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 and the concrete notations for writing down maps are somewhat different.

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.

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

Documentation for the standard library can be found at https://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 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, standard, 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, standard, 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, standard, 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, standard, 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 *eqb_id*.

Exercise: 2 stars, standard (eqb_idP) Use the proof of *eqb_natP* in chapter *IndProp* as a template to prove the following:

```
Lemma eqb_idP : ∀ x y, reflect (x = y) (x =? y).
Proof.
  Admitted.
```

□

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

Exercise: 2 stars, standard (t_update_same) Using the example in chapter *IndProp* as a template, use eqb_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, standard, especially useful (t_update_permute) Use eqb_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 : forall (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.
```

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 -> `Extract` -> Redblack
with many optional chapters whose dependencies are,

- `Sort` -> Multiset or Selection or Decide
- SearchTree -> ADT
- 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 8.9.1 or later.

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 https://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.

2.3.7 Recommended Citation Format

If you want to refer to this volume in your own writing, please do so as follows:

@book {Appel:SF3, author = {Andrew W. Appel}, title = “Verified Functional Algorithms”, series = “Software Foundations”, volume = “3”, year = “2020”, publisher = “Electronic textbook”, note = {Version 1.4, \URLhttp://softwarefoundations.cis.upenn.edu }, }

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 Comparisons and Permutations

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 comparisons, and about whether two collections have the same contents. In this chapter, we introduce some techniques for reasoning about:

- less-than comparisons on natural numbers, and
- permutations (rearrangements of lists).

In later chapters, we'll apply these proof techniques to reasoning about algorithms and data structures.

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Import Strings.String. From Coq Require Export Bool.Bool.
From Coq Require Export Arith.Arith.
From Coq Require Export Arith.EqNat.
From Coq Require Export Omega.
From Coq Require Export Lists.List.
Export ListNotations.
From Coq Require Export Permutation.
```

3.2 The Less-Than Order on the Natural Numbers

In our proofs about searching and sorting algorithms, we often have to reason about the less-than order on natural numbers. greater-than. Recall that the Coq standard library contains both propositional and Boolean less-than operators on natural numbers. We write $x < y$ for the proposition that x is less than y:

```
Locate "_< _". Check lt : nat → nat → Prop.
```

And we write $x <? y$ for the computation that returns true or false depending on whether x is less than y:

```
Locate "_<? _". Check Nat.ltb : nat → nat → bool.
```

Operation $<$ is a reflection of $<?$, as discussed in Logic and *IndProp*. The *Nat* module has a theorem showing how they relate:

```
Check Nat.ltb_lt : ∀ n m : nat, (n <? m) = true ↔ n < m.
```

The *Nat* module contains a synonym for lt.

```
Print Nat.lt.
```

For unknown reasons, *Nat* does not define notations for $>?$ or $>=?$. So we define them here:

```
Notation "a >=? b" := (Nat.leb b a)
                                (at level 70) : nat_scope.
Notation "a >? b" := (Nat.ltb b a)
                               (at level 70) : nat_scope.
```

3.2.1 The Omega Tactic

Reasoning about inequalities by hand can be a little painful. Luckily, Coq provides a tactic called omega that is quite helpful.

```
Theorem omega_example1:
 ∀ i j k,
    i < j →
    ¬ (k - 3 ≤ j) →
   k > i.
Proof.
  intros.
```

The hard way to prove this is by hand.

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

```
Abort.

Theorem truncated_subtraction: ¬ (∀ k:nat, k > k - 3).
Proof.
  intros contra.
  specialize (contra 0).
  simpl in contra.
  inversion contra.
Qed.
```

Since subtraction is truncated, does omega_example1 actually hold? It does. 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.
```

That was tedious. Here's a much easier way:

```
Theorem omega_example2:
 ∀ i j k,
     i < j →
     ¬ (k - 3 ≤ j) →
    k > i.
Proof.
  intros.
  omega.
Qed.
```

Omega is a decision procedure invented in 1991 by William Pugh for integer linear programming (ILP). The `omega` tactic was made available by importing Coq.omega.Omega, at the beginning of the file. It is an implementation of Pugh's algorithm. The tactic works with

Coq 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 expressions such as $a \times b$ and $f\ x\ y$ as 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 $u > v \rightarrow u + 3 \geq v$. But it cannot reason about, e.g., multiplication.

```
Theorem omega_example_3 : ∀ (f : nat → nat → nat) a b x y,
    f x y > a × b → f x y + 3 ≥ a × b.
Proof.
  intros. omega.
Qed.
```

```
Theorem omega_example_4 : ∀ a b,
    a × b = b × a.
Proof.
  intros. Fail omega.
Abort.
```

The Omega algorithm is NP-complete, so we might expect that this tactic is exponential-time in the worst case. Indeed, if you have N equations, it could take 2^N time. But in the typical cases that result from reasoning about programs, `omega` is much faster than that.

3.3 Swapping

Consider trying to sort a list of natural numbers. As a small piece of a sorting algorithm, we might need to 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.
```

Applying `maybe_swap` twice should give the same result as applying it once. That is, `maybe_swap` is *idempotent*.

```
Theorem maybe_swap_idempotent: ∀ al,
    maybe_swap (maybe_swap al) = maybe_swap al.
Proof.
  intros [ | a [ | b al]]; simpl; try reflexivity.
```

```
  destruct (b <? a) eqn:Hb_lt_a; simpl.
  - destruct (a <? b) eqn:Ha_lt_b; simpl.
```

+ Now what? We have a contradiction in the hypotheses: it cannot hold that a is less than b and b is less than a. Unfortunately, omega cannot immediately show that for us, because it reasons about comparisons in Prop not **bool**. *Fail* omega.

```
Abort.
```

Of course we could finish the proof by reasoning directly about inequalities in **bool**. But this situation is going to occur repeatedly in our study of sorting.

Let's set up some machinery to enable using omega on boolean tests.

3.3.1 Reflection

The **reflect** type, defined in the standard library (and presented in *IndProp*), relates a proposition to a Boolean. That is, a value of type **reflect** P b contains a proof of P if b is true, or a proof of $\neg$ P if b is false.

```
Print reflect.
```

The standard library proves a theorem that says if P is provable whenever b = true is provable, then P reflects b.

```
Check iff_reflect : ∀ (P : Prop) (b : bool),
    P ↔ b = true → reflect P b.
```

Using that theorem, we can quickly prove that the propositional (in)equality operators are reflections of the Boolean operators.

```
Lemma eqb_reflect : ∀ x y, reflect (x = y) (x =? y).
Proof.
  intros x y. apply iff_reflect. symmetry.
  apply Nat.eqb_eq.
Qed.

Lemma ltb_reflect : ∀ x y, reflect (x < y) (x <? y).
Proof.
  intros x y. apply iff_reflect. symmetry.
  apply Nat.ltb_lt.
Qed.

Lemma leb_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 ltb_reflect "by hand":

```
Example reflect_example1: ∀ a,
    (if a <? 5 then a else 2) < 6.
Proof.
  intros a.
  assert (R: reflect (a < 5) (a <? 5)) by apply ltb_reflect.
  remember (a <? 5) as guard.
  destruct R as [H|H] eqn:HR.
  × omega.
  × omega.
Qed.
```

For the ReflectT constructor, the guard $a <? 5$ must be equal to true. The if expression in the goal has already been simplified to take advantage of that fact. Also, for ReflectT to have been used, there must be evidence H that $a < 5$ holds. From there, all that remains is to show $a < 5$ entails $a < 6$. The omega tactic, which is capable of automatically proving some theorems about inequalities, succeeds.

For the ReflectF constructor, the guard $a <? 5$ must be equal to false. So the if expression simplifies to $2 < 6$, which is immediately provable by omega.

A less didactic version of the above proof wouldn't do the assert and *remember*: we can directly skip to destruct.

```
Example reflect_example1': ∀ a,
    (if a <? 5 then a else 2) < 6.
Proof.
  intros a. destruct (ltb_reflect a 5); omega.
Qed.
```

But even that proof is a little unsatisfactory. The original expression, $a <? 5$, is not perfectly apparent from the expression ltb_reflect a 5 that we pass to destruct.

It would be nice to be able to just say something like destruct ($a <? 5$) and get the reflection "for free." That's what we'll engineer, next.

3.3.2 A Tactic for Boolean Destruction

We're now going to build a tactic that you'll want to *use*, but you won't need to understand the details of how to *build* it yourself.

Let's put several of these **reflect** lemmas into a Hint database. We call it *bdestruct*, because we'll use it in our boolean-destruction tactic:

```
Hint Resolve ltb_reflect leb_reflect eqb_reflect : bdestruct.
```

Here is the tactic, the body of which you do not need to understand. Invoking *bdestruct* on Boolean expression b does the same kind of reasoning we did above: reflection and destruction. It also attempts to simplify negations involving inequalities in hypotheses.

```
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]]].
```

This tactic makes quick, easy-to-read work of our running example.

```
Example reflect_example2: ∀ a,
    (if a <? 5 then a else 2) < 6.
Proof.
  intros.
  bdestruct (a <? 5);
  omega.
Qed.
```

3.3.3 Finishing the maybe_swap Proof

Now that we have *bdestruct*, we can finish the proof of maybe_swap's idempotence.

```
Theorem maybe_swap_idempotent: ∀ al,
    maybe_swap (maybe_swap al) = maybe_swap al.
Proof.
  intros [ | a [ | b al]]; simpl; try reflexivity.
  bdestruct (a >? b); simpl.
```

Note how $b < a$ is a hypothesis, rather than $b <?\ a = $ true.

```
  - bdestruct (b >? a); simpl.
```

+ omega can take care of the contradictory propositional inequalities.

```
      omega.
    + reflexivity.
  - bdestruct (a >? b); simpl.
    + omega.
    + reflexivity.
Qed.
```

When proving theorems about a program that uses Boolean comparisons, use *bdestruct* followed by omega, rather than destruct followed by application of various theorems about Boolean operators.

3.4 Permutations

Another useful fact about maybe_swap is that it doesn't add or remove elements from the list: it only reorders them. That is, the output list is a permutation of the input. List *al* is a *permutation* of list *bl* if the elements of *al* can be reordered to get the list *bl*. Note that reordering does not permit adding or removing duplicate elements.

Coq's Permutation library has an inductive definition of permutations.

```
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 our specifications of 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, but that's not obvious.) To gain confidence that we have the right specification, let's use it prove some properties that permutations ought to have.

