Contents

Chapter 1

Library VFA.Maps

1.1 Maps: Total and Partial Maps

This file is almost identical to the Maps chapter of Software Foundations volume 1 (Logical Foundations), except that it implements functions from **nat** to A rather than functions from id to A.

Maps (or dictionaries) are ubiquitous data structures, both in software construction generally and in the theory of programming languages in particular; we're going to need them in many places in the coming chapters. They also make a nice case study using ideas we've seen in previous chapters, including building data structures out of higher-order functions (from *Basics* and *Poly*) and the use of reflection to streamline proofs (from *IndProp*).

We'll define two flavors of maps: *total* maps, which include a "default" element to be returned when a key being looked up doesn't exist, and *partial* maps, which return an **option** to indicate success or failure. The latter is defined in terms of the former, using None as the default element.

1.2 The Coq Standard Library

One small digression before we start.

Unlike the chapters we have seen so far, this one does not `Require Import` the chapter before it (and, transitively, all the earlier chapters). Instead, in this chapter and from now, on we're going to import the definitions and theorems we need directly from Coq's standard library stuff. You should not notice much difference, though, because we've been careful to name our own definitions and theorems the same as their counterparts in the standard library, wherever they overlap.

```
Require Import Coq.Arith.Arith.
Require Import Coq.Bool.Bool.
Require Import Coq.Logic.FunctionalExtensionality.
```

Documentation for the standard library can be found at http://coq.inria.fr/library/.

The `Search` command is a good way to look for theorems involving objects of specific types.

1.3 Total Maps

Our main job in this chapter will be to build a definition of partial maps that is similar in behavior to the one we saw in the *Lists* chapter, plus accompanying lemmas about their behavior.

This time around, though, we're going to use *functions*, rather than lists of key-value pairs, to build maps. The advantage of this representation is that it offers a more *extensional* view of maps, where two maps that respond to queries in the same way will be represented as literally the same thing (the same function), rather than just "equivalent" data structures. This, in turn, simplifies proofs that use maps.

We build partial maps in two steps. First, we define a type of *total maps* that return a default value when we look up a key that is not present in the map.

```
Definition total_map (A:Type) := nat → A.
```

Intuitively, a total map over an element type A *is* just a function that can be used to look up ids, yielding As.

The function t_empty yields an empty total map, given a default element; this map always returns the default element when applied to any id.

```
Definition t_empty {A:Type} (v : A) : total_map A :=
  (fun _ ⇒ v).
```

More interesting is the update function, which (as before) takes a map m, a key x, and a value v and returns a new map that takes x to v and takes every other key to whatever m does.

```
Definition t_update {A:Type} (m : total_map A)
                    (x : nat) (v : A) :=
  fun x' ⇒ if beq_nat x x' then v else m x'.
```

This definition is a nice example of higher-order programming. The t_update function takes a *function* m and yields a new function `fun x' ⇒ ...` that behaves like the desired map.

For example, we can build a map taking ids to **bool**s, where *Id* 3 is mapped to true and every other key is mapped to false, like this:

```
Definition examplemap :=
  t_update (t_update (t_empty false) 1 false) 3 true.
```

This completes the definition of total maps. Note that we don't need to define a find operation because it is just function application!

```
Example update_example1 : examplemap 0 = false.
Proof. reflexivity. Qed.
Example update_example2 : examplemap 1 = false.
```

```
Proof. reflexivity. Qed.
```

```
Example update_example3 : examplemap 2 = false.
Proof. reflexivity. Qed.
```

```
Example update_example4 : examplemap 3 = true.
Proof. reflexivity. Qed.
```

To use maps in later chapters, we'll need several fundamental facts about how they behave. Even if you don't work the following exercises, make sure you thoroughly understand the statements of the lemmas! (Some of the proofs require the functional extensionality axiom, which is discussed in the Logic chapter and included in the Coq standard library.)

Exercise: 1 star, optional (t_apply_empty) First, the empty map returns its default element for all keys: `Lemma t_apply_empty: ∀ A x v, @t_empty A v x = v.`

```
Proof.
  Admitted.
```

□

Exercise: 2 stars, optional (t_update_eq) Next, if we update a map m at a key x with a new value v and then look up x in the map resulting from the update, we get back v:

```
Lemma t_update_eq : ∀ A (m: total_map A) x v,
  (t_update m x v) x = v.
Proof.
  Admitted.
```

□

Exercise: 2 stars, optional (t_update_neq) On the other hand, if we update a map m at a key $x1$ and then look up a *different* key $x2$ in the resulting map, we get the same result that m would have given:

```
Theorem t_update_neq : ∀ (X:Type) v x1 x2
                            (m : total_map X),
  x1 ≠ x2 →
  (t_update m x1 v) x2 = m x2.
Proof.
  Admitted.
```

□

Exercise: 2 stars, optional (t_update_shadow) If we update a map m at a key x with a value $v1$ and then update again with the same key x and another value $v2$, the resulting map behaves the same (gives the same result when applied to any key) as the simpler map obtained by performing just the second update on m:

```
Lemma t_update_shadow : ∀ A (m: total_map A) v1 v2 x,
    t_update (t_update m x v1) x v2
  = t_update m x v2.
Proof.
  Admitted.
```

□

For the final two lemmas about total maps, it's convenient to use the reflection idioms introduced in chapter *IndProp*. We begin by proving a fundamental *reflection lemma* relating the equality proposition on ids with the boolean function *beq_id*.

Exercise: 2 stars (beq_idP) Use the proof of *beq_natP* in chapter *IndProp* as a template to prove the following:

```
Lemma beq_idP : ∀ x y, reflect (x = y) (beq_nat x y).
Proof.
  Admitted.
```

□

Now, given ids *x1* and *x2*, we can use the destruct (beq_idP *x1 x2*) to simultaneously perform case analysis on the result of *beq_id x1 x2* and generate hypotheses about the equality (in the sense of =) of *x1* and *x2*.

Exercise: 2 stars (t_update_same) Using the example in chapter *IndProp* as a template, use beq_idP to prove the following theorem, which states that if we update a map to assign key *x* the same value as it already has in *m*, then the result is equal to *m*:

```
Theorem t_update_same : ∀ X x (m : total_map X),
  t_update m x (m x) = m.
Proof.
  Admitted.
```

□

Exercise: 3 stars, recommended (t_update_permute) Use beq_idP to prove one final property of the update function: If we update a map *m* at two distinct keys, it doesn't matter in which order we do the updates.

```
Theorem t_update_permute : ∀ (X:Type) v1 v2 x1 x2
                                 (m : total_map X),
  x2 ≠ x1 →
    (t_update (t_update m x2 v2) x1 v1)
  = (t_update (t_update m x1 v1) x2 v2).
Proof.
  Admitted.
```

□

1.4 Partial maps

Finally, we define *partial maps* on top of total maps. A partial map with elements of type A is simply a total map with elements of type **option** A and default element None.

```
Definition partial_map (A:Type) := total_map (option A).

Definition empty {A:Type} : partial_map A :=
  t_empty None.

Definition update {A:Type} (m : partial_map A)
                  (x : nat) (v : A) :=
  t_update m x (Some v).
```

We can now lift all of the basic lemmas about total maps to partial maps.

```
Lemma apply_empty : ∀ A x, @empty A x = None.
Proof.
  intros. unfold empty. rewrite t_apply_empty.
  reflexivity.
Qed.

Lemma update_eq : ∀ A (m: partial_map A) x v,
  (update m x v) x = Some v.
Proof.
  intros. unfold update. rewrite t_update_eq.
  reflexivity.
Qed.

Theorem update_neq : ∀ (X:Type) v x1 x2
                       (m : partial_map X),
  x2 ≠ x1 →
  (update m x2 v) x1 = m x1.
Proof.
  intros X v x1 x2 m H.
  unfold update. rewrite t_update_neq. reflexivity.
  apply H. Qed.

Lemma update_shadow : ∀ A (m: partial_map A) v1 v2 x,
  update (update m x v1) x v2 = update m x v2.
Proof.
  intros A m v1 v2 x1. unfold update. rewrite t_update_shadow.
  reflexivity.
Qed.

Theorem update_same : ∀ X v x (m : partial_map X),
  m x = Some v →
  update m x v = m.
Proof.
```

```
  intros X v x m H. unfold update. rewrite ← H.
  apply t_update_same.
Qed.

Theorem update_permute : ∀ (X:Type) v1 v2 x1 x2
                                        (m : partial_map X),
  x2 ≠ x1 →
    (update (update m x2 v2) x1 v1)
  = (update (update m x1 v1) x2 v2).
Proof.
  intros X v1 v2 x1 x2 m. unfold update.
  apply t_update_permute.
Qed.
```

Date

Chapter 2

Library VFA.Preface

2.1 Preface

2.2 Welcome

Here's a good way to build formally verified correct software:

- Write your program in an expressive language with a good proof theory (the Gallina language embedded in Coq's logic).
- Prove it correct in Coq.
- Compile it with an optimizing ML compiler.

Since you want your programs to be *efficient*, you'll want to implement sophisticated data structures and algorithms. Since Gallina is a *purely functional* language, it helps to have purely functional algorithms.

In this volume you will learn how to specify and verify (prove the correctness of) sorting algorithms, binary search trees, balanced binary search trees, and priority queues. Before using this book, you should have some understanding of these algorithms and data structures, available in any standard undergraduate algorithms textbook.

This electronic book is Volume 3 of the *Software Foundations* series, which presents the mathematical underpinnings of reliable software. It builds on *Software Foundations Volume 1* (Logical Foundations), but does not depend on Volume 2. The exposition here is intended for a broad range of readers, from advanced undergraduates to PhD students and researchers.

The principal novelty of *Software Foundations* is that it is one hundred percent formalized and machine-checked: the entire text is literally a script for Coq. It is intended to be read alongside an interactive session with Coq. All the details in the text are fully formalized in Coq, and the exercises are designed to be worked using Coq.

2.3 Practicalities

2.3.1 Chapter Dependencies

Before using *Verified Functional Algorithms*, read (and do the exercises in) these chapters of *Software Foundations Volume I*: Preface, Basics, Induction, Lists, Poly, Tactics, Logic, IndProp, Maps, and perhaps (ProofObjects), (IndPrinciples).

In this volume, the core path is:
Preface -> Perm -> `Sort` -> **SearchTree** -> Redblack
with many optional chapters whose dependencies are,

- `Sort` -> Multiset or Selection or Decide
- **SearchTree** -> ADT or `Extract`
- Perm -> Trie
- `Sort` -> Selection -> **SearchTree** -> ADT -> Priqueue -> Binom

The Color chapter is advanced material that should not be attempted until the student has had experience with most of the earlier chapters, or other experience using Coq.

2.3.2 System Requirements

Coq runs on Windows, Linux, and OS X. The Preface of Volume 1 describes the Coq installation you will need. This edition was built with Coq 8.8.0.

In addition, two of the chapters ask you to compile and run an OCaml program; having OCaml installed on your computer is helpful, but not essential.

2.3.3 Exercises

Each chapter includes numerous exercises. Each is marked with a "star rating," which can be interpreted as follows:

- One star: easy exercises that underscore points in the text and that, for most readers, should take only a minute or two. Get in the habit of working these as you reach them.
- Two stars: straightforward exercises (five or ten minutes).
- Three stars: exercises requiring a bit of thought (ten minutes to half an hour).
- Four and five stars: more difficult exercises (half an hour and up).

Also, some exercises are marked “advanced”, and some are marked “optional.” Doing just the non-optional, non-advanced exercises should provide good coverage of the core material. Optional exercises provide a bit of extra practice with key concepts and introduce secondary themes that may be of interest to some readers. Advanced exercises are for readers who want an extra challenge (and, in return, a deeper contact with the material).

Please do not post solutions to the exercises in any public place: Software Foundations is widely used both for self-study and for university courses. Having solutions easily available makes it much less useful for courses, which typically have graded homework assignments. The authors especially request that readers not post solutions to the exercises anyplace where they can be found by search engines.

2.3.4 Downloading the Coq Files

A tar file containing the full sources for the “release version” of this book (as a collection of Coq scripts and HTML files) is available at http://softwarefoundations.cis.upenn.edu.

(If you are using the book as part of a class, your professor may give you access to a locally modified version of the files, which you should use instead of the release version.)

2.3.5 Lecture Videos

Lectures on for an intensive summer course based on some chapters of this book at the DeepSpec summer school in 2017 can be found at https://deepspec.org/event/dsss17/lecture_appel.html.

2.3.6 For Instructors and Contributors

If you plan to use these materials in your own course, you will undoubtedly find things you’d like to change, improve, or add. Your contributions are welcome! Please see the Preface to *Logical Foundations* for instructions.

2.4 Thanks

Development of the *Software Foundations* series has been supported, in part, by the National Science Foundation under the NSF Expeditions grant 1521523, *The Science of Deep Specification.*

Chapter 3

Library VFA.Perm

3.1 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 Import Coq.Strings.String.
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.
```

3.2 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. Check lt. Goal Nat.lt = lt. Proof. reflexivity. Qed. Check Nat.ltb. Locate
"_< _". Locate "<?".
```

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.
```

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.
```

3.2.1 Relating `Prop` to **bool**

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

```
Print reflect.
```

That is, **reflect** P b means that $P \leftrightarrow$**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].
  ×
     omega.    ×
     apply not_lt in H.        omega.
Qed.
```

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

3.2.2 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 :=
  let H := fresh in let e := fresh "e" in
   evar (e: Prop);
   assert (H: reflect e X); subst e;
    [eauto with bdestruct
    | destruct H as [H|H];
       [ | try first [apply not_lt in H | apply not_le in H]]].
```

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).    ×
     omega.    ×
     omega.
Qed.
```

3.2.3 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.
```

3.2.4 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.

```
  Search (¬ _ ≤ _ → _).
  apply not_le in H0.
  Search (_ > _ → _ > _ → _ > _).
  apply gt_trans with j.
  apply gt_trans with (k-3).
Abort.

Theorem bogus_subtraction: ¬ (∀ k:nat, k > k - 3).
Proof.
  intro.
  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.   intros.
  apply not_le in H0.
  unfold gt in H0.
  unfold gt.
```

```
  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.
```

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 2ˆN 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: $<\ =\ >\ \leq\ \geq\ +$ - $\neg$, as well as multiplication by small integer literals (such as 0,1,2,3...) and some uses of $\vee$ and $\wedge$.

Omega does *not* understand other operators. It treats things like $a \times b$ and $f\ x\ y$ as if they were variables. That is, it can prove $f\ x\ y > a \times b \rightarrow f\ x\ y + 3 \geq a \times b$, in the same way it would prove $u > v \rightarrow u{+}3 \geq v$.

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). Check (1>?2).
```

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

```
Locate ">?".
```

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

```
Print Nat.ltb.
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.
```

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). ×
  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.
×
  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.
```

3.3 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. Check Permutation.
```

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.
```

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.
```

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

```
Definition manual_grade_for_Permutation_properties : option (prod nat string) := None.
```

□

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.
 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.
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.
 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.
  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.
  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.
  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.
  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.
```

3.4 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: $\forall$ $\{A\}$ $(f\colon A \rightarrow$ Prop) al bl,
  Permutation al bl $\rightarrow$
  Forall f al $\rightarrow$ **Forall** f bl.

Proof.

Admitted.

□

Date

Chapter 4

Library VFA.Sort

4.1 Sort: Insertion Sort

Sorting can be done in O(N log N) time by various algorithms (quicksort, mergesort, heapsort, etc.). But for smallish inputs, a simple quadratic-time algorithm such as insertion sort can actually be faster. And it's certainly easier to implement – and to prove correct.

4.2 Recommended Reading

If you don't already know how insertion sort works, see Wikipedia or read any standard textbook; for example:

Sections 2.0 and 2.1 of *Algorithms, Fourth Edition*, by Sedgewick and Wayne, Addison Wesley 2011; or

Section 2.1 of *Introduction to Algorithms, 3rd Edition*, by Cormen, Leiserson, and Rivest, MIT Press 2009.

4.3 The Insertion-Sort Program

Insertion sort is usually presented as an imperative program operating on arrays. But it works just as well as a functional program operating on linked lists!

```
From VFA Require Import Perm.

Fixpoint insert (i:nat) (l: list nat) :=
  match l with
  | nil ⇒ i::nil
  | h::t ⇒ if i <=? h then i::h::t else h :: insert i t
 end.

Fixpoint sort (l: list nat) : list nat :=
  match l with
```

```
  | nil ⇒ nil
  | h::t ⇒ insert h (sort t)
end.

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].
Proof. simpl. reflexivity. Qed.
```

What Sedgewick/Wayne and Cormen/Leiserson/Rivest don't acknowlege is that the arrays-and-swaps model of sorting is not the only one in the world. We are writing *functional programs*, where our sequences are (typically) represented as linked lists, and where we do *not* destructively splice elements into those lists. Instead, we build new lists that (sometimes) share structure with the old ones.

So, for example:

```
Eval compute in insert 7 [1; 3; 4; 8; 12; 14; 18].
```

The tail of this list, 12::14::18::nil, is not disturbed or rebuilt by the insert algorithm. The nodes 1::3::4::7::_ are new, constructed by insert. The first three nodes of the old list, 1::3::4::_ will likely be garbage-collected, if no other data structure is still pointing at them. Thus, in this typical case,

- Time cost = 4X
- Space cost = (4-3)Y = Y

where X and Y are constants, independent of the length of the tail. The value Y is the number of bytes in one list node: 2 to 4 words, depending on how the implementation handles constructor-tags. We write (4-3) to indicate that four list nodes are constructed, while three list nodes become eligible for garbage collection.

We will not *prove* such things about the time and space cost, but they are *true* anyway, and we should keep them in consideration.

4.4 Specification of Correctness

A sorting algorithm must rearrange the elements into a list that is totally ordered.

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

Is this really the right definition of what it means for a list to be sorted? One might have thought that it should go more like this:

```
Definition sorted' (al: list nat) :=
  ∀ i j, i < j < length al → nth i al 0 ≤ nth j al 0.
```

This is a reasonable definition too. It should be equivalent. Later on, we'll prove that the two definitions really are equivalent. For now, let's use the first one to define what it means to be a correct sorting algorthm.

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

The result $(f\ al)$ should not only be a **sorted** sequence, but it should be some rearrangement (Permutation) of the input sequence.

4.5 Proof of Correctness

Exercise: 3 stars (insert_perm) Prove the following auxiliary lemma, insert_perm, which will be useful for proving sort_perm below. Your proof will be by induction, but you'll need some of the permutation facts from the library, so first remind yourself by doing Search.

```
Search Permutation.

Lemma insert_perm: ∀ x l, Permutation (x::l) (insert x l).
Proof.
  Admitted.
```

□

Exercise: 3 stars (sort_perm) Now prove that sort is a permutation.

```
Theorem sort_perm: ∀ l, Permutation l (sort l).
Proof.
  Admitted.
```

□

Exercise: 4 stars (insert_sorted) This one is a bit tricky. However, there just a single induction right at the beginning, and you do *not* need to use insert_perm or sort_perm.

```
Lemma insert_sorted:
  ∀ a l, sorted l → sorted (insert a l).
Proof.
  Admitted.
```

□

Exercise: 2 stars (sort_sorted) This one is easy.

```
Theorem sort_sorted: ∀ l, sorted (sort l).
Proof.
```

Admitted.

☐

Now we wrap it all up.

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

4.6 Making Sure the Specification is Right

It's really important to get the *specification* right. You can prove that your program satisfies its specification (and Coq will check that proof for you), but you can't prove that you have the right specification. Therefore, we take the trouble to write two different specifications of sortedness (**sorted** and sorted'), and prove that they mean the same thing. This increases our confidence that we have the right specification, though of course it doesn't *prove* that we do.

Exercise: 4 stars, optional (sorted_sorted') Lemma sorted_sorted': ∀ *al*, **sorted** *al* → sorted' *al*.

Hint: Instead of doing induction on the list *al*, do induction on the *sortedness* of *al*. This proof is a bit tricky, so you may have to think about how to approach it, and try out one or two different ideas.

Admitted.

☐

Exercise: 3 stars, optional (sorted'_sorted) Lemma sorted'_sorted: ∀ *al*, sorted' *al* → **sorted** *al*.

Here, you can't do induction on the sorted'-ness of the list, because sorted' is not an inductive predicate.

Proof.

Admitted.

☐

4.7 Proving Correctness from the Alternate Spec

Depending on how you write the specification of a program, it can be *much* harder or easier to prove correctness. We saw that the predicates **sorted** and sorted' are equivalent; but it is really difficult to prove correctness of insertion sort directly from sorted'.

Try it yourself, if you dare! I managed it, but my proof is quite long and complicated. I found that I needed all these facts:

- insert_perm, sort_perm
- Forall_perm, Permutation_length
- *Permutation_sym*, *Permutation_trans*
- a new lemma Forall_nth, stated below.

Maybe you will find a better way that's not so complicated.

DO NOT USE sorted_sorted', sorted'_sorted, insert_sorted, or sort_sorted in these proofs!

Exercise: 3 stars, optional (Forall_nth) Lemma Forall_nth:
 ∀ {*A*: Type} (*P*: *A* → Prop) *d* (*al*: **list** *A*),
  Forall *P* *al* ↔ (∀ *i*, *i* < length *al* → *P* (nth *i* *al* *d*)).
Proof.
 Admitted.
 □

Exercise: 4 stars, optional (insert_sorted') Lemma insert_sorted':
 ∀ *a* *l*, sorted' *l* → sorted' (insert *a* *l*).
 Admitted.
 □

Exercise: 4 stars, optional (insert_sorted') Theorem sort_sorted': ∀ *l*, sorted' (sort *l*).
 Admitted.
 □

4.7.1 The Moral of This Story

The proofs of insert_sorted and sort_sorted were easy; the proofs of insert_sorted' and sort_sorted' were difficult; and yet **sorted** *al* ↔ sorted' *al*. *Different formulations of the functional specification can lead to great differences in the difficulty of the correctness proofs.*

Suppose someone required you to prove sort_sorted', and never mentioned the **sorted** predicate to you. Instead of proving sort_sorted' directly, it would be much easier to design a new predicate (**sorted**), and then prove sort_sorted and sorted_sorted'.

Date

Chapter 5

Library VFA.Multiset

5.1 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 Coq.Strings.String.
From VFA Require Import Perm.
From VFA 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,
    union a (union b c) = union (union a b) c.
Proof.
  intros.
  extensionality x.
   Admitted.
```
□

Exercise: 1 star (union_comm)

```
Lemma union_comm: ∀ a b : multiset,
    union a b = union b a.
Proof.
   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].
Proof. simpl. reflexivity. Qed.

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).
Qed.
```

5.2 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.
```

Admitted.

☐

Exercise: 3 stars (sort_contents) Now prove that sort preserves contents.

```
Theorem sort_contents: ∀ l, contents l = contents (sort l).
```

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. `Definition manual_grade_for_permutations_vs_multiset : option (prod nat string) := None.`

☐

5.3 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.
    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).
    Admitted.
```

□

Exercise: 2 stars (contents_perm_aux) Lemma contents_perm_aux:

```
 ∀ v b, empty = union (singleton v) b → False.
Proof.
    Admitted.
```

□

Exercise: 2 stars (contents_in) Lemma contents_in:

∀ (a: value) (bl: list value) , contents bl a > 0 → In a bl.

Proof.

Admitted.

☐

Exercise: 2 stars (in_perm_delete) Lemma in_perm_delete:

∀ a bl,

In a bl → Permutation (a :: list_delete bl a) bl.

Proof.

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.

Admitted.

☐

5.4 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.
```

Chapter 6

Library VFA.Selection

6.1 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

6.2 The Selection-Sort Program

```
Require Export Coq.Lists.List.
From VFA 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.

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
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).
```

6.3 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 *.
   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`.

```
   Admitted.

Theorem selection_sort_perm:
  ∀ l, Permutation l (selection_sort l).
Proof.
   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.
   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 *.
   admit.
bdestruct (x <=? a).
×
destruct (select x al) eqn:?H.
   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.
   Admitted.
```

```
Theorem selection_sort_sorted: ∀ al, sorted (selection_sort al).
Proof.
intros.
unfold selection_sort.
   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.
```

6.4 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.

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.
```

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.

Admitted.

□

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

Date

Chapter 7

Library VFA.SearchTree

7.1 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 $a<b \leftrightarrow \tilde{}(b<a)$ – 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 Coq.Strings.String.
From VFA Require Import Perm.
Require Import FunctionalExtensionality.
```

7.2 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.

```
From VFA Require Import Maps.
```

7.3 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.
```

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.
```

7.4 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.
```

7.5 Search Tree Examples

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

7.6 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 $k' \neq k$, 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. Check t_update_neq. Check t_update_shadow. Check t_update_same.
Check t_update_permute. Check t_apply_empty.
```

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.

7.7 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.

7.8 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) Definition example_map ($v2$ $v4$ $v5$: V) : total_map V

. *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.
```

Admitted.

☐

You can ignore this lemma, unless it fails. Lemma check_too_clever: ∀ *v2 v4 v5*: *V*, **True**.

```
Proof.
intros.
evar (m: total_map V).
assert (Abs (example_tree v2 v4 v5) m).
repeat constructor.
(change m with (example_map v2 v4 v5) in H || auto);
```

fail "Did you use copy-and-paste, from your check_example_map proof, into your example_map definition? If so, very clever. Please try it again with an example_map definition that you make up from first principles. Or, to skip that, uncomment the (* auto; *) above.".
Qed.

