SOFTWARE FOUNDATIONS

VOLUME 3: VERIFIED FUNCTIONAL ALGORITHMS

SEARCHTREE

BINARY SEARCH TREES

Binary search trees are an efficient data structure for lookup tables, that is, mappings from keys to values. The `total_map` type from Maps.v is an *inefficient* implementation: if you add N items to your total_map, then looking them up takes N comparisons in the worst case, and N/2 comparisons in the average case.

In contrast, if your `key` type is a total order — that is, if it has a less-than comparison that's transitive and antisymmetric `a<b ↔ ~(b<a)` — then one can implement binary search trees (BSTs). We will assume you know how BSTs work; you can learn this from:

- Section 3.2 of *Algorithms, Fourth Edition*, by Sedgewick and Wayne, Addison Wesley 2011; or
- Chapter 12 of *Introduction to Algorithms, 3rd Edition*, by Cormen, Leiserson, and Rivest, MIT Press 2009.

Our focus here is to *prove the correctness of an implementation* of binary search trees.

```
Require Import Perm.
Require Import FunctionalExtensionality.
```

Total and Partial Maps

Recall the `Maps` chapter of Volume 1 (Logical Foundations), describing functions from identifiers to some arbitrary type `A`. VFA's `Maps` module is almost exactly the same, except that it implements functions from `nat` to some arbitrary type `A`.

```
Require Import Maps.
```

Sections

We will use Coq's `Section` feature to structure this development, so first a brief introduction to Sections. We'll use the example of lookup tables implemented by lists.

```
Module SectionExample1.
  Definition mymap (V: Type) := list (nat*V).
  Definition empty (V: Type) : mymap V := nil.
  Fixpoint lookup (V: Type) (default: V) (x: nat) (m: mymap V) : V :=
    match m with
     | (a,v)::al ⇒ if x =? a then v else lookup V default x al
     | nil ⇒ default
    end.
```

```
  Theorem lookup_empty (V: Type) (default: V):
       ∀ x, lookup V default x (empty V) = default.
   Proof. reflexivity. Qed.
End SectionExample1.
```

It sure is tedious to repeat the `V` and `default` parameters in every definition and every theorem. The `Section` feature allows us to declare them as parameters to every definition and theorem in the entire section:

```
Module SectionExample2.
  Section MAPS.
  Variable V : Type.
  Variable default: V.

  Definition mymap := list (nat*V).
  Definition empty : mymap := nil.
  Fixpoint lookup (x: nat) (m: mymap) : V :=
    match m with
    | (a,v)::al ⇒ if x =? a then v else lookup x al
    | nil ⇒ default
    end.
  Theorem lookup_empty:
       ∀ x, lookup x empty = default.
   Proof. reflexivity. Qed.
  End MAPS.
End SectionExample2.
```

At the close of the section, this produces exactly the same result: the functions that "need" to be parametrized by `V` or `default` are given extra parameters. We can test this claim, as follows:

```
Goal SectionExample1.empty = SectionExample2.empty.
Proof. reflexivity.
Qed.

Goal SectionExample1.lookup = SectionExample2.lookup.
Proof.
  unfold SectionExample1.lookup, SectionExample2.lookup.
  try reflexivity. (* doesn't do anything. *)
```

Well, not exactly the same; but certainly equivalent. Functions `f` and `g` are "extensionally equal" if, for every argument `x`, `f x = g x`. The Axiom of Extensionality says that if two functions are "extensionally equal" then they are *equal*. The `extensionality` tactic is just a convenient way of applying the axiom of extensionality.

```
  extensionality V; extensionality default; extensionality x.
  extensionality m; simpl.
  induction m as [| [? ?] ]; auto.
  destruct (x=?n); auto.
Qed.
```

Program for Binary Search Trees

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

Definition key := nat.

Inductive tree : Type :=
 | E : tree
```

```
  | T: tree → key → V → tree → tree.

Definition empty_tree : tree := E.

Fixpoint lookup (x: key) (t : tree) : V :=
  match t with
  | E ⇒ default
  | T tl k v tr ⇒ if x <? k then lookup x tl
                         else if k <? x then lookup x tr
                         else v
  end.