Exercise: 2 stars, standard (Permutation_properties) Think of some desirable 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 TASK: 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 (nat×string) := None.
```

□

Let's use the permutation theorems 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.
```

Let's group [*u*;t;t;*e*;*r*] together on both sides. Tactic `change t with` *u* replaces `t` with *u*. Terms `t` and *u* must be *convertible*, here meaning that they evalute to the same term.

```
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]).
```

We don't actually need to know the list elements in [*u*;t;t;*e*;*r*]. Let's forget about them and just remember them as a variable named *utter*. `remember [u;t;t;e;r] as utter.`
`clear Hequtter.`

Likewise, let's group [*f*;*l*] and remember it as a variable. `change [f;l;y] with ([f;l]++[y]).`

remember [*f* ; *l*] as *fl*. clear *Heqfl*.

Next, let's cancel *fl* from both sides. In order to do that, we need to bring it to the beginning of each list. For the right list, that follows easily from the associativity of ++.

```
replace ((fl ++ utter) ++ [b ; y]) with (fl ++ utter ++ [b ; y])
  by apply app_assoc.
```

But for the left list, we can't just use associativity. Instead, we need to reason about permutations and use some library theorems. `apply perm_trans with (fl ++ [y] ++ ([b] ++ utter)).`

```
- replace (fl ++ [y] ++ [b] ++ utter) with ((fl ++ [y]) ++ [b] ++ utter).
  + apply Permutation_app_comm.
  + rewrite ← app_assoc. reflexivity.
```

- A library theorem will now help us cancel *fl*. `apply Permutation_app_head.`

Next let's cancel *utter*. `apply perm_trans with (utter ++ [y] ++ [b]).`

```
  + replace ([y] ++ [b] ++ utter) with (([y] ++ [b]) ++ utter).
    × apply Permutation_app_comm.
    × rewrite app_assoc. reflexivity.
  + apply Permutation_app_head.
```

Finally we're left with just *y* and *b*. `apply perm_swap.`

```
Qed.
```

That example illustrates a general method for proving permutations involving cons :: and append ++:

- Identify some portion appearing in both sides.
- 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.
- Use Permutation_app_head to cancel an appended head. You can also use perm_skip to cancel a single element.

Exercise: 3 stars, standard (permut_example) Use the permutation rules in the library to prove the following theorem. The following `Check` commands are a hint about useful lemmas. You don't need all of them, and depending on your approach you will find lemmas to be more useful than others. Use `Search Permutation` to find others, if you like.

```
Check perm_skip.
Check perm_trans.
Check Permutation_refl.
Check Permutation_app_comm.
Check app_assoc.
Check app_nil_r.
Check app_comm_cons.
```

```
Example permut_example: ∀ (a b: list nat),
  Permutation (5 :: 6 :: a ++ b) ((5 :: b) ++ (6 :: a ++ [])).
Proof.
    Admitted.
    □
```

Exercise: 2 stars, standard (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.
    □
```

3.4.1 Correctness of maybe_swap

Now we can prove that maybe_swap is a permutation: it reorders elements but does not add or remove any.

```
Theorem maybe_swap_perm: ∀ al,
  Permutation al (maybe_swap al).
Proof.
  unfold maybe_swap.
  destruct al as [ | a [ | b al]].
  - simpl. apply perm_nil.
  - apply Permutation_refl.
  - bdestruct (b <? a).
    + apply perm_swap.
    + apply Permutation_refl.
Qed.
```

And, we can prove that maybe_swap permutes elements such that the first is less than or equal to 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.
  -
    unfold maybe_swap.
    destruct al as [ | a [ | b al]]; simpl; auto.
    bdestruct (a >? b); simpl; omega.
Qed.
```

3.5 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: 3 stars, standard (Forall_perm) To close, we define a utility tactic and lemma. First, the tactic.

Coq's `inversion` *H* tactic is so good at extracting information from the hypothesis *H* that *H* sometimes 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` ? Tactic *inv* does just that.

```
Ltac inv H := inversion H; clear H; subst.
```

Second, the lemma. You will find *inv* useful in proving it.

Forall is Coq library's version of the `All` proposition defined in Logic, but defined as an inductive proposition rather than a fixpoint. Prove this lemma by induction. You will need to decide what to induct on: *al*, *bl*, Permutation *al bl*, and Forall *f al* are possibilities.

```
Theorem Forall_perm: ∀ {A} (f: A → Prop) al bl,
  Permutation al bl →
  Forall f al → Forall f bl.
Proof.
   Admitted.
```

□

Chapter 4

Library VFA.Sort

4.1 Sort: Insertion Sort

Sorting can be done in expected 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. It's certainly easier to implement – and to verify.

If you don't recall insertion sort or haven't seen it in a while, 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.

```
From VFA Require Import Perm.
```

4.2 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.

```
Fixpoint insert (i : nat) (l : list nat) :=
  match l with
  | [] ⇒ [i]
  | 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
  | [] ⇒ []
  | 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.
```

We won't analyze or prove anything about the efficiency of sort. Instead, we will verify its correctness: that it produces the correct output for a given input.

4.3 Specification of Correctness

A sorting algorithm must rearrange the elements into a list that is totally ordered. There are many ways we might express that idea formally in Coq. One is with an inductively-defined relation that says:

- The empty list is sorted.
    - Any single-element list is sorted.
    - For any two adjacent elements, they must be in the proper order.

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

This definition might not be the most obvious. Another definition, perhaps more familiar, might be: for any two elements of the list (regardless of whether they are adjacent), they should be in the proper order. Let's try formalizing that.

We can think in terms of indices into a list *lst*, and say: for any valid indices i and j, if $i < j$ then *index lst* $i \leq$ *index lst* j, where *index lst* n means the element of *lst* at index n. Unfortunately, formalizing this idea becomes messy, because any Coq implementing *index* must be total: it must return some result even if the index is out of range for the list. The Coq standard library contains two such functions:

```
Check nth : ∀ A : Type, nat → list A → A → A.
Check nth_error : ∀ A : Type, list A → nat → option A.
```

These two functions ensure totality in different ways:

- nth takes an additional argument of type A –a *default* value– to be returned if the index is out of range, whereas

- nth_error returns Some v if the index is in range and None

  - -an error– otherwise.

If we use nth, we must ensure that indices are in range:

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

The choice of default value, here 0, is unimportant, because it will never be returned for the i and j we pass.

If we use nth_error, we must add additional antecedents:

```
Definition sorted' (al : list nat) := ∀ i j iv jv,
  i < j →
  nth_error al i = Some iv →
  nth_error al j = Some jv →
  iv ≤ jv.
```

Here, the validity of i and j are implicit in the fact that we get Some results back from each call to nth_error.

All three definitions of sortedness are reasonable. In practice, sorted' is easier to work with than sorted'' because it doesn't need to mention the length function. And **sorted** is easiest, because it doesn't need to mention indices.

Using **sorted**, we specify what it means to be a correct sorting algorithm:

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

Function f is a correct sorting algorithm if f al is **sorted** and is a permutation of its input.

4.4 Proof of Correctness

In the following exercises, you will prove the correctness of insertion sort.

Exercise: 3 stars, standard (insert_sorted) Lemma insert_sorted:

```
  ∀ a l, sorted l → sorted (insert a l).
Proof.
  intros a l S. induction S; simpl.
  (* FILL IN HERE *) Admitted.
```

☐

Exercise: 2 stars, standard (sort_sorted) Using insert_sorted, prove that insertion sort makes a list sorted.

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

☐

Exercise: 3 stars, standard (insert_perm) The following lemma will be useful soon as a helper. Take advantage of helpful theorems from the Permutation library.

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

☐

Exercise: 3 stars, standard (sort_perm) Prove that sort is a permutation, using insert_perm.

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

☐

Exercise: 1 star, standard (insertion_sort_correct) Finish the proof of correctness!

```
Theorem insertion_sort_correct:
    is_a_sorting_algorithm sort.
Proof.
  Admitted.
```

☐

4.5 Validating the Specification (Advanced)

You can prove that a program satisfies a specification, but how can you prove you have the right specification? Actually, you cannot. The specification is an informal requirement in your mind. As Alan Perlis quipped, "One can't proceed from the informal to the formal by formal means."

But one way to build confidence in a specification is to state it in two different ways, then prove they are equivalent.

Exercise: 4 stars, advanced (sorted_sorted') Lemma sorted_sorted': $\forall$ *al*, **sorted** *al* $\rightarrow$ 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. Proof.

Admitted.

□

Exercise: 3 stars, advanced (sorted'_sorted) Lemma sorted'_sorted : $\forall$ *al*, sorted' *al* $\rightarrow$ **sorted** *al*.

Proof.

Here, you can't do induction on the sortedness of the list, because sorted' is not an inductive predicate. But the proof is not hard. *Admitted.*

□

4.6 Proving Correctness from the Alternative Spec (Optional)

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

Give it a try! The best proof we know of makes essential use of the auxiliary lemma *nth_default_insert*, so you may want to prove that first. And some other auxiliary lemmas may be needed too. But maybe you will find a simpler appraoch!

DO NOT USE sorted_sorted', sorted'_sorted, insert_sorted, or sort_sorted in these proofs. That would defeat the purpose!

Exercise: 5 stars, standard, optional (insert_sorted') Lemma nth_error_insert : $\forall$ *l a i iv*,

nth_error (insert *a l*) *i* = Some *iv* $\rightarrow$
a = *iv* $\lor$ $\exists$ *i'*, nth_error *l i'* = Some *iv*.

Proof.

Admitted.

Lemma insert_sorted':
$\forall$ *a l*, sorted' *l* $\rightarrow$ sorted' (insert *a l*).

Proof.

Admitted.

□

Theorem sort_sorted': $\forall$ *l*, sorted' (sort *l*).

Proof.

```
  induction l.
  - unfold sorted'. intros. destruct i; inv H0.
  - simpl. apply insert_sorted'. auto.
Qed.
```

If you complete the proofs above, you will note that the proof of insert_sorted is relatively easy compared to the proof of insert_sorted', even though **sorted** *al* $\leftrightarrow$ sorted' *al*. So, suppose someone asked you to prove sort_sorted'. Instead of proving it directly, it would be much easier to design predicate **sorted**, then prove sort_sorted and sorted_sorted'.

The moral of the story is therefore: *Different formulations of the functional specification can lead to great differences in the difficulty of the correctness proofs.*

Chapter 5

Library VFA.Multiset

5.1 Multiset: Insertion Sort Verified With Multisets

Our specification of **sorted** in `Sort` was based in part on permutations, which enabled us to express the idea that sorting rearranges list elements but does not add or remove any.