```
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.
```

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.
```

Admitted.

□

7.9 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.

```
Abort.
Definition manual_grade_for_elements_relate_informal : option (prod nat string) := None.
```

□

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*.

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.

7.10 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).
```

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.
  Admitted.
```
□

7.10.1 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).
Proof.
intros.
induction al as [ | [k v]].
right; intros [w H]; inv H.
destruct IHal as [[w H] | H].
left; ∃ w; right; auto.
bdestruct (k =? i).
subst k.
left; eauto.
∃ v; left; auto.
right. intros [w H1].
destruct H1. inv H1. omega.
apply H; eauto.
Qed.
```

```
Lemma list2map_app_left:
  ∀ (al bl: list (key×V)) (i: key) v,
     In (i,v) al → list2map (al++bl) i = list2map al i.
Proof.
intros.
revert H; induction al as [| [j w] al]; intro; simpl; auto.
inv H.
destruct H. inv H.
unfold t_update.
bdestruct (i=?i); [ | omega].
auto.
```

```
unfold t_update.
bdestruct (j=?i); auto.
Qed.

Lemma list2map_app_right:
  ∀ (al bl: list (key×V)) (i: key),
      ~(∃ v, In (i,v) al) → list2map (al++bl) i = list2map bl i.
Proof.
intros.
revert H; induction al as [| [j w] al]; intro; simpl; auto.
unfold t_update.
bdestruct (j=?i).
subst j.
contradiction H.
∃ w; left; auto.
apply IHal.
contradict H.
destruct H as [u ?].
∃ u; right; auto.
Qed.

Lemma list2map_not_in_default:
 ∀ (al: list (key×V)) (i: key),
    ~(∃ v, In (i,v) al) → list2map al i = default.
Proof.
intros.
induction al as [| [j w] al].
reflexivity.
simpl.
unfold t_update.
bdestruct (j=?i).
subst.
contradiction H.
∃ w; left; auto.
apply IHal.
intros [v ?].
apply H. ∃ v; right; auto.
Qed.
```

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.
×
inv H0.
reflexivity.
×
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.
  Admitted.
```

□

7.11 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. Admitted.
```

□

Exercise: 3 stars (insert_SearchTree)

```
Theorem insert_SearchTree:
 ∀ k v t,
  SearchTree t → SearchTree (insert k v t).
Proof.
clear default. Admitted.
```

□

7.12 We Got Lucky

Recall the statement of lookup_relate:

```
Check lookup_relate.
```

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.
```

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.
```

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).
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.

7.13 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.
  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:
$\forall$ *t*, $\exists$ *cts* , **Abs** *t* *cts*.

```
Proof.
    Admitted.
    □
```

7.14 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.
```

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.

Admitted.

☐

7.14.1 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.

Chapter 8

Library VFA.ADT

8.1 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 k empty = default.
 Axiom gss: ∀ k v t,
      get k (set k v t) = v.
 Axiom gso: ∀ j k v t,
     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`.

```
From VFA 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)).
```

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.

```
From VFA 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.
  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.
```

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.

8.2 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: $A \rightarrow$ 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 ($\exists$ x, *proof*) is written as $\{x \mid P\ x\}$.

```
Check exist. Check proj1_sig. Check proj2_sig.
```

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].
   simpl.
   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.
  Admitted.
  □ End TreeTable2.
```

(End of the brief excursion into dependent types.)

8.3 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).
Check (insert_SearchTree V).
```

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).
```

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). Check (lookup_relate' V default). Check (insert_relate' V default).
```

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.

End ADT_SUMMARY.
```

8.4 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.
Proof.
induction i; destruct j,al; simpl; intros; auto; try omega.
apply IHi. omega.
Qed.
```

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 :=

 .

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

Lemma create_relate : True. Admitted.

Lemma cons_relate : True. Admitted.

Lemma nth_relate : True. 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.
   Admitted.

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

Lemma fibish_correct: ∀ n, fibish n = fib n.
Proof. 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: fib: fibish:

□

Chapter 9

Library VFA.Extract

9.1 Extract: Running Coq programs in ML

```
Require Extraction.
```

Coq's `Extraction` feature allows you to write a functional program inside Coq; (presumably) use Coq's logic to prove some correctness properties about it; then print it out as an ML (or Haskell) program that you can compile with your optimizing ML (or Haskell) compiler.

The `Extraction` chapter of *Logical Foundations* gave a simple example of Coq's program extraction features. In this chapter, we'll take a deeper look.

```
Set Warnings "-extraction-inside-module". From VFA Require Import Perm.

Module SORT1.
Fixpoint insert (i:nat) (l: list nat) :=
  match l with
  | nil ⇒ i::nil
  | h::t ⇒ if i <=? h then i::h::t else h :: insert i t
 end.

Fixpoint sort (l: list nat) : list nat :=
  match l with
  | nil ⇒ nil
  | h::t ⇒ insert h (sort t)
end.
```

The `Extraction` command prints out a function as Ocaml code.

```
Require Coq.extraction.Extraction.
Extraction sort.
```

You can see the translation of "sort" from Coq to Ocaml, in the "Messages" window of your IDE. Examine it there, and notice the similarities and differences.

However, we really want the whole program, including the insert function. We get that as follows:

`Recursive Extraction` *sort*.

The first thing you see there is a redefinition of the **bool** type. But Ocaml already has a **bool** type whose inductive structure is isomorphic. We want our extracted functions to be compatible with, callable by, ordinary Ocaml code. So we want to use Ocaml's standard notation for the inductive definition, **bool**. The following directive accomplishes that:

`Extract Inductive` **bool** ⇒ "bool" ["true" "false"].
`Extract Inductive` **list** ⇒ "list" ["[]" "(::)"].
`Recursive Extraction` *sort*.

`End` SORT1.

This is better. But the program still uses a unary representation of natural numbers: the number 7 is really (S (S (S (S (S (S (S O))))))). which in Ocaml will be a data structure that's seven pointers deep. The leb function takes time proportional to the difference in value between n and m, which is terrible. We'd like natural numbers to be represented as Ocaml *int*. Unfortunately, there are only a finite number of *int* values in Ocaml (2ˆ31, or 2ˆ63, depending on your implementation); so there are things you could prove about some programs, in Coq, that wouldn't be true in Ocaml.

There are two solutions to this problem:

- Instead of using **nat**, use a more efficient constructive type, such as **Z**.
- Instead of using **nat**, use an *abstract type*, and instantiate it with Ocaml integers.

The first alternative uses Coq's **Z** type, an inductive type with constructors xI xH etc. **Z** represents 7 as Zpos (xI (xI xH)), that is, +(1+2*(1+2* 1)). A number n is represented as a data structure of size log(n), and the operations (plus, less-than) also take about log(n) each.

Z's log-time per operation is much better than linear time; but in Ocaml we are used to having constant-time operations. Thus, here we will explore the second alternative: program with abstract types, then use an extraction directive to get efficiency.

`Require Import` ZArith.
`Open Scope` *Z_scope*.

We will be using `Parameter` and `Axiom` in Coq. You already saw these keywords, in a `Module Type`, in the ADT chapter. There, they describe interface components that must be instantiated by any `Module` that satisfies the type. Here, we will use this feature in a different (and more dangerous) way: To axiomatize a mathematical theory without actually constructing it. The reason that's dangerous is that if your axioms are inconsistent, then you can prove **False**, or in fact, you can prove *anything*, so all your proofs are worthless. So we must take care!

Here, we will axiomatize a *very weak* mathematical theory: We claim that there exists some type *int* with a function *ltb*, so that *int* injects into **Z**, and *ltb* corresponds to the $<$ relation on Z. That seems true enough (for example, take *int*=**Z**), but we're not *proving* it here.

Parameter *int* : Type. Extract *Inlined Constant int* ⇒ "int".

Parameter *ltb*: *int* → *int* → **bool**. Extract *Inlined Constant ltb* ⇒ "(<)".

Now, we need to axiomatize *ltb* so that we can reason about programs that use it. We need to take great care here: the axioms had better be consistent with Ocaml's behavior, otherwise our proofs will be meaningless.

One axiomatization of *ltb* is just that it's a total order, irreflexive and transitive. This would work just fine. But instead, I choose to claim that there's an injection from "int" into the mathematical integers, Coq's **Z** type. The reason to do this is then we get to use the omega tactic, and other Coq libraries about integer comparisons.

Parameter *int2Z*: *int* → **Z**.
Axiom *ltb_lt* : ∀ *n m* : *int*, *ltb n m* = true ↔ *int2Z n* < *int2Z m*.

Both of these axioms are sound. There does (abstractly) exist a function from "int" to **Z**, and that function *is* consistent with the *ltb_lt* axiom. But you should think about this until you are convinced.

Notice that we do not give extraction directives for *int2Z* or *ltb_lt*. That's because they will not appear in *programs*, only in proofs that are not meant to be extracted.

Now, here's a dangerous axiom:

Parameter *ocaml_plus* : *int* → *int* → *int*.

Extract *Inlined Constant ocaml_plus* ⇒ "(+)".

Axiom *ocaml_plus_plus*: ∀ *a b c*: *int*, *ocaml_plus a b* = *c* ↔ *int2Z a* + *int2Z b* = *int2Z c*.

The first two lines are OK: there really is a "+" function in Ocaml, and its type really is *int* → *int* → *int*.

But *ocaml_plus_plus* is unsound! From it, you could prove,

(*int2Z max_int* + *int2Z max_int*) = *int2Z* (*ocaml_plus max_int max_int*),

which is not true in Ocaml, because overflow wraps around, modulo 2^(*wordsize*-1).

So we won't axiomatize Ocaml addition.

9.2 Utilities for OCaml Integer Comparisons

Just like in Perm.v, but for *int* and **Z** instead of **nat**.

```
Lemma int_blt_reflect : ∀ x y, reflect (int2Z x < int2Z y) (ltb x y).
Proof.
  intros x y.
  apply iff_reflect. symmetry. apply ltb_lt.
Qed.
```

```
Lemma Z_eqb_reflect : ∀ x y, reflect (x=y) (Z.eqb x y).
Proof.
  intros x y.
  apply iff_reflect. symmetry. apply Z.eqb_eq.
```

```
Qed.

Lemma Z_ltb_reflect : ∀ x y, reflect (x<y) (Z.ltb x y).
Proof.
  intros x y.
  apply iff_reflect. symmetry. apply Z.ltb_lt.
Qed.

Hint Resolve int_blt_reflect Z_eqb_reflect Z_ltb_reflect : bdestruct.
```

9.3 SearchTrees, Extracted

Let us re-do binary search trees, but with Ocaml integers instead of Coq nats.

9.3.1 Maps, on Z Instead of nat

Our original proof with nats used *Maps.total_map* in its abstraction relation, but that won't work here because we need maps over **Z** rather than **nat**. So, we copy-paste-edit to make total_map over **Z**.

```
Require Import Coq.Logic.FunctionalExtensionality.

Module IntMaps.
Definition total_map (A:Type) := Z → A.
Definition t_empty {A:Type} (v : A) : total_map A := (fun _ ⇒ v).
Definition t_update {A:Type} (m : total_map A) (x : Z) (v : A) :=
  fun x' ⇒ if Z.eqb x x' then v else m x'.
Lemma t_update_eq : ∀ A (m: total_map A) x v, (t_update m x v) x = v.
Proof.
  intros. unfold t_update.
  bdestruct (x=?x); auto.
  omega.
Qed.

Theorem t_update_neq : ∀ (X:Type) v x1 x2 (m : total_map X),
  x1 ≠ x2 → (t_update m x1 v) x2 = m x2.
Proof.
  intros. unfold t_update.
  bdestruct (x1=?x2); auto.
  omega.
Qed.

Lemma t_update_shadow : ∀ A (m: total_map A) v1 v2 x,
    t_update (t_update m x v1) x v2 = t_update m x v2.
Proof.
  intros. unfold t_update.
```

```
  extensionality x'.
  bdestruct (x=?x'); auto.
Qed.

End IntMaps.

Import IntMaps.
```

9.3.2 Trees, on *int* Instead of nat

```
Module SearchTree2.
Section TREES.
Variable V : Type.
Variable default: V.

Definition key := int.

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 ltb x k then lookup x tl
                         else if ltb 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 ltb x y then T (insert x v a) y v' b
                        else if ltb 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.

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

```
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 (int2Z k) a b) (int2Z k) v).
```

```
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 (int2Z k).
Proof. Admitted.
```

☐

Exercise: 3 stars (insert_relate)

```
Theorem insert_relate:
 ∀ k v t cts,
    Abs t cts →
    Abs (insert k v t) (t_update cts (int2Z k) v).
Proof. Admitted.
```

☐

Exercise: 1 star (unrealistically_strong_can_relate)

```
Lemma unrealistically_strong_can_relate:
 ∀ t, ∃ cts , Abs t cts.
Proof. Admitted.
```

☐

```
End TREES.
```

Now, run this command and examine the results in the "results" window of your IDE:

```
Recursive Extraction empty_tree insert lookup elements.
```

Next, we will extract it into an Ocaml source file, and measure its performance.

```
Extraction "searchtree.ml" empty_tree insert lookup elements.
```

Note: we've done the extraction *inside* the module, even though Coq warns against it, for a specific reason: We want to extract only the program, not the proofs.

```
End SearchTree2.
```

9.4 Performance Tests

Read the Ocaml program, test_searchtree.ml:

let test (f: int -> int) (n: int) = let rec build (j, t) = if j=0 then t else build(j-1, insert (f j) 1 t) in let t1 = build(n,empty_tree) in let rec g (j,count) = if j=0 then count else if lookup 0 (f j) t1 = 1 then g(j-1,count+1) else g(j-1,count) in let start = Sys.time() in let answer = g(n,0) in (answer, Sys.time() -. start)

let print_test name (f: int -> int) n = let (answer, time) = test f n in (print_string "Insert and lookup "; print_int n; print_string " "; print_string name; print_string " integers in "; print_float time; print_endline " seconds.")

let test_random n = print_test "random" (fun _-> Random.int n) n let test_consec n = print_test "consecutive" (fun i -> n-i) n

let run_tests() = (test_random 1000000; test_random 20000; test_consec 20000)

let _= run_tests () »

You can run this inside the ocaml top level by:

use "test_searchtree.ml";; run_tests();;

On my machine, in the byte-code interpreter this prints,

Insert and lookup 1000000 random integers in 1.076 seconds. Insert and lookup 20000 random integers in 0.015 seconds. Insert and lookup 20000 consecutive integers in 5.054 seconds.

You can compile and run this with the ocaml native-code compiler by:

ocamlopt searchtree.mli searchtree.ml -open Searchtree test_searchtree.ml -o test_searchtree ./test_searchtree

On my machine this prints,

Insert and lookup 1000000 random integers in 0.468 seconds. Insert and lookup 20000 random integers in 0. seconds. Insert and lookup 20000 consecutive integers in 0.374 seconds.

9.5 Unbalanced Binary Search Trees

Why is the performance of the algorithm so much worse when the keys are all inserted consecutively? To examine this, let's compute with some searchtrees inside Coq. We cannot do this with the search trees defined thus far in this file, because they use a key-comparison function *ltb* that is abstract and uninstantiated (only during Extraction to Ocaml does *ltb* get instantiated).

So instead, we'll use the SearchTree module, where everything runs inside Coq.

```
From VFA Require SearchTree.
Module EXPERIMENTS.
Open Scope nat_scope.
Definition empty_tree := SearchTree.empty_tree nat.
Definition insert := SearchTree.insert nat.
Definition lookup := SearchTree.lookup nat 0.
Definition E := SearchTree.E nat.
Definition T := SearchTree.T nat.
```

Goal insert 5 1 (insert 4 1 (insert 3 1 (insert 2 1 (insert 1 1 (insert 0 1 empty_tree))))) $\neq$ E.
simpl. fold E; repeat fold T.

Look here! The tree is completely unbalanced. Looking up 5 will take linear time. That's why the runtime on consecutive integers is so bad.

Abort.

9.6 Balanced Binary Search Trees

To achieve robust performance (that stays N log N for a sequence of N operations, and does not degenerate to N*N), we must keep the search trees balanced. The next chapter, Redblack, implements that idea.

End EXPERIMENTS.

Chapter 10

Library VFA.Redblack

10.1 Redblack: Implementation and Proof of Red-Black Trees

10.2 Required Reading

(1) General background on red-black trees,

- Section 3.3 of *Algorithms, Fourth Edition*, by Sedgewick and Wayne, Addison Wesley 2011; or
- Chapter 13 of *Introduction to Algorithms, 3rd Edition*, by Cormen, Leiserson, and Rivest, MIT Press 2009
- or Wikipedia.

(2) an explanation of the particular implementation we use here. Red-Black Trees in a Functional Setting, by Chris Okasaki. *Journal of Functional Programming*, 9(4):471-477, July 1999. http://www.westpoint.edu/eecs/SiteAssets/SitePages/Faculty20Publication20Documents/Oka

(3) Optional reading: Efficient Verified Red-Black Trees, by Andrew W. Appel, September 2011. http://www.cs.princeton.edu/~appel/papers/redblack.pdf

Red-black trees are a form of binary search tree (BST), but with *balance*. Recall that the *depth* of a node in a tree is the distance from the root to that node. The *height* of a tree is the depth of the deepest node. The insert or lookup function of the BST algorithm (Chapter **SearchTree**) takes time proportional to the depth of the node that is found (or inserted). To make these functions run fast, we want trees where the worst-case depth (or the average depth) is as small as possible.

In a perfectly balanced tree of N nodes, every node has depth less than or or equal to log N, using logarithms base 2. In an approximately balanced tree, every node has depth less than or equal to 2 log N. That's good enough to make insert and lookup run in time proportional to log N.

The trick is to mark the nodes Red and Black, and by these marks to know when to locally rebalance the tree. For more explanation and pictures, see the Required Reading above.

We will use the same framework as in Extract.v: keys are Ocaml integers. We don't repeat the `Extract` commands, because they are imported implicitly from Extract.v

```
From VFA Require Import Perm.
From VFA Require Import Extract.
Require Import Coq.Lists.List.
Export ListNotations.
Require Import Coq.Logic.FunctionalExtensionality.
Require Import ZArith.
Open Scope Z_scope.

Definition key := int.

Inductive color := Red | Black.

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

 Inductive tree : Type :=
 | E : tree
 | T: color → tree → key → V → tree → tree.

 Definition empty_tree := E.
```

lookup is exactly as in our (unbalanced) search-tree algorithm in Extract.v, except that the T constructor carries a color component, which we can ignore here.

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

The balance function is copied directly from Okasaki's paper. Now, the nice thing about machine-checked proof in Coq is that you can prove this correct without actually understanding it! So, do read Okasaki's paper, but don't worry too much about the details of this balance function.

In contrast, Sedgewick has proposed *left-leaning red-black trees*, which have a simpler balance function (but a more complicated invariant). He does this in order to make the proof of correctness easier: there are fewer cases in the balance function, and therefore fewer cases in the case-analysis of the proof of correctness of balance. But as you will see, our proofs about balance will have automated case analyses, so we don't care how many cases there are!

```
Definition balance rb t1 k vk t2 :=
 match rb with Red ⇒ T Red t1 k vk t2
 | _ ⇒
 match t1 with
 | T Red (T Red a x vx b) y vy c ⇒
      T Red (T Black a x vx b) y vy (T Black c k vk t2)
 | T Red a x vx (T Red b y vy c) ⇒
      T Red (T Black a x vx b) y vy (T Black c k vk t2)
 | a ⇒ match t2 with
            | T Red (T Red b y vy c) z vz d ⇒
                 T Red (T Black t1 k vk b) y vy (T Black c z vz d)
            | T Red b y vy (T Red c z vz d) ⇒
                 T Red (T Black t1 k vk b) y vy (T Black c z vz d)
            | _ ⇒ T Black t1 k vk t2
            end
  end
 end.

Definition makeBlack t :=
  match t with
  | E ⇒ E
  | T _ a x vx b ⇒ T Black a x vx b
  end.

Fixpoint ins x vx s :=
 match s with
 | E ⇒ T Red E x vx E
 | T c a y vy b ⇒ if ltb x y then balance c (ins x vx a) y vy b
                        else if ltb y x then balance c a y vy (ins x vx b)
                        else T c a x vx b
 end.

Definition insert x vx s := makeBlack (ins x vx s).
```

Now that the program has been defined, it's time to prove its properties. A red-black tree has two kinds of properties:

- **SearchTree**: the keys in each left subtree are all less than the node's key, and the keys in each right subtree are greater
- *Balanced*: there is the same number of black nodes on any path from the root to each leaf; and there are never two red nodes in a row.

First, we'll treat the **SearchTree** property.

10.3 Proof Automation for Case-Analysis Proofs.

```
Lemma T_neq_E:
  ∀ c l k v r, T c l k v r ≠ E.
Proof.
intros. intro Hx. inversion Hx.
Qed.
```

Several of the proofs for red-black trees require a big case analysis over all the clauses of the balance function. These proofs are very tedious to do "by hand," but are easy to automate.

```
Lemma ins_not_E: ∀ x vx s, ins x vx s ≠ E.
Proof.
intros. destruct s; simpl.
apply T_neq_E.
remember (ins x vx s1) as a1.
unfold balance.
```

Here we go! Let's just "destruct" on the topmost case. Right, here it's *ltb* x k. We can use `destruct` instead of *bdestruct* because we don't need to remember whether $x<k$ or $x\geq k$.

```
destruct (ltb x k).
destruct c.
apply T_neq_E.
destruct a1.
destruct s2.
intro Hx; inversion Hx.
```

How long will this go on? A long time! But it will terminate. Just keep typing. Better yet, let's automate. The following tactic applies whenever the current goal looks like, `match ?c with Red ⇒ _ | Black ⇒ _ end ≠ _` , and what it does in that case is, `destruct c`

```
match goal with
| ⊢ match ?c with Red ⇒ _ | Black ⇒ _ end ≠ _⇒ destruct c
end.
```

The following tactic applies whenever the current goal looks like,
`match ?s with E ⇒ _ | T _ _ _ _ _ ⇒ _ end ≠ _`,
and what it does in that case is, `destruct s`

```
match goal with
   | ⊢ match ?s with E ⇒ _ | T _ _ _ _ _ ⇒ _ end ≠ _⇒destruct s
end.
```

Let's apply that tactic again, and then try it on the subgoals, recursively. Recall that the `repeat` tactical keeps trying the same tactic on subgoals.

```
repeat match goal with
```

```
    | ⊢ match ?s with E ⇒ _ | T _ _ _ _ _ ⇒ _ end ≠ _⇒destruct s
end.
match goal with
  | ⊢ T _ _ _ _ _ ≠ E ⇒ apply T_neq_E
end.
```

Let's start the proof all over again.

```
Abort.

Lemma ins_not_E: ∀ x vx s, ins x vx s ≠ E.
Proof.
intros. destruct s; simpl.
apply T_neq_E.
remember (ins x vx s1) as a1.
unfold balance.
```

This is the beginning of the big case analysis. This time, let's combine several tactics together:

```
repeat match goal with
  | ⊢ (if ?x then _ else _) ≠ _ ⇒ destruct x
  | ⊢ match ?c with Red ⇒ _ | Black ⇒ _ end ≠ _⇒ destruct c
  | ⊢ match ?s with E ⇒ _ | T _ _ _ _ _ ⇒ _ end ≠ _⇒destruct s
end.
```

What we have left is 117 cases, every one of which can be proved the same way:

```
apply T_neq_E.
apply T_neq_E.
apply T_neq_E.
apply T_neq_E.
apply T_neq_E.
apply T_neq_E.
apply T_neq_E.
apply T_neq_E.
apply T_neq_E.
apply T_neq_E.
Abort.

Lemma ins_not_E: ∀ x vx s, ins x vx s ≠ E.
Proof.
intros. destruct s; simpl.
apply T_neq_E.
remember (ins x vx s1) as a1.
unfold balance.
```

This is the beginning of the big case analysis. This time, we add one more clause to the `match goal` command:

```
repeat match goal with
  | ⊢ (if ?x then _ else _) ≠ _ ⇒ destruct x
  | ⊢ match ?c with Red ⇒ _ | Black ⇒ _ end ≠ _⇒ destruct c
  | ⊢ match ?s with E ⇒ _ | T _ _ _ _ _ _ ⇒ _ end ≠ _⇒destruct s
  | ⊢ T _ _ _ _ _ _ ≠ E ⇒ apply T_neq_E
 end.
Qed.
```

10.4 The SearchTree Property

The SearchTree property for red-black trees is exactly the same as for ordinary searchtrees (we just ignore the color c of each node).

```
Inductive SearchTree' : Z → tree → Z → Prop :=
| ST_E : ∀ lo hi, lo ≤ hi → SearchTree' lo E hi
| ST_T: ∀ lo c l k v r hi,
    SearchTree' lo l (int2Z k) →
    SearchTree' (int2Z k + 1) r hi →
    SearchTree' lo (T c l k v r) hi.
```

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

Now we prove that if t is a SearchTree, then the rebalanced version of t is also a SearchTree. Lemma balance_SearchTree:

```
 ∀ c s1 k kv s2 lo hi,
   SearchTree' lo s1 (int2Z k) →
   SearchTree' (int2Z k + 1) s2 hi →
   SearchTree' lo (balance c s1 k kv s2) hi.
Proof.
intros.
unfold balance.
```

Use proof automation for this case analysis.

