTable2.get j (TreeTable2.set k v t) = TreeTable2.get j t  *)

End ADT_SUMMARY.
```

Exercise in Data Abstraction

The rest of this chapter is optional.

```
Require Import List.
Import ListNotations.
```

Here's the Fibonacci function.

```
Fixpoint fibonacci (n: nat) :=
 match n with
 | 0 ⇒ 1
 | S i ⇒ match i with 0 ⇒ 1 | S j ⇒ fibonacci i + fibonacci j end
 end.

Eval compute in map fibonacci [0;1;2;3;4;5;6].
```

Here's a silly little program that computes the Fibonacci function.

```
Fixpoint repeat {A} (f: A→A) (x: A) n :=
 match n with O ⇒ x | S n' ⇒ f (repeat f x n') end.

Definition step (al: list nat) : list nat :=
 List.cons (nth 0 al 0 + nth 1 al 0) al.
```

```
Eval compute in map (repeat step [1;0;0]) [0;1;2;3;4;5].

Definition fib n := nth 0 (repeat step [1;0;0] n) 0.

Eval compute in map fib [0;1;2;3;4;5;6].
```

Here's a strange "List" module.

```
Module Type LISTISH.
 Parameter list: Type.
 Parameter create : nat → nat → nat → list.
 Parameter cons: nat → list → list.
 Parameter nth: nat → list → nat.
End LISTISH.

Module L <: LISTISH.
 Definition list := (nat*nat*nat)%type.
 Definition create (a b c: nat) : list := (a,b,c).
 Definition cons (i: nat) (il : list) := match il with (a,b,c) ⇒
(i,a,b) end.
 Definition nth (n: nat) (al: list) :=
   match al with (a,b,c) ⇒
      match n with 0 ⇒ a | 1 ⇒ b | 2 ⇒ c | _ ⇒ 0 end
   end.
End L.

Definition sixlist := L.cons 0 (L.cons 1 (L.cons 2 (L.create 3 4 5))).

Eval compute in map (fun i ⇒ L.nth i sixlist) [0;1;2;3;4;5;6;7].
```

Module `L` implements *approximations* of lists: it can remember the first three elements, and forget the rest. Now watch:

```
Definition stepish (al: L.list) : L.list :=
 L.cons (L.nth 0 al + L.nth 1 al) al.

Eval compute in map (repeat stepish (L.create 1 0 0)) [0;1;2;3;4;5].

Definition fibish n := L.nth 0 (repeat stepish (L.create 1 0 0) n).

Eval compute in map fibish [0;1;2;3;4;5;6].
```

This little theorem may be useful in the next exercise.

```
Lemma nth_firstn:
  ∀ A d i j (al: list A), i<j → nth i (firstn j al) d = nth i al d.
 +
```

Exercise: 4 stars, optional (listish_abstraction)

In this exercise we will not need a *representation invariant*. Define an abstraction relation:

```
Inductive L_Abs: L.list → List.list nat → Prop :=
 (* FILL IN HERE *)
 .

Definition O_Abs al al' := L_Abs al al'.

(* State these theorems using O_Abs, not L_Abs.
   You'll see why below, at "Opaque". *)
Lemma create_relate : True. (* change this line appropriately *)
(* FILL IN HERE *) Admitted.
```

```
Lemma cons_relate : True. (* change this line appropriately *)
(* FILL IN HERE *) Admitted.

Lemma nth_relate : True. (* change this line appropriately *)
(* FILL IN HERE *) Admitted.
```

Now, we will make these operators opaque. Therefore, in the rest of the proofs in this exercise, you will not unfold their definitions. Instead, you will just use the theorems `create_relate`, `cons_relate`, `nth_relate`.

```
Opaque L.list.
Opaque L.create.
Opaque L.cons.
Opaque L.nth.
Opaque O_Abs.

Lemma step_relate:
  ∀ al al',
   O_Abs al al' →
   O_Abs (stepish al) (step al').
Proof.
(* FILL IN HERE *) Admitted.

Lemma repeat_step_relate:
 ∀ n al al',
 O_Abs al al' →
 O_Abs (repeat stepish al n) (repeat step al' n).
Proof.
(* FILL IN HERE *) Admitted.

Lemma fibish_correct: ∀ n, fibish n = fib n.
Proof. (* No induction needed in this proof! *)
(* FILL IN HERE *) Admitted.
```

□

Exercise: 2 stars, optional (fib_time_complexity)

Suppose you run these three programs call-by-value, that is, as if they were ML programs. `fibonacci N` `fib N` `fibish N` What is the asymptotic time complexity (big-Oh run time) of each, as a function of N? Assume that the `plus` function runs in constant time. You can use terms like "linear," "N log N," "quadratic," "cubic," "exponential." Explain your answers briefly.

fibonacci: `(* fill in here *)`
fib: `(* fill  in here *)`
fibish: `(* fill in here *)`

□