Another way to express that idea is to use multisets, aka bags. A *set* is like a list in which element order is irrelevant, and in which no duplicate elements are permitted. That is, an element can either be *in* the set or not in the set, but it can't be in the set multiple times. A *multiset* relaxes that restriction: an element can be in the multiset multiple times. The number of times the element occurs in the multiset is the element's *multiplicity*.

For example:

- {1, 2} is a set, and is the same as set {2, 1}.

1; 1; 2 is a list, and is different than list [2; 1; 1].

- {1, 1, 2} is a multiset and the same as multiset {2, 1, 1}.

In this chapter we'll explore using multisets to specify sortedness.

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

5.2 Multisets

We will represent multisets as functions: if m is a multiset, then $m\ n$ will be the multiplicity of n in m. Since we are sorting lists of natural numbers, the type of multisets would be **nat** $\rightarrow$ **nat**. The input is the value, the output is its multiplicity. To help avoid confusion between those two uses of **nat**, we'll introduce a synonym, value.

```
Definition value := nat.
```

```
Definition multiset := value → nat.
```

The empty multiset has multiplicity 0 for every value.

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

Multiset singleton v contains only v, and exactly once.

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

The union of two multisets is their *pointwise* sum.

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

Exercise: 1 star, standard (union_assoc) Prove that multiset union is associative.

To prove that one multiset equals another we use the axiom of functional extensionality, which was introduced in Logic. We begin the proof below by using Coq's tactic `extensionality`, which applies that axiom.

```
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, standard (union_comm) Prove that multiset union is commutative.

```
Lemma union_comm: ∀ a b : multiset,
   union a b = union b a.
Proof.
   Admitted.
```

□

Exercise: 2 stars, standard (union_swap) Prove that the multisets in a nested union can be swapped. You do not need `extensionality` if you use the previous two lemmas.

```
Lemma union_swap : ∀ a b c : multiset,
    union a (union b c) = union b (union a c).
Proof.
   Admitted.
```

□

Note that this is not an efficient implementation of multisets. We wouldn't want to use it for programs with high performance requirements. But we are using multisets for specifications, not for programs. We don't intend to build large multisets, only to use them in verifying algorithms such as insertion sort. So this inefficiency is not a problem.

5.3 Specification of Sorting

A sorting algorithm must rearrange the elements into a list that is totally ordered. Using multisets, we can restate that as: the algorithm must produce a list *with the same multiset of values*, and this list must be totally ordered. Let's formalize that idea.

The *contents* of a list are the elements it contains, without any notion of order. Function contents extracts the contents of a list as a multiset.

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

The insertion-sort program sort from Sort preserves the contents of a list. Here's an example of that:

```
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.
  repeat (destruct x; try reflexivity).
Qed.
```

A sorting algorithm must preserve contents and totally order the list.

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

That definition is similar to is_a_sorting_algorithm from Sort, except that we're now using contents instead of Permutation.

5.4 Verification of Insertion Sort

The following series of exercises will take you through a verification of insertion sort using multisets.

Exercise: 3 stars, standard (insert_contents) Prove that insertion sort's insert function produces the same contents as merely prepending the inserted element to the front of the list.

Proceed by induction. You do not need extensionality if you make use of the above lemmas about union.

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

□

Exercise: 2 stars, standard (sort_contents) Prove that insertion sort preserves contents. Proceed by induction. Make use of insert_contents.

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

□

Exercise: 1 star, standard (insertion_sort_correct) Finish the proof of correctness!

```
Theorem insertion_sort_correct :
  is_a_sorting_algorithm' sort.
Proof.
   Admitted.
```

□

Exercise: 1 star, standard (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 or find easier to think about?

- permutations
- multisets

Put an X in one box in each list.

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

□

5.5 Equivalence of Permutation and Multiset Specifications

We have developed two specifications of sorting, one based on permutations (is_a_sorting_algorithm) and one based on multisets (is_a_sorting_algorithm'). These two specifications are actually equivalent, which will be the final theorem in this chapter.

One reason for that equivalence is that permutations and multisets are closely related. We'll begin by proving:

Permutation al bl <-> contents al = contents bl

The forward direction is relatively easy, but the backward direction is surprisingly difficult.

5.5.1 The Forward Direction

Exercise: 3 stars, standard (perm_contents) The forward direction is the easier one. Proceed by induction on the evidence for Permutation *al bl*:

```
Lemma perm_contents: ∀ al bl : list nat,
  Permutation al bl → contents al = contents bl.
Proof.
  Admitted.
```

□

5.5.2 The Backward Direction (Advanced)

The backward direction is surprisingly difficult. This proof approach is due to Zhong Sheng Hu. The first three lemmas are used to prove the fourth one. Don't forget that union, singleton, and empty must be explicitly unfolded to access their definitions.

Exercise: 2 stars, advanced (contents_nil_inv)

```
Lemma contents_nil_inv : ∀ l, (∀ x, 0 = contents l x) → l = nil.
Proof.
  Admitted.
```

□

Exercise: 3 stars, advanced (contents_cons_inv)

```
Lemma contents_cons_inv : ∀ l x n,
  S n = contents l x →
  ∃ l1 l2,
    l = l1 ++ x :: l2
    ∧ contents (l1 ++ l2) x = n.
Proof.
  Admitted.
```

□

Exercise: 2 stars, advanced (contents_insert_other) Lemma contents_insert_other : ∀ *l1 l2 x y*,
    $y \neq x \rightarrow$ contents (*l1* ++ *x* :: *l2*) *y* = contents (*l1* ++ *l2*) *y*.

```
Proof.
  Admitted.
```

□

Exercise: 3 stars, advanced (contents_perm) Lemma contents_perm: ∀ *al bl*,
    contents *al* = contents *bl* → **Permutation** *al bl*.

```
Proof.
  intros al bl H0.
  assert (H: ∀ x, contents al x = contents bl x).
  { rewrite H0. auto. }
  clear H0.
  generalize dependent bl.
  Admitted.
```

□

5.5.3 The Main Theorem

With both directions proved, we can establish the correspondence between multisets and permutations.

Exercise: 1 star, standard (same_contents_iff_perm) Use contents_perm (even if you haven't proved it) and perm_contents to quickly prove the next theorem.

Theorem same_contents_iff_perm: ∀ *al bl*,
    contents *al* = contents *bl* ↔ **Permutation** *al bl*.

```
Proof.
  Admitted.
```

□

Therefore the two specifications are equivalent.

Exercise: 2 stars, standard (sort_specifications_equivalent) Theorem sort_specifications_equivalent:
∀ *sort*,
    is_a_sorting_algorithm *sort* ↔ is_a_sorting_algorithm' *sort*.

```
Proof.
  Admitted.
```

□

That means we can verify sorting algorithms using either permutations or multisets, whichever we find more convenient.

Chapter 6

Library VFA.BagPerm

6.1 BagPerm: Insertion Sort With Bags

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 *bags* (also called *multisets*), which we introduced in *Lists*. 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. Whereas the principal query on a set is whether a given element appears in it, the principal query on a bag is *how many* times a given element appears in it.

```
From Coq Require Import Strings.String. From Coq Require Import Setoid Morphisms.
From VFA Require Import Perm.
From VFA Require Import Sort.
```

To keep this chapter more self-contained, we restate the critical definitions from *Lists*.

```
Definition bag := list nat.

Fixpoint count (v:nat) (s:bag) : nat :=
  match s with
  | nil ⇒ 0
  | h :: t ⇒
      (if h =? v then 1 else 0) + count v t
  end.
```

We will say two bags are *equivalent* if they have the same number of copies of every possible element.

```
Definition bag_eqv (b1 b2: bag) : Prop :=
  ∀ n, count n b1 = count n b2.
```

Exercise: 2 stars, standard (bag_eqv_properties) Lemma bag_eqv_refl : ∀ b, bag_eqv b b.

Proof.
Admitted.

Lemma bag_eqv_sym: ∀ $b1$ $b2$, bag_eqv $b1$ $b2$ → bag_eqv $b2$ $b1$.
Proof.
Admitted.

Lemma bag_eqv_trans: ∀ $b1$ $b2$ $b3$, bag_eqv $b1$ $b2$ → bag_eqv $b2$ $b3$ → bag_eqv $b1$ $b3$.
Proof.
Admitted.

The following little lemma is handy in a couple of places.

Lemma bag_eqv_cons : ∀ x $b1$ $b2$, bag_eqv $b1$ $b2$ → bag_eqv (x::$b1$) (x::$b2$).
Proof.
Admitted.
□

6.2 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, bag_eqv al (f al) ∧ sorted (f al).

Exercise: 3 stars, standard (insert_bag) First, prove the auxiliary lemma insert_bag, which will be useful for proving sort_bag below. Your proof will be by induction.

Lemma insert_bag: ∀ x l, bag_eqv (x::l) (insert x l).
Proof.
Admitted.
□

Exercise: 2 stars, standard (sort_bag) Now prove that sort preserves bag contents.
Theorem sort_bag: ∀ l, bag_eqv l (sort l).
Admitted.
□

Now we wrap it all up.

Theorem insertion_sort_correct:
is_a_sorting_algorithm' sort.
Proof.
split. apply *sort_bag*. apply *sort_sorted*.
Qed.

Exercise: 1 star, standard (permutations_vs_multiset) Compare your proofs of insert_perm, sort_perm with your proofs of insert_bag, sort_bag. 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** (**nat**×**string**) := None.
□

6.3 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* ↔ bag_eqv *al bl*.

Exercise: 3 stars, standard (perm_bag) The forward direction is straighforward, by induction on the evidence for Permutation: `Lemma` perm_bag:
∀ *al bl* : **list nat**,
Permutation *al bl* → bag_eqv *al bl*.
Admitted.
□

The other direction, bag_eqv *al bl* → Permutation *al bl*, is surprisingly difficult. This proof approach is due to Zhong Sheng Hu. The first three lemmas are used to prove the fourth one.

Exercise: 2 stars, advanced (bag_nil_inv) `Lemma` bag_nil_inv : ∀ *b*, bag_eqv [] *b* → *b* = [].
`Proof.`
Admitted.
□

Exercise: 3 stars, advanced (bag_cons_inv) Lemma bag_cons_inv : ∀ l x n,
S n = count x l →
∃ $l1$ $l2$,
l = $l1$ ++ x :: $l2$
∧ count x ($l1$ ++ $l2$) = n.

Proof.
Admitted.
□

Exercise: 2 stars, advanced (count_insert_other) Lemma count_insert_other : ∀ $l1$ $l2$ x y,
$y \neq x$ → count y ($l1$ ++ x :: $l2$) = count y ($l1$ ++ $l2$).

Proof.
Admitted.
□

Exercise: 3 stars, advanced (bag_perm) Lemma bag_perm:
∀ al bl, bag_eqv al bl → Permutation al bl.

Proof.
Admitted.
□

6.4 The Main Theorem: Equivalence of Multisets and Permutations