Fixpoint insert (x: key) (v: V) (s: tree) : tree :=
 match s with
 | E ⇒ T E x v E
 | T a y v' b ⇒ if x <? y then T (insert x v a) y v' b
                        else if y <? x then T a y v' (insert x v b)
                        else T a x v b
 end.

Fixpoint elements' (s: tree) (base: list (key*V)) : list (key * V) :=
 match s with
 | E ⇒ base
 | T a k v b ⇒ elements' a ((k,v) :: elements' b base)
 end.

Definition elements (s: tree) : list (key * V) := elements' s nil.
```

Search Tree Examples

```
Section EXAMPLES.
Variables v2 v4 v5 : V.
Eval compute in insert 5 v5 (insert 2 v2 (insert 4 v5 empty_tree)).
  (*      = T (T E 2 v2 E) 4 v5 (T E 5 v5 E) *)
Eval compute in lookup 5 (T (T E 2 v2 E) 4 v5 (T E 5 v5 E)).
  (*      = v5 *)
Eval compute in lookup 3 (T (T E 2 v2 E) 4 v5 (T E 5 v5 E)).
  (*      = default *)
Eval compute in elements (T (T E 2 v2 E) 4 v5 (T E 5 v5 E)).
  (*      = (2, v2); (4, v5); (5, v5) *)
End EXAMPLES.
```

What Should We Prove About Search trees?

Search trees are meant to be an implementation of maps. That is, they have an `insert` function that corresponds to the `update` function of a map, and a `lookup` function that corresponds to applying the map to an argument. To prove the correctness of a search-tree algorithm, we can prove:

- Any search tree corresponds to some map, using a function or relation that we demonstrate.
- The lookup function gives the same result as applying the map
- The insert function returns a corresponding map.
- Maps have the properties we actually wanted. It would do no good to prove that searchtrees correspond to some abstract type X, if X didn't have useful properties!

What properties do we want searchtrees to have? If I insert the binding `(k,v)` into a searchtree `t`, then look up `k`, I should get `v`. If I look up `k'` in `insert (k,v) t`, where `k'≠k`, then I should get the same result as `lookup k t`. There are several more properties. Fortunately, all these properties are already proved about `total_map` in the `Maps` module:

```
Check t_update_eq.
(*  : forall (A : Type) (m : total_map A) (x : id) (v : A),
       t_update m x v x = v   *)
Check t_update_neq.
(* : forall (X : Type) (v : X) (x1 x2 : id) (m : total_map X),
       x1 <> x2 -> t_update m x1 v x2 = m x2    *)
Check t_update_shadow.
(* : forall (A : Type) (m : total_map A) (v1 v2 : A) (x : id),
       t_update (t_update m x v1) x v2 = t_update m x v2    *)
Check t_update_same. (* : forall (X : Type) (x : id) (m : total_map X),
        t_update m x (m x) = m    *)
Check t_update_permute.
(* forall (X : Type) (v1 v2 : X) (x1 x2 : id) (m : total_map X),
       x2 <> x1 ->
       t_update (t_update m x2 v2) x1 v1 =
         t_update (t_update m x1 v1) x2 v2    *)
Check t_apply_empty. (* : forall (A : Type) (x : id) (v : A),
       t_empty v x = v *)
```

So, if we like those properties that `total_map` is proved to have, and we can prove that searchtrees behave like maps, then we don't have to reprove each individual property about searchtrees.

More generally: a job worth doing is worth doing well. It does no good to prove the "correctness" of a program, if you prove that it satisfies a wrong or useless specification.

Efficiency of Search Trees

We use binary search trees because they are efficient. That is, if there are `N` elements in a (reasonably well balanced) BST, each insertion or lookup takes about logN time.

What could go wrong?

1. The search tree might not be balanced. In that case, each insertion or lookup will take as much as linear time. - SOLUTION: use an algorithm, such as "red-black trees", that ensures the trees stay balanced. We'll do that in Chapter RedBlack.

2. Our keys are natural numbers, and Coq's `nat` type takes linear time *per comparison*. That is, computing (j <? k) takes time proportional to the *value* of `k-j`. - SOLUTION: represent keys by a data type that has a more efficient comparison operator. We just use `nat` in this chapter because it's something you're already familiar with.

3. There's no notion of "run time" in Coq. That is, we can't say what it means that a Coq function "takes N steps to evaluate." Therefore, we can't prove that binary search trees are efficient. - SOLUTION 1: Don't prove (in Coq) that they're efficient; just prove that they are correct. Prove things about their efficiency the old-fashioned way, on pencil and paper. - SOLUTION 2: Prove in Coq some facts about the height of the trees, which have direct bearing on their efficiency. We'll explore that in later chapters.

4. Our functions in Coq aren't real implementations; they are just pretend models of real implementations. What if there are bugs in the correspondence between the Coq function and the

real implementation?

- SOLUTION: Use Coq's `extraction` feature to derive the real implementation (in Ocaml or Haskell) automatically from the Coq function. Or, use Coq's `vm_compute` or `native_compute` feature to compile and run the programs efficiently inside Coq. We'll explore `extraction` in a later chapter.

Proof of Correctness

We claim that a `tree` "corresponds" to a `total_map`. So we must exhibit an "abstraction relation" `Abs: tree → total_map V → Prop`.

The idea is that `Abs t m` says that tree `t` is a representation of map `m`; or that map `m` is an abstraction of tree `t`. How should we define this abstraction relation?

The empty tree is easy: `Abs E (fun x ⇒ default)`.

Now, what about this tree?:

```
Definition example_tree (v2 v4 v5 : V) :=
   T (T E 2 v2 E) 4 v4 (T E 5 v5 E).
```

Exercise: 2 stars (example_map)

```
(* Fill in the definition of example_map with a total_map that
  you think example_tree should correspond to.  Use
  t_update and (t_empty default). *)

Definition example_map (v2 v4 v5: V) : total_map V
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *). Admitted.
```

☐

To build the `Abs` relation, we'll use these two auxiliary functions that construct maps:

```
Definition combine {A} (pivot: key) (m1 m2: total_map A) : total_map A :=
  fun x ⇒ if x <? pivot then m1 x else m2 x.
```

`combine pivot a b` uses the map `a` on any input less than `pivot`, and uses map `b` on any input ≥ `pivot`.

```
Inductive Abs: tree → total_map V → Prop :=
| Abs_E: Abs E (t_empty default)
| Abs_T: ∀ a b l k v r,
      Abs l a →
      Abs r b →
      Abs (T l k v r) (t_update (combine k a b) k v).
```

Exercise: 3 stars (check_example_map)

Prove that your `example_map` is the right one. If it isn't, go back and fix your definition of `example_map`. You will probably need the `bdestruct` tactic, and `omega`.

```
Lemma check_example_map:
  ∀ v2 v4 v5, Abs (example_tree v2 v4 v5) (example_map v2 v4 v5).
Proof.
intros.
unfold example_tree.
evar (m: total_map V).
replace (example_map v2 v4 v5) with m; subst m.
```

```
  repeat constructor.
  extensionality x.
  (* HINT:
    First,    unfold example_map, t_update, combine, t_empty, beq_id.
    Then, repeat the following procedure:  If you see something like
    if 4 =? x then ... else ...,    use the tactic bdestruct (4 =? x).
    If the arithmetic facts above the line can't all be true, use omega.
    If you're too lazy to check for yourself whether they are true,
     use bdestruct (4 =? x); try omega.
  *)
  (* FILL IN HERE *) Admitted.
```

☐

You can ignore this lemma, unless it fails.

```
Lemma check_too_clever: ∀ v2 v4 v5: V, True.
+

Theorem empty_tree_relate: Abs empty_tree (t_empty default).
Proof.
constructor.
Qed.
```

Exercise: 3 stars (lookup_relate)

```
Theorem lookup_relate:
  ∀ k t cts ,
    Abs t cts → lookup k t = cts k.
Proof.
(* FILL IN HERE *) Admitted.
```

☐

Exercise: 4 stars (insert_relate)

```
Theorem insert_relate:
 ∀ k v t cts,
    Abs t cts →
    Abs (insert k v t) (t_update cts k v).
Proof.
(* FILL IN HERE *) Admitted.
```

☐

Correctness Proof of the `elements` Function

How should we specify what `elements` is supposed to do? Well, `elements t` returns a list of pairs `(k1,v1);(k2;v2);...;(kn,vn)` that ought to correspond to the total_map, `t_update ... (t_update (t_update (t_empty default) (Id k1) v1) (Id k2) v2) ... (Id kn) vn`.

We can formalize this quite easily.