```
repeat match goal with
  | ⊢ SearchTree' _ (match ?c with Red ⇒ _ | Black ⇒ _ end) _ ⇒ destruct c
  | ⊢ SearchTree' _ (match ?s with E ⇒ _ | T _ _ _ _ _ _ ⇒ _ end) _ ⇒ destruct s
  end.
```

58 cases to consider!

```
× constructor; auto.
× constructor; auto.
× constructor; auto.
× constructor; auto.
  constructor; auto. constructor; auto.
```

```
inv H. inv H0. inv H8. inv H9.
auto.
constructor; auto.
inv H. inv H0. inv H8. inv H9. auto.
inv H. inv H0. inv H8. inv H9. auto.
```

There's a pattern here. Whenever we have a hypothesis above the line that looks like,

- H: SearchTree' _E _
- H: SearchTree' _(T _) _

we should invert it. Let's build that idea into our proof automation.

```
Abort.
```

```
Lemma balance_SearchTree:
 ∀ c s1 k kv s2 lo hi,
   SearchTree' lo s1 (int2Z k) →
   SearchTree' (int2Z k + 1) s2 hi →
   SearchTree' lo (balance c s1 k kv s2) hi.
Proof.
intros.
unfold balance.
```

Use proof automation for this case analysis.

```
repeat match goal with
  | ⊢ SearchTree' _ (match ?c with Red ⇒ _ | Black ⇒ _ end) _ ⇒
              destruct c
  | ⊢ SearchTree' _ (match ?s with E ⇒ _ | T _ _ _ _ _ ⇒ _ end) _ ⇒
              destruct s
  | H: SearchTree' _ E _ ⊢ _ ⇒ inv H
  | H: SearchTree' _ (T _ _ _ _ _) _ ⊢ _ ⇒ inv H
  end.
```

58 cases to consider!

```
× constructor; auto.
× constructor; auto. constructor; auto. constructor; auto.
× constructor; auto. constructor; auto. constructor; auto. constructor; auto.
constructor; auto.
× constructor; auto. constructor; auto. constructor; auto. constructor; auto.
constructor; auto.
× constructor; auto. constructor; auto. constructor; auto. constructor; auto.
constructor; auto.
```

Do we see a pattern here? We can add that to our automation!

```
Abort.
```

```
Lemma balance_SearchTree:
 ∀ c s1 k kv s2 lo hi,
    SearchTree' lo s1 (int2Z k) →
    SearchTree' (int2Z k + 1) s2 hi →
    SearchTree' lo (balance c s1 k kv s2) hi.
Proof.
intros.
unfold balance.
```

Use proof automation for this case analysis.

```
repeat match goal with
  | ⊢ SearchTree' _ (match ?c with Red ⇒ _ | Black ⇒ _ end) _ ⇒
              destruct c
  | ⊢ SearchTree' _ (match ?s with E ⇒ _ | T _ _ _ _ _ ⇒ _ end) _ ⇒
              destruct s
  | H: SearchTree' _ E _ ⊢ _ ⇒ inv H
  | H: SearchTree' _ (T _ _ _ _ _) _ ⊢ _ ⇒ inv H
  end;
 repeat (constructor; auto).
Qed.
```

Exercise: 2 stars (ins_SearchTree) This one is pretty easy, even without proof automation. Copy-paste your proof of insert_SearchTree from Extract.v. You will need to apply balance_SearchTree in two places. Lemma ins_SearchTree:

```
  ∀ x vx s lo hi,
                    lo ≤ int2Z x →
                    int2Z x < hi →
                    SearchTree' lo s hi →
                    SearchTree' lo (ins x vx s) hi.
Proof.
```

Admitted.

☐

Exercise: 2 stars (valid) Lemma empty_tree_SearchTree: **SearchTree** empty_tree.

Admitted.

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

```
Lemma expand_range_SearchTree':
  ∀ s lo hi,
```

```
    SearchTree' lo s hi →
    ∀ lo' hi',
    lo' ≤ lo → hi ≤ hi' →
    SearchTree' lo' s hi'.
Proof.
induction 1; intros.
constructor.
omega.
constructor.
apply IHSearchTree'1; omega.
apply IHSearchTree'2; omega.
Qed.
```

```
Lemma insert_SearchTree: ∀ x vx s,
     SearchTree s → SearchTree (insert x vx s).
    Admitted.
```

□

```
Import IntMaps.
```

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

```
Inductive Abs: tree → total_map V → Prop :=
| Abs_E: Abs E (t_empty default)
| Abs_T: ∀ a b c l k vk r,
        Abs l a →
        Abs r b →
        Abs (T c l k vk r) (t_update (combine (int2Z k) a b) (int2Z k) vk).
```

```
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 (int2Z k).
Proof. Admitted.
```

□

```
Lemma Abs_helper:
  ∀ m' t m, Abs t m' → m' = m → Abs t m.
Proof.
   intros. subst. auto.
Qed.
```

```
Ltac contents_equivalent_prover :=
```

```
extensionality x; unfold t_update, combine, t_empty;
repeat match goal with
 | ⊢ context [if ?A then _ else _] ⇒ bdestruct A
end;
auto;
omega.
```

Exercise: 4 stars (balance_relate) You will need proof automation for this one. Study the methods used in ins_not_E and balance_SearchTree, and try them here. Add one clause at a time to your `match goal`.

```
Theorem balance_relate:
  ∀ c l k vk r m,
    SearchTree (T c l k vk r) →
    Abs (T c l k vk r) m →
    Abs (balance c l k vk r) m.
Proof.
intros.
inv H.
unfold balance.
repeat match goal with
 | H: Abs E _ ⊢ _ ⇒ inv H
end.
```

Add these clauses, one at a time, to your `repeat match goal` tactic, and try it out:

- 1. Whenever a clause *H*: Abs E _ is above the line, invert it by *inv H*. Take note: with just this one clause, how many subgoals remain?
- 2. Whenever Abs (T _ _ _ _ _) _ is above the line, invert it. Take note: with just these two clause, how many subgoals remain?
- 3. Whenever **SearchTree'** _ E _ is above the line, invert it. Take note after this step and each step: how many subgoals remain?
- 4. Same for **SearchTree'** _ (T _ _ _ _ _) _.
- 5. When Abs `match` *c* `with` Red ⇒ _ | Black ⇒ _ `end` _ is below the line, `destruct` *c*.
- 6. When Abs `match` *s* `with` E ⇒ _ | T ... ⇒ _ `end` _ is below the line, `destruct` *s*.
- 7. Whenever Abs (T _ _ _ _ _) _ is below the line, prove it by `apply` Abs_T. This won't always work; Sometimes the "cts" in the proof goal does not exactly match the form of the "cts" required by the Abs_T constructor. But it's all right if a clause fails; in that case, the `match goal` will just try the next clause. Take note, as usual: how many clauses remain?

- 8. Whenever Abs E _ is below the line, solve it by `apply Abs_E`.
- 9. Whenever the current proof goal matches a hypothesis above the line, just use it. That is, just add this clause: | |- _=> assumption
- 10. At this point, if all has gone well, you should have exactly 21 subgoals. Each one should be of the form, Abs (T ...) (t_update...) What you want to do is replace (t_update...) with a different "contents" that matches the form required by the Abs_T constructor. In the first proof goal, do this: `eapply Abs_helper`. Notice that you have two subgoals. The first subgoal you can prove by: apply Abs_T. apply Abs_T. apply Abs_E. apply Abs_E. apply Abs_T. eassumption. eassumption. Step through that, one at a time, to see what it's doing. Now, undo those 7 commands, and do this instead: repeat econstructor; eassumption. That solves the subgoal in exactly the same way. Now, wrap this all up, by adding this clause to your `match goal`: | |- _=> eapply Abs_helper; `repeat econstructor`; *eassumption* |
- 11. You should still have exactly 21 subgoals, each one of the form, t_update... = t_update... . Notice above the line you have some assumptions of the form, *H*: **SearchTree'** *lo* _ *hi* . For this equality proof, we'll need to know that *lo* ≤ *hi*. So, add a clause at the end of your `match goal` to apply SearchTree'_le in any such assumption, when below the line the proof goal is an equality _ = _ .
- 12. Still exactly 21 subgoals. In the first subgoal, try: *contents_equivalent_prover*. That should solve the goal. Look above, at `Ltac` *contents_equivalent_prover*, to see how it works. Now, add a clause to `match goal` that does this for all the subgoals.
- Qed!

Admitted.

Extend this list, so that the nth entry shows how many subgoals were remaining after you followed the nth instruction in the list above. Your list should be exactly 13 elements long; there was one subgoal *before* step 1, after all.

```
Definition how_many_subgoals_remaining :=
    [1; 1; 1; 1; 1; 2

  ].
```

□

Exercise: 3 stars (ins_relate) `Theorem ins_relate:`

```
∀ k v t cts,
    SearchTree t →
    Abs t cts →
    Abs (ins k v t) (t_update cts (int2Z k) v).
```

```
Proof. Admitted.
□

Lemma makeBlack_relate:
 ∀ t cts,
    Abs t cts →
    Abs (makeBlack t) cts.
Proof.
intros.
destruct t; simpl; auto.
inv H; constructor; auto.
Qed.

Theorem insert_relate:
 ∀ k v t cts,
    SearchTree t →
    Abs t cts →
    Abs (insert k v t) (t_update cts (int2Z k) v).
Proof.
intros.
unfold insert.
apply makeBlack_relate.
apply ins_relate; auto.
Qed.
```

OK, we're almost done! We have proved all these main theorems:

```
Check empty_tree_SearchTree.
Check empty_tree_relate.
Check lookup_relate.
Check insert_SearchTree.
Check insert_relate.
```

Together these imply that this implementation of red-black trees (1) preserves the representation invariant, and (2) respects the abstraction relation.

Exercise: 4 stars, optional (elements) Prove the correctness of the elements function. Because elements does not pay attention to colors, and does not rebalance the tree, then its proof should be a simple copy-paste from SearchTree.v, with only minor edits.

```
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.
```

```
Definition elements_property (t: tree) (cts: total_map V) : Prop :=
  ∀ k v,
    (In (k,v) (elements t) → cts (int2Z k) = v) ∧
    (cts (int2Z k) ≠ default →
     ∃ k', int2Z k = int2Z k' ∧ In (k', cts (int2Z k)) (elements t)).

Theorem elements_relate:
  ∀ t cts,
  SearchTree t →
  Abs t cts →
  elements_property t cts.
Proof.
   Admitted.
   □
```

10.5 Proving Efficiency

Red-black trees are supposed to be more efficient than ordinary search trees, because they stay balanced. In a perfectly balanced tree, any two leaves have exactly the same depth, or the difference in depth is at most 1. In an approximately balanced tree, no leaf is more than twice as deep as another leaf. Red-black trees are approximately balanced. Consequently, no node is more then 2logN deep, and the run time for insert or lookup is bounded by a constant times 2logN.

We can't prove anything *directly* about the run time, because we don't have a cost model for Coq functions. But we can prove that the trees stay approximately balanced; this tells us important information about their efficiency.

Exercise: 4 stars (is_redblack_properties) The relation **is_redblack** ensures that there are exactly n black nodes in every path from the root to a leaf, and that there are never two red nodes in a row.

```
Inductive is_redblack : tree → color → nat → Prop :=
| IsRB_leaf: ∀ c, is_redblack E c 0
| IsRB_r: ∀ tl k kv tr n,
          is_redblack tl Red n →
          is_redblack tr Red n →
          is_redblack (T Red tl k kv tr) Black n
| IsRB_b: ∀ c tl k kv tr n,
          is_redblack tl Black n →
          is_redblack tr Black n →
          is_redblack (T Black tl k kv tr) c (S n).

Lemma is_redblack_toblack:
  ∀ s n, is_redblack s Red n → is_redblack s Black n.
```

```
Proof.
  Admitted.
Lemma makeblack_fiddle:
  ∀ s n, is_redblack s Black n →
              ∃ n, is_redblack (makeBlack s) Red n.
Proof.
  Admitted.
```

nearly_redblack expresses, "the tree is a red-black tree, except that it's nonempty and it is permitted to have two red nodes in a row at the very root (only)."

```
Inductive nearly_redblack : tree → nat → Prop :=
| nrRB_r: ∀ tl k kv tr n,
          is_redblack tl Black n →
          is_redblack tr Black n →
          nearly_redblack (T Red tl k kv tr) n
| nrRB_b: ∀ tl k kv tr n,
          is_redblack tl Black n →
          is_redblack tr Black n →
          nearly_redblack (T Black tl k kv tr) (S n).
Lemma ins_is_redblack:
  ∀ x vx s n,
    (is_redblack s Black n → nearly_redblack (ins x vx s) n) ∧
    (is_redblack s Red n → is_redblack (ins x vx s) Black n).
Proof.
induction s; intro n; simpl; split; intros; inv H; repeat constructor; auto.
×
destruct (IHs1 n); clear IHs1.
destruct (IHs2 n); clear IHs2.
specialize (H0 H6).
specialize (H2 H7).
clear H H1.
unfold balance.
```

You will need proof automation, in a similar style to the proofs of ins_not_E and balance_relate.

```
  Admitted.
Lemma insert_is_redblack:
  ∀ x xv s n, is_redblack s Red n →
                    ∃ n', is_redblack (insert x xv s) Red n'.
Proof.
  Admitted.
  □
End TREES.
```

10.6 Extracting and Measuring Red-Black Trees

`Extraction` "redblack.ml" *empty_tree insert lookup elements*.

You can run this inside the ocaml top level by:

use "test_searchtree.ml";; run_tests();;

On my machine, in the byte-code interpreter this prints,

Insert and lookup 1000000 random integers in 0.889 seconds. Insert and lookup 20000 random integers in 0.016 seconds. Insert and lookup 20000 consecutive integers in 0.015 seconds.

You can compile and run this with the ocaml native-code compiler by:

ocamlopt redblack.mli redblack.ml -open Redblack test_searchtree.ml -o test_redblack ./test_redblack

On my machine this prints,

Insert and lookup 1000000 random integers in 0.436 seconds. Insert and lookup 20000 random integers in 0. seconds. Insert and lookup 20000 consecutive integers in 0. seconds.

10.7 Success!

The benchmark measurements above (and in Extract.v) demonstrate that:

- On random insertions, red-black trees are slightly faster than ordinary BSTs (red-black 0.436 seconds, vs ordinary 0.468 seconds)

- On consecutive insertions, red-black trees are *much* faster than ordinary BSTs (red-black 0. seconds, vs ordinary 0.374 seconds)

In particular, red-black trees are almost exactly as fast on the consecutive insertions (0.015 seconds) as on the random (0.016 seconds).

Chapter 11

Library VFA.Trie

11.1 Trie: Number Representations and Efficient Lookup Tables

11.2 LogN Penalties in Functional Programming

Purely functional algorithms sometimes suffer from an asymptotic slowdown of order logN compared to imperative algorithms. The reason is that imperative programs can do *indexed array update* in constant time, while functional programs cannot.

Let's take an example. Give an algorithm for detecting duplicate values in a sequence of N integers, each in the range 0..2N. As an imperative program, there's a very simple linear-time algorithm:

collisions=0; for (i=0; i<2N; i++) ai=0; for (j=0; j<N; j++) { i = input$j$; if (ai != 0) collisions++; ai=1; } return collisions;

In a functional program, we must replace $a[i]$=1 with the update of a finite map. If we use the inefficient maps in *Maps.v*, each lookup and update will take (worst-case) linear time, and the whole algorithm is quadratic time. If we use balanced binary search trees *Redblack.v*, each lookup and update will take (worst-case) logN time, and the whole algorithm takes NlogN. Comparing O(NlogN) to O(N), we see that there is a logN asymptotic penalty for using a functional implementation of finite maps. This penalty arises not only in this "duplicates" algorithm, but in any algorithm that relies on random access in arrays.

One way to avoid this problem is to use the imperative (array) features of a not-really-functional language such as ML. But that's not really a functional program! In particular, in *Verified Functional Algorithms* we prove program correct by relying on the *tractable proof theory* of purely functional programs; if we use nonfunctional features of ML, then this style of proof will not work. We'd have to use something like Hoare logic instead (see *Hoare.v* in volume 2 of *Software Foundations*), and that is not *nearly* as nice.

Another choice is to use a purely functional programming language designed for imperative programming: Haskell with the IO monad. The IO monad provides a pure-functional interface to efficient random-access arrays. This might be a reasonable approach, but we will not cover it here.

Here, we accept the logN penalty, and focus on making the "constant factors" small: that is, let us at least have efficient functional finite maps.

`Extract` showed one approach: use Ocaml integers. The advantage: constant-time greater-than comparison. The disadvantages: (1) Need to make sure you axiomatize them correctly in Coq, otherwise your proofs are unsound. (2) Can't easily axiomatize addition, multiplication, subtraction, because Ocaml integers don't behave like the "mathematical" integers upon 31-bit (or 63-bit) overflow. (3) Can *only* run the programs in Ocaml, not inside Coq.

So let's examine another approach, which is quite standard inside Coq: use a construction in Coq of arbitrary-precision binary numbers, with logN-time addition, subtraction, and comparison.

11.3 A Simple Program That's Waaaaay Too Slow.

```
Require Import Coq.Strings.String.
From VFA Require Import Perm.
From VFA Require Import Maps.
Import FunctionalExtensionality.

Module VerySlow.

Fixpoint loop (input: list nat) (c: nat) (table: total_map bool) : nat :=
  match input with
  | nil ⇒ c
  | a::al ⇒ if table a
                  then loop al (c+1) table
                  else loop al c (t_update table a true)
 end.

Definition collisions (input: list nat) : nat :=
       loop input 0 (t_empty false).

Example collisions_pi: collisions [3;1;4;1;5;9;2;6] = 1.
Proof. reflexivity. Qed.
```

This program takes cubic time, O(N^3). Let's assume that there are few duplicates, or none at all. There are N iterations of loop, each iteration does a table lookup, most iterations do a t_update as well, and those operations each do N comparisons. The average length of the table (the number of elements) averages only N/2, and (if there are few duplicates) the lookup will have to traverse the entire list, so really in each iteration there will be only $N/2$ comparisons instead of N, but in asymptotic analysis we ignore the constant factors.

So far it seems like this is a quadratic-time algorithm, O(N^2). But to compare Coq natural numbers for equality takes O(N) time as well:

```
Print beq_nat.
```

Remember, **nat** is a unary representation, with a number of S constructors proportional to the number being represented!

```
End VerySlow.
```

11.4 Efficient Positive Numbers

We can do better; we *must* do better. In fact, Coq's integer type, called **Z**, is a binary representation (not unary), so that operations such as *plus* and *leq* take time linear in the number of bits, that is, logarithmic in the value of the numbers. Here we will explore how **Z** is built.

```
Module Integers.
```

We start with positive numbers.

```
Inductive positive : Set :=
  | xI : positive → positive
  | xO : positive → positive
  | xH : positive.
```

A positive number is either

- 1, that is, xH
- 0+2n, that is, xO n
- 1+2n, that is, xI n.

For example, ten is 0+2(1+2(0+2(1))).

```
Definition ten := xO (xI (xO xH)).
```

To interpret a **positive** number as a **nat**,

```
Fixpoint positive2nat (p: positive) : nat :=
  match p with
  | xI q ⇒ 1 + 2 × positive2nat q
  | xO q ⇒ 0 + 2 × positive2nat q
  | xH ⇒ 1
 end.

Eval compute in positive2nat ten.
```

We can read the binary representation of a positive number as the *backwards* sequence of xO (meaning 0) and xI/xH (1). Thus, ten is 1010 in binary.

```
Fixpoint print_in_binary (p: positive) : list nat :=
  match p with
```

```
  | xI q ⇒ print_in_binary q ++ [1]
  | xO q ⇒print_in_binary q ++ [0]
  | xH ⇒ [1]
 end.
```

```
Eval compute in print_in_binary ten.
```

Another way to see the "binary representation" is to make up postfix notation for xI and xO, as follows

```
Notation "p ~ 1" := (xI p)
 (at level 7, left associativity, format "p '~' '1'").
Notation "p ~ 0" := (xO p)
 (at level 7, left associativity, format "p '~' '0'").
```

```
Print ten.
```

Why are we using positive numbers anyway? Since the zero was invented 2300 years ago by the Babylonians, it's sort of old-fashioned to use number systems that start at 1.

The answer is that it's highly inconvenient to have number systems with several different representations of the same number. For one thing, we don't want to worry about 00110=110. Then, when we extend this to the integers, with a "minus sign", we don't have to worry about -0 = +0.

To find the successor of a binary number–that is to increment– we work from low-order to high-order, until we hit a zero bit.

```
Fixpoint succ x :=
  match x with
    | p~1 ⇒ (succ p)~0
    | p~0 ⇒ p~1
    | xH ⇒ xH~0
  end.
```

To add binary numbers, we work from low-order to high-order, keeping track of the carry.

```
Fixpoint addc (carry: bool) (x y: positive) {struct x} : positive :=
  match carry, x, y with
    | false, p~1, q~1 ⇒ (addc true p q)~0
    | false, p~1, q~0 ⇒ (addc false p q)~1
    | false, p~1, xH ⇒ (succ p)~0
    | false, p~0, q~1 ⇒ (addc false p q)~1
    | false, p~0, q~0 ⇒ (addc false p q)~0
    | false, p~0, xH ⇒ p~1
    | false, xH, q~1 ⇒ (succ q)~0
    | false, xH, q~0 ⇒ q~1
    | false, xH, xH ⇒ xH~0
    | true, p~1, q~1 ⇒ (addc true p q)~1
```

```
    | true, p~1, q~0 ⇒ (addc true p q)~0
    | true, p~1, xH ⇒ (succ p)~1
    | true, p~0, q~1 ⇒ (addc true p q)~0
    | true, p~0, q~0 ⇒ (addc false p q)~1
    | true, p~0, xH ⇒ (succ p)~0
    | true, xH, q~1 ⇒ (succ q)~1
    | true, xH, q~0 ⇒ (succ q)~0
    | true, xH, xH ⇒ xH~1
  end.
Definition add (x y: positive) : positive := addc false x y.
```

Exercise: 2 stars (succ_correct) Lemma succ_correct: ∀ p,

```
    positive2nat (succ p) = S (positive2nat p).
Proof.
    Admitted.
```

☐

Exercise: 3 stars (addc_correct) You may use `omega` in this proof if you want, along with induction of course. But really, using `omega` is an anachronism in a sense: Coq's `omega` uses theorems about **Z** that are proved from theorems about Coq's standard-library **positive** that, in turn, rely on a theorem much like this one. So the authors of the Coq standard library had to do the associative-commutative rearrangement proofs "by hand." But really, here you can use `omega` without penalty.

```
Lemma addc_correct: ∀ (c: bool) (p q: positive),
    positive2nat (addc c p q) =
          (if c then 1 else 0) + positive2nat p + positive2nat q.
Proof.
    Admitted.

Theorem add_correct: ∀ (p q: positive),
    positive2nat (add p q) = positive2nat p + positive2nat q.
Proof.
intros.
unfold add.
apply addc_correct.
Qed.
```

☐

Claim: the add function on positive numbers takes worst-case time proportional to the log base 2 of the result.

We can't prove this in Coq, since Coq has no cost model for execution. But we can prove it informally. Notice that addc is structurally recursive on p, that is, the number of recursive calls is at most the height of the p structure; that's equal to log base 2 of p (rounded up to the nearest integer). The last call may call succ q, which is structurally recursive on q, but this q argument is what remained of the original q after stripping off a number of constructors equal to the height of p.

To implement comparison algorithms on positives, the recursion (Fixpoint) is easier to implement if we compute not only "less-than / not-less-than", but actually, "less / equal / greater". To express these choices, we use an Inductive data type.

```
Inductive comparison : Set :=
  Eq : comparison | Lt : comparison | Gt : comparison.
```

Exercise: 5 stars (compare_correct) `Fixpoint compare x y {struct x}:=`

```
 match x, y with
   | p~1, q~1 ⇒ compare p q
   | p~1, q~0 ⇒ match compare p q with Lt ⇒ Lt | _ ⇒ Gt end
   | p~1, xH ⇒ Gt



   | _, _ ⇒ Lt
 end.
```

```
Lemma positive2nat_pos:
 ∀ p, positive2nat p > 0.
Proof.
intros.
induction p; simpl; omega.
Qed.
```

```
Theorem compare_correct:
 ∀ x y,
  match compare x y with
  | Lt ⇒ positive2nat x < positive2nat y
  | Eq ⇒ positive2nat x = positive2nat y
  | Gt ⇒ positive2nat x > positive2nat y
 end.
Proof.
induction x; destruct y; simpl.
```

Admitted.

☐

Claim: compare x y takes time proportional to the log base 2 of x. Proof: it's structurally inductive on the height of x.

11.4.1 Coq's Integer Type, Z

Coq's integer type is constructed from positive numbers:

```
Inductive Z : Set :=
  | Z0 : Z
  | Zpos : positive → Z
  | Zneg : positive → Z.
```

We can construct efficient (logN time) algorithms for operations on **Z**: add, *subtract*, compare, and so on. These algorithms call upon the efficient algorithms for **positive**s.

We won't show these here, because in this chapter we now turn to efficient maps over positive numbers.

```
End Integers.
```

These types, **positive** and **Z**, are part of the Coq standard library. We can access them here, because (above) the `Import Perm` has also exported `ZArith` to us.