```
Theorem bag_eqv_iff_perm:
  ∀ al bl, bag_eqv al bl ↔ Permutation al bl.
Proof.
  intros. split. apply bag_perm. apply perm_bag.
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 bag_eqv_iff_perm; auto.
Qed.
```

Date

Chapter 7

Library VFA.Selection

7.1 Selection: Selection Sort

If you don't recall selection sort or haven't seen it in a while, see Wikipedia or read any standard textbook; some suggestions can be found in `Sort`.

The specification for sorting algorithms we developed in `Sort` can also be used to verify selection sort. The selection-sort program itself is interesting, because writing it in Coq will cause us to explore a new technique for convincing Coq that a function terminates.

A couple of notes on efficiency:

- Selection sort, like insertion sort, runs in quadratic time. But selection sort typically makes many more comparisons than insertion sort, so insertion sort is usually preferable for sorting small inputs. Selection sort can beat insertion sort if the cost of swapping elements is vastly higher than the cost of comparing them, but that doesn't apply to functional lists.

- What you should really never use is bubble sort. "Bubble sort would be the wrong way to go." Everybody should know that! See this video for a definitive statement: https://www.youtube.com/watch?v=k4RRi_ntQc8&t=34

```
From VFA Require Import Perm.
Hint Constructors Permutation.
From Coq Require Export Lists.List.
```

7.2 The Selection-Sort Program

Selection sort on lists is more challenging to code in Coq than insertion sort was. First, we write a helper function to select the smallest element.

```
Fixpoint select (x: nat) (l: list nat) : nat * list nat :=
  match l with
```

```
  | [] => (x, [])
  | 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.
```

Selection sort should repeatedly extract the smallest element and make a list of the results. But the following attempted definition fails:

```
Fail Fixpoint selsort (l : list nat) : list nat :=
  match l with
  | [] => []
  | x :: r => let (y, r') := select x r
              in y :: selsort r'
  end.
```

Coq rejects selsort because it doesn't satisfy Coq's requirements for termination. The problem is that the recursive call in selsort is not *structurally decreasing*: the argument r' at the call site is not known to be a smaller part of the original input l. Indeed, select might not return such a list. For example, select 1 [0; 2] is (0, [1; 2], but [1; 2] is not a part of [0; 2].

There are severals ways to fix this problem. One programming pattern is to provide *fuel*: an extra argument that has no use in the algorithm except to bound the amount of recursion. The n argument, below, is the fuel. When it reaches 0, the recursion terminates.

```
Fixpoint selsort (l : list nat) (n : nat) : list nat :=
  match l, n with
  | _, O => []
  | [], _ => []
  | x :: r, S n' => let (y, r') := select x r
                    in y :: selsort r' n'
end.
```

If fuel runs out, we get the wrong output.

```
Example out_of_fuel: selsort [3;1;4;1;5] 3 ≠ [1;1;3;4;5].
Proof.
  simpl. intro. discriminate.
Qed.
```

Extra fuel isn't a problem though.

```
Example extra_fuel: selsort [3;1;4;1;5] 10 = [1;1;3;4;5].
Proof.
  simpl. reflexivity.
```

```
Qed.
```

The exact amount of fuel needed is the length of the input list. So that's how we define selection_sort:

```
Definition selection_sort (l : list nat) : list nat :=
  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.
```

7.3 Proof of Correctness

We begin by repeating from `Sort` the specification of a correct sorting algorithm: it rearranges the elements into a list that is totally ordered.

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

Hint Constructors sorted.

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

In the following exercises, you will prove that selection sort is a correct sorting algorithm. You might wish to keep track of the lemmas you have proved, so that you can spot places to use them later.

Depending on the path you have followed through *Software Foundations* it might have been a while since you have worked with pairs. Here's a brief reminder of how `destruct` can be used to break a pair apart into its components. A similar technique will be needed in many of the following proofs.

```
Example pairs_example : ∀ (a c x : nat) (b d l : list nat),
    (a, b) = (let (c, d) := select x l in (c, d)) →
    (a, b) = select x l.
Proof.
  intros. destruct (select x l) eqn:E. auto.
Qed.
```

Exercise: 3 stars, standard (select_perm) Prove that select returns a permutation of its input. Proceed by induction on l. The *inv* tactic defined at the end of Perm will be helpful.

```
Lemma select_perm: ∀ x l y r,
```

(y , r) = select x l → **Permutation** $(x :: l)$ $(y :: r)$.

```
Proof.
```

Admitted.

☐

Exercise: 3 stars, standard (selsort_perm) Prove that if you provide sufficient fuel, selsort produces a permutation. Proceed by induction on n.

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

```
Proof.
```

Admitted.

☐

Exercise: 1 star, standard (selection_sort_perm) Prove that selection_sort produces a permutation.

Lemma selection_sort_perm: ∀ l,
    Permutation l (selection_sort l).

```
Proof.
```

Admitted.

☐

Exercise: 2 stars, standard (select_rest_length) Prove that select returns a list that has the correct length. You can do this without induction if you make use of select_perm.

Lemma select_rest_length : ∀ x l y r,
    select x l = (y , r) → length l = length r.

```
Proof.
```

Admitted.

☐

Exercise: 3 stars, standard (select_fst_leq) Prove that the first component of select x _ is no bigger than x. Proceed by induction on al.

Lemma select_fst_leq: ∀ al bl x y,
    select x al = (y , bl) →
    $y \leq x$.

```
Proof.
```

Admitted.

☐

Exercise: 3 stars, standard (select_smallest) Prove that the first component of select _ _ is no bigger than any of the elements in the second component. To represent that concept of comparing an element to a list, we introduce a new notation:

```
Definition le_all x xs := Forall (fun y ⇒ x ≤ y) xs.
Infix "<=*" := le_all (at level 70, no associativity).
```

Proceed by induction on *al*.

```
Lemma select_smallest: ∀ al bl x y,
    select x al = (y , bl) →
    y <=* bl.
Proof.
  Admitted.
```

□

Exercise: 3 stars, standard (select_in) Prove that the element returned by select must be one of the elements in its input. Proceed by induction on *al*.

```
Lemma select_in : ∀ al bl x y,
    select x al = (y , bl) →
    In y (x :: al).
Proof.
  Admitted.
```

□

Exercise: 3 stars, standard (cons_of_small_maintains_sort) Prove that adding an element to the beginning of a selection-sorted list maintains sortedness, as long as the element is small enough and enough fuel is provided. Proceed by induction on *bl*.

```
Lemma cons_of_small_maintains_sort: ∀ bl y n,
    n = length bl →
    y <=* bl →
    sorted (selsort bl n) →
    sorted (y :: selsort bl n).
Proof.
  Admitted.
```

□

Exercise: 3 stars, standard (selsort_sorted) Prove that selsort produced a sorted list when given sufficient fuel. Proceed by induction on *n*. This proof will make use of a few previous lemmas.

```
Lemma selsort_sorted : ∀ n al,
    length al = n → sorted (selsort al n).
```

```
Proof.
  Admitted.
```

☐

Exercise: 1 star, standard (selection_sort_sorted) Prove that selection_sort produces a sorted list.

```
Lemma selection_sort_sorted : ∀ al,
    sorted (selection_sort al).
Proof.
  Admitted.
```

☐

Exercise: 1 star, standard (selection_sort_is_correct) Finish the proof of correctness!

```
Theorem selection_sort_is_correct :
  is_a_sorting_algorithm selection_sort.
Proof.
  Admitted.
```

☐

Exercise: 5 stars, advanced, optional (selection_sort_is_correct_multiset) Uncomment the next line, and prove the correctness of selection_sort using multisets instead of permutations. We haven't tried this yet! Send us your proof so we can add it as a solution.

☐

7.4 Recursive Functions That are Not Structurally Recursive

We used fuel above to create a structurally recursive version of selsort that Coq would accept as terminating. The amount of fuel decreased at each call, until it reached zero. Since the fuel argument was structurally decreasing, Coq accepted the definition. But it complicated the implementation of selsort and the proofs about it.

Coq provides an experimental command `Function` that implements a similar idea as fuel, but without requiring the function definition to be structurally recursive. Instead, the function is annotated with a *measure* that is decreasing at each recursive call. To activate this experimental command, we need to load a library.

```
Require Import Recdef.
```

Now we can add a `measure` annotation on the definition of selsort to tell Coq that each recursive call decreases the length of l:

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

The measure annotation takes two parameters, a measure function and an argument name. Above, the function is length and the argument is l. The function must return a **nat** when applied to the argument. Coq then challenges us to prove that length applied to l is actually decreasing at every recursive call.

```
Proof.
  intros.
  assert (Hperm: Permutation (x :: r) (y :: r')).
  { apply select_perm. auto. }
  apply Permutation_length in Hperm.
  inv Hperm. simpl. omega.
Defined.
```

The proof must end with Defined instead of Qed. That ensures the function's body can be used in computation. For example, the following unit test succeeds, but try changing Defined to Qed and see how it gets stuck.