```
Fixpoint list2map (el: list (key*V)) : total_map V :=
 match el with
  | nil ⇒ t_empty default
  | (i,v)::el' ⇒ t_update (list2map el') i v
 end.
```

Exercise: 3 stars (elements_relate_informal)

```
Theorem elements_relate:
  ∀ t cts, Abs t cts → list2map (elements t) = cts.
Proof.
```

Don't prove this yet. Instead, explain in your own words, with examples, why this must be true. It's OK if your explanation is not a formal proof; it's even OK if your explanation is subtly wrong! Just make it convincing.

```
    (* FILL IN YOUR EXPLANATION HERE *)
  Abort.
```

☐

Instead of doing a *formal* proof that `elements_relate` is true, prove that it's false! That is, as long as type `V` contains at least two distinct values.

Exercise: 4 stars (not_elements_relate)

```
Theorem not_elements_relate:
  ∀ v, v ≠ default →
  ¬ (∀ t cts, Abs t cts → list2map (elements t) = cts).
Proof.
intros.
intro.
pose (bogus := T (T E 3 v E) 2 v E).
pose (m := t_update (t_empty default) 2 v).
pose (m' := t_update
        (combine 2
           (t_update (combine 3 (t_empty default) (t_empty default)) 3 v)
           (t_empty default)) 2 v).
assert (Paradox: list2map (elements bogus) = m ∧ list2map (elements bogus)
≠ m).
split.
```

To prove the first subgoal, prove that `m=m'` (by `extensionality`) and then use `H`.

To prove the second subgoal, do an `intro` so that you can assume `update_list (t_empty default) (elements bogus) = m`, then show that `update_list (t_empty default) (elements bogus) (Id 3) ≠ m (Id 3)`. That's a contradiction.

To prove the third subgoal, just destruct `Paradox` and use the contradiction.

In all 3 goals, when you need to unfold local definitions such as `bogus` you can use `unfold bogus` or `subst bogus`.

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

☐

What went wrong? Clearly, `elements_relate` is true; you just explained why. And clearly, it's not true, because `not_elements_relate` is provable in Coq. The problem is that the tree `(T (T E 3 v E) 2 v E)` is bogus: it's not a well-formed binary search tree, because there's a 3 in the left subtree of the 2 node, and 3 is not less than 2.

If you wrote a good answer to the `elements_relate_informal` exercise, (that is, an answer that is only subtly wrong), then the subtlety is that you assumed that the search tree is well formed. That's a reasonable assumption; but we will have to prove that all the trees we operate on will be well formed.

Representation Invariants

A `tree` has the `SearchTree` property if, at any node with key `k`, all the keys in the left subtree are less than `k`, and all the keys in the right subtree are greater than `k`. It's not completely obvious how to formalize that! Here's one way: it's correct, but not very practical.

```
Fixpoint forall_nodes (t: tree) (P: tree→key→V→tree→Prop) : Prop :=
 match t with
 | E ⇒ True
 | T l k v r ⇒ P l k v r ∧ forall_nodes l P ∧ forall_nodes r P
 end.

Definition SearchTreeX (t: tree) :=
 forall_nodes t
   (fun l k v r ⇒
      forall_nodes l (fun _ j _ _ ⇒ j<k) ∧
      forall_nodes r (fun _ j _ _ ⇒ j>k)).

Lemma example_SearchTree_good:
   ∀ v2 v4 v5, SearchTreeX (example_tree v2 v4 v5).
Proof.
intros.
hnf. simpl.
repeat split; auto.
Qed.

Lemma example_SearchTree_bad:
   ∀ v, ¬SearchTreeX (T (T E 3 v E) 2 v E).
Proof.
intros.
intro.
hnf in H; simpl in H.
do 3 destruct H.
omega.
Qed.