```
Print positive.

Check Pos.compare. Check Pos.add.

Check Z.add.
```

11.4.2 From $N \times N \times N$ to $N \times N \times logN$

This program runs in $(N\hat{}2)$*$(log\ N)$ time. The loop does N iterations; the table lookup does O(N) comparisons, and each comparison takes O$(log\ N)$ time.

```
Module RatherSlow.

Definition total_mapz (A: Type) := Z → A.

Definition empty {A:Type} (default: A) : total_mapz A := fun _ ⇒ default.
Definition update {A:Type} (m : total_mapz A)
                           (x : Z) (v : A) :=
  fun x' ⇒ if Z.eqb x x' then v else m x'.

Fixpoint loop (input: list Z) (c: Z) (table: total_mapz bool) : Z :=
  match input with
  | nil ⇒ c
  | a::al ⇒ if table a
                  then loop al (c+1) table
                  else loop al c (update table a true)
 end.

Definition collisions (input: list Z) := loop input 0 (empty false).

Example collisions_pi: collisions [3;1;4;1;5;9;2;6]%Z = 1%Z.
Proof. reflexivity. Qed.

End RatherSlow.
```

11.4.3 From $N \times N \times logN$ to $N \times logN \times logN$

We can use balanced binary search trees (red-black trees), with keys of type **Z**. Then the loop does N iterations; the table lookup does O($logN$) comparisons, and each comparison takes O($log\ N$) time. Overall, the asymptotic run time is N*($logN$)^2.

11.5 Tries: Efficient Lookup Tables on Positive Binary Numbers

Binary search trees are very nice, because they can implement lookup tables from *any* totally ordered type to any other type. But when the type of keys is known specifically to be (small-to-medium size) integers, then we can use a more specialized representation.

By analogy, in imperative programming languages (C, Java, ML), when the index of a table is the integers in a certain range, you can use arrays. When the keys are not integers, you have to use something like hash tables or binary search trees.

A *trie* is a tree in which the edges are labeled with letters from an alphabet, and you look up a word by following edges labeled by successive letters of the word. In fact, a trie is a special case of a Deterministic Finite Automaton (DFA) that happens to be a tree rather than a more general graph.

A *binary trie* is a trie in which the alphabet is just {0,1}. The "word" is a sequence of bits, that is, a binary number. To look up the "word" 10001, use 0 as a signal to "go left", and 1 as a signal to "go right."

The binary numbers we use will be type **positive**:

```
Print positive.
```

```
Goal 10%positive = xO (xI (xO xH)).
Proof. reflexivity. Qed.
```

Given a **positive** number such as ten, we will go left to right in the xO/xI/ *constructors* (*which* is *from the low-order bit to the high-order bit*), using [xO] as *a signal to go* left, [xI] as *a signal to go* right, *and* [xH] as *a signal to stop.*

```
Inductive trie (A : Type) :=
    | Leaf : trie A
    | Node : trie A → A → trie A → trie A.
Arguments Leaf {A}.
Arguments Node {A} _ _ _.
```

```
Definition trie_table (A: Type) : Type := (A × trie A)%type.
```

```
Definition empty {A: Type} (default: A) : trie_table A :=
      (default, Leaf).
```

```
Fixpoint look {A: Type} (default: A) (i: positive) (m: trie A): A :=
    match m with
    | Leaf ⇒ default
```

```
    | Node l x r ⇒
        match i with
        | xH ⇒ x
        | xO i' ⇒ look default i' l
        | xI i' ⇒ look default i' r
        end
    end.

Definition lookup {A: Type} (i: positive) (t: trie_table A) : A :=
  look (fst t) i (snd t).

Fixpoint ins {A: Type} default (i: positive) (a: A) (m: trie A): trie A :=
    match m with
    | Leaf ⇒
        match i with
        | xH ⇒ Node Leaf a Leaf
        | xO i' ⇒ Node (ins default i' a Leaf) default Leaf
        | xI i' ⇒ Node Leaf default (ins default i' a Leaf)
        end
    | Node l o r ⇒
        match i with
        | xH ⇒ Node l a r
        | xO i' ⇒ Node (ins default i' a l) o r
        | xI i' ⇒ Node l o (ins default i' a r)
        end
    end.

Definition insert {A: Type} (i: positive) (a: A) (t: trie_table A)
                  : trie_table A :=
  (fst t, ins (fst t) i a (snd t)).

Definition three_ten : trie_table bool :=
 insert 3 true (insert 10 true (empty false)).

Eval compute in three_ten.

Eval compute in
   map (fun i ⇒ lookup i three_ten) [3;1;4;1;5]%positive.
```

11.5.1 From $N \times logN \times logN$ to $N \times logN$

```
Module FastEnough.

Fixpoint loop (input: list positive) (c: nat) (table: trie_table bool) : nat :=
  match input with
  | nil ⇒ c
  | a::al ⇒ if lookup a table
```

```
                                then loop al (1+c) table
                                else loop al c (insert a true table)
 end.
```

Definition collisions (*input*: **list positive**) := loop *input* 0 (empty false).

Example collisions_pi: collisions [3;1;4;1;5;9;2;6]%*positive* = 1.
Proof. reflexivity. Qed.

End FastEnough.

This program takes O(N *log* N) time: the loop executes N iterations, the lookup takes *log* N time, the insert takes *log* N time. One might worry about $1+c$ computed in the natural numbers (unary representation), but this evaluates in one step to S c, which takes constant time, no matter how long c is. In "real life", one might be advised to use **Z** instead of **nat** for the c variables, in which case, $1+c$ takes worst-case *log* N, and average-case constant time.

Exercise: 2 stars (successor_of_Z_constant_time) Explain why the average-case time for successor of a binary integer, with carry, is constant time. Assume that the input integer is random (uniform distribution from 1 to N), or assume that we are iterating successor starting at 1, so that each number from 1 to N is touched exactly once – whichever way you like.

Definition manual_grade_for_successor_of_Z_constant_time : **option** (**prod nat string**) := None.

□

11.6 Proving the Correctness of Trie Tables

Trie tables are just another implementation of the Maps abstract data type. What we have to prove is the same as usual for an ADT: define a representation invariant, define an abstraction relation, prove that the operations respect the invariant and the abstraction relation.

We will indeed do that. But this time we'll take a different approach. Instead of defining a "natural" abstraction relation based on what we see in the data structure, we'll define an abstraction relation that says, "what you get is what you get." This will work, but it means we've moved the work into directly proving some things about the relation between the lookup and the insert operators.

11.6.1 Lemmas About the Relation Between **lookup** and **insert**

Exercise: 1 star (look_leaf) Lemma look_leaf:
 ∀ A (a:A) j, look a j Leaf = a.
 Admitted.

□

Exercise: 2 stars (look_ins_same) This is a rather simple induction.

Lemma look_ins_same: $\forall$ $\{A\}$ a k $(v{:}A)$ t, look a k (ins a k v t) = v.
Admitted.
☐

Exercise: 3 stars (look_ins_same) Induction on j? Induction on t? Do you feel lucky?

Lemma look_ins_other: $\forall$ $\{A\}$ a j k $(v{:}A)$ t,
$j \neq k \rightarrow$ look a j (ins a k v t) = look a j t.
Admitted.
☐

11.6.2 Bijection Between positive and nat.

In order to relate lookup on positives to total_map on nats, it's helpful to have a bijection between positive and nat. We'll relate 1%positive to 0%nat, 2%positive to 1%nat, and so on.

```
Definition nat2pos (n: nat) : positive := Pos.of_succ_nat n.
Definition pos2nat (n: positive) : nat := pred (Pos.to_nat n).
```

```
Lemma pos2nat2pos: ∀ p, nat2pos (pos2nat p) = p.
Proof. intro. unfold nat2pos, pos2nat.
rewrite ← (Pos2Nat.id p) at 2.
destruct (Pos.to_nat p) eqn:?.
pose proof (Pos2Nat.is_pos p). omega.
rewrite ← Pos.of_nat_succ.
reflexivity.
Qed.
```

```
Lemma nat2pos2nat: ∀ i, pos2nat (nat2pos i) = i.
Proof. intro. unfold nat2pos, pos2nat.
rewrite SuccNat2Pos.id_succ.
reflexivity.
Qed.
```

Now, use those two lemmas to prove that it's really a bijection!

Exercise: 2 stars (pos2nat_bijective) Lemma pos2nat_injective: $\forall$ p q, pos2nat p = pos2nat q $\rightarrow$ p=q.
Admitted.

Lemma nat2pos_injective: $\forall$ i j, nat2pos i = nat2pos j $\rightarrow$ i=j.
Admitted.
☐

11.6.3 Proving That Tries are a "Table" ADT.

Representation invariant. Under what conditions is a trie well-formed? Fill in the simplest thing you can, to start; then correct it later as necessary.

```
Definition is_trie {A: Type} (t: trie_table A) : Prop
. Admitted.
```

Abstraction relation. This is what we mean by, "what you get is what you get." That is, the abstraction of a trie_table is the total function, from naturals to A values, that you get by running the lookup function. Based on this abstraction relation, it'll be trivial to prove lookup_relate. But insert_relate will NOT be trivial.

```
Definition abstract {A: Type} (t: trie_table A) (n: nat) : A :=
  lookup (nat2pos n) t.
```

```
Definition Abs {A: Type} (t: trie_table A) (m: total_map A) :=
  abstract t = m.
```

Exercise: 2 stars (is_trie) If you picked a *really simple* representation invariant, these should be easy. Later, if you need to change the representation invariant in order to get the *_relate* proofs to work, then you'll need to fix these proofs.

```
Theorem empty_is_trie: ∀ {A} (default: A), is_trie (empty default).
   Admitted.
```

```
Theorem insert_is_trie: ∀ {A} i x (t: trie_table A),
   is_trie t → is_trie (insert i x t).
   Admitted.
   □
```

Exercise: 2 stars (empty_relate) Just unfold a bunch of definitions, use `extensionality`, and use one of the lemmas you proved above, in the section "Lemmas about the relation between lookup and insert."

```
Theorem empty_relate: ∀ {A} (default: A),
    Abs (empty default) (t_empty default).
Proof.
   Admitted.
   □
```

Exercise: 2 stars (lookup_relate) Given the abstraction relation we've chosen, this one should be really simple.

```
Theorem lookup_relate: ∀ {A} i (t: trie_table A) m,
    is_trie t → Abs t m → lookup i t = m (pos2nat i).
   Admitted.
   □
```

Exercise: 3 stars (insert_relate) Given the abstraction relation we've chosen, this one should NOT be simple. However, you've already done the heavy lifting, with the lemmas look_ins_same and look_ins_other. You will not need induction here. Instead, unfold a bunch of things, use extensionality, and get to a case analysis on whether pos2nat k =? pos2nat j. To handle that case analysis, use *bdestruct*. You may also need pos2nat_injective.

```
Theorem insert_relate: ∀ {A} k (v: A) t cts,
    is_trie t →
    Abs t cts →
    Abs (insert k v t) (t_update cts (pos2nat k) v).
  Admitted.
```

□

11.6.4 Sanity Check

```
Example Abs_three_ten:
    Abs
      (insert 3 true (insert 10 true (empty false)))
      (t_update (t_update (t_empty false) (pos2nat 10) true) (pos2nat 3) true).
Proof.
try (apply insert_relate; [hnf; auto | ]).
try (apply insert_relate; [hnf; auto | ]).
try (apply empty_relate).
  Admitted.
```

11.7 Conclusion

Efficient functional maps with (positive) integer keys are one of the most important data structures in functional programming. They are used for symbol tables in compilers and static analyzers; to represent directed graphs (the mapping from node-ID to edge-list); and (in general) anywhere that an imperative algorithm uses an array or *requires* a mutable pointer.

Therefore, these *tries* on positive numbers are very important in Coq programming. They were introduced by Xavier Leroy and Sandrine Blazy in the CompCert compiler (2006), and are now available in the Coq standard library as the *PositiveMap* module, which implements the FMaps interface. The core implementation of *PositiveMap* is just as shown in this chapter, but FMaps uses different names for the functions insert and lookup, and also provides several other operations on maps.

Chapter 12

Library VFA.Priqueue

12.1 Priqueue: Priority Queues

A *priority queue* is an abstract data type with the following operations:

- empty: priqueue
- insert: key → priqueue → priqueue
- delete_max: priqueue → **option** (key × priqueue)

The idea is that you can find (and remove) the highest-priority element. Priority queues have applications in:

- Discrete-event simulations: The highest-priority event is the one whose scheduled time is the earliest. Simulating one event causes new events to be scheduled in the future.
- Sorting: *heap sort* puts all the elements in a priority queue, then removes them one at a time.
- Computational geometry: algorithms such as *convex hull* use priority queues.
- Graph algorithms: Dijkstra's algorithm for finding the shortest path uses a priority queue.

We will be considering *mergeable* priority queues, with one additional operator:

- merge: priqueue → priqueue → priqueue

The classic data structure for priority queues is the "heap", a balanced binary tree in which the the key at any node is *bigger* than all the keys in nodes below it. With heaps, empty is constant time, insert and delete_max are logN time. But merge takes NlogN time, as one must take all the elements out of one queue and insert them into the other queue.

Another way to do priority queues is by *balanced binary search trees* (such as red-black trees); again, empty is constant time, insert and delete_max are logN time, and merge takes NlogN time, as one must take all the elements out of one queue and insert them into the other queue.

In the *Binom* chapter we will examine an algorithm in which empty is constant time, insert, delete_max, and merge are logN time.

In *this* chapter we will consider a much simpler (and slower) implementation, using unsorted lists, in which:

- empty takes constant time
- insert takes constant time
- delete_max takes linear time
- merge takes linear time

12.2 Module Signature

This is the "signature" of a correct implementation of priority queues where the keys are natural numbers. Using **nat** for the key type is a bit silly, since the comparison function Nat.ltb takes linear time in the value of the numbers! But you have already seen in the `Extract` chapter how to define these kinds of algorithms on key types that have efficient comparisons, so in this chapter (and the Binom chapter) we simply won't worry about the time per comparison.

```
From VFA Require Import Perm.

Module Type PRIQUEUE.
  Parameter priqueue: Type.
  Definition key := nat.

  Parameter empty: priqueue.
  Parameter insert: key → priqueue → priqueue.
  Parameter delete_max: priqueue → option (key × priqueue).
  Parameter merge: priqueue → priqueue → priqueue.

  Parameter priq: priqueue → Prop.
  Parameter Abs: priqueue → list key → Prop.
  Axiom can_relate: ∀ p, priq p → ∃ al, Abs p al.
  Axiom abs_perm: ∀ p al bl,
   priq p → Abs p al → Abs p bl → Permutation al bl.
  Axiom empty_priq: priq empty.
  Axiom empty_relate: Abs empty nil.
  Axiom insert_priq: ∀ k p, priq p → priq (insert k p).
```

```
Axiom insert_relate:
      ∀ p al k, priq p → Abs p al → Abs (insert k p) (k::al).
Axiom delete_max_None_relate:
      ∀ p, priq p → (Abs p nil ↔ delete_max p = None).
Axiom delete_max_Some_priq:
    ∀ p q k, priq p → delete_max p = Some(k,q) → priq q.
Axiom delete_max_Some_relate:
∀ (p q: priqueue) k (pl ql: list key), priq p →
  Abs p pl →
  delete_max p = Some (k,q) →
  Abs q ql →
  Permutation pl (k::ql) ∧ Forall (ge k) ql.
Axiom merge_priq: ∀ p q, priq p → priq q → priq (merge p q).
Axiom merge_relate:
   ∀ p q pl ql al,
      priq p → priq q →
      Abs p pl → Abs q ql → Abs (merge p q) al →
      Permutation al (pl++ql).
End PRIQUEUE.
```

Take some time to consider whether this is the right specification! As always, if we get the specification wrong, then proofs of "correctness" are not so useful.

12.3 Implementation

Module List_Priqueue <: PRIQUEUE.

Now we are responsible for providing *Definitions* of all those `Parameters`, and proving *Theorems* for all those `Axioms`, so that the values in the `Module` match the types in the `Module Type`. If we try to `End` List_Priqueue before everything is provided, we'll get an error. Uncomment the next line and try it!

12.3.1 Some Preliminaries

A copy of the select function from Selection.v, but getting the max element instead of the min element:

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

Exercise: 3 stars (select_perm_and_friends) Lemma select_perm: ∀ i l,

```
  let (j,r) := select i l in
   Permutation (i::l) (j::r).
Proof. intros i l; revert i.
induction l; intros; simpl in *.
    Admitted.

Lemma select_biggest_aux:
  ∀ i al j bl,
     Forall (fun x ⇒ j ≥ x) bl →
     select i al = (j,bl) →
     j ≥ i.
Proof. Admitted.

Theorem select_biggest:
  ∀ i al j bl, select i al = (j,bl) →
       Forall (fun x ⇒ j ≥ x) bl.
Proof. intros i al; revert i; induction al; intros; simpl in *.
    admit.
bdestruct (i >=? a).
×
destruct (select i al) eqn:?H.
    Admitted.
```

☐

12.3.2 The Program

```
Definition key := nat.

Definition priqueue := list key.

Definition empty : priqueue := nil.
Definition insert (k: key)(p: priqueue) := k::p.
Definition delete_max (p: priqueue) :=
  match p with
  | i::p' ⇒ Some (select i p')
  | nil ⇒ None
  end.
Definition merge (p q: priqueue) : priqueue := p++q.
```

12.4 Predicates on Priority Queues

12.4.1 The Representation Invariant

In this implementation of priority queues as unsorted lists, the representation invariant is trivial.

```
Definition priq (p: priqueue) := True.
```

The abstraction relation is trivial too.

```
Inductive Abs': priqueue → list key → Prop :=
Abs_intro: ∀ p, Abs' p p.
Definition Abs := Abs'.
```

12.4.2 Sanity Checks on the Abstraction Relation

```
Lemma can_relate : ∀ p, priq p → ∃ al, Abs p al.
Proof.
  intros. ∃ p; constructor.
Qed.
```

When the Abs relation says, "priority queue p contains elements al", it is free to report the elements in any order. It could even relate p to two different lists al and bl, as long as one is a permutation of the other.

```
Lemma abs_perm: ∀ p al bl,
   priq p → Abs p al → Abs p bl → Permutation al bl.
Proof.
intros.
inv H0. inv H1. apply Permutation_refl.
Qed.
```

12.4.3 Characterizations of the Operations on Queues

```
Lemma empty_priq: priq empty.
Proof. constructor. Qed.
Lemma empty_relate: Abs empty nil.
Proof. constructor. Qed.
Lemma insert_priq: ∀ k p, priq p → priq (insert k p).
Proof. intros; constructor. Qed.
Lemma insert_relate:
    ∀ p al k, priq p → Abs p al → Abs (insert k p) (k::al).
Proof. intros. unfold insert. inv H0. constructor. Qed.
```

```
Lemma delete_max_Some_priq:
      ∀ p q k, priq p → delete_max p = Some(k,q) → priq q.
Proof. constructor. Qed.
```

Exercise: 2 stars (simple_priq_proofs)

```
Lemma delete_max_None_relate:
  ∀ p, priq p →
      (Abs p nil ↔ delete_max p = None).
Proof.
   Admitted.

Lemma delete_max_Some_relate:
  ∀ (p q: priqueue) k (pl ql: list key), priq p →
   Abs p pl →
   delete_max p = Some (k,q) →
   Abs q ql →
   Permutation pl (k::ql) ∧ Forall (ge k) ql.
Proof.
   Admitted.

Lemma merge_priq:
  ∀ p q, priq p → priq q → priq (merge p q).
Proof. intros. constructor. Qed.

Lemma merge_relate:
    ∀ p q pl ql al,
       priq p → priq q →
       Abs p pl → Abs q ql → Abs (merge p q) al →
       Permutation al (pl++ql).
Proof.
   Admitted.
```

☐

End LIST_PRIQUEUE.

Chapter 13

Library VFA.Binom

13.1 Binom: Binomial Queues

Implementation and correctness proof of fast mergeable priority queues using binomial queues.

Operation empty is constant time, insert, delete_max, and merge are logN time. (Well, except that comparisons on **nat** take linear time. Read the `Extract` chapter to see what can be done about that.)

13.2 Required Reading

Binomial Queues http://www.cs.princeton.edu/~appel/Binom.pdf by Andrew W. Appel, 2016.

Binomial Queues http://www.cs.princeton.edu/~appel/BQ.pdf Section 9.7 of *Algorithms 3rd Edition in Java, Parts 1-4: Fundamentals, Data Structures, Sorting, and Searching*, by Robert Sedgewick. Addison-Wesley, 2002.

13.3 The Program

```
Require Import Coq.Strings.String.
From VFA Require Import Perm.
From VFA Require Import Priqueue.

Module BinomQueue <: PRIQUEUE.

Definition key := nat.

Inductive tree : Type :=
| Node: key → tree → tree → tree
| Leaf : tree.
```

A priority queue (using the binomial queues data structure) is a list of trees. The i'th element of the list is either Leaf or it is a power-of-2-heap with exactly 2ˆi nodes.

This program will make sense to you if you've read the Sedgewick reading; otherwise it is rather mysterious.

```
Definition priqueue := list tree.

Definition empty : priqueue := nil.

Definition smash (t u: tree) : tree :=
  match t , u with
  | Node x t1 Leaf, Node y u1 Leaf ⇒
                    if x >? y then Node x (Node y u1 t1) Leaf
                                  else Node y (Node x t1 u1) Leaf
  | _ , _ ⇒ Leaf
  end.

Fixpoint carry (q: list tree) (t: tree) : list tree :=
  match q, t with
  | nil, Leaf ⇒ nil
  | nil, _ ⇒ t :: nil
  | Leaf :: q', _ ⇒ t :: q'
  | u :: q', Leaf ⇒ u :: q'
  | u :: q', _ ⇒ Leaf :: carry q' (smash t u)
 end.

Definition insert (x: key) (q: priqueue) : priqueue :=
    carry q (Node x Leaf Leaf).

Eval compute in fold_left (fun x q ⇒insert q x) [3;1;4;1;5;9;2;3;5] empty.
   = Node 5 Leaf Leaf; Leaf; Leaf; Node 9 (Node 4 (Node 3 (Node 1 Leaf Leaf) (Node 1 Leaf
Leaf)) (Node 3 (Node 2 Leaf Leaf) (Node 5 Leaf Leaf))) Leaf : priqueue »

Fixpoint join (p q: priqueue) (c: tree) : priqueue :=
  match p, q, c with
    [], _ , _ ⇒ carry q c
  | _, [], _ ⇒ carry p c
  | Leaf::p', Leaf::q', _ ⇒ c :: join p' q' Leaf
  | Leaf::p', q1::q', Leaf ⇒ q1 :: join p' q' Leaf
  | Leaf::p', q1::q', Node _ _ _ ⇒ Leaf :: join p' q' (smash c q1)
  | p1::p', Leaf::q', Leaf ⇒ p1 :: join p' q' Leaf
  | p1::p', Leaf::q',Node _ _ _ ⇒ Leaf :: join p' q' (smash c p1)
  | p1::p', q1::q', _ ⇒ c :: join p' q' (smash p1 q1)
 end.

Fixpoint unzip (t: tree) (cont: priqueue → priqueue) : priqueue :=
  match t with
  | Node x t1 t2 ⇒ unzip t2 (fun q ⇒ Node x t1 Leaf :: cont q)
```

```
  | Leaf ⇒ cont nil
  end.

Definition heap_delete_max (t: tree) : priqueue :=
  match t with
    Node x t1 Leaf ⇒ unzip t1 (fun u ⇒ u)
  | _ ⇒ nil
  end.

Fixpoint find_max' (current: key) (q: priqueue) : key :=
  match q with
  | [] ⇒ current
  | Leaf::q' ⇒ find_max' current q'
  | Node x _ _ :: q' ⇒ find_max' (if x >? current then x else current) q'
  end.

Fixpoint find_max (q: priqueue) : option key :=
  match q with
  | [] ⇒ None
  | Leaf::q' ⇒ find_max q'
  | Node x _ _ :: q' ⇒ Some (find_max' x q')
 end.

Fixpoint delete_max_aux (m: key) (p: priqueue) : priqueue × priqueue :=
  match p with
  | Leaf :: p' ⇒ let (j,k) := delete_max_aux m p' in (Leaf::j, k)
  | Node x t1 Leaf :: p' ⇒
       if m >? x
       then (let (j,k) := delete_max_aux m p'
             in (Node x t1 Leaf::j,k))
       else (Leaf::p', heap_delete_max (Node x t1 Leaf))
  | _ ⇒ (nil, nil)
  end.

Definition delete_max (q: priqueue) : option (key × priqueue) :=
  match find_max q with
  | None ⇒ None
  | Some m ⇒ let (p',q') := delete_max_aux m q
                                in Some (m, join p' q' Leaf)
  end.