```
Example selsort'_example : selsort' [3;1;4;1;5;9;2;6;5] = [1;1;2;3;4;5;5;6;9].
Proof. reflexivity. Qed.
```

The definition of selsort' is completed by the Function command using a helper function that it generates, selsort'_terminate. Neither of them is going to be useful to unfold in proofs:

```
Print selsort'.
Print selsort'_terminate.
```

Instead, anywhere you want to unfold or simplify selsort', you should now rewrite with selsort'_equation, which was automatically defined by the Function command:

```
Check selsort'_equation.
```

Exercise: 2 stars, standard (selsort'_perm) Hint: Follow the same strategy as selsort_perm. In our solution, there was only a one-line change.

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

□

Exercise: 5 stars, advanced, optional (selsort'_correct) Prove the correctness of selsort'. We haven't tried this yet! Send us your proof so we can add it as a solution.

□

Chapter 8

Library VFA.Merge

8.1 Merge: Merge Sort, With Specification and Proof of Correctness

```
From VFA Require Import Perm.
From VFA Require Import Sort.
From Coq Require Import Recdef.
```

Mergesort is a well-known sorting algorithm, normally presented as an imperative algorithm on arrays, that has worst-case O(n log n) execution time and requires O(n) auxiliary space.

The basic idea is simple: we divide the data to be sorted into two halves, recursively sort each of them, and then merge together the (sorted) results from each half:

```
mergesort xs =
  split xs into ys,zs;
  ys' = mergesort ys;
  zs' = mergesort zs;
  return (merge ys' zs')
```

(As usual, if you are unfamiliar with mergesort see Wikipedia or your favorite algorithms textbook.)

Mergesort on lists works essentially the same way: we split the original list into two halves, recursively sort each sublist, and then merge the two sublists together again. The only difference, compared to the imperative algorithm, is that splitting the list takes O(n) rather than O(1) time; however, that does not affect the asymptotic cost, since the merge step already takes O(n) anyhow.

8.1.1 Split and its properties

Let us try to write down the Gallina code for mergesort. The first step is to write a splitting function. There are several ways to do this, since the exact splitting method does not matter as long as the results are (roughly) equal in size. For example, if we know the length of the list, we could use that to split at the half-way point. But here is an attractive alternative, which simply alternates assigning the elements into left and right sublists:

```
Fixpoint split {X:Type} (l:list X) : (list X × list X) :=
  match l with
  | [] ⇒ ([],[])
  | [x] ⇒ ([x],[])
  | x1::x2::l' ⇒
    let (l1,l2) := split l' in
    (x1::l1,x2::l2)
  end.
```

Note: For generality, we made this function polymorphic, since the type of the values in the list is irrelevant to the splitting process.

While this function is straightforward to define, it can be a bit challenging to work with. Let's try to prove the following lemma, which is obviously true:

```
Lemma split_len_first_try: ∀ {X} (l:list X) (l1 l2: list X),
    split l = (l1,l2) →
    length l1 ≤ length l ∧
    length l2 ≤ length l.
Proof.
  induction l; intros.
  - inv H. simpl. omega.
  - destruct l as [| x l'].
    + inv H.
      split; simpl; auto.
    + inv H. destruct (split l') as [l1' l2'] eqn:E. inv H1.
Abort.
```

The problem here is that the standard induction principle for lists requires us to show that the property being proved follows for any non-empty list if it holds for the tail of that list. What we want here is a "two-step" induction principle, that instead requires us to show that the property being proved follows for a list of length at least two, if it holds for the tail of the tail of that list. Formally:

```
Definition list_ind2_principle:=
    ∀ (A : Type) (P : list A → Prop),
      P [] →
      (∀ (a:A), P [a]) →
      (∀ (a b : A) (l : list A), P l → P (a :: b :: l)) →
      ∀ l : list A, P l.
```

If we assume the correctness of this "non-standard" induction principle, our split_len proof is easy, using a form of the induction tactic that lets us specify the induction principle to use:

```
Lemma split_len': list_ind2_principle →
    ∀ {X} (l:list X) (l1 l2: list X),
    split l = (l1 ,l2) →
    length l1 ≤ length l ∧
    length l2 ≤ length l.
Proof.
  unfold list_ind2_principle; intro IP.
  induction l using IP; intros.
  - inv H. omega.
  - inv H. simpl; omega.
  - inv H. destruct (split l) as [l1' l2']. inv H1.
    simpl.
    destruct (IHl l1' l2') as [P1 P2]; auto; omega.
Qed.
```

We still need to prove list_ind2_principle. There are several ways to do this, but one direct way is to write an explicit proof term, thus:

```
Definition list_ind2 :
  ∀ (A : Type) (P : list A → Prop),
      P [] →
      (∀ (a:A), P [a]) →
      (∀ (a b : A) (l : list A), P l → P (a :: b :: l)) →
      ∀ l : list A, P l :=
  fun (A : Type)
      (P : list A → Prop)
      (H : P [])
      (H0 : ∀ a : A, P [a])
      (H1 : ∀ (a b : A) (l : list A), P l → P (a :: b :: l)) ⇒
    fix IH (l : list A) : P l :=
    match l with
    | [] ⇒ H
    | [x] ⇒ H0 x
    | x::y::l' ⇒ H1 x y l' (IH l')
    end.
```

Here, the fix keyword defines a local recursive function *IH* of type ∀ *l*:list *A*, *P l*, which is returned as the overall value of list_ind2. As usual, this function must be obviously terminating to Coq (which it is because the recursive call is on a sublist *l'* of the original argument *l*) and the match must be exhaustive over all possible lists (which it evidently is).

With our induction principle in hand, we can finally prove split_len free and clear:

```
Lemma split_len: ∀ {X} (l:list X) (l1 l2: list X),
    split l = (l1 ,l2 ) →
    length l1 ≤ length l ∧
    length l2 ≤ length l.
Proof.
 apply (@split_len' list_ind2).
Qed.
```

Exercise: 3 stars, standard (split_perm) Here's another fact about `split` that we will find useful later on.

```
Lemma split_perm : ∀ {X:Type} (l l1 l2: list X),
    split l = (l1 ,l2 ) → Permutation l (l1 ++ l2).
Proof.
  induction l as [| x | x1 x2 l1' IHl'] using list_ind2; intros.
   Admitted.
```

□

8.1.2 Defining Merge

Next, we need a merge function, which takes two sorted lists (of naturals) and returns their sorted result. This would seem easy to write:

```
Fixpoint merge l1 l2 :=
  match l1, l2 with
  | [], _ ⇒ l2
  | _, [] ⇒ l1
  | a1::l1', a2::l2' ⇒
      if a1 <=? a2 then a1 :: merge l1' l2 else a2 :: merge l1 l2'
  end.
```

But Coq will reject this definition with the message:

Error: Cannot guess decreasing argument `of fix`.

Coq insists the every `Fixpoint` definition be structurally recursive on some specified argument, meaning that at each recursive call the callee is passed a value that is a subterm of the caller's argument value. This check guarantees that every `Fixpoint` is actually terminating.

It is fairly obvious that this function is in fact terminating, because at each call, either *l1* or *l2* is passed the tail of its original value. But unfortunately, `Fixpoint` recursive calls must always decrease on a *single fixed* argument – and neither *l1* nor *l2* will do. (That's why Coq couldn't guess the one to use.) We might reasonably wish that Coq was a little smarter, but it isn't.

There are a number of ways to get around the problem of convincing Coq that a function is actually terminating when the "natural" Fixpoint doesn't work. In this case, a little creativity (or a peek at the Coq library) might lead us to the following definition:

```
Fixpoint merge l1 l2 {struct l1} :=
  let fix merge_aux l2 :=
  match l1, l2 with
  | [], _ ⇒ l2
  | _, [] ⇒ l1
  | a1::l1', a2::l2' ⇒
      if a1 <=? a2 then a1 :: merge l1' l2 else a2 :: merge_aux l2'
  end
  in merge_aux l2.
```

Coq accepts the outer definition because it is structurally decreasing on *l1* (we specify that with the {struct *l1*} annotation, although Coq would have guessed this even if we didn't write it), and it accepts the inner definition because it is structurally recursive on its (sole) argument. (Note that let fix ... in ... end is just a mechanism for defining a local recursive function.)

This definition will turn out to work pretty well; the only irritation is that simplification will show the definition of *merge_aux*, as illustrated by the following examples.

First, let's remind ourselves that Coq desugars a match over multiple arguments into a nested sequence of matches:

```
Print merge.
```

==> (after a little renaming for clarity)

```
 fix merge (l1 l2 : list nat) {struct l1} : list nat :=
   let
     fix merge_aux (l2 : list nat) : list nat :=
       match l1 with
       | [] ⇒ l2
       | a1 :: l1' ⇒
           match l2 with
           | [] ⇒ l1
           | a2 :: l2' ⇒
               if a1 <=? a2 then a1 :: merge l1' l2 else a2 :: merge_aux l2'
           end
       end in
   merge_aux l2.
```

Let's prove the following simple lemmas about merge:

```
Lemma merge2 : ∀ (x1 x2:nat) r1 r2,
    x1 ≤ x2 →
    merge (x1::r1) (x2::r2) =
    x1::merge r1 (x2::r2).
Proof.
  intros.
  simpl.   bdestruct (x1 <=? x2).
  - auto.
  -
    simpl.    destruct r2; simpl.    + omega.
    + omega.
Qed.

Lemma merge_nil_l : ∀ l, merge [] l = l.
Proof.
  intros. simpl.
  destruct l.
  - auto.
  - auto.
Qed.
```

Morals:

(1) Even though the proof state involving local recursive functions can can be hard to read, persevere!

(2) If Coq won't simplify an "obvious" application, try destructing the argument.

We will defer stating and proving other properties of merge until later.

8.1.3 Defining Mergesort

Finally, we need to define the main mergesort function itself. Once again, we might hope to write something simple like this:

```
Fixpoint mergesort (l: list nat) : list nat :=
   let (l1,l2) := split l in
   merge (mergesort l1) (mergesort l2).
```

Since this function has only one argument, Coq guesses that it is intended to be structurally decreasing, but still rejects the definition, this time with the complaint:

```
Recursive call to mergesort has principal argument equal to
"l1" instead of a subterm of "l".
```

Again, the problem is that Coq has no way to know that $l1$ and $l2$ are "smaller" than

l. And this time, it is hard to complain that Coq is being stupid, since the fact that split returns smaller lists than it is passed is nontrivial.

In fact, it isn't true! Consider the behavior of split on empty or singleton lists... This is case where Coq's totality requirements can actually help us correct the definition of our code. What we really want to write is something more like:

```
Fixpoint mergesort (l: list nat) : list nat :=
    match l with
    | [] ⇒ []
    | [x] ⇒ [x]
    | _ ⇒ let (l1,l2) := split l in merge (mergesort l1) (mergesort l2).
```

Now this function really is terminating! But Coq still won't let us write it with a Fixpoint. Instead, we need to use a mechanism (there are several available) for defining functions that accommodates an explicit way to show that the function only calls itself on smaller arguments. We will use the Function command:

```
Function mergesort (l: list nat) {measure length l} : list nat :=
  match l with
  | [] ⇒ []
  | [x] ⇒ [x]
  | _ ⇒ let (l1,l2) := split l in
         merge (mergesort l1) (mergesort l2)
  end.
```

Function is similar to Fixpoint, but it lets us specify an explicit *measure* on the function arguments. The annotation {measure length l} says that the function length applied to argument l serves as a decreasing measure. After processing this definition, Coq enters proof mode and demands proofs that each recursive call is indeed on a shorter list. Happily, we proved that fact already.