Theorem elements_relate_second_attempt:
  ∀ t cts,
  SearchTreeX t →
  Abs t cts →
  list2map (elements t) = cts.
Proof.
```

This is probably provable. But the `SearchTreeX` property is quite unwieldy, with its two Fixpoints nested inside a Fixpoint. Instead of using `SearchTreeX`, let's reformulate the searchtree property as an inductive proposition without any nested induction.

```
Abort.

Inductive SearchTree' : key → tree → key → Prop :=
| ST_E : ∀ lo hi, lo ≤ hi → SearchTree' lo E hi
| ST_T: ∀ lo l k v r hi,
    SearchTree' lo l k →
    SearchTree' (S k) r hi →
    SearchTree' lo (T l k v r) hi.

Inductive SearchTree: tree → Prop :=
| ST_intro: ∀ t hi, SearchTree' 0 t hi → SearchTree t.

Lemma SearchTree'_le:
  ∀ lo t hi, @SearchTree' lo t hi → lo ≤ hi.
Proof.
induction 1; omega.
Qed.
```

Before we prove that `elements` is correct, let's consider a simpler version.

```
Fixpoint slow_elements (s: tree) : list (key * V) :=
 match s with
```

```
  | E ⇒ nil
  | T a k v b ⇒ slow_elements a ++ [(k,v)] ++ slow_elements b
  end.
```

This one is easier to understand than the `elements` function, because it doesn't carry the `base` list around in its recursion. Unfortunately, its running time is quadratic, because at each of the `T` nodes it does a linear-time list-concatentation. The original `elements` function takes linear time overall; that's much more efficient.

To prove correctness of `elements`, it's actually easier to first prove that it's equivalent to `slow_elements`, then prove the correctness of `slow_elements`. We don't care that `slow_elements` is quadratic, because we're never going to really run it; it's just there to support the proof.

Exercise: 3 stars, optional (elements_slow_elements)

```
Theorem elements_slow_elements: elements = slow_elements.
Proof.
extensionality s.
unfold elements.
assert (∀ base, elements' s base = slow_elements s ++ base).
(* FILL IN HERE *) Admitted.
```

☐

Exercise: 3 stars, optional (slow_elements_range)

```
Lemma slow_elements_range:
 ∀ k v lo t hi,
  SearchTree' lo t hi →
  In (k,v) (slow_elements t) →
  lo ≤ k < hi.
Proof.
(* FILL IN HERE *) Admitted.
```

☐

Auxiliary Lemmas About `In` and `list2map`

```
Lemma In_decidable:
  ∀ (al: list (key*V)) (i: key),
  (∃ v, In (i,v) al) ∨ (¬∃ v, In (i,v) al).
+

Lemma list2map_app_left:
  ∀ (al bl: list (key*V)) (i: key) v,
     In (i,v) al → list2map (al++bl) i = list2map al i.
+

Lemma list2map_app_right:
  ∀ (al bl: list (key*V)) (i: key),
     ~(∃ v, In (i,v) al) → list2map (al++bl) i = list2map bl i.
+

Lemma list2map_not_in_default:
 ∀ (al: list (key*V)) (i: key),
   ~(∃ v, In (i,v) al) → list2map al i = default.
+
```

Exercise: 3 stars, optional (elements_relate)

```
Theorem elements_relate:
  ∀ t cts,
```

```
    SearchTree t →
    Abs t cts →
    list2map (elements t) = cts.
  Proof.
  rewrite elements_slow_elements.
  intros until 1. inv H.
  revert cts; induction H0; intros.
  * (* ST_E case *)
  inv H0.
  reflexivity.
  * (* ST_T case *)
  inv H.
  specialize (IHSearchTree'1 _ H5). clear H5.
  specialize (IHSearchTree'2 _ H6). clear H6.
  unfold slow_elements; fold slow_elements.
  subst.
  extensionality i.
  destruct (In_decidable (slow_elements l) i) as [[w H] | Hleft].
  rewrite list2map_app_left with (v:=w); auto.
  pose proof (slow_elements_range _ _ _ _ _ H0_ H).
  unfold combine, t_update.
  bdestruct (k=?i); [ omega | ].
  bdestruct (i<?k); [ | omega].
  auto.
  (* FILL IN HERE *) Admitted.
```

□

Preservation of Representation Invariant

How do we know that all the trees we will encounter (particularly, that the `elements` function will encounter), have the `SearchTree` property? Well, the empty tree is a `SearchTree`; and if you insert into a tree that's a `SearchTree`, then the result is a `SearchTree`; and these are the only ways that you're supposed to build trees. So we need to prove those two theorems.

Exercise: 1 star (empty_tree_SearchTree)