Definition merge (p q: priqueue) := join p q Leaf.
```

13.4 Characterization Predicates

t is a complete binary tree of depth n, with every key $<= m$

```
Fixpoint pow2heap' (n: nat) (m: key) (t: tree) :=
 match n, m, t with
    0, m, Leaf ⇒ True
  | 0, m, Node _ _ _ ⇒ False
  | S _, m,Leaf ⇒ False
  | S n', m, Node k l r ⇒
       m ≥ k ∧ pow2heap' n' k l ∧ pow2heap' n' m r
 end.
```

t is a power-of-2 heap of depth n

```
Definition pow2heap (n: nat) (t: tree) :=
  match t with
    Node m t1 Leaf ⇒ pow2heap' n m t1
  | _ ⇒ False
  end.
```

l is the ith tail of a binomial heap

```
Fixpoint priq' (i: nat) (l: list tree) : Prop :=
   match l with
  | t :: l' ⇒ (t=Leaf ∨ pow2heap i t) ∧ priq' (S i) l'
  | nil ⇒ True
 end.
```

q is a binomial heap

```
Definition priq (q: priqueue) : Prop := priq' 0 q.
```

13.5 Proof of Algorithm Correctness

13.5.1 Various Functions Preserve the Representation Invariant

...that is, the priq property, or the closely related property pow2heap.

Exercise: 1 star (empty_priq) `Theorem empty_priq: priq empty.`

Admitted.

☐

Exercise: 2 stars (smash_valid) `Theorem smash_valid:`

```
      ∀ n t u, pow2heap n t → pow2heap n u → pow2heap (S n) (smash t u).
```

Admitted.

☐

Exercise: 3 stars (carry_valid) `Theorem` carry_valid:
  ∀ n q, priq' n q →
  ∀ t, (t=Leaf ∨ pow2heap n t) → priq' n (carry q t).

Admitted.

☐

Exercise: 2 stars, optional (insert_valid) `Theorem` insert_priq: ∀ x q, priq q → priq (insert x q).

Admitted.

☐

Exercise: 3 stars, optional (join_valid) `Theorem` join_valid: ∀ p q c n, priq' n p → priq' n q → (c=Leaf ∨ pow2heap n c) → priq' n (join p q c).

Admitted.

☐

`Theorem` merge_priq: ∀ p q, priq p → priq q → priq (merge p q).
`Proof`.
 `intros`. `unfold` merge. `apply` *join_valid*; `auto`.
`Qed`.

Exercise: 5 stars, optional (delete_max_Some_priq) `Theorem` delete_max_Some_priq:
  ∀ p q k, priq p → delete_max p = Some(k,q) → priq q.

Admitted.

☐

13.5.2 The Abstraction Relation

tree_elems t l means that the keys in t are the same as the elements of l (with repetition)

`Inductive` **tree_elems**: **tree** → **list** key → `Prop` :=
| tree_elems_leaf: **tree_elems** Leaf nil
| tree_elems_node: ∀ bl br v tl tr b,
  tree_elems tl bl →
  tree_elems tr br →
  Permutation b (v::bl++br) →
  tree_elems (Node v tl tr) b.

Exercise: 3 stars (priqueue_elems) Make an inductive definition, similar to **tree_elems**, to relate a priority queue "l" to a list of all its elements.

As you can see in the definition of **tree_elems**, a **tree** relates to *any* permutation of its keys, not just a single permutation. You should make your **priqueue_elems** relation behave similarly, using (basically) the same technique as in **tree_elems**.

```
Inductive priqueue_elems: list tree → list key → Prop :=
```

.

```
Definition manual_grade_for_priqueue_elems : option (prod nat string) := None.
```

☐

```
Definition Abs (p: priqueue) (al: list key) := priqueue_elems p al.
```

13.5.3 Sanity Checks on the Abstraction Relation

Exercise: 2 stars (tree_elems_ext) Extensionality theorem for the tree_elems relation

```
Theorem tree_elems_ext: ∀ t e1 e2,
  Permutation e1 e2 → tree_elems t e1 → tree_elems t e2.
   Admitted.
```

☐

Exercise: 2 stars (tree_perm)

```
Theorem tree_perm: ∀ t e1 e2,
  tree_elems t e1 → tree_elems t e2 → Permutation e1 e2.
   Admitted.
```

☐

Exercise: 2 stars (priqueue_elems_ext) To prove priqueue_elems_ext, you should almost be able to cut-and-paste the proof of tree_elems_ext, with just a few edits.

```
Theorem priqueue_elems_ext: ∀ q e1 e2,
  Permutation e1 e2 → priqueue_elems q e1 → priqueue_elems q e2.
   Admitted.
```

☐

Exercise: 2 stars (abs_perm)

```
Theorem abs_perm: ∀ p al bl,
   priq p → Abs p al → Abs p bl → Permutation al bl.
Proof.
   Admitted.
```

☐

Exercise: 2 stars (can_relate)

```
Lemma tree_can_relate: ∀ t, ∃ al, tree_elems t al.
Proof.
   Admitted.
Theorem can_relate: ∀ p, priq p → ∃ al, Abs p al.
Proof.
   Admitted.
```

☐

13.5.4 Various Functions Preserve the Abstraction Relation

Exercise: 1 star (empty_relate) `Theorem` empty_relate: Abs empty nil.
`Proof.`
Admitted.
☐

Exercise: 3 stars (smash_elems) Warning: This proof is rather long.

`Theorem` smash_elems: $\forall\ n\ t\ u\ bt\ bu,$
pow2heap $n\ t \rightarrow$ pow2heap $n\ u \rightarrow$
tree_elems $t\ bt \rightarrow$ **tree_elems** $u\ bu \rightarrow$
tree_elems (smash $t\ u$) (bt ++ bu).
Admitted.
☐

13.5.5 Optional Exercises

Some of these proofs are quite long, but they're not especially tricky.

Exercise: 4 stars, optional (carry_elems) `Theorem` carry_elems:
$\forall\ n\ q,$ priq' $n\ q \rightarrow$
$\forall\ t,$ (t=Leaf $\lor$ pow2heap $n\ t$) $\rightarrow$
$\forall\ eq\ et,$ **priqueue_elems** $q\ eq \rightarrow$
tree_elems $t\ et \rightarrow$
priqueue_elems (carry $q\ t$) (eq++et).
Admitted.
☐

Exercise: 2 stars, optional (insert_elems) `Theorem` insert_relate:
$\forall\ p\ al\ k,$ priq $p \rightarrow$ Abs $p\ al \rightarrow$ Abs (insert $k\ p$) (k::al).
Admitted.
☐

Exercise: 4 stars, optional (join_elems) `Theorem` join_elems:
$\forall\ p\ q\ c\ n,$
priq' $n\ p \rightarrow$
priq' $n\ q \rightarrow$
(c=Leaf $\lor$ pow2heap $n\ c$) $\rightarrow$
$\forall\ pe\ qe\ ce,$
priqueue_elems $p\ pe \rightarrow$
priqueue_elems $q\ qe \rightarrow$
tree_elems $c\ ce \rightarrow$

priqueue_elems (join p q c) (ce++pe++qe).

Admitted.

☐

Exercise: 2 stars, optional (merge_relate) `Theorem` merge_relate:

$\forall$ p q pl ql al,

priq p $\rightarrow$ priq q $\rightarrow$

Abs p pl $\rightarrow$ Abs q ql $\rightarrow$ Abs (merge p q) al $\rightarrow$

Permutation al (pl++ql).

`Proof`.

Admitted.

☐

Exercise: 5 stars, optional (delete_max_None_relate) `Theorem` delete_max_None_relate:

$\forall$ p, priq p $\rightarrow$ (Abs p nil $\leftrightarrow$ delete_max p = None).

Admitted.

☐

Exercise: 5 stars, optional (delete_max_Some_relate) `Theorem` delete_max_Some_relate:

$\forall$ (p q: priqueue) k (pl ql: list key), priq p $\rightarrow$

Abs p pl $\rightarrow$

delete_max p = Some (k,q) $\rightarrow$

Abs q ql $\rightarrow$

Permutation pl (k::ql) $\wedge$ Forall (ge k) ql.

Admitted.

☐

With the following line, we're done! We have demonstrated that Binomial Queues are a correct implementation of mergeable priority queues. That is, we have exhibited a `Module` BinomQueue that satisfies the `Module Type` PRIQUEUE.

`End` BinomQueue.

13.6 Measurement.

Exercise: 5 stars, optional (binom_measurement) Adapt the program (but not necessarily the proof) to use Ocaml integers as keys, in the style shown in `Extract`. Write an ML program to exercise it with random inputs. Compare the runtime to the implementation from Priqueue, also adapted for Ocaml integers. ☐

Chapter 14

Library VFA.Decide

14.1 Decide: Programming with Decision Procedures

```
Set Warnings "-notation-overridden,-parsing".
From VFA Require Import Perm.
```

14.2 Using reflect to characterize decision procedures

Thus far in *Verified Functional Algorithms* we have been using

- propositions (`Prop`) such as $a<b$ (which is Notation for lt a b)
- booleans (**bool**) such as $a<?b$ (which is Notation for *ltb* a b).

```
Check Nat.lt. Check Nat.ltb.
```

The Perm chapter defined a tactic called *bdestruct* that does case analysis on (x <? y) while giving you hypotheses (above the line) of the form (x<y). This tactic is built using the **reflect** type and the blt_reflect theorem.

```
Print reflect.
```

```
Check blt_reflect.
```

The name **reflect** for this type is a reference to *computational reflection*, a technique in logic. One takes a logical formula, or proposition, or predicate, and designs a syntactic embedding of this formula as an "object value" in the logic. That is, *reflect* the formula back into the logic. Then one can design computations expressible inside the logic that manipulate these syntactic object values. Finally, one proves that the computations make transformations that are equivalent to derivations (or equivalences) in the logic.

The first use of computational reflection was by Goedel, in 1931: his syntactic embedding encoded formulas as natural numbers, a "Goedel numbering." The second and third uses of reflection were by Church and Turing, in 1936: they encoded (respectively) lambda-expressions and Turing machines.

In Coq it is easy to do reflection, because the Calculus of Inductive Constructions (CiC) has Inductive data types that can easily encode syntax trees. We could, for example, take some of our propositional operators such as *and*, *or*, and make an `Inductive` type that is an encoding of these, and build a computational reasoning system for boolean satisfiability.

But in this chapter I will show something much simpler. When reasoning about less-than comparisons on natural numbers, we have the advantage that **nat** already an inductive type; it is "pre-reflected," in some sense. (The same for **Z**, list, **bool**, etc.)

Now, let's examine how **reflect** expresses the coherence between lt and *ltb*. Suppose we have a value v whose type is **reflect** (3<7) (3<?7). What is v? Either it is

- ReflectT P (3<?7), where P is a proof of 3<7, and 3<?7 is true, or
- ReflectF Q (3<?7), where Q is a proof of ~(3<7), and 3<?7 is false.

In the case of 3,7, we are well advised to use ReflectT, because (3<?7) cannot match the false required by ReflectF.

```
Goal (3<?7 = true). Proof. reflexivity. Qed.
```

So v cannot be ReflectF Q (3<?7) for any Q, because that would not type-check. Now, the next question: must there exist a value of type **reflect** (3<7) (3<?7) ? The answer is yes; that is the blt_reflect theorem. The result of `Check` blt_reflect, above, says that for any x,y, there does exist a value (blt_reflect x y) whose type is exactly **reflect** $(x<y)(x<?y)$. So let's look at that value! That is, examine what H, and P, and Q are equal to at "Case 1" and "Case 2":

```
Theorem three_less_seven_1: 3<7.
Proof.
assert (H := blt_reflect 3 7).
remember (3<?7) as b.
destruct H as [P|Q] eqn:?.
×
apply P.
×
compute in Heqb.
inversion Heqb.
Qed.
```

Here is another proof that uses `inversion` instead of `destruct`. The ReflectF case is eliminated automatically by `inversion` because 3<?7 does not match false.

```
Theorem three_less_seven_2: 3<7.
Proof.
assert (H := blt_reflect 3 7).
inversion H as [P|Q].
```

```
apply P.
Qed.
```

The **reflect** inductive data type is a way of relating a *decision procedure* (a function from X to **bool**) with a predicate (a function from X to `Prop`). The convenience of **reflect**, in the verification of functional programs, is that we can do `destruct` (blt_reflect a b), which relates $a <? b$ (in the program) to the $a < b$ (in the proof). That's just how the *bdestruct* tactic works; you can go back to *Perm.v* and examine how it is implemented in the `Ltac` tactic-definition language.

14.3 Using sumbool to Characterize Decision Procedures

```
Module ScratchPad.
```

An alternate way to characterize decision procedures, widely used in Coq, is via the inductive type **sumbool**.

Suppose Q is a proposition, that is, Q: `Prop`. We say Q is *decidable* if there is an algorithm for computing a proof of Q or $\neg Q$. More generally, when P is a predicate (a function from some type T to `Prop`), we say P is decidable when $\forall$ x:T, *decidable*(P).

We represent this concept in Coq by an inductive datatype:

```
Inductive sumbool (A B : Prop) : Set :=
 | left : A → sumbool A B
 | right : B → sumbool A B.
```

Let's consider **sumbool** applied to two propositions:

```
Definition t1 := sumbool (3<7) (3>2).
Lemma less37: 3<7. Proof. omega. Qed.
Lemma greater23: 3>2. Proof. omega. Qed.

Definition v1a: t1 := left (3<7) (3>2) less37.
Definition v1b: t1 := right (3<7) (3>2) greater23.
```

A value of type **sumbool** (3<7) (3>2) is either one of:

- `left` applied to a proof of (3<7), or
- `right` applied to a proof of (3>2).

Now let's consider:

```
Definition t2 := sumbool (3<7) (2>3).
Definition v2a: t2 := left (3<7) (2>3) less37.
```

A value of type **sumbool** (3<7) (2>3) is either one of:

- `left` applied to a proof of (3<7), or

- `right` applied to a proof of (2>3).

But since there are no proofs of 2>3, only `left` values (such as v2a) exist. That's OK.

sumbool is in the Coq standard library, where there is `Notation` for it: the expression $\{A\}+\{B\}$ means **sumbool** A B.

```
Notation "{ A } + { B }" := (sumbool A B) : type_scope.
```

A very common use of **sumbool** is on a proposition and its negation. For example,

```
Definition t4 := ∀ a b, {a<b}+{~(a<b)}.
```

That expression, $\forall\ a\ b$, $\{a<b\}+\{\tilde{}(a<b)\}$, says that for any natural numbers a and b, either $a<b$ or $a\geq b$. But it is *more* than that! Because **sumbool** is an Inductive type with two constructors `left` and `right`, then given the $\{3<7\}+\{\tilde{}(3<7)\}$ you can pattern-match on it and learn *constructively* which thing is true.

```
Definition v3: {3<7}+{~(3<7)} := left _ _ less37.

Definition is_3_less_7: bool :=
 match v3 with
 | left _ _ _ ⇒ true
 | right _ _ _ ⇒ false
 end.

Eval compute in is_3_less_7.

Print t4.
```

Suppose there existed a value lt_dec of type t4. That would be a *decision procedure* for the less-than function on natural numbers. For any nats a and b, you could calculate lt_dec a b, which would be either `left` ... (if $a<b$ was provable) or `right` ... (if $\tilde{}(a<b)$ was provable).

Let's go ahead and implement lt_dec. We can base it on the function *ltb*: **nat** → **nat** → **bool** which calculates whether a is less than b, as a boolean. We already have a theorem that this function on booleans is related to the proposition $a<b$; that theorem is called blt_reflect.

```
Check blt_reflect.
```

It's not too hard to use blt_reflect to define lt_dec

```
Definition lt_dec (a: nat) (b: nat) : {a<b}+{~(a<b)} :=
match blt_reflect a b with
| ReflectT _ P ⇒ left (a < b) (¬ a < b) P
| ReflectF _ Q ⇒ right (a < b) (¬ a < b) Q
end.
```

Another, equivalent way to define lt_dec is to use definition-by-tactic:

```
Definition lt_dec' (a: nat) (b: nat) : {a<b}+{~(a<b)}.
  destruct (blt_reflect a b) as [P|Q]. left. apply P. right. apply Q.
Defined.
```

```
Print lt_dec.
Print lt_dec'.
Theorem lt_dec_equivalent: ∀ a b, lt_dec a b = lt_dec' a b.
Proof.
intros.
unfold lt_dec, lt_dec'.
reflexivity.
Qed.
```

Warning: these definitions of lt_dec are not as nice as the definition in the Coq standard library, because these are not fully computable. See the discussion below.

```
End ScratchPad.
```

14.3.1 sumbool in the Coq Standard Library

```
Module ScratchPad2.
Locate sumbool. Print sumbool.
```

The output of `Print` **sumbool** explains that the first two arguments of `left` and `right` are implicit. We use them as follows (notice that `left` has only one explicit argument P:

```
Definition lt_dec (a: nat) (b: nat) : {a<b}+{~(a<b)} :=
match blt_reflect a b with
| ReflectT _ P ⇒ left P
| ReflectF _ Q ⇒ right Q
end.
Definition le_dec (a: nat) (b: nat) : {a≤b}+{~(a≤b)} :=
match ble_reflect a b with
| ReflectT _ P ⇒ left P
| ReflectF _ Q ⇒ right Q
end.
```

Now, let's use le_dec directly in the implementation of insertion sort, without mentioning *ltb* at all.

```
Fixpoint insert (x:nat) (l: list nat) :=
  match l with
  | nil ⇒ x::nil
  | h::t ⇒ if le_dec x h then x::h::t else h :: insert x t
 end.
Fixpoint sort (l: list nat) : list nat :=
  match l with
  | nil ⇒ nil
  | h::t ⇒ insert h (sort t)
end.
```

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

Exercise: 2 stars (insert_sorted_le_dec) Lemma insert_sorted:

```
  ∀ a l, sorted l → sorted (insert a l).
Proof.
  intros a l H.
  induction H.
  - constructor.
  - unfold insert.
    destruct (le_dec a x) as [ Hle | Hgt].
```

Look at the proof state now. In the first subgoal, we have above the line, *Hle*: $a \leq x$. In the second subgoal, we have *Hgt*: $\neg\ (a < x)$. These are put there automatically by the `destruct (le_dec a x)`. Now, the rest of the proof can proceed as it did in *Sort.v*, but using `destruct (le_dec _ _)` instead of *bdestruct* (_ <=? _).

Admitted.

□

14.4 Decidability and Computability

Before studying the rest of this chapter, it is helpful to study the *ProofObjects* chapter of *Software Foundations volume 1* if you have not done so already.

A predicate P: T→Prop is *decidable* if there is a computable function f: T→**bool** such that, forall x:T, $f\ x$ = true ↔ $P\ x$. The second and most famous example of an *undecidable* predicate is the Halting Problem (Turing, 1936): T is the type of Turing-machine descriptions, and $P(x)$ is, Turing machine x halts. The first, and not as famous, example is due to Church, 1936 (six months earlier): test whether a lambda-expression has a normal form. In 1936-37, as a first-year PhD student before beginning his PhD thesis work, Turing proved these two problems are equivalent.

Classical logic contains the axiom $\forall\ P,\ P \vee \neg P$. This is not provable in core Coq, that is, in the bare Calculus of Inductive Constructions. But its negation is not provable either. You could add this axiom to Coq and the system would still be consistent (i.e., no way to prove **False**).

But $P \vee \neg P$ is a weaker statement than $\{P\}+\{\tilde{}P\}$, that is, **sumbool** P ($\tilde{}P$). From $\{P\}+\{\tilde{}P\}$ you can actually *calculate* or `compute` either `left` (x:P) or `right`(y: $\neg P$). From

$P \lor \neg P$ you cannot `compute` whether P is true. Yes, you can `destruct` it in a proof, but not in a calculation.

For most purposes its unnecessary to add the axiom $P \lor \neg P$ to Coq, because for specific predicates there's a specific way to prove $P \lor \neg P$ as a theorem. For example, less-than on natural numbers is decidable, and the existence of blt_reflect or lt_dec (as a theorem, not as an axiom) is a demonstration of that.

Furthermore, in this "book" we are interested in *algorithms*. An axiom $P \lor \neg P$ does not give us an algorithm to compute whether P is true. As you saw in the definition of insert above, we can use lt_dec not only as a theorem that either 3<7 or ~(3<7), we can use it as a function to compute whether 3<7. In Coq, you can't compute with axioms! Let's try it:

```
Axiom lt_dec_axiom_1: ∀ i j: nat, i<j ∨ ~(i<j).
```

Now, can we use this axiom to compute with?

That doesn't work, because an `if` statement requires an `Inductive` data type with exactly two constructors; but *lt_dec_axiom_1* i j has type $i<j \lor$ ~$(i<j)$, which is not Inductive. But let's try a different axiom:

```
Axiom lt_dec_axiom_2: ∀ i j: nat, {i<j} + {~(i<j)}.
```

```
Definition max_with_axiom (i j: nat) : nat :=
  if lt_dec_axiom_2 i j then j else i.
```

This typechecks, because *lt_dec_axiom_2* i j belongs to type **sumbool** $(i<j)$ (~$(i<j)$) (also written $\{i<j\}$ + $\{$~$(i<j)\}$), which does have two constructors.

Now, let's use this function:

```
Eval compute in max_with_axiom 3 7.
```

This `compute` didn't compute very much! Let's try to evaluate it using `unfold`:

```
Lemma prove_with_max_axiom: max_with_axiom 3 7 = 7.
Proof.
unfold max_with_axiom.
try reflexivity. destruct (lt_dec_axiom_2 3 7).
reflexivity.
contradiction n. omega.
Qed.
```

It is dangerous to add Axioms to Coq: if you add one that's inconsistent, then it leads to the ability to prove **False**. While that's a convenient way to get a lot of things proved, it's unsound; the proofs are useless.

The Axioms above, *lt_dec_axiom_1* and *lt_dec_axiom_2*, are safe enough: they are consistent. But they don't help in computation. Axioms are not useful here.

```
End ScratchPad2.
```

14.5 Opacity of Qed

This lemma prove_with_max_axiom turned out to be *provable*, but the proof could not go by *computation*. In contrast, let's use lt_dec, which was built without any axioms:

```
Lemma compute_with_lt_dec: (if ScratchPad2.lt_dec 3 7 then 7 else 3) = 7.
Proof.
compute.
Abort.
```

Unfortunately, even though blt_reflect was proved without any axioms, it is an *opaque theorem* (proved with Qed instead of with Defined), and one cannot compute with opaque theorems. Not only that, but it is proved with other opaque theorems such as *iff_sym* and *Nat.ltb_lt*. If we want to compute with an implementation of lt_dec built from blt_reflect, then we will have to rebuild blt_reflect without using Qed anywhere, only Defined.

Instead, let's use the version of lt_dec from the Coq standard library, which *is* carefully built without any opaque (Qed) theorems.

```
Lemma compute_with_StdLib_lt_dec: (if lt_dec 3 7 then 7 else 3) = 7.
Proof.
compute.
reflexivity.
Qed.
```

The Coq standard library has many decidability theorems. You can examine them by doing the following Search command. The results shown here are only for the subset of the library that's currently imported (by the Import commands above); there's even more out there.

```
Search ({_}+{¬_}).
```

The type of list_eq_dec is worth looking at. It says that if you have a decidable equality for an element type A, then list_eq_dec calculates for you a decidable equality for type list A. Try it out:

```
Definition list_nat_eq_dec:
    (∀ al bl : list nat, {al=bl}+{al≠bl}) :=
  list_eq_dec eq_nat_dec.

Eval compute in if list_nat_eq_dec [1;3;4] [1;4;3] then true else false.

Eval compute in if list_nat_eq_dec [1;3;4] [1;3;4] then true else false.
```

Exercise: 2 stars (list_nat_in) Use *in_dec* to build this function.

```
Definition list_nat_in: ∀ (i: nat) (al: list nat), {In i al}+{¬ In i al}
 . Admitted.

Example in_4_pi: (if list_nat_in 4 [3;1;4;1;5;9;2;6] then true else false) = true.
Proof.
```

`simpl`.

Admitted.

□

In general, beyond list_eq_dec and *in_dec*, one can construct a whole programmable calculus of decidability, using the programs-as-proof language of Coq. But is it a good idea? Read on!

14.6 Advantages and Disadvantages of reflect Versus sumbool

I have shown two ways to program decision procedures in Coq, one using **reflect** and the other using {_}+{~_}, i.e., **sumbool**.

- With **sumbool**, you define *two* things: the operator in `Prop` such as lt: **nat** → **nat** → `Prop` and the decidability "theorem" in **sumbool**, such as lt_dec: ∀ i j, {lt i j}+{~ lt i j}. I say "theorem" in quotes because it's not *just* a theorem, it's also a (nonopaque) computable function.
- With **reflect**, you define *three* things: the operator in `Prop`, the operator in **bool** (such as *ltb*: **nat** → **nat** → **bool**, and the theorem that relates them (such as *ltb_reflect*).

Defining three things seems like more work than defining two. But it may be easier and more efficient. Programming in **bool**, you may have more control over how your functions are implemented, you will have fewer difficult uses of dependent types, and you will run into fewer difficulties with opaque theorems.

However, among Coq programmers, **sumbool** seems to be more widely used, and it seems to have better support in the Coq standard library. So you may encounter it, and it is worth understanding what it does. Either of these two methods is a reasonable way of programming with proof.

Chapter 15

Library VFA.Color

15.1 Color: Graph Coloring

Required reading: , by Andrew W. Appel, 2016.

Suggested reading: , by Sandrine Blazy, Benoit Robillard, and Andrew W. Appel. ESOP 2010: 19th European Symposium on Programming, pp. 145-164, March 2010.

Coloring an undirected graph means, assigning a color to each node, so that any two nodes directly connected by an edge have different colors. The *chromatic number* of a graph is the minimum number of colors needed to color the graph. Graph coloring is NP-complete, so there is no polynomial-time algorithm; but we need to do it anyway, for applications such as register allocation in compilers. So therefore we often use incomplete algorithms: ones that work only on certain classes of graphs, or ones that color *most* but not all of the nodes. Those algorithms are often good enough for important applications.

In this chapter we will study Kempe's algorithm for K-coloring a graph. It was invented by Alfred Kempe in 1879, for use in his attempt to prove the four-color theorem (that every planar graph is 4-colorable). His 4-color proof had a bug; but his algorithm continues to be useful: a (major) variation of it was used in the successful 1976 proof of the 4-color theorem, and in 1979 Kempe's algorithm was adapted by Gregory Chaitin for application to register allocation. It is the Kempe-Chaitin algorithm that we'll prove here.

We implement a program to K-color an undirected graph, perhaps leaving some nodes uncolored. In a register-allocation problem, the graph nodes correspond to variables in a program, the colors correspond to registers, and the graph edges are interference constraints: two nodes connected by an edge cannot be assigned the same color. Nodes left uncolored are "spilled," that is, a register allocator would implement such nodes in memory locations instead of in registers. We desire to have as few uncolored nodes as possible, but this desire is not formally specified.

In this exercise we show a simple and unsophisticated algorithm; the program described by Blazy et al. (cited above) is more sophisticated in several ways, such as the use of "register coalescing" to get better results and the use of worklists to make it run faster.

Our algorithm does, at least, make use of efficient data structures for representing undirected graphs, adjacency sets, and so on.

15.2 Preliminaries: Representing Graphs

In the Trie chapter we saw how to represent efficient maps (lookup tables) where the keys are **positive** numbers in Coq. Those tries are implemented in the Coq standard library as FMaps, functional maps, and we will use them directly from the standard library. FMaps represent *partial functions*, that is, mapping keys to **option**(t) for whatever t.

We will also use FSets, efficient sets of keys; you can *think* of those as FMaps from keys to **unit**, where None means absent and Some *tt* means present; but their implementation is a bit more efficient.