```
Proof.
  -
    intros.
    simpl in *. destruct (split l1) as [l1' l2'] eqn:E. inv teq1.
    destruct (split_len _ _ _ E).
    simpl. omega.
  -
    intros.
    simpl in *. destruct (split l1) as [l1' l2'] eqn:E. inv teq1.
    destruct (split_len _ _ _ E).
    simpl. omega.
Defined.
```

Notice that the Proof must end with the keyword Defined rather than Qed; if we don't do this, we won't be able to actually compute with mergesort.

Defining mergesort with `Function` rather than `Fixpoint` causes the automatic generation of some useful auxiliary definitions that we will need when working with it. First, we get a lemma mergesort_equation, which performs a one-level unfolding of the function.

```
Check mergesort_equation.
```

==>

```
mergesort_equation
  : ∀ l : list nat,
    mergesort l =
    match l with
    | [] ⇒ []
    | [x] ⇒ [x]
    | x :: _ :: _ ⇒
        let (l2, l3) := split l in merge (mergesort l2) (mergesort l3)
    end
```

We should always use `apply` mergesort_equation to simplify a call to mergesort rather than trying to `unfold` or `simpl` it, which will lead to ugly or mysterious results.

Second, we get an induction principle mergesort_ind; performing induction using this principle can be much easier than trying to use list induction over the argument l.

```
Check mergesort_ind.
```

==>

```
mergesort_ind
  : ∀ P : list nat → list nat → Prop,
    (∀ l : list nat, l = [] → P [] []) →
    (∀ (l : list nat) (x : nat), l = [x] → P [x] [x]) →
    (∀ l _x : list nat,
     l = _x →
     match _x with
     | _ :: _ :: _ ⇒ True
     | _ ⇒ False
     end →
     ∀ l1 l2 : list nat,
     split l = (l1, l2) →
     P l1 (mergesort l1) →
     P l2 (mergesort l2) → P _x (merge (mergesort l1) (mergesort l2))) →
     ∀ l : list nat, P l (mergesort l)
```

8.1.4 Correctness: Sortedness

As with insertion sort, our goal is to prove that mergesort produces a sorted list that is a permutation of the original list, i.e. to prove

is_a_sorting_algorithm mergesort

We will start by showing that mergesort produces a sorted list. The key lemma is to show that merge of two sorted lists produces a sorted list. It is perhaps easiest to break out a sub-lemma first:

Exercise: 2 stars, standard (sorted_merge1) Lemma sorted_merge1 : $\forall$ x $x1$ $l1$ $x2$ $l2$,
$x \leq x1 \rightarrow x \leq x2 \rightarrow$
sorted (merge ($x1$ :: $l1$) ($x2$ :: $l2$)) $\rightarrow$
sorted (x :: merge ($x1$ :: $l1$) ($x2$ :: $l2$)).

Proof.
Admitted.
□

Exercise: 4 stars, standard (sorted_merge) Lemma sorted_merge : $\forall$ $l1$, **sorted** $l1$ $\rightarrow$
$\forall$ $l2$, **sorted** $l2$ $\rightarrow$
sorted (merge $l1$ $l2$).

Proof.
Admitted.
□

Exercise: 2 stars, standard (mergesort_sorts) Lemma mergesort_sorts: $\forall$ l, **sorted** (mergesort l).

Proof.
apply mergesort_ind; intros. *Admitted.*
□

8.1.5 Correctness: Permutation

Finally, we must show that mergesort returns a permutation of its input.

As usual, the key lemma is for merge.

Incidentally, you are welcome to import the alternative characterizations of permutations as multisets given in Multiset or BagPerm and use that instead of Permutation if you think it will be easier. (I'm not sure!)

Exercise: 3 stars, advanced (merge_perm) Lemma merge_perm: $\forall$ ($l1$ $l2$: **list nat**),
Permutation ($l1$ ++ $l2$) (merge $l1$ $l2$).

Proof.

Admitted.

□

Exercise: 3 stars, advanced (mergesort_perm) Lemma mergesort_perm: ∀ l, **Permutation** l (mergesort l).

Proof.

Admitted.

□

Putting it all together:

```
Theorem mergesort_correct:
  is_a_sorting_algorithm mergesort.
Proof.
  split.
  apply mergesort_perm.
  apply mergesort_sorts.
Qed.
```

Date

Chapter 9

Library VFA.SearchTree

9.1 SearchTree: Binary Search Trees

We have implemented maps twice so far: with lists in *Lists*, and with higher-order functions in Maps. Those are simple but inefficient implementations: looking up the value bound to a given key takes time linear in the number of bindings, both in the worst and expected case.

If the type of keys can be totally ordered – that is, it supports a well-behaved $\leq$ comparison – then maps can be implemented with *binary search trees* (BSTs). Insert and lookup operations on BSTs take time proportional to the height of the tree. If the tree is balanced, the operations therefore take logarithmic time.

If you don't recall BSTs or haven't seen them in a while, see Wikipedia or read any standard textbook; for example:

- 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.

```
From Coq Require Import String. From Coq Require Import Logic.FunctionalExtensionality.
From VFA Require Import Perm.
From VFA Require Import Maps.
From VFA Require Import Sort.
```

9.2 BST Implementation

We use **nat** as the key type in our implementation of BSTs, since it has a convenient total order <=? with lots of theorems and automation available.

```
Definition key := nat.
```

E represents the empty map. T l k v r represents the map that binds k to v, along with all the bindings in l and r. No key may be bound more than once in the map.

```
Inductive tree (V : Type) : Type :=
| E
| T (l : tree V) (k : key) (v : V) (r : tree V).
```

Arguments E {V}.
Arguments T {V}.

An example tree:

4 -> "four" / \ / \ 2 -> "two" 5 -> "five"

```
Definition ex_tree : tree string :=
  (T (T E 2 "two" E) 4 "four" (T E 5 "five" E))%string.
```

empty_tree contains no bindings.

```
Definition empty_tree {V : Type} : tree V :=
  E.
```

bound k t is whether k is bound in t.

```
Fixpoint bound {V : Type} (x : key) (t : tree V) :=
  match t with
  | E ⇒ false
  | T l y v r ⇒ if x <? y then bound x l
                else if x >? y then bound x r
                     else true
  end.
```

lookup d k t is the value bound to k in t, or is default value d if k is not bound in t.

```
Fixpoint lookup {V : Type} (d : V) (x : key) (t : tree V) : V :=
  match t with
  | E ⇒ d
  | T l y v r ⇒ if x <? y then lookup d x l
                else if x >? y then lookup d x r
                     else v
  end.
```

insert k v t is the map containing all the bindings of t along with a binding of k to v.

```
Fixpoint insert {V : Type} (x : key) (v : V) (t : tree V) : tree V :=
  match t with
  | E ⇒ T E x v E
  | T l y v' r ⇒ if x <? y then T (insert x v l) y v' r
                 else if x >? y then T l y v' (insert x v r)
                      else T l x v r
  end.
```

Note that insert is a *functional* aka *persistent* implementation: t is not changed.

Module TESTS.

Here are some unit tests to check that BSTs behave the way we expect.

```
Open Scope string_scope.

Example bst_ex1 :
  insert 5 "five" (insert 2 "two" (insert 4 "four" empty_tree)) = ex_tree.
Proof. reflexivity. Qed.

Example bst_ex2 : lookup "" 5 ex_tree = "five".
Proof. reflexivity. Qed.

Example bst_ex3 : lookup "" 3 ex_tree = "".
Proof. reflexivity. Qed.

Example bst_ex4 : bound 3 ex_tree = false.
Proof. reflexivity. Qed.
```

End TESTS.

Although we can spot-check the behavior of BST operations with unit tests like these, we of course should prove general theorems about their correctness. We will do that later in the chapter.

9.3 BST Invariant

The implementations of lookup and insert assume that values of type **tree** obey the *BST invariant*: for any non-empty node with key k, all the values of the left subtree are less than k and all the values of the right subtree are greater than k. But that invariant is not part of the definition of **tree**. For example, the following tree is not a BST:

Module NOTBST.

```
Open Scope string_scope.

Definition t : tree string :=
  T (T E 5 "five" E) 4 "four" (T E 2 "two" E).
```

The insert function we wrote above would never produce such a tree, but we can still construct it by manually applying T. When we try to lookup 2 in that tree, we get the wrong answer, because lookup assumes 2 is in the left subtree:

```
Example not_bst_lookup_wrong :
  lookup "" 2 t ≠ "two".
Proof.
  simpl. unfold not. intros contra. discriminate.
Qed.
```

End NOTBST.

So, let's formalize the BST invariant. Here's one way to do so. First, we define a helper ForallT to express that idea that a predicate holds at every node of a tree:

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

Second, we define the BST invariant:

- An empty tree is a BST.
- A non-empty tree is a BST if all its left nodes have a lesser key, its right nodes have a greater key, and the left and right subtrees are themselves BSTs.

```
Inductive BST {V : Type} : tree V → Prop :=
| BST_E : BST E
| BST_T : ∀ l x v r,
    ForallT (fun y _ ⇒ y < x) l →
    ForallT (fun y _ ⇒ y > x) r →
    BST l →
    BST r →
    BST (T l x v r).
Hint Constructors BST.
```

Let's check that **BST** correctly classifies a couple of example trees:

```
Example is_BST_ex :
  BST ex_tree.
Proof.
  unfold ex_tree.
  repeat (constructor; try omega).
Qed.

Example not_BST_ex :
  ¬ BST NotBst.t.
Proof.
  unfold NotBst.t. intros contra.
  inv contra. inv H3. omega.
Qed.
```

Exercise: 1 star, standard (empty_tree_BST) Prove that the empty tree is a BST.

```
Theorem empty_tree_BST : ∀ (V : Type),
    BST (@empty_tree V).
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 4 stars, standard (insert_BST) Prove that insert produces a BST, assuming it is given one.

Start by proving this helper lemma, which says that insert preserves any node predicate. Proceed by induction on t.

```
Lemma ForallT_insert : ∀ (V : Type) (P : key → V → Prop) (t : tree V),
    ForallT P t → ∀ (k : key) (v : V),
      P k v → ForallT P (insert k v t).
Proof.
  Admitted.
```

Now prove the main theorem. Proceed by induction on the evidence that t is a BST.

```
Theorem insert_BST : ∀ (V : Type) (k : key) (v : V) (t : tree V),
    BST t → BST (insert k v t).
Proof.
  Admitted.
```

☐

Since empty_tree and insert are the only operations that create BSTs, we are guaranteed that any **tree** is a BST – unless it was constructed manually with T. It would therefore make sense to limit the use of T to only within the tree operations, rather than expose it. Coq, like OCaml and other functional languages, can do this with its module system. See ADT for details.

9.4 Correctness of BST Operations

To prove the correctness of lookup and bound, we need specifications for them. We'll study two different techniques for that in this chapter.

The first is called *algebraic specification*. With it, we write down equations relating the results of operations. For example, we could write down equations like the following to specify the + and × operations:

(a + b) + c = a + (b + c) a + b = b + a a + 0 = a (a * b) * c = a * (b * c) a * b = b * a a * 1 = a a * 0 = 0 a * (b + c) = a * b + a * c

For BSTs, let's examine how lookup should interact with when applied to other operations. It is easy to see what needs to be true for empty_tree: looking up any value at all in the empty tree should fail and return the default value:

lookup d k empty_tree = d

What about non-empty trees? The only way to build a non-empty tree is by applying insert k v t to an existing tree t. So it suffices to describe the behavior of lookup on the result of an arbitrary insert operation. There are two cases. If we look up the same key that was just inserted, we should get the value that was inserted with it:

lookup d k (insert k v t) = v

If we look up a different key than was just inserted, the insert should not affect the answer – which should be the same as if we did the lookup in the original tree before the insert occured:

lookup d k' (insert k v t) = lookup d k' t if k <> k'

These three basic equations specify the correct behavior of maps. Let's prove that they hold.

```
Theorem lookup_empty : ∀ (V : Type) (d : V) (k : key),
    lookup d k empty_tree = d.
Proof.
  auto.
Qed.

Theorem lookup_insert_eq : ∀ (V : Type) (t : tree V) (d : V) (k : key) (v : V),
    lookup d k (insert k v t) = v.
Proof.
  induction t; intros; simpl.
  - bdestruct (k <? k); try omega; auto.
  - bdestruct (k <? k0); bdestruct (k0 <? k); simpl; try omega; auto.
    + bdestruct (k <? k0); bdestruct (k0 <? k); try omega; auto.
    + bdestruct (k <? k0); bdestruct (k0 <? k); try omega; auto.
    + bdestruct (k0 <? k0); try omega; auto.
Qed.
```

The basic method of that proof is to repeatedly *bdestruct* everything in sight, followed by generous use of `omega` and `auto`. Let's automate that.

```
Ltac bdestruct_guard :=
  match goal with
  | ⊢ context [ if ?X =? ?Y then _ else _ ] ⇒ bdestruct (X =? Y)
  | ⊢ context [ if ?X <=? ?Y then _ else _ ] ⇒ bdestruct (X <=? Y)
  | ⊢ context [ if ?X <? ?Y then _ else _ ] ⇒ bdestruct (X <? Y)
  end.

Ltac bdall :=
  repeat (simpl; bdestruct_guard; try omega; auto).

Theorem lookup_insert_eq' :
  ∀ (V : Type) (t : tree V) (d : V) (k : key) (v : V),
    lookup d k (insert k v t) = v.
Proof.
  induction t; intros; bdall.
Qed.
```

The tactic immediately pays off in proving the third equation.

```
Theorem lookup_insert_neq :
  ∀ (V : Type) (t : tree V) (d : V) (k k' : key) (v : V),
   k ≠ k' → lookup d k' (insert k v t) = lookup d k' t.
Proof.
```

```
  induction t; intros; bdall.
Qed.
```

Perhaps surprisingly, the proofs of these results do not depend on whether t satisfies the BST invariant. That's because lookup and insert follow the same path through the tree, so even if nodes are in the "wrong" place, they are consistently "wrong".

Exercise: 3 stars, standard, optional (bound_correct) Specify and prove the correctness of bound. State and prove three theorems, inspired by those we just proved for lookup. If you have the right theorem statements, the proofs should all be quite easy – thanks to *bdall*.

```
Definition manual_grade_for_bound_correct : option (nat×string) := None.
```

□

Exercise: 3 stars, standard, optional (bound_default) Prove that if bound returns false, then lookup returns the default value. Proceed by induction on the tree.

```
Theorem bound_default :
  ∀ (V : Type) (k : key) (d : V) (t : tree V),
    bound k t = false →
    lookup d k t = d.
Proof.
  Admitted.
```

□

9.5 BSTs vs. Higher-order Functions (Optional)

The three theorems we just proved for lookup should seem familiar: we proved equivalent theorems in Maps for maps defined as higher-order functions.

- lookup_empty and t_apply_empty both state that the empty map binds all keys to the default value.

```
Check lookup_empty : ∀ (V : Type) (d : V) (k : key),
    lookup d k empty_tree = d.
Check t_apply_empty : ∀ (V : Type) (k : key) (d : V),
    t_empty d k = d.
```

- lookup_insert_eq and t_update_eq both state that updating a map then looking for the updated key produces the updated value.

Check lookup_insert_eq : ∀ (V : Type) (t : **tree** V) (d : V) (k : key) (v : V),
lookup d k (insert k v t) = v.

Check *t_update_eq* : ∀ (V : Type) (m : total_map V) (k : key) (v : V),
(t_update m k v) k = v.

- lookup_insert_neq and t_update_neq both state that updating a map then looking for a different key produces the same value as the original map.

Check lookup_insert_neq :
∀ (V : Type) (t : **tree** V) (d : V) (k k' : key) (v : V),
$k \neq k'$ → lookup d k' (insert k v t) = lookup d k' t.

Check *t_update_neq* : ∀ (V : Type) (v : V) (k k' : key) (m : total_map V),
$k \neq k'$ → (t_update m k v) k' = m k'.

In Maps, we also proved three other theorems about the behavior of functional maps on various combinations of updates and lookups:

Check *t_update_shadow* : ∀ (V : Type) (m : total_map V) ($v1$ $v2$: V) (k : key),
t_update (t_update m k $v1$) k $v2$ = t_update m k $v2$.

Check *t_update_same* : ∀ (V : Type) (k : key) (m : total_map V),
t_update m k (m k) = m.

Check *t_update_permute* :
∀ (V : Type) ($v1$ $v2$: V) ($k1$ $k2$: key) (m : total_map V),
$k2 \neq k1$ →
t_update (t_update m $k2$ $v2$) $k1$ $v1$ = t_update (t_update m $k1$ $v1$) $k2$ $v2$.

Let's prove analogues to these three theorems for search trees.

Hint: you do not need to unfold the definitions of empty_tree, insert, or lookup. Instead, use lookup_insert_eq and lookup_insert_neq.

Exercise: 2 stars, standard, optional (lookup_insert_shadow) Lemma lookup_insert_shadow
:
∀ (V : Type) (t : **tree** V) (v v' d: V) (k k' : key),
lookup d k' (insert k v (insert k v' t)) = lookup d k' (insert k v t).
Proof.
intros. *bdestruct* (k =? k').
Admitted.

□

Exercise: 2 stars, standard, optional (lookup_insert_same) Lemma lookup_insert_same
:
∀ (V : Type) (k k' : key) (d : V) (t : **tree** V),

```
    lookup d k' (insert k (lookup d k t) t) = lookup d k' t.
Proof.
  Admitted.
```

□

Exercise: 2 stars, standard, optional (lookup_insert_permute) Lemma lookup_insert_permute :

```
  ∀ (V : Type) (v1 v2 d : V) (k1 k2 k': key) (t : tree V),
    k1 ≠ k2 →
    lookup d k' (insert k1 v1 (insert k2 v2 t))
    = lookup d k' (insert k2 v2 (insert k1 v1 t)).
Proof.
  Admitted.
```

□

Our ability to prove these lemmas without reference to the underlying tree implementation demonstrates they hold for any map implementation that satisfies the three basic equations.

Each of these lemmas just proved was phrased as an equality between the results of looking up an arbitrary key *k'* in two maps. But the lemmas for the function-based maps were phrased as direct equalities between the maps themselves.

Could we state the tree lemmas with direct equalities? For *insert_shadow*, the answer is yes:

```
Lemma insert_shadow_equality : ∀ (V : Type) (t : tree V) (k : key) (v v' : V),
    insert k v (insert k v' t) = insert k v t.
Proof.
  induction t; intros; bdall.
  - rewrite IHt1; auto.
  - rewrite IHt2; auto.
Qed.
```

But the other two direct equalities on BSTs do not necessarily hold.

Exercise: 3 stars, standard, optional (direct_equalities_break) Prove that the other equalities do not hold. Hint: find a counterexample first on paper, then use the ∃ tactic to instantiate the theorem on your counterexample. The simpler your counterexample, the simpler the rest of the proof will be.

```
Lemma insert_same_equality_breaks :
  ∃ (V : Type) (d : V) (t : tree V) (k : key),
      insert k (lookup d k t) t ≠ t.
Proof.
  Admitted.
```

```
Lemma insert_permute_equality_breaks :
  ∃ (V : Type) (v1 v2 : V) (k1 k2 : key) (t : tree V),
    k1 ≠ k2 ∧ insert k1 v1 (insert k2 v2 t) ≠ insert k2 v2 (insert k1 v1 t).
Proof.
    Admitted.
```

□

9.6 Converting a BST to a List

Let's add a new operation to our BST: converting it to an *association list* that contains the key–value bindings from the tree stored as pairs. If that list is sorted by the keys, then any two trees that represent the same map would be converted to the same list. Here's a function that does so with an in-order traversal of the tree:

```
Fixpoint elements {V : Type} (t : tree V) : list (key × V) :=
  match t with
  | E ⇒ []
  | T l k v r ⇒ elements l ++ [(k, v)] ++ elements r
  end.
```

```
Example elements_ex :
    elements ex_tree = [(2, "two"); (4, "four"); (5, "five")]%string.
Proof. reflexivity. Qed.
```

Here are three desirable properties for elements:
1. The list has the same bindings as the tree.
2. The list is sorted by keys.
3. The list contains no duplicates.
Let's formally specify and verify them.

9.6.1 Part 1: Same Bindings

We want to show that a binding is in elements t iff it's in t. We'll prove the two directions of that bi-implication separately:

- elements is *complete*: if a binding is in t then it's in elements t.
- elements is *correct*: if a binding is in elements t then it's in t.

Getting the specification of completeness right is a little tricky. It's tempting to start off with something too simple like this:

```
Definition elements_complete_broken_spec :=
  ∀ (V : Type) (k : key) (v d : V) (t : tree V),
    BST t →
```

lookup d k t = v →
In (k , v) (elements t).

The problem with that specification is how it handles the default element d: the specification would incorrectly require elements t to contain a binding (k, d) for all keys k unbound in t. That would force elements t to be infinitely long, since it would have to contain a binding for every natural number. We can observe this problem right away if we begin the proof:

```
Theorem elements_complete_broken : elements_complete_broken_spec.
Proof.
  unfold elements_complete_broken_spec. intros. induction t.
  - simpl.
```

We have nothing to work with, since elements E is []. Abort.

The solution is to check first to see whether k is bound in t. Only bound keys need be in the list of elements:

```
Definition elements_complete_spec :=
  ∀ (V : Type) (k : key) (v d : V) (t : tree V),
    BST t →
    bound k t = true →
    lookup d k t = v →
    In (k , v) (elements t).
```

Exercise: 3 stars, standard (elements_complete) Prove that elements is complete. Proceed by induction on t.

```
Theorem elements_complete : elements_complete_spec.
Proof.
  Admitted.
```

□

The specification for correctness likewise mentions that the key must be bound:

```
Definition elements_correct_spec :=
  ∀ (V : Type) (k : key) (v d : V) (t : tree V),
    BST t →
    In (k , v) (elements t) →
    bound k t = true ∧ lookup d k t = v.
```

Proving correctness requires more work than completeness.

BST uses ForallT to say that all nodes in the left/right subtree have smaller/greater keys than the root. We need to relate ForallT, which expresses that all nodes satisfy a property, to **Forall**, which expresses that all list elements satisfy a property.

We begin with this lemma about **Forall**, which is missing from the standard library.

```
Lemma Forall_app : ∀ (A: Type) (P : A → Prop) (l1 l2 : list A),
    Forall P l1 → Forall P l2 → Forall P (l1 ++ l2).
```

```
Proof.
  induction l1; intros; simpl; auto; inv H; constructor; auto.
Qed.
```

Exercise: 2 stars, standard (elements_preserves_forall) Prove that if a property P holds of every node in a tree t, then that property holds of every pair in elements t. Proceed by induction on t.

There is a little mismatch between the type of P in ForallT and the type of the property accepted by Forall, so we have to *uncurry* P when we pass it to Forall. (See *Poly* for more about uncurrying.) The single quote used below is the Coq syntax for doing a pattern match in the function arguments.

```
Definition uncurry {X Y Z : Type} (f : X → Y → Z) '(a, b) :=
  f a b.

Hint Transparent uncurry.

Lemma elements_preserves_forall : ∀ (V : Type) (P : key → V → Prop) (t : tree V),
    ForallT P t →
    Forall (uncurry P) (elements t).
Proof.
   Admitted.
```

□

Exercise: 2 stars, standard (elements_preserves_relation) Prove that if all the keys in t are in a relation R with a distinguished key k', then any key k in elements t is also related by R to k'. For example, R could be $<$, in which case the lemma says that if all the keys in t are less than k', then all the keys in elements t are also less than k'.

Hint: you don't need induction. Immediately look for a way to use elements_preserves_forall and library theorem *Forall_forall*.

```
Lemma elements_preserves_relation :
  ∀ (V : Type) (k k' : key) (v : V) (t : tree V) (R : key → key → Prop),
    ForallT (fun y _ ⇒ R y k') t
    → In (k , v) (elements t)
    → R k k'.
Proof.
   Admitted.
```

□

Exercise: 4 stars, standard (elements_correct) Prove that elements is correct. Proceed by induction on the evidence that t is a BST.

```
Theorem elements_correct : elements_correct_spec.
Proof.
```

Admitted.

□

The inverses of completeness and correctness also should hold:

- inverse completeness: if a binding is not in t then it's not in elements t.
- inverse correctness: if a binding is not in elements t then it's not in t.

Let's prove that they do.

Exercise: 2 stars, advanced (elements_complete_inverse) This inverse doesn't require induction. Look for a way to use elements_correct to quickly prove the result.

```
Theorem elements_complete_inverse :
  ∀ (V : Type) (k : key) (v : V) (t : tree V),
    BST t →
    bound k t = false →
    ¬ In (k, v) (elements t).
Proof.
  Admitted.
```

□

Exercise: 4 stars, advanced (elements_correct_inverse) Prove the inverse. First, prove this helper lemma by induction on t.

```
Lemma bound_value : ∀ (V : Type) (k : key) (t : tree V),
    bound k t = true → ∃ v, ∀ d, lookup d k t = v.
Proof.
  Admitted.
```

Prove the main result. You don't need induction.

```
Theorem elements_correct_inverse :
  ∀ (V : Type) (k : key) (t : tree V),
    BST t →
    (∀ v, ¬ In (k, v) (elements t)) →
    bound k t = false.
Proof.
  Admitted.
```

□

9.6.2 Part 2: Sorted (Advanced)

We want to show that elements is sorted by keys. We follow a proof technique contributed by Lydia Symmons et al.

Exercise: 3 stars, advanced (sorted_app) Prove that inserting an intermediate value between two lists maintains sortedness. Proceed by induction on the evidence that *l1* is sorted.

```
Lemma sorted_app: ∀ l1 l2 x,
  Sort.sorted l1 → Sort.sorted l2 →
  Forall (fun n ⇒ n < x) l1 → Forall (fun n ⇒ n > x) l2 →
  Sort.sorted (l1 ++ x :: l2).
Proof.
    Admitted.
```

□

Exercise: 4 stars, advanced (sorted_elements) The keys in an association list are the first elements of every pair:

```
Definition list_keys {V : Type} (lst : list (key × V)) :=
  map fst lst.
```

Prove that elements t is sorted by keys. Proceed by induction on the evidence that t is a BST.

```
Theorem sorted_elements : ∀ (V : Type) (t : tree V),
    BST t → Sort.sorted (list_keys (elements t)).
Proof.
    Admitted.
```

□

9.6.3 Part 3: No Duplicates (Advanced and Optional)

We want to show that elements t contains no duplicate bindings. Tree t itself cannot contain any duplicates, so the list that elements produces shouldn't either. The standard library already contains a helpful inductive proposition, NoDup.

```
Print NoDup.
```

The library is missing a theorem, though, about NoDup and ++. To state that theorem, we first need to formalize what it means for two lists to be disjoint:

```
Definition disjoint {X:Type} (l1 l2: list X) := ∀ (x : X),
    In x l1 → ¬ In x l2.
```

Exercise: 3 stars, advanced, optional (NoDup_append) Prove that if two lists are disjoint, appending them preserves NoDup. Hint: You might already have proved this theorem in an advanced exercise in *IndProp*.

```
Lemma NoDup_append : ∀ (X:Type) (l1 l2: list X),
  NoDup l1 → NoDup l2 → disjoint l1 l2 →
```

NoDup (l1 ++ l2).
Proof.
    Admitted.
    □

Exercise: 4 stars, advanced, optional (elements_nodup_keys) Prove that there are no duplicate keys in the list returned by elements. Proceed by induction on the evidence that t is a BST. Make use of library theorems about map as needed.

```
Theorem elements_nodup_keys : ∀ (V : Type) (t : tree V),
    BST t →
    NoDup (list_keys (elements t)).
Proof.
    Admitted.
```

□

That concludes the proof of correctness of elements.

9.7 A Faster elements Implementation

The implemention of elements is inefficient because of how it uses the ++ operator. On a balanced tree its running time is linearithmic, because it does a linear number of concatentations at each level of the tree. On an unbalanced tree it's quadratic time. Here's a tail-recursive implementation than runs in linear time, regardless of whether the tree is balanced:

```
Fixpoint fast_elements_tr {V : Type} (t : tree V)
         (acc : list (key × V)) : list (key × V) :=
  match t with
  | E ⇒ acc
  | T l k v r ⇒ fast_elements_tr l ((k, v) :: fast_elements_tr r acc)
  end.

Definition fast_elements {V : Type} (t : tree V) : list (key × V) :=
  fast_elements_tr t [].
```

Exercise: 3 stars, standard (fast_elements_eq_elements) Prove that fast_elements and elements compute the same function.

```
Lemma fast_elements_tr_helper :
  ∀ (V : Type) (t : tree V) (lst : list (key × V)),
    fast_elements_tr t lst = elements t ++ lst.
Proof.
    Admitted.

Lemma fast_elements_eq_elements : ∀ (V : Type) (t : tree V),
```

```
    fast_elements t = elements t.
Proof.
   Admitted.
```

□

Since the two implementations compute the same function, all the results we proved about the correctness of elements also hold for fast_elements. For example:

```
Corollary fast_elements_correct :
  ∀ (V : Type) (k : key) (v d : V) (t : tree V),
    BST t →
    In (k, v) (fast_elements t) →
    bound k t = true ∧ lookup d k t = v.
Proof.
  intros. rewrite fast_elements_eq_elements in *.
  apply elements_correct; assumption.
Qed.
```

This corollary illustrates a general technique: prove the correctness of a simple, slow implementation; then prove that the slow version is functionally equivalent to a fast implementation. The proof of correctness for the fast implementation then comes "for free".

9.8 An Algebraic Specification of elements

The verification of elements we did above did not adhere to the algebraic specification approach, which would suggest that we look for equations of the form

elements empty_tree = ... elements (insert k v t) = ... (elements t) ...

The first of these is easy; we can trivially prove the following:

```
Lemma elements_empty : ∀ (V : Type),
    @elements V empty_tree = [].
Proof.
  intros. simpl. reflexivity.
Qed.
```

But for the second equation, we have to express the result of inserting (k, v) into the elements list for t, accounting for ordering and the possibility that t may already contain a pair (k, v') which must be replaced. The following rather ugly function will do the trick:

```
Fixpoint kvs_insert {V : Type} (k : key) (v : V) (kvs : list (key × V)) :=
  match kvs with
  | [] ⇒ [(k, v)]
  | (k', v') :: kvs' ⇒
    if k <? k' then (k, v) :: kvs
    else if k >? k' then (k', v') :: kvs_insert k v kvs'
         else (k, v) :: kvs'
```

```
end.
```

That's not satisfactory, because the definition of kvs