```
  Theorem empty_tree_SearchTree: SearchTree empty_tree.
  Proof.
  clear default.
  (* This is here to avoid a nasty interaction between Admitted
     and Section/Variable.  It's also a hint that the default value
     is not needed in this theorem. *)
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars (insert_SearchTree)

```
  Theorem insert_SearchTree:
    ∀ k v t,
     SearchTree t → SearchTree (insert k v t).
  Proof.
  clear default.
  (* This is here to avoid a nasty interaction between Admitted and Section/Variable *)
  (* FILL IN HERE *) Admitted.
```

□

We Got Lucky

Recall the statement of `lookup_relate`:

```
Check lookup_relate.
(*  forall (k : key) (t : tree) (cts : total_map V),
       Abs t cts -> lookup k t = cts (Id k)   *)
```

In general, to prove that a function satisfies the abstraction relation, one also needs to use the representation invariant. That was certainly the case with `elements_relate`:

```
Check elements_relate.
(*  : forall (t : tree) (cts : total_map V),
       SearchTree t -> Abs t cts -> elements_property t cts    *)
```

To put that another way, the general form of `lookup_relate` should be:

```
Lemma lookup_relate':
  ∀ (k : key) (t : tree) (cts : total_map V),
    SearchTree t → Abs t cts → lookup k t = cts k.
```

That is certainly provable, since it's a weaker statement than what we proved:

```
Proof.
intros.
apply lookup_relate.
apply H0.
Qed.

Theorem insert_relate':
 ∀ k v t cts,
    SearchTree t →
    Abs t cts →
    Abs (insert k v t) (t_update cts k v).
Proof. intros. apply insert_relate; auto.
Qed.
```

The question is, why did we not need the representation invariant in the proof of `lookup_relate`? The answer is that our particular Abs relation is much more clever than necessary:

```
Print Abs.
(* Inductive Abs : tree -> total_map V -> Prop :=
    Abs_E : Abs E (t_empty default)
  | Abs_T : forall (a b: total_map V) (l: tree) (k: key) (v: V) (r: tree),
            Abs l a ->
            Abs r b ->
            Abs (T l k v r) (t_update (combine k a b) (Id k) v)
*)
```

Because the `combine` function is chosen very carefully, it turns out that this abstraction relation even works on bogus trees!

```
Remark abstraction_of_bogus_tree:
 ∀ v2 v3,
   Abs (T (T E 3 v3 E) 2 v2 E) (t_update (t_empty default) 2 v2).
Proof.
intros.
evar (m: total_map V).
replace (t_update (t_empty default) 2 v2) with m; subst m.
repeat constructor.
extensionality x.
unfold t_update, combine, t_empty.
bdestruct (2 =? x).
auto.
bdestruct (x <? 2).
```

```
bdestruct (3 =? x).
(* LOOK HERE! *)
omega.
bdestruct (x <? 3).
auto.
auto.
auto.
Qed.
```

Step through the proof to `LOOK HERE`, and notice what's going on. Just when it seems that `(T (T E 3 v3 E) 2 v2 E)` is about to produce `v3` while `(t_update (t_empty default) (Id 2) v2)` is about to produce `default`, `omega` finds a contradiction. What's happening is that `combine 2` is careful to ignore any keys >= 2 in the left-hand subtree.

For that reason, `Abs` matches the *actual* behavior of `lookup`, even on bogus trees. But that's a really strong condition! We should not have to care about the behavior of `lookup` (and `insert`) on bogus trees. We should not need to prove anything about it, either.

Sure, it's convenient in this case that the abstraction relation is able to cope with ill-formed trees. But in general, when proving correctness of abstract-data-type (ADT) implementations, it may be a lot of extra effort to make the abstraction relation as heavy-duty as that. It's often much easier for the abstraction relation to assume that the representation is well formed. Thus, the general statement of our correctness theorems will be more like `lookup_relate'` than like `lookup_relate`.