```
Require Import List.
Require Import FSets. Require Import FMaps. From VFA Require Import Perm.
```

The nodes in our graph will be named by positive numbers. FSets and FMaps are interfaces for sets and maps over an element type. One instance is when the element type is **positive**, with a particular comparison operator corresponding to easy lookup in tries. The Coq module for this element type (with its total order) is *PositiveOrderedTypeBits*. We'll use E as an abbreviation for this module name.

```
Module E := PositiveOrderedTypeBits.
Print Module E.
Print E.t.
```

The `Module Type` *FSetInterface.S* gives the API of "functional sets." One instance of this, *PositiveSet*, has keys = positive numbers. We abbreviate this as `Module` S.

```
Module S <: FSetInterface.S := PositiveSet.
Print Module S.
Print S.elt.
```

And similarly for functional maps over positives

```
Module M <: FMapInterface.S := PositiveMap.
Print Module M.
Print M.E.
```

15.3 Lemmas About Sets and Maps

In order to reason about a graph coloring algorithm, we need to prove lemmas such as, "if you remove an element (one domain->range binding) from a finite map, then the result is a new finite map whose domain has fewer elements." (Duh!) But to prove this, we need to build up some definitions and lemmas. We start by importing some modules that have some already-proved properties of FMaps.

Module WF := WFACTS_FUN E M. Module WP := WPROPERTIES_FUN E M. Print Module *WF*.
Print Module *WP*.

Check E.lt.

E.lt is a comparison predicate on **positive** numbers. It is *not* the usual less-than operator; it is a different ordering that is more compatible with the order that a Positive Trie arranges its keys. In the application of certain lemmas about maps and sets, we will need the facts that *E.lt* is a **StrictOrder** (irreflexive and transitive) and respects a congruence over equality (is **Proper** for *eq* ==> *eq* ==> iff). As shown here, we just have to dig up these facts from a submodule of a submodule of a submodule of M.

Lemma lt_strict: **StrictOrder** E.lt.
Proof. exact M.ME.MO.IsTO.lt_strorder. Qed.

Lemma lt_proper: **Proper** (**eq** ==> **eq** ==> iff) E.lt.
Proof. exact M.ME.MO.IsTO.lt_compat. Qed.

The domain of a map is the set of elements that map to Some(_). To calculate the domain, we can use *M.fold*, an operation that comes with the FMaps abstract data type. It takes a map *m*, function *f* and base value *b*, and calculates *f x1 y1* (*f x2 y2* (*f x3 y3* (... (*f xn yn b*)...))), where (*xi*,*yi*) are the individual elements of *m*. That is, *M.find xi m* = Some *yi*, for each *i*.

So, to compute the domain, we just use an *f* function that adds *xi* to a set; mapping this over all the nodes will add all the keys in *m* to the set *S.empty*.

Definition Mdomain {*A*} (*m*: M.t *A*) : S.t :=
  M.fold (fun *n* _ *s* ⇒ S.add *n s*) *m* S.empty.

Example: Make a map from *node* (represented as **positive**) to *set of node* (represented as *S.t*), in which nodes 3,9,2 each map to the empty set, and no other nodes map to anything.

Definition example_map : M.t S.t :=
(M.add 3%*positive* S.empty
  (M.add 9%*positive* S.empty
   (M.add 2%*positive* S.empty (M.empty S.t)))).

Example domain_example_map:
  S.elements (Mdomain example_map) = [2;9;3]%*positive*.
Proof. compute. reflexivity. Qed.

15.3.1 equivlistA

Print equivlistA.

Suppose two lists *al*,*bl* both contain the same elements, not necessarily in the same order. That is, ∀ *x*:*A*, In *x al* ↔ In *x bl*. In fact from this definition you can see that *al* or *bl* might even have different numbers of repetitions of certain elements. Then we say the lists are "equivalent."

We can generalize this. Suppose instead of In *x al*, which says that the value *x* is in the list *al*, we use a different equivalence relation on that *A*. That is, **InA** *eqA x al* says that some element of *al* is *equivalent* to *x*, using the equivalence relation *eqA*. For example:

```
Definition same_mod_10 (i j: nat) := i mod 10 = j mod 10.
Example InA_example: InA same_mod_10 27 [3;17;2].
Proof. right. left. compute. reflexivity. Qed.
```

The predicate equivlistA *eqA al bl* says that lists *al* and *bl* have equivalent sets of elements, using the equivalence relation *eqA*. For example:

```
Example equivlistA_example: equivlistA same_mod_10 [3; 17] [7; 3; 27].
Proof.
split; intro.
inv H. right; left. auto.
inv H1. left. apply H0.
inv H0.
inv H. right; left. apply H1.
inv H1. left. apply H0.
inv H0. right. left. apply H1.
inv H1.
Qed.
```

15.3.2 SortA_equivlistA_eqlistA

Suppose two lists *al*,*bl* are "equivalent:" they contain the same set of elements (modulo an equivalence relation *eqA* on elements, perhaps in different orders, and perhaps with different numbers of repetitions). That is, suppose equivlistA *eqA al bl*.

And suppose list *al* is sorted, in some strict total order (respecting the same equivalence relation *eqA*). And suppose list *bl* is sorted. Then the lists must be *equal* (modulo *eqA*).

Just to make this easier to think about, suppose *eqA* is just ordinary equality. Then if *al* and *bl* contain the same set of elements (perhaps reordered), and each list is sorted (by *less-than*, not by *less-or-equal*), then they must be equal. Obviously.

That's what the theorem SortA_equivlistA_eqlistA says, in the Coq library:

```
Check SortA_equivlistA_eqlistA.
```

That is, suppose *eqA* is an equivalence relation on type *A*, that is, *eqA* is reflexive, symmetric, and transitive. And suppose *ltA* is a strict order, that is, irreflexive and transitive. And suppose *ltA* respects the equivalence relation, that is, if *eqA x x'* and *eqA y y'*, then *ltA x y* $\leftrightarrow$ *ltA x' y'*. THEN, if *l* is sorted (using the comparison *ltA*), and l' is sorted, and *l*,*l'* contain equivalent sets of elements, then *l*,*l'* must be equal lists, modulo the equivalence relation.

To make this easier to think about, let's use ordinary equality for eqA. We will be making sets and maps over the "node" type, *E.t*, but that's just type **positive**. Therefore, the equivalence *E.eq*: *E.t* $\rightarrow$ *E.t* $\rightarrow$ `Prop` is just the same as *eq*.

```
Goal E.t = positive. Proof. reflexivity. Qed.
Goal E.eq = @eq positive. Proof. reflexivity. Qed.
```

And therefore, **eqlistA** *E.eq al bl* means the same as *al*=*bl*.

```
Lemma eqlistA_Eeq_eq: ∀ al bl, eqlistA E.eq al bl ↔ al=bl.
Proof.
split; intro.
× induction H. reflexivity. unfold E.eq in H. subst. reflexivity.
× subst. induction bl. constructor. constructor.
  unfold E.eq. reflexivity. assumption.
Qed.
```

So now, the theorem: if *al* and *bl* are sorted, and contain "the same" elements, then they are equal:

```
Lemma SortE_equivlistE_eqlistE:
 ∀ al bl, Sorted E.lt al →
                     Sorted E.lt bl →
                     equivlistA E.eq al bl → eqlistA E.eq al bl.
Proof.
  apply SortA_equivlistA_eqlistA; auto.
  apply lt_strict.
  apply lt_proper.
Qed.
```

If list *l* is sorted, and you apply *List.filter* to remove the elements on which *f* is false, then the result is still sorted. Obviously.

```
Lemma filter_sortE: ∀ f l,
    Sorted E.lt l → Sorted E.lt (List.filter f l).
Proof.
  apply filter_sort with E.eq; auto.
  apply lt_strict.
  apply lt_proper.
Qed.
```

15.3.3 S.remove and S.elements

The FSets interface (and therefore our `Module` S) provides these two functions:

```
Check S.remove. Check S.elements.
```

In module S, of course, *S.elt*=**positive**, as these are sets of positive numbers.

Now, this relationship between *S.remove* and *S.elements* will soon be useful:

```
Lemma Sremove_elements: ∀ (i: E.t) (s: S.t),
  S.In i s →
     S.elements (S.remove i s) =
```

```
            List.filter (fun x ⇒ if E.eq_dec x i then false else true) (S.elements s).
Abort.
```

That is, if i is in the set s, then the elements of *S.remove* i s is the list that you get by filtering i out of *S.elements* s. Go ahead and prove it!

Exercise: 3 stars (Sremove_elements)

```
Lemma Proper_eq_eq:
  ∀ f, Proper (E.eq ==> @eq bool) f.
Proof.
unfold Proper. unfold respectful.
    Admitted.

Lemma Sremove_elements: ∀ (i: E.t) (s: S.t),
  S.In i s →
      S.elements (S.remove i s) =
            List.filter (fun x ⇒ if E.eq_dec x i then false else true) (S.elements s).
Proof.
intros.
apply eqlistA_Eeq_eq.
apply SortE_equivlistE_eqlistE.
×
admit.
×
admit.
×
intro j.
rewrite filter_InA; [ | apply Proper_eq_eq].
destruct (E.eq_dec j i).
 +
admit.
 +
admit.
    Admitted.
```

☐

15.3.4 Lists of (key,value) Pairs

The elements of a finite map from positives to type A (that is, the *M.elements* of a *M.t A*) is a list of pairs (**positive**×A).

```
Check M.elements.
```

Let's start with a little lemma about lists of pairs: Suppose l: list (**positive**×A). Then j is in map fst l iff there is some e such that (j,e) is in l.

Exercise: 2 stars (InA_map_fst_key) Lemma InA_map_fst_key:
$\forall$ A j l,
InA E.eq j (map (@fst M.E.t A) l) $\leftrightarrow$ $\exists$ e, **InA** (@M.eq_key_elt A) (j, e) l.

Admitted.

☐

Exercise: 3 stars (Sorted_lt_key) The function *M.lt_key* compares two elements of an *M.elements* list, that is, two pairs of type **positive**$\times A$, by just comparing their first elements using *E.lt*. Therefore, an elements list (of type list(**positive**$\times A$) is **Sorted** by *M.lt_key* iff its list-of-first-elements is **Sorted** by *E.lt*.

Lemma Sorted_lt_key:
$\forall$ A (al: **list** (**positive**$\times A$)),
Sorted (@M.lt_key A) al $\leftrightarrow$ **Sorted** E.lt (map (@fst **positive** A) al).
Proof.

Admitted.

☐

15.3.5 Cardinality

The *cardinality* of a set is the number of distinct elements. The cardinality of a finite map is, essentially, the cardinality of its domain set.

Exercise: 4 stars (cardinal_map) Lemma cardinal_map: $\forall$ A B (f: $A \rightarrow B$) g,
M.cardinal (M.map f g) = M.cardinal g.

Hint: To prove this theorem, I used these lemmas. You might find a different way.

Check M.cardinal_1.
Check M.elements_1.
Check M.elements_2.
Check M.elements_3.
Check map_length.
Check eqlistA_length.
Check SortE_equivlistE_eqlistE.
Check *InA_map_fst_key*.
Check WF.map_mapsto_iff.
Check *Sorted_lt_key*.

Admitted.

☐

Exercise: 4 stars (Sremove_cardinal_less) Lemma Sremove_cardinal_less: $\forall$ i s,
S.In i s $\rightarrow$ S.cardinal (S.remove i s) < S.cardinal s.
Proof.

```
intros.
repeat rewrite S.cardinal_1.
generalize (Sremove_elements _ _ H); intro.
rewrite H0; clear H0.
```

Admitted.

□

We have a lemma SortA_equivlistA_eqlistA that talks about arbitrary equivalence relations and arbitrary total-order relations (as long as they are compatible. Here is a specialization to a particular equivalence (*M.eq_key_elt*) and order (*M.lt_key*).

```
Lemma specialize_SortA_equivlistA_eqlistA:
  ∀ A al bl,
  Sorted (@M.lt_key A) al →
  Sorted (@M.lt_key A) bl →
  equivlistA (@M.eq_key_elt A) al bl →
  eqlistA (@M.eq_key_elt A) al bl.
Proof.
intros.
apply SortA_equivlistA_eqlistA with (@M.lt_key A); auto.
apply M.eqke_equiv.
apply M.ltk_strorder.
clear.
repeat intro.
unfold M.lt_key, M.eq_key_elt in *.
destruct H, H0. rewrite H,H0. split; auto.
Qed.
```

```
Lemma Proper_eq_key_elt:
 ∀ A,
   Proper (@M.eq_key_elt A ==> @M.eq_key_elt A ==> iff)
                (fun x y : E.t × A ⇒ E.lt (fst x) (fst y)).
Proof.
 repeat intro. destruct H,H0. rewrite H,H0. split; auto.
Qed.
```

Exercise: 4 stars (Mremove_elements) Lemma Mremove_elements: ∀ A i s,
  M.In i s →
    eqlistA (@M.eq_key_elt A) (M.elements (M.remove i s))
            (List.filter (fun x ⇒ if E.eq_dec (fst x) i then false else true) (M.elements s)).

```
Check specialize_SortA_equivlistA_eqlistA.
Check M.elements_1.
Check M.elements_2.
Check M.elements_3.
```

```
Check M.remove_1.
Check M.eqke_equiv.
Check M.ltk_strorder.
Check Proper_eq_key_elt.
Check filter_InA.
```

Admitted.

☐

Exercise: 3 stars (Mremove_cardinal_less) `Lemma Mremove_cardinal_less: ∀ A i (s: M.t A), M.In i s →`

```
      M.cardinal (M.remove i s) < M.cardinal s.
```

Look at the proof of Sremove_cardinal_less, if you succeeded in that, for an idea of how to do this one.

Admitted.

☐

Exercise: 2 stars (two_little_lemmas) `Lemma fold_right_rev_left:`

```
  ∀ (A B: Type) (f: A → B → A) (l: list B) (i: A),
  fold_left f l i = fold_right (fun x y ⇒ f y x) i (rev l).
```

Admitted.

```
Lemma Snot_in_empty: ∀ n, ¬ S.In n S.empty.
```

Admitted.

☐

Exercise: 3 stars (Sin_domain) `Lemma Sin_domain: ∀ A n (g: M.t A), S.In n (Mdomain g) ↔ M.In n g.`

This seems so obvious! But I didn't find a really simple proof of it.

Admitted.

☐

15.4 Now Begins the Graph Coloring Program

```
Definition node := E.t.
Definition nodeset := S.t.
Definition nodemap: Type → Type := M.t.
Definition graph := nodemap nodeset.

Definition adj (g: graph) (i: node) : nodeset :=
  match M.find i g with Some a ⇒ a | None ⇒ S.empty end.

Definition undirected (g: graph) :=
```

$\forall$ i j, S.In j (adj g i) $\rightarrow$ S.In i (adj g j).

Definition no_selfloop (g: graph) := $\forall$ i, $\neg$ S.In i (adj g i).

Definition nodes (g: graph) := Mdomain g.

Definition subset_nodes
(P: node $\rightarrow$ nodeset $\rightarrow$ **bool**)
(g: graph) :=
M.fold (fun n adj s $\Rightarrow$ if P n adj then S.add n s else s) g S.empty.

A node has "low degree" if the cardinality of its adjacency set is less than K

Definition low_deg (K: **nat**) (n: node) (adj: nodeset) : **bool** := S.cardinal adj <? K.

Definition remove_node (n: node) (g: graph) : graph :=
M.map (S.remove n) (M.remove n g).

15.4.1 Some Proofs in Support of Termination

We need to prove some lemmas related to the termination of the algorithm before we can actually define the Function.

Exercise: 3 stars (subset_nodes_sub) Lemma subset_nodes_sub: $\forall$ P g, S.Subset (subset_nodes P g) (nodes g).

Admitted.

☐

Exercise: 3 stars (select_terminates) Lemma select_terminates:
$\forall$ (K: **nat**) (g : graph) (n : S.elt),
S.choose (subset_nodes (low_deg K) g) = Some n $\rightarrow$
M.cardinal (remove_node n g) < M.cardinal g.

Admitted.

☐

15.4.2 The Rest of the Algorithm

Require Import Recdef.

```
Function select (K: nat) (g: graph) {measure M.cardinal g}: list node :=
  match S.choose (subset_nodes (low_deg K) g) with
  | Some n => n :: select K (remove_node n g)
  | None => nil
  end.
Proof. apply select_terminates.
Defined.
```

Definition coloring := M.t node.

```
Definition colors_of (f: coloring) (s: S.t) : S.t :=
   S.fold (fun n s ⇒ match M.find n f with Some c ⇒ S.add c s | None ⇒ s end) s S.empty.
```

```
Definition color1 (palette: S.t) (g: graph) (n: node) (f: coloring) : coloring :=
   match S.choose (S.diff palette (colors_of f (adj g n))) with
   | Some c ⇒ M.add n c f
   | None ⇒ f
   end.
```

```
Definition color (palette: S.t) (g: graph) : coloring :=
  fold_right (color1 palette g) (M.empty _) (select (S.cardinal palette) g).
```

15.5 Proof of Correctness of the Algorithm.

We want to show that any coloring produced by the color function actually respects the interference constraints. This property is called coloring_ok.

```
Definition coloring_ok (palette: S.t) (g: graph) (f: coloring) :=
 ∀ i j, S.In j (adj g i) →
     (∀ ci, M.find i f = Some ci → S.In ci palette) ∧
     (∀ ci cj, M.find i f = Some ci → M.find j f = Some cj → ci≠cj).
```

Exercise: 2 stars (adj_ext) Lemma adj_ext: ∀ g i j, E.eq i j → S.eq (adj g i) (adj g j).
Admitted.
☐

Exercise: 3 stars (in_colors_of_1) Lemma in_colors_of_1:
 ∀ i s f c, S.In i s → M.find i f = Some c → S.In c (colors_of f s).
Admitted.
☐

Exercise: 4 stars (color_correct) Theorem color_correct:
 ∀ palette g,
     no_selfloop g →
     undirected g →
     coloring_ok palette g (color palette g).
Admitted.
☐

That concludes the proof that the algorithm is correct.

15.6 Trying Out the Algorithm on an Actual Test Case

```
Local Open Scope positive.
```

```
Definition palette: S.t := fold_right S.add S.empty [1; 2; 3].

Definition add_edge (e: (E.t×E.t)) (g: graph) : graph :=
 M.add (fst e) (S.add (snd e) (adj g (fst e)))
  (M.add (snd e) (S.add (fst e) (adj g (snd e))) g).

Definition mk_graph (el: list (E.t×E.t)) :=
  fold_right add_edge (M.empty _) el.

Definition G :=
     mk_graph [ (5,6); (6,2); (5,2); (1,5); (1,2); (2,4); (1,4)].

Compute (S.elements (Mdomain G)).

Compute (M.elements (color palette G)).
```

That is our graph coloring: Node 4 is colored with color 1, node 2 with color 3, nodes 6 and 1 with 2, and node 5 with color 1.

Chapter 16

Library VFA.MapsTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From VFA Require Import Maps.
Parameter MISSING: Type.
```

```
Module Check.
```

```
Ltac check_type A B :=
match type of A with
| context[MISSING] ⇒ idtac "Missing:" A
| ?T ⇒ first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B ")"]
end.
```

```
Ltac print_manual_grade A :=
match eval compute in A with
| Some (pair ?S ?C) ⇒
idtac "Score:" S;
match eval compute in C with
| ""%string ⇒ idtac "Comment: None"
| _ ⇒ idtac "Comment:" C
end
| None ⇒
idtac "Score: Ungraded";
idtac "Comment: None"
end.
```

```
End Check.
```

```
From VFA Require Import Maps.
Import Check.
```

```
Goal True.
```

```
idtac "------------------- beq_idP --------------------".
idtac " ".
```

```
idtac "#> beq_idP".
idtac "Possible points: 2".
check_type @beq_idP ((∀ x y : nat, Bool.reflect (x = y) (PeanoNat.Nat.eqb x y))).
idtac "Assumptions:".
Abort.
Print Assumptions beq_idP.
Goal True.
idtac " ".

idtac "-------------------- t_update_same ---------------------".
idtac " ".

idtac "#> t_update_same".
idtac "Possible points: 2".
check_type @t_update_same (
(∀ (X : Type) (x : nat) (m : total_map X), @t_update X m x (m x) = m)).
idtac "Assumptions:".
Abort.
Print Assumptions t_update_same.
Goal True.
idtac " ".

idtac "-------------------- t_update_permute ---------------------".
idtac " ".

idtac "#> t_update_permute".
idtac "Possible points: 3".
check_type @t_update_permute (
(∀ (X : Type) (v1 v2 : X) (x1 x2 : nat) (m : total_map X),
 x2 ≠ x1 →
 @t_update X (@t_update X m x2 v2) x1 v1 =
 @t_update X (@t_update X m x1 v1) x2 v2)).
idtac "Assumptions:".
Abort.
Print Assumptions t_update_permute.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 7".
idtac "Max points - advanced: 7".
Abort.
```

Chapter 17

Library VFA.PrefaceTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From VFA Require Import Preface.
Parameter MISSING: Type.

Module Check.

Ltac check_type A B :=
match type of A with
| context[MISSING] => idtac "Missing:" A
| ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B ")"]
end.

Ltac print_manual_grade A :=
match eval compute in A with
| Some (pair ?S ?C) =>
idtac "Score:" S;
match eval compute in C with
| ""%string => idtac "Comment: None"
| _ => idtac "Comment:" C
end
| None =>
idtac "Score: Ungraded";
idtac "Comment: None"
end.

End Check.

From VFA Require Import Preface.
Import Check.

Goal True.

idtac " ".

idtac "Max points - standard: 0".
```

```
idtac "Max points - advanced: 0".
Abort.
```

Chapter 18

Library VFA.PermTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From VFA Require Import Perm.
Parameter MISSING: Type.

Module Check.

Ltac check_type A B :=
match type of A with
| context[MISSING] => idtac "Missing:" A
| ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B ")"]
end.

Ltac print_manual_grade A :=
match eval compute in A with
| Some (pair ?S ?C) =>
idtac "Score:" S;
match eval compute in C with
| ""%string => idtac "Comment: None"
| _ => idtac "Comment:" C
end
| None =>
idtac "Score: Ungraded";
idtac "Comment: None"
end.

End Check.

From VFA Require Import Perm.
Import Check.

Goal True.

idtac "-------------------  Permutation_properties ----------------".
idtac " ".
```

```
idtac "#> Manually graded: Exploration1.Permutation_properties".
idtac "Possible points: 2".
print_manual_grade Exploration1.manual_grade_for_Permutation_properties.
idtac " ".

idtac "-------------------  permut_example ----------------".
idtac " ".

idtac "#> Exploration1.permut_example".
idtac "Possible points: 3".
check_type @Exploration1.permut_example (
(∀ a b : list nat,
 @Permutation nat (5 :: 6 :: a ++ b) ((5 :: b) ++ 6 :: a ++ []))).
idtac "Assumptions:".
Abort.
Print Assumptions Exploration1.permut_example.
Goal True.
idtac " ".

idtac "-------------------  not_a_permutation ----------------".
idtac " ".

idtac "#> Exploration1.not_a_permutation".
idtac "Possible points: 1".
check_type @Exploration1.not_a_permutation ((¬ @Permutation nat [1; 1] [1; 2])).
idtac "Assumptions:".
Abort.
Print Assumptions Exploration1.not_a_permutation.
Goal True.
idtac " ".

idtac "-------------------  Forall_perm ----------------".
idtac " ".

idtac "#> Forall_perm".
idtac "Possible points: 2".
check_type @Forall_perm (
(∀ (A : Type) (f : A → Prop) (al bl : list A),
 @Permutation A al bl → @Forall A f al → @Forall A f bl)).
idtac "Assumptions:".
Abort.
Print Assumptions Forall_perm.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 8".
```

`idtac` "Max points - advanced: 8".
`Abort`.

Chapter 19

Library VFA.SortTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From VFA Require Import Sort.
Parameter MISSING: Type.

Module CHECK.

Ltac check_type A B :=
match type of A with
| context[MISSING] ⇒ idtac "Missing:" A
| ?T ⇒ first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B ")"]
end.

Ltac print_manual_grade A :=
match eval compute in A with
| Some (pair ?S ?C) ⇒
idtac "Score:" S;
match eval compute in C with
| ""%string ⇒ idtac "Comment: None"
| _ ⇒ idtac "Comment:" C
end
| None ⇒
idtac "Score: Ungraded";
idtac "Comment: None"
end.

End CHECK.

From VFA Require Import Sort.
Import Check.

Goal True.

idtac "------------------- insert_perm --------------------".
idtac " ".
```

```
idtac "#> insert_perm".
idtac "Possible points: 3".
check_type @insert_perm (
(∀ (x : nat) (l : list nat),
 @Permutation.Permutation nat (x :: l) (insert x l))).
idtac "Assumptions:".
Abort.
Print Assumptions insert_perm.
Goal True.
idtac " ".

idtac "------------------- sort_perm --------------------".
idtac " ".

idtac "#> sort_perm".
idtac "Possible points: 3".
check_type @sort_perm ((∀ l : list nat, @Permutation.Permutation nat l (sort l))).
idtac "Assumptions:".
Abort.
Print Assumptions sort_perm.
Goal True.
idtac " ".

idtac "------------------- insert_sorted --------------------".
idtac " ".

idtac "#> insert_sorted".
idtac "Possible points: 4".
check_type @insert_sorted (
(∀ (a : nat) (l : list nat), sorted l → sorted (insert a l))).
idtac "Assumptions:".
Abort.
Print Assumptions insert_sorted.
Goal True.
idtac " ".

idtac "------------------- sort_sorted --------------------".
idtac " ".

idtac "#> sort_sorted".
idtac "Possible points: 2".
check_type @sort_sorted ((∀ l : list nat, sorted (sort l))).
idtac "Assumptions:".
Abort.
Print Assumptions sort_sorted.
Goal True.
idtac " ".
```

`idtac` " ".

`idtac` "Max points - standard: 12".
`idtac` "Max points - advanced: 12".
`Abort`.

Chapter 20

Library VFA.MultisetTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From VFA Require Import Multiset.
Parameter MISSING: Type.

Module CHECK.

Ltac check_type A B :=
match type of A with
| context[MISSING] => idtac "Missing:" A
| ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B ")"]
end.

Ltac print_manual_grade A :=
match eval compute in A with
| Some (pair ?S ?C) =>
idtac "Score:" S;
match eval compute in C with
| ""%string => idtac "Comment: None"
| _ => idtac "Comment:" C
end
| None =>
idtac "Score: Ungraded";
idtac "Comment: None"
end.

End CHECK.

From VFA Require Import Multiset.
Import Check.

Goal True.

idtac "------------------- union_assoc --------------------".
idtac " ".
```

```
idtac "#> union_assoc".
idtac "Possible points: 1".
check_type @union_assoc (
(∀ a b c : multiset, union a (union b c) = union (union a b) c)).
idtac "Assumptions:".
Abort.
Print Assumptions union_assoc.
Goal True.
idtac " ".

idtac "-------------------  union_comm  ---------------".
idtac " ".

idtac "#> union_comm".
idtac "Possible points: 1".
check_type @union_comm ((∀ a b : multiset, union a b = union b a)).
idtac "Assumptions:".
Abort.
Print Assumptions union_comm.
Goal True.
idtac " ".

idtac "-------------------  insert_contents  ---------------".
idtac " ".

idtac "#> insert_contents".
idtac "Possible points: 3".
check_type @insert_contents (
(∀ (x : value) (l : list value),
 contents (x :: l) = contents (Sort.insert x l))).
idtac "Assumptions:".
Abort.
Print Assumptions insert_contents.
Goal True.
idtac " ".

idtac "-------------------  sort_contents  ---------------".
idtac " ".

idtac "#> sort_contents".
idtac "Possible points: 3".
check_type @sort_contents ((∀ l : list value, contents l = contents (Sort.sort l))).
idtac "Assumptions:".
Abort.
Print Assumptions sort_contents.
Goal True.
idtac " ".
```

```
idtac "------------------- permutations_vs_multiset --------------------".
idtac " ".

idtac "#> Manually graded: permutations_vs_multiset".
idtac "Possible points: 1".
print_manual_grade manual_grade_for_permutations_vs_multiset.
idtac " ".

idtac "------------------- perm_contents --------------------".
idtac " ".

idtac "#> perm_contents".
idtac "Possible points: 3".
check_type @perm_contents (
(∀ al bl : list nat,
 @Permutation.Permutation nat al bl → contents al = contents bl)).
idtac "Assumptions:".
Abort.
Print Assumptions perm_contents.
Goal True.
idtac " ".

idtac "------------------- delete_contents --------------------".
idtac " ".

idtac "#> delete_contents".
idtac "Possible points: 3".
check_type @delete_contents (
(∀ (v : value) (al : list value),
 contents (list_delete al v) = multiset_delete (contents al) v)).
idtac "Assumptions:".
Abort.
Print Assumptions delete_contents.
Goal True.
idtac " ".

idtac "------------------- contents_perm_aux --------------------".
idtac " ".

idtac "#> contents_perm_aux".
idtac "Possible points: 2".
check_type @contents_perm_aux (
(∀ (v : value) (b : multiset), empty = union (singleton v) b → False)).
idtac "Assumptions:".
Abort.
Print Assumptions contents_perm_aux.
Goal True.
idtac " ".
```

```
idtac "-------------------  contents_in  --------------------".
idtac " ".

idtac "#> contents_in".
idtac "Possible points: 2".
check_type @contents_in (
(∀ (a : value) (bl : list value),
 contents bl a > 0 → @List.In value a bl)).
idtac "Assumptions:".
Abort.
Print Assumptions contents_in.
Goal True.
idtac " ".

idtac "-------------------  in_perm_delete  --------------------".
idtac " ".

idtac "#> in_perm_delete".
idtac "Possible points: 2".
check_type @in_perm_delete (
(∀ (a : value) (bl : list value),
 @List.In value a bl →
 @Permutation.Permutation value (a :: list_delete bl a) bl)).
idtac "Assumptions:".
Abort.
Print Assumptions in_perm_delete.
Goal True.
idtac " ".

idtac "-------------------  contents_perm  --------------------".
idtac " ".

idtac "#> contents_perm".
idtac "Possible points: 4".
check_type @contents_perm (
(∀ al bl : list value,
 contents al = contents bl → @Permutation.Permutation value al bl)).
idtac "Assumptions:".
Abort.
Print Assumptions contents_perm.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 25".
idtac "Max points - advanced: 25".
Abort.
```

Chapter 21

Library VFA.SelectionTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From VFA Require Import Selection.
Parameter MISSING: Type.

Module Check.

Ltac check_type A B :=
match type of A with
| context[MISSING] ⇒ idtac "Missing:" A
| ?T ⇒ first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B ")"]
end.

Ltac print_manual_grade A :=
match eval compute in A with
| Some (pair ?S ?C) ⇒
idtac "Score:" S;
match eval compute in C with
| ""%string ⇒ idtac "Comment: None"
| _ ⇒ idtac "Comment:" C
end
| None ⇒
idtac "Score: Ungraded";
idtac "Comment: None"
end.

End Check.

From VFA Require Import Selection.
Import Check.

Goal True.

idtac "-------------------  select_perm  --------------------".
idtac " ".
```

```
idtac "#> select_perm".
idtac "Possible points: 3".
check_type @select_perm (
(∀ (x : nat) (l : list nat),
 let (y, r) := select x l in @Permutation.Permutation nat (x :: l) (y :: r))).
idtac "Assumptions:".
Abort.
Print Assumptions select_perm.
Goal True.
idtac " ".

idtac "-------------------  selection_sort_perm ----------------------".
idtac " ".

idtac "#> selection_sort_perm".
idtac "Possible points: 3".
check_type @selection_sort_perm (
(∀ l : list nat, @Permutation.Permutation nat l (selection_sort l))).
idtac "Assumptions:".
Abort.
Print Assumptions selection_sort_perm.
Goal True.
idtac " ".

idtac "-------------------  select_smallest ----------------------".
idtac " ".

idtac "#> select_smallest".
idtac "Possible points: 3".
check_type @select_smallest (
(∀ (x : nat) (al : list nat) (y : nat) (bl : list nat),
 select x al = (y, bl) → @Forall nat (fun z : nat ⇒ y ≤ z) bl)).
idtac "Assumptions:".
Abort.
Print Assumptions select_smallest.
Goal True.
idtac " ".

idtac "-------------------  selection_sort_sorted ----------------------".
idtac " ".

idtac "#> selection_sort_sorted".
idtac "Possible points: 3".
check_type @selection_sort_sorted ((∀ al : list nat, sorted (selection_sort al))).
idtac "Assumptions:".
Abort.
Print Assumptions selection_sort_sorted.
```

```
Goal True.
idtac " ".
idtac "-------------------  selsort'_perm  --------------------".
idtac " ".
idtac "#> selsort'_perm".
idtac "Possible points: 3".
check_type @selsort'_perm (
(∀ (n : nat) (l : list nat),
 @length nat l = n → @Permutation.Permutation nat l (selsort' l))).
idtac "Assumptions:".
Abort.
Print Assumptions selsort'_perm.
Goal True.
idtac " ".
idtac " ".
idtac "Max points - standard: 15".
idtac "Max points - advanced: 15".
Abort.
```

Chapter 22

Library VFA.SearchTreeTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From VFA Require Import SearchTree.
Parameter MISSING: Type.

Module CHECK.

Ltac check_type A B :=
match type of A with
| context[MISSING] ⇒ idtac "Missing:" A
| ?T ⇒ first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B ")"]
end.

Ltac print_manual_grade A :=
match eval compute in A with
| Some (pair ?S ?C) ⇒
idtac "Score:" S;
match eval compute in C with
| ""%string ⇒ idtac "Comment: None"
| _ ⇒ idtac "Comment:" C
end
| None ⇒
idtac "Score: Ungraded";
idtac "Comment: None"
end.

End CHECK.

From VFA Require Import SearchTree.
Import Check.

Goal True.

idtac "——————————- example_map ——————————".
idtac " ".
```

```
idtac "#> example_map".
idtac "Possible points: 2".
check_type @example_map ((∀ V : Type, V → V → V → V → Maps.total_map V)).
idtac "Assumptions:".
Abort.
Print Assumptions example_map.
Goal True.
idtac " ".

idtac "-------------------- check_example_map ---------------------".
idtac " ".

idtac "#> check_example_map".
idtac "Possible points: 3".
check_type @check_example_map (
(∀ (V : Type) (default v2 v4 v5 : V),
 Abs V default (example_tree V v2 v4 v5) (example_map V default v2 v4 v5))).
idtac "Assumptions:".
Abort.
Print Assumptions check_example_map.
Goal True.
idtac " ".

idtac "-------------------- lookup_relate ---------------------".
idtac " ".

idtac "#> lookup_relate".
idtac "Possible points: 3".
check_type @lookup_relate (
(∀ (V : Type) (default : V) (k : key) (t : tree V)
   (cts : Maps.total_map V),
 Abs V default t cts → lookup V default k t = cts k)).
idtac "Assumptions:".
Abort.
Print Assumptions lookup_relate.
Goal True.
idtac " ".

idtac "-------------------- insert_relate ---------------------".
idtac " ".

idtac "#> insert_relate".
idtac "Possible points: 4".
check_type @insert_relate (
(∀ (V : Type) (default : V) (k : key) (v : V)
   (t : tree V) (cts : Maps.total_map V),
 Abs V default t cts →
```

```
 Abs V default (insert V k v t) (@Maps.t_update V cts k v))).
idtac "Assumptions:".
Abort.
Print Assumptions insert_relate.
Goal True.
idtac " ".

idtac "-------------------  elements_relate_informal --------------------".
idtac " ".

idtac "#> Manually graded: elements_relate_informal".
idtac "Possible points: 3".
print_manual_grade manual_grade_for_elements_relate_informal.
idtac " ".

idtac "-------------------  not_elements_relate --------------------".
idtac " ".

idtac "#> not_elements_relate".
idtac "Possible points: 4".
check_type @not_elements_relate (
(∀ (V : Type) (default v : V),
 v ≠ default →
 ¬
 (∀ (t : tree V) (cts : Maps.total_map V),
  Abs V default t cts → list2map V default (elements V t) = cts))).
idtac "Assumptions:".
Abort.
Print Assumptions not_elements_relate.
Goal True.
idtac " ".

idtac "-------------------  empty_tree_SearchTree --------------------".
idtac " ".

idtac "#> empty_tree_SearchTree".
idtac "Possible points: 1".
check_type @empty_tree_SearchTree ((∀ V : Type, SearchTree V (empty_tree V))).
idtac "Assumptions:".
Abort.
Print Assumptions empty_tree_SearchTree.
Goal True.
idtac " ".

idtac "-------------------  insert_SearchTree --------------------".
idtac " ".

idtac "#> insert_SearchTree".
```

```
idtac "Possible points: 3".
check_type @insert_SearchTree (
(∀ (V : Type) (k : key) (v : V) (t : tree V),
 SearchTree V t → SearchTree V (insert V k v t))).
idtac "Assumptions:".
Abort.
Print Assumptions insert_SearchTree.
Goal True.
idtac " ".

idtac "-------------------- can_relate --------------------".
idtac " ".

idtac "#> can_relate".
idtac "Possible points: 2".
check_type @can_relate (
(∀ (V : Type) (default : V) (t : tree V),
 SearchTree V t → ∃ cts : Maps.total_map V, Abs V default t cts)).
idtac "Assumptions:".
Abort.
Print Assumptions can_relate.
Goal True.
idtac " ".

idtac "-------------------- unrealistically_strong_can_relate --------------------".
idtac " ".

idtac "#> unrealistically_strong_can_relate".
idtac "Possible points: 2".
check_type @unrealistically_strong_can_relate (
(∀ (V : Type) (default : V) (t : tree V),
 ∃ cts : Maps.total_map V, Abs V default t cts)).
idtac "Assumptions:".
Abort.
Print Assumptions unrealistically_strong_can_relate.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 27".
idtac "Max points - advanced: 27".
Abort.
```

Chapter 23

Library VFA.ADTTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From VFA Require Import ADT.
Parameter MISSING: Type.
```

```
Module Check.
```

```
Ltac check_type A B :=
match type of A with
| context[MISSING] ⇒ idtac "Missing:" A
| ?T ⇒ first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B ")"]
end.
```

```
Ltac print_manual_grade A :=
match eval compute in A with
| Some (pair ?S ?C) ⇒
idtac "Score:" S;
match eval compute in C with
| ""%string ⇒ idtac "Comment: None"
| _ ⇒ idtac "Comment:" C
end
| None ⇒
idtac "Score: Ungraded";
idtac "Comment: None"
end.
```

```
End Check.
```

```
From VFA Require Import ADT.
Import Check.
```

```
Goal True.
```

```
idtac "————————- TreeTable_gso ————————".
idtac " ".
```

```
idtac "#> TreeTable.gso".
idtac "Possible points: 3".
check_type @TreeTable.gso (
(∀ (j k : TreeTable.key) (v : TreeTable.V) (t : TreeTable.table),
 j ≠ k → TreeTable.get j (TreeTable.set k v t) = TreeTable.get j t)).
idtac "Assumptions:".
Abort.
Print Assumptions TreeTable.gso.
Goal True.
idtac " ".

idtac "-------------------  TreeTable_gso --------------------".
idtac " ".

idtac "#> TreeTable2.gso".
idtac "Possible points: 3".
check_type @TreeTable2.gso (
(∀ (j k : TreeTable2.key) (v : TreeTable2.V) (t : TreeTable2.table),
 j ≠ k → TreeTable2.get j (TreeTable2.set k v t) = TreeTable2.get j t)).
idtac "Assumptions:".
Abort.
Print Assumptions TreeTable2.gso.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 6".
idtac "Max points - advanced: 6".
Abort.
```

Chapter 24

Library VFA.ExtractTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From VFA Require Import Extract.
Parameter MISSING: Type.

Module Check.

Ltac check_type A B :=
match type of A with
| context[MISSING] ⇒ idtac "Missing:" A
| ?T ⇒ first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B ")"]
end.

Ltac print_manual_grade A :=
match eval compute in A with
| Some (pair ?S ?C) ⇒
idtac "Score:" S;
match eval compute in C with
| ""%string ⇒ idtac "Comment: None"
| _ ⇒ idtac "Comment:" C
end
| None ⇒
idtac "Score: Ungraded";
idtac "Comment: None"
end.

End Check.

From VFA Require Import Extract.
Import Check.

Goal True.

idtac "—————————- lookup_relate ——————————".
idtac " ".
```

```
idtac "#> SearchTree2.lookup_relate".
idtac "Possible points: 3".
check_type @SearchTree2.lookup_relate (
(∀ (V : Type) (default : V) (k : SearchTree2.key)
   (t : SearchTree2.tree V) (cts : IntMaps.total_map V),
 SearchTree2.Abs V default t cts →
 SearchTree2.lookup V default k t = cts (int2Z k))).
idtac "Assumptions:".
Abort.
Print Assumptions SearchTree2.lookup_relate.
Goal True.
idtac " ".
idtac "-------------------  insert_relate --------------------".
idtac " ".
idtac "#> SearchTree2.insert_relate".
idtac "Possible points: 3".
check_type @SearchTree2.insert_relate (
(∀ (V : Type) (default : V) (k : SearchTree2.key)
   (v : V) (t : SearchTree2.tree V) (cts : IntMaps.total_map V),
 SearchTree2.Abs V default t cts →
 SearchTree2.Abs V default (SearchTree2.insert V k v t)
   (@IntMaps.t_update V cts (int2Z k) v))).
idtac "Assumptions:".
Abort.
Print Assumptions SearchTree2.insert_relate.
Goal True.
idtac " ".
idtac "-------------------  unrealistically_strong_can_relate --------------------".
idtac " ".
idtac "#> SearchTree2.unrealistically_strong_can_relate".
idtac "Possible points: 1".
check_type @SearchTree2.unrealistically_strong_can_relate (
(∀ (V : Type) (default : V) (t : SearchTree2.tree V),
 ∃ cts : IntMaps.total_map V , SearchTree2.Abs V default t cts)).
idtac "Assumptions:".
Abort.
Print Assumptions SearchTree2.unrealistically_strong_can_relate.
Goal True.
idtac " ".
idtac " ".
idtac "Max points - standard: 7".
```

`idtac` "Max points - advanced: 7".
`Abort`.

Chapter 25

Library VFA.RedblackTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From VFA Require Import Redblack.
Parameter MISSING: Type.

Module Check.

Ltac check_type A B :=
match type of A with
| context[MISSING] => idtac "Missing:" A
| ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B ")"]
end.

Ltac print_manual_grade A :=
match eval compute in A with
| Some (pair ?S ?C) =>
idtac "Score:" S;
match eval compute in C with
| ""%string => idtac "Comment: None"
| _ => idtac "Comment:" C
end
| None =>
idtac "Score: Ungraded";
idtac "Comment: None"
end.

End Check.

From VFA Require Import Redblack.
Import Check.

Goal True.

idtac "------------------- ins_SearchTree --------------------".
idtac " ".
```

```
idtac "#> ins_SearchTree".
idtac "Possible points: 2".
check_type @ins_SearchTree (
(∀ (V : Type) (x : Extract.int) (vx : V) (s : tree V)
   (lo hi : BinNums.Z),
 BinInt.Z.le lo (Extract.int2Z x) →
 BinInt.Z.lt (Extract.int2Z x) hi →
 SearchTree' V lo s hi → SearchTree' V lo (ins V x vx s) hi)).
idtac "Assumptions:".
Abort.
Print Assumptions ins_SearchTree.
Goal True.
idtac " ".

idtac "-------------------- valid --------------------".
idtac " ".

idtac "#> empty_tree_SearchTree".
idtac "Possible points: 1".
check_type @empty_tree_SearchTree ((∀ V : Type, SearchTree V (empty_tree V))).
idtac "Assumptions:".
Abort.
Print Assumptions empty_tree_SearchTree.
Goal True.
idtac " ".

idtac "#> insert_SearchTree".
idtac "Possible points: 1".
check_type @insert_SearchTree (
(∀ (V : Type) (x : key) (vx : V) (s : tree V),
 SearchTree V s → SearchTree V (insert V x vx s))).
idtac "Assumptions:".
Abort.
Print Assumptions insert_SearchTree.
Goal True.
idtac " ".

idtac "-------------------- lookup_relate --------------------".
idtac " ".

idtac "#> lookup_relate".
idtac "Possible points: 3".
check_type @lookup_relate (
(∀ (V : Type) (default : V) (k : key) (t : tree V)
   (cts : Extract.IntMaps.total_map V),
 Abs V default t cts → lookup V default k t = cts (Extract.int2Z k))).
```

```
idtac "Assumptions:".
Abort.
Print Assumptions lookup_relate.
Goal True.
idtac " ".

idtac "-------------------- balance_relate ---------------------".
idtac " ".

idtac "#> balance_relate".
idtac "Possible points: 4".
check_type @balance_relate (
(∀ (V : Type) (default : V) (c : color) (l : tree V)
   (k : key) (vk : V) (r : tree V) (m : Extract.IntMaps.total_map V),
 SearchTree V (T V c l k vk r) →
 Abs V default (T V c l k vk r) m → Abs V default (balance V c l k vk r) m)).
idtac "Assumptions:".
Abort.
Print Assumptions balance_relate.
Goal True.
idtac " ".

idtac "-------------------- ins_relate ---------------------".
idtac " ".

idtac "#> ins_relate".
idtac "Possible points: 3".
check_type @ins_relate (
(∀ (V : Type) (default : V) (k : key) (v : V)
   (t : tree V) (cts : Extract.IntMaps.total_map V),
 SearchTree V t →
 Abs V default t cts →
 Abs V default (ins V k v t)
   (@Extract.IntMaps.t_update V cts (Extract.int2Z k) v))).
idtac "Assumptions:".
Abort.
Print Assumptions ins_relate.
Goal True.
idtac " ".

idtac "-------------------- is_redblack_properties ---------------------".
idtac " ".

idtac "#> is_redblack_toblack".
idtac "Possible points: 1".
check_type @is_redblack_toblack (
(∀ (V : Type) (s : tree V) (n : nat),
```

```
 is_redblack V s Red n → is_redblack V s Black n)).
idtac "Assumptions:".
Abort.
Print Assumptions is_redblack_toblack.
Goal True.
idtac " ".

idtac "#> makeblack_fiddle".
idtac "Possible points: 1".
check_type @makeblack_fiddle (
(∀ (V : Type) (s : tree V) (n : nat),
 is_redblack V s Black n →
 ∃ n0 : nat, is_redblack V (makeBlack V s) Red n0)).
idtac "Assumptions:".
Abort.
Print Assumptions makeblack_fiddle.
Goal True.
idtac " ".

idtac "#> ins_is_redblack".
idtac "Possible points: 1".
check_type @ins_is_redblack (
(∀ (V : Type) (x : key) (vx : V) (s : tree V) (n : nat),
 (is_redblack V s Black n → nearly_redblack V (ins V x vx s) n) ∧
 (is_redblack V s Red n → is_redblack V (ins V x vx s) Black n))).
idtac "Assumptions:".
Abort.
Print Assumptions ins_is_redblack.
Goal True.
idtac " ".