Every Well-Formed Tree Does Actually Relate to an Abstraction

We're not quite done yet. We would like to know that *every tree that satisfies the representation invariant, means something*.

So as a general sanity check, we need the following theorem:

Exercise: 2 stars (can_relate)

```
Lemma can_relate:
 ∀ t, SearchTree t → ∃ cts, Abs t cts.
Proof.
(* FILL IN HERE *) Admitted.
```
☐

Now, because we happen to have a super-strong abstraction relation, that even works on bogus trees, we can prove a super-strong can_relate function:

Exercise: 2 stars (unrealistically_strong_can_relate)

```
Lemma unrealistically_strong_can_relate:
 ∀ t, ∃ cts, Abs t cts.
Proof.
(* FILL IN HERE *) Admitted.
```
☐

It Wasn't Really Luck, Actually

In the previous section, I said, "we got lucky that the abstraction relation that I happened to choose had this super-strong property."

But actually, the first time I tried to prove correctness of search trees, I did *not* get lucky. I chose a different abstraction relation:

```
Definition AbsX (t: tree) (m: total_map V) : Prop :=
    list2map (elements t) = m.
```

It's easy to prove that `elements` respects this abstraction relation:

```
Theorem elements_relateX:
  ∀ t cts,
  SearchTree t →
  AbsX t cts →
  list2map (elements t) = cts.
Proof.
intros.
apply H0.
Qed.
```

But it's not so easy to prove that `lookup` and `insert` respect this relation. For example, the following claim is not true.

```
Theorem naive_lookup_relateX:
  ∀ k t cts ,
    AbsX t cts → lookup k t = cts k.
Abort. (* Not true *)
```

In fact, `naive_lookup_relateX` is provably false, as long as the type `V` contains at least two different values.

```
Theorem not_naive_lookup_relateX:
   ∀ v, default ≠ v →
    ¬ (∀ k t cts , AbsX t cts → lookup k t = cts k).
Proof.
unfold AbsX.
intros v H.
intros H0.
pose (bogus := T (T E 3 v E) 2 v E).
pose (m := t_update (t_update (t_empty default) 2 v) 3 v).
assert (list2map (elements bogus) = m).
  reflexivity.
assert (¬ lookup 3 bogus = m 3). {
  unfold bogus, m, t_update, t_empty.
  simpl.
  apply H.
}
(** Right here you see how it is proved.  bogus is our old friend,
    the bogus tree that does not satisfy the SearchTree property.
    m is the total_map that corresponds to the elements of bogus.
    The lookup function returns default at key 3,
    but the map m returns v at key 3.  And yet, assumption H0
    claims that they should return the same thing. *)
apply H2.
apply H0.
apply H1.
Qed.
```

Exercise: 4 stars, optional (lookup_relateX)

```
Theorem lookup_relateX:
  ∀ k t cts ,
    SearchTree t → AbsX t cts → lookup k t = cts k.
Proof.
intros.
unfold AbsX in H0. subst cts.
inv H. remember 0 as lo in H0.
clear - H0.
rewrite elements_slow_elements.
```

To prove this, you'll need to use this collection of facts: `In_decidable`, `list2map_app_left`, `list2map_app_right`, `list2map_not_in_default`, `slow_elements_range`. The point is, it's not very pretty.

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

□

Coherence With `elements` Instead of `lookup`

The first definition of the abstraction relation, `Abs`, is "coherent" with the `lookup` operation, but not very coherent with the `elements` operation. That is, `Abs` treats all trees, including ill-formed ones, much the way `lookup` does, so it was easy to prove `lookup_relate`. But it was harder to prove `elements_relate`.

The alternate abstraction relation, `AbsX`, is coherent with `elements`, but not very coherent with `lookup`. So proving `elements_relateX` is trivial, but proving `lookup_relate` is difficult.

This kind of thing comes up frequently. The important thing to remember is that you often have choices in formulating the abstraction relation, and the choice you make will affect the simplicity and elegance of your proofs. If you find things getting too difficult, step back and reconsider your abstraction relation.

```
End TREES.
```