idtac "#> insert_is_redblack".
idtac "Possible points: 1".
check_type @insert_is_redblack (
(∀ (V : Type) (x : key) (xv : V) (s : tree V) (n : nat),
 is_redblack V s Red n →
 ∃ n' : nat, is_redblack V (insert V x xv s) Red n')).
idtac "Assumptions:".
Abort.
Print Assumptions insert_is_redblack.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 18".
```

`idtac` "Max points - advanced: 18".
`Abort`.

Chapter 26

Library VFA.TrieTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From VFA Require Import Trie.
Parameter MISSING: Type.

Module Check.

Ltac check_type A B :=
match type of A with
| context[MISSING] ⇒ idtac "Missing:" A
| ?T ⇒ first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B ")"]
end.

Ltac print_manual_grade A :=
match eval compute in A with
| Some (pair ?S ?C) ⇒
idtac "Score:" S;
match eval compute in C with
| ""%string ⇒ idtac "Comment: None"
| _ ⇒ idtac "Comment:" C
end
| None ⇒
idtac "Score: Ungraded";
idtac "Comment: None"
end.

End Check.

From VFA Require Import Trie.
Import Check.

Goal True.

idtac "------------------- succ_correct --------------------".
idtac " ".
```

```
idtac "#> Integers.succ_correct".
idtac "Possible points: 2".
check_type @Integers.succ_correct (
(∀ p : Integers.positive,
 Integers.positive2nat (Integers.succ p) = S (Integers.positive2nat p))).
idtac "Assumptions:".
Abort.
Print Assumptions Integers.succ_correct.
Goal True.
idtac " ".

idtac "-------------------- addc_correct ---------------------".
idtac " ".

idtac "#> Integers.addc_correct".
idtac "Possible points: 3".
check_type @Integers.addc_correct (
(∀ (c : bool) (p q : Integers.positive),
 Integers.positive2nat (Integers.addc c p q) =
 (if c then 1 else 0) + Integers.positive2nat p + Integers.positive2nat q)).
idtac "Assumptions:".
Abort.
Print Assumptions Integers.addc_correct.
Goal True.
idtac " ".

idtac "-------------------- compare_correct ---------------------".
idtac " ".

idtac "#> Integers.compare_correct".
idtac "Possible points: 5".
check_type @Integers.compare_correct (
(∀ x y : Integers.positive,
 match Integers.compare x y with
 | Integers.Eq ⇒ Integers.positive2nat x = Integers.positive2nat y
 | Integers.Lt ⇒ Integers.positive2nat x < Integers.positive2nat y
 | Integers.Gt ⇒ Integers.positive2nat x > Integers.positive2nat y
 end)).
idtac "Assumptions:".
Abort.
Print Assumptions Integers.compare_correct.
Goal True.
idtac " ".

idtac "-------------------- successor_of_Z_constant_time ---------------------".
idtac " ".
```

```
idtac "#> Manually graded: successor_of_Z_constant_time".
idtac "Possible points: 2".
print_manual_grade manual_grade_for_successor_of_Z_constant_time.
idtac " ".

idtac "-------------------  look_leaf ---------------------".
idtac " ".

idtac "#> look_leaf".
idtac "Possible points: 1".
check_type @look_leaf (
(forall (A : Type) (a : A) (j : BinNums.positive), @look A a j (@Leaf A) = a)).
idtac "Assumptions:".
Abort.
Print Assumptions look_leaf.
Goal True.
idtac " ".

idtac "-------------------  look_ins_same ---------------------".
idtac " ".

idtac "#> look_ins_same".
idtac "Possible points: 2".
check_type @look_ins_same (
(forall (A : Type) (a : A) (k : BinNums.positive) (v : A) (t : trie A),
 @look A a k (@ins A a k v t) = v)).
idtac "Assumptions:".
Abort.
Print Assumptions look_ins_same.
Goal True.
idtac " ".

idtac "-------------------  look_ins_same ---------------------".
idtac " ".

idtac "#> look_ins_same".
idtac "Possible points: 3".
check_type @look_ins_same (
(forall (A : Type) (a : A) (k : BinNums.positive) (v : A) (t : trie A),
 @look A a k (@ins A a k v t) = v)).
idtac "Assumptions:".
Abort.
Print Assumptions look_ins_same.
Goal True.
idtac " ".

idtac "-------------------  pos2nat_bijective ---------------------".
idtac " ".
```

```
idtac "#> pos2nat_injective".
idtac "Possible points: 1".
check_type @pos2nat_injective (
(∀ p q : BinNums.positive, pos2nat p = pos2nat q → p = q)).
idtac "Assumptions:".
Abort.
Print Assumptions pos2nat_injective.
Goal True.
idtac " ".

idtac "#> nat2pos_injective".
idtac "Possible points: 1".
check_type @nat2pos_injective ((∀ i j : nat, nat2pos i = nat2pos j → i = j)).
idtac "Assumptions:".
Abort.
Print Assumptions nat2pos_injective.
Goal True.
idtac " ".

idtac "-------------------  is_trie --------------------".
idtac " ".

idtac "#> is_trie".
idtac "Possible points: 2".
check_type @is_trie ((∀ A : Type, trie_table A → Prop)).
idtac "Assumptions:".
Abort.
Print Assumptions is_trie.
Goal True.
idtac " ".

idtac "-------------------  empty_relate --------------------".
idtac " ".

idtac "#> empty_relate".
idtac "Possible points: 2".
check_type @empty_relate (
(∀ (A : Type) (default : A),
 @Abs A (@empty A default) (@Maps.t_empty A default))).
idtac "Assumptions:".
Abort.
Print Assumptions empty_relate.
Goal True.
idtac " ".

idtac "-------------------  lookup_relate --------------------".
idtac " ".
```

```
idtac "#> lookup_relate".
idtac "Possible points: 2".
check_type @lookup_relate (
(∀ (A : Type) (i : BinNums.positive) (t : trie_table A)
   (m : Maps.total_map A),
 @is_trie A t → @Abs A t m → @lookup A i t = m (pos2nat i))).
idtac "Assumptions:".
Abort.
Print Assumptions lookup_relate.
Goal True.
idtac " ".

idtac "-------------------  insert_relate ---------------------".
idtac " ".

idtac "#> insert_relate".
idtac "Possible points: 3".
check_type @insert_relate (
(∀ (A : Type) (k : BinNums.positive) (v : A)
   (t : trie_table A) (cts : Maps.total_map A),
 @is_trie A t →
 @Abs A t cts →
 @Abs A (@insert A k v t) (@Maps.t_update A cts (pos2nat k) v))).
idtac "Assumptions:".
Abort.
Print Assumptions insert_relate.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 29".
idtac "Max points - advanced: 29".
Abort.
```

Chapter 27

Library VFA.PriqueueTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From VFA Require Import Priqueue.
Parameter MISSING: Type.
```

```
Module Check.
```

```
Ltac check_type A B :=
match type of A with
| context[MISSING] ⇒ idtac "Missing:" A
| ?T ⇒ first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B ")"]
end.
```

```
Ltac print_manual_grade A :=
match eval compute in A with
| Some (pair ?S ?C) ⇒
idtac "Score:" S;
match eval compute in C with
| ""%string ⇒ idtac "Comment: None"
| _ ⇒ idtac "Comment:" C
end
| None ⇒
idtac "Score: Ungraded";
idtac "Comment: None"
end.
```

```
End Check.
```

```
From VFA Require Import Priqueue.
Import Check.
```

```
Goal True.
```

```
idtac "------------------- select_perm_and_friends --------------------".
idtac " ".
```

```
idtac "#> List_Priqueue.select_perm".
idtac "Possible points: 1".
check_type @List_Priqueue.select_perm (
(∀ (i : nat) (l : list nat),
 let (j, r) := List_Priqueue.select i l in
 @Permutation.Permutation nat (i :: l) (j :: r))).
idtac "Assumptions:".
Abort.
Print Assumptions List_Priqueue.select_perm.
Goal True.
idtac " ".

idtac "#> List_Priqueue.select_biggest_aux".
idtac "Possible points: 1".
check_type @List_Priqueue.select_biggest_aux (
(∀ (i : nat) (al : list nat) (j : nat) (bl : list nat),
 @List.Forall nat (fun x : nat ⇒ j ≥ x) bl →
 List_Priqueue.select i al = (j , bl) → j ≥ i)).
idtac "Assumptions:".
Abort.
Print Assumptions List_Priqueue.select_biggest_aux.
Goal True.
idtac " ".

idtac "#> List_Priqueue.select_biggest".
idtac "Possible points: 1".
check_type @List_Priqueue.select_biggest (
(∀ (i : nat) (al : list nat) (j : nat) (bl : list nat),
 List_Priqueue.select i al = (j , bl) →
 @List.Forall nat (fun x : nat ⇒ j ≥ x) bl)).
idtac "Assumptions:".
Abort.
Print Assumptions List_Priqueue.select_biggest.
Goal True.
idtac " ".

idtac "—————————- simple_priq_proofs ——————————".
idtac " ".

idtac "#> List_Priqueue.delete_max_None_relate".
idtac "Possible points: 0.5".
check_type @List_Priqueue.delete_max_None_relate (
(∀ p : List_Priqueue.priqueue,
 List_Priqueue.priq p →
 List_Priqueue.Abs p (@nil List_Priqueue.key) ↔
```

```
 List_Priqueue.delete_max p = @None (nat × list nat))).
idtac "Assumptions:".
Abort.
Print Assumptions List_Priqueue.delete_max_None_relate.
Goal True.
idtac " ".

idtac "#> List_Priqueue.delete_max_Some_relate".
idtac "Possible points: 1".
check_type @List_Priqueue.delete_max_Some_relate (
(∀ (p q : List_Priqueue.priqueue) (k : nat)
   (pl ql : list List_Priqueue.key),
 List_Priqueue.priq p →
 List_Priqueue.Abs p pl →
 List_Priqueue.delete_max p = @Some (nat × List_Priqueue.priqueue) (k, q) →
 List_Priqueue.Abs q ql →
 @Permutation.Permutation List_Priqueue.key pl (k :: ql) ∧
 @List.Forall nat (ge k) ql)).
idtac "Assumptions:".
Abort.
Print Assumptions List_Priqueue.delete_max_Some_relate.
Goal True.
idtac " ".

idtac "#> List_Priqueue.delete_max_Some_relate".
idtac "Possible points: 0.5".
check_type @List_Priqueue.delete_max_Some_relate (
(∀ (p q : List_Priqueue.priqueue) (k : nat)
   (pl ql : list List_Priqueue.key),
 List_Priqueue.priq p →
 List_Priqueue.Abs p pl →
 List_Priqueue.delete_max p = @Some (nat × List_Priqueue.priqueue) (k, q) →
 List_Priqueue.Abs q ql →
 @Permutation.Permutation List_Priqueue.key pl (k :: ql) ∧
 @List.Forall nat (ge k) ql)).
idtac "Assumptions:".
Abort.
Print Assumptions List_Priqueue.delete_max_Some_relate.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 5".
idtac "Max points - advanced: 5".
```

```
Abort.
```

Chapter 28

Library VFA.BinomTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From VFA Require Import Binom.
Parameter MISSING: Type.
```

```
Module Check.
```

```
Ltac check_type A B :=
match type of A with
| context[MISSING] ⇒ idtac "Missing:" A
| ?T ⇒ first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B ")"]
end.
```

```
Ltac print_manual_grade A :=
match eval compute in A with
| Some (pair ?S ?C) ⇒
idtac "Score:" S;
match eval compute in C with
| ""%string ⇒ idtac "Comment: None"
| _ ⇒ idtac "Comment:" C
end
| None ⇒
idtac "Score: Ungraded";
idtac "Comment: None"
end.
```

```
End Check.
```

```
From VFA Require Import Binom.
Import Check.
```

```
Goal True.
```

```
idtac "—————————- empty_priq ——————————".
idtac " ".
```

```
idtac "#> BinomQueue.empty_priq".
idtac "Possible points: 1".
check_type @BinomQueue.empty_priq ((BinomQueue.priq BinomQueue.empty)).
idtac "Assumptions:".
Abort.
Print Assumptions BinomQueue.empty_priq.
Goal True.
idtac " ".

idtac "-------------------  smash_valid --------------------".
idtac " ".

idtac "#> BinomQueue.smash_valid".
idtac "Possible points: 2".
check_type @BinomQueue.smash_valid (
(∀ (n : nat) (t u : BinomQueue.tree),
 BinomQueue.pow2heap n t →
 BinomQueue.pow2heap n u → BinomQueue.pow2heap (S n) (BinomQueue.smash t u))).
idtac "Assumptions:".
Abort.
Print Assumptions BinomQueue.smash_valid.
Goal True.
idtac " ".

idtac "-------------------  carry_valid --------------------".
idtac " ".

idtac "#> BinomQueue.carry_valid".
idtac "Possible points: 3".
check_type @BinomQueue.carry_valid (
(∀ (n : nat) (q : list BinomQueue.tree),
 BinomQueue.priq' n q →
 ∀ t : BinomQueue.tree,
 t = BinomQueue.Leaf ∨ BinomQueue.pow2heap n t →
 BinomQueue.priq' n (BinomQueue.carry q t))).
idtac "Assumptions:".
Abort.
Print Assumptions BinomQueue.carry_valid.
Goal True.
idtac " ".

idtac "-------------------  priqueue_elems --------------------".
idtac " ".

idtac "#> Manually graded: BinomQueue.priqueue_elems".
idtac "Possible points: 3".
print_manual_grade BinomQueue.manual_grade_for_priqueue_elems.
```

```
idtac " ".
idtac "-------------------- tree_elems_ext ---------------------".
idtac " ".
idtac "#> BinomQueue.tree_elems_ext".
idtac "Possible points: 2".
check_type @BinomQueue.tree_elems_ext (
(∀ (t : BinomQueue.tree) (e1 e2 : list BinomQueue.key),
 @Permutation.Permutation BinomQueue.key e1 e2 →
 BinomQueue.tree_elems t e1 → BinomQueue.tree_elems t e2)).
idtac "Assumptions:".
Abort.
Print Assumptions BinomQueue.tree_elems_ext.
Goal True.
idtac " ".
idtac "-------------------- tree_perm ---------------------".
idtac " ".
idtac "#> BinomQueue.tree_perm".
idtac "Possible points: 2".
check_type @BinomQueue.tree_perm (
(∀ (t : BinomQueue.tree) (e1 e2 : list BinomQueue.key),
 BinomQueue.tree_elems t e1 →
 BinomQueue.tree_elems t e2 → @Permutation.Permutation BinomQueue.key e1 e2)).
idtac "Assumptions:".
Abort.
Print Assumptions BinomQueue.tree_perm.
Goal True.
idtac " ".
idtac "-------------------- priqueue_elems_ext ---------------------".
idtac " ".
idtac "#> BinomQueue.priqueue_elems_ext".
idtac "Possible points: 2".
check_type @BinomQueue.priqueue_elems_ext (
(∀ (q : list BinomQueue.tree) (e1 e2 : list BinomQueue.key),
 @Permutation.Permutation BinomQueue.key e1 e2 →
 BinomQueue.priqueue_elems q e1 → BinomQueue.priqueue_elems q e2)).
idtac "Assumptions:".
Abort.
Print Assumptions BinomQueue.priqueue_elems_ext.
Goal True.
idtac " ".
idtac "-------------------- abs_perm ---------------------".
```

```
idtac " ".
idtac "#> BinomQueue.abs_perm".
idtac "Possible points: 2".
check_type @BinomQueue.abs_perm (
(∀ (p : BinomQueue.priqueue) (al bl : list BinomQueue.key),
 BinomQueue.priq p →
 BinomQueue.Abs p al →
 BinomQueue.Abs p bl → @Permutation.Permutation BinomQueue.key al bl)).
idtac "Assumptions:".
Abort.
Print Assumptions BinomQueue.abs_perm.
Goal True.
idtac " ".
idtac "-------------------- can_relate --------------------".
idtac " ".
idtac "#> BinomQueue.can_relate".
idtac "Possible points: 2".
check_type @BinomQueue.can_relate (
(∀ p : BinomQueue.priqueue,
 BinomQueue.priq p → ∃ al : list BinomQueue.key, BinomQueue.Abs p al)).
idtac "Assumptions:".
Abort.
Print Assumptions BinomQueue.can_relate.
Goal True.
idtac " ".
idtac "-------------------- empty_relate --------------------".
idtac " ".
idtac "#> BinomQueue.empty_relate".
idtac "Possible points: 1".
check_type @BinomQueue.empty_relate (
(BinomQueue.Abs BinomQueue.empty (@nil BinomQueue.key))).
idtac "Assumptions:".
Abort.
Print Assumptions BinomQueue.empty_relate.
Goal True.
idtac " ".
idtac "-------------------- smash_elems --------------------".
idtac " ".
idtac "#> BinomQueue.smash_elems".
idtac "Possible points: 3".
check_type @BinomQueue.smash_elems (
```

```
(∀ (n : nat) (t u : BinomQueue.tree) (bt bu : list BinomQueue.key),
 BinomQueue.pow2heap n t →
 BinomQueue.pow2heap n u →
 BinomQueue.tree_elems t bt →
 BinomQueue.tree_elems u bu →
 BinomQueue.tree_elems (BinomQueue.smash t u) (bt ++ bu))).
idtac "Assumptions:".
Abort.
Print Assumptions BinomQueue.smash_elems.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 23".
idtac "Max points - advanced: 23".
Abort.
```

Chapter 29

Library VFA.DecideTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From VFA Require Import Decide.
Parameter MISSING: Type.

Module Check.

Ltac check_type A B :=
match type of A with
| context[MISSING] => idtac "Missing:" A
| ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B ")"]
end.

Ltac print_manual_grade A :=
match eval compute in A with
| Some (pair ?S ?C) =>
idtac "Score:" S;
match eval compute in C with
| ""%string => idtac "Comment: None"
| _ => idtac "Comment:" C
end
| None =>
idtac "Score: Ungraded";
idtac "Comment: None"
end.

End Check.

From VFA Require Import Decide.
Import Check.

Goal True.

idtac "------------------- insert_sorted_le_dec -----------------".
idtac " ".
```

```
idtac "#> ScratchPad2.insert_sorted".
idtac "Possible points: 2".
check_type @ScratchPad2.insert_sorted (
(∀ (a : nat) (l : list nat),
 ScratchPad2.sorted l → ScratchPad2.sorted (ScratchPad2.insert a l))).
idtac "Assumptions:".
Abort.
Print Assumptions ScratchPad2.insert_sorted.
Goal True.
idtac " ".

idtac "-------------------  list_nat_in --------------------".
idtac " ".

idtac "#> list_nat_in".
idtac "Possible points: 2".
check_type @list_nat_in (
(∀ (i : nat) (al : list nat),
 {@List.In nat i al} + {¬ @List.In nat i al})).
idtac "Assumptions:".
Abort.
Print Assumptions list_nat_in.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 4".
idtac "Max points - advanced: 4".
Abort.
```

Chapter 30

Library VFA.ColorTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From VFA Require Import Color.
Parameter MISSING: Type.

Module Check.

Ltac check_type A B :=
match type of A with
| context[MISSING] ⇒ idtac "Missing:" A
| ?T ⇒ first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B ")"]
end.

Ltac print_manual_grade A :=
match eval compute in A with
| Some (pair ?S ?C) ⇒
idtac "Score:" S;
match eval compute in C with
| ""%string ⇒ idtac "Comment: None"
| _ ⇒ idtac "Comment:" C
end
| None ⇒
idtac "Score: Ungraded";
idtac "Comment: None"
end.

End Check.

From VFA Require Import Color.
Import Check.

Goal True.

idtac "———————- Sremove_elements ————————".
idtac " ".
```

```
idtac "#> Sremove_elements".
idtac "Possible points: 3".
check_type @Sremove_elements (
(∀ (i : E.t) (s : S.t),
 S.In i s →
 S.elements (S.remove i s) =
 @List.filter BinNums.positive
   (fun x : BinNums.positive ⇒ if WP.F.eq_dec x i then false else true)
   (S.elements s))).
idtac "Assumptions:".
Abort.
Print Assumptions Sremove_elements.
Goal True.
idtac " ".

idtac "-------------------- InA_map_fst_key --------------------".
idtac " ".

idtac "#> InA_map_fst_key".
idtac "Possible points: 2".
check_type @InA_map_fst_key (
(∀ (A : Type) (j : BinNums.positive) (l : list (M.E.t × A)),
 S.InL j (@List.map (M.E.t × A) M.E.t (@fst M.E.t A) l) ↔
 (∃ e : A, @SetoidList.InA (M.key × A) (@M.eq_key_elt A) (j, e) l))).
idtac "Assumptions:".
Abort.
Print Assumptions InA_map_fst_key.
Goal True.
idtac " ".

idtac "-------------------- Sorted_lt_key --------------------".
idtac " ".

idtac "#> Sorted_lt_key".
idtac "Possible points: 3".
check_type @Sorted_lt_key (
(∀ (A : Type) (al : list (BinNums.positive × A)),
 @Sorted.Sorted (M.key × A) (@M.lt_key A) al ↔
 @Sorted.Sorted BinNums.positive E.lt
   (@List.map (BinNums.positive × A) BinNums.positive
      (@fst BinNums.positive A) al))).
idtac "Assumptions:".
Abort.
Print Assumptions Sorted_lt_key.
Goal True.
```

```
idtac " ".
idtac "-------------------- cardinal_map ---------------------".
idtac " ".
idtac "#> cardinal_map".
idtac "Possible points: 4".
check_type @cardinal_map (
(∀ (A B : Type) (f : A → B) (g : M.t A),
 @M.cardinal B (@M.map A B f g) = @M.cardinal A g)).
idtac "Assumptions:".
Abort.
Print Assumptions cardinal_map.
Goal True.
idtac " ".
idtac "-------------------- Sremove_cardinal_less ---------------------".
idtac " ".
idtac "#> Sremove_cardinal_less".
idtac "Possible points: 4".
check_type @Sremove_cardinal_less (
(∀ (i : S.elt) (s : S.t),
 S.In i s → S.cardinal (S.remove i s) < S.cardinal s)).
idtac "Assumptions:".
Abort.
Print Assumptions Sremove_cardinal_less.
Goal True.
idtac " ".
idtac "-------------------- Mremove_elements ---------------------".
idtac " ".
idtac "#> Mremove_elements".
idtac "Possible points: 4".
check_type @Mremove_elements (
(∀ (A : Type) (i : M.key) (s : M.t A),
 @M.In A i s →
 @SetoidList.eqlistA (M.key × A) (@M.eq_key_elt A)
   (@M.elements A (@M.remove A i s))
   (@List.filter (BinNums.positive × A)
      (fun x : BinNums.positive × A ⇒
       if WP.F.eq_dec (@fst BinNums.positive A x) i then false else true)
      (@M.elements A s)))).
idtac "Assumptions:".
Abort.
Print Assumptions Mremove_elements.
```

```
Goal True.
idtac " ".
idtac "-------------------  Mremove_cardinal_less  ----------------".
idtac " ".

idtac "#> Mremove_cardinal_less".
idtac "Possible points: 3".
check_type @Mremove_cardinal_less (
(forall (A : Type) (i : M.key) (s : M.t A),
 @M.In A i s -> @M.cardinal A (@M.remove A i s) < @M.cardinal A s)).
idtac "Assumptions:".
Abort.
Print Assumptions Mremove_cardinal_less.
Goal True.
idtac " ".

idtac "-------------------  two_little_lemmas  ----------------".
idtac " ".

idtac "#> fold_right_rev_left".
idtac "Possible points: 1".
check_type @fold_right_rev_left (
(forall (A B : Type) (f : A -> B -> A) (l : list B) (i : A),
 @List.fold_left A B f l i =
 @List.fold_right A B (fun (x : B) (y : A) => f y x) i (@List.rev B l))).
idtac "Assumptions:".
Abort.
Print Assumptions fold_right_rev_left.
Goal True.
idtac " ".

idtac "#> Snot_in_empty".
idtac "Possible points: 1".
check_type @Snot_in_empty ((forall n : S.elt, ~ S.In n S.empty)).
idtac "Assumptions:".
Abort.
Print Assumptions Snot_in_empty.
Goal True.
idtac " ".

idtac "-------------------  Sin_domain  ----------------".
idtac " ".

idtac "#> Sin_domain".
idtac "Possible points: 3".
check_type @Sin_domain (
(forall (A : Type) (n : S.elt) (g : M.t A),
```

```
 S.In n (@Mdomain A g) ↔ @M.In A n g)).
idtac "Assumptions:".
Abort.
Print Assumptions Sin_domain.
Goal True.
idtac " ".

idtac "-------------------- subset_nodes_sub ---------------------".
idtac " ".

idtac "#> subset_nodes_sub".
idtac "Possible points: 3".
check_type @subset_nodes_sub (
(∀ (P : node → nodeset → bool) (g : graph),
 S.Subset (subset_nodes P g) (nodes g))).
idtac "Assumptions:".
Abort.
Print Assumptions subset_nodes_sub.
Goal True.
idtac " ".

idtac "-------------------- select_terminates ---------------------".
idtac " ".

idtac "#> select_terminates".
idtac "Possible points: 3".
check_type @select_terminates (
(∀ (K : nat) (g : graph) (n : S.elt),
 S.choose (subset_nodes (low_deg K) g) = @Some S.elt n →
 @M.cardinal nodeset (remove_node n g) < @M.cardinal nodeset g)).
idtac "Assumptions:".
Abort.
Print Assumptions select_terminates.
Goal True.
idtac " ".

idtac "-------------------- adj_ext ---------------------".
idtac " ".

idtac "#> adj_ext".
idtac "Possible points: 2".
check_type @adj_ext (
(∀ (g : graph) (i j : BinNums.positive),
 E.eq i j → S.eq (adj g i) (adj g j))).
idtac "Assumptions:".
Abort.
Print Assumptions adj_ext.
```

```
Goal True.
idtac " ".
idtac "-------------------- in_colors_of_1 ---------------------".
idtac " ".
idtac "#> in_colors_of_1".
idtac "Possible points: 3".
check_type @in_colors_of_1 (
(∀ (i : S.elt) (s : S.t) (f : M.t S.elt) (c : S.elt),
 S.In i s → @M.find S.elt i f = @Some S.elt c → S.In c (colors_of f s))).
idtac "Assumptions:".
Abort.
Print Assumptions in_colors_of_1.
Goal True.
idtac " ".
idtac "-------------------- color_correct ---------------------".
idtac " ".
idtac "#> color_correct".
idtac "Possible points: 4".
check_type @color_correct (
(∀ (palette : S.t) (g : graph),
 no_selfloop g → undirected g → coloring_ok palette g (color palette g))).
idtac "Assumptions:".
Abort.
Print Assumptions color_correct.
Goal True.
idtac " ".
idtac " ".
idtac "Max points - standard: 43".
idtac "Max points - advanced: 43".
Abort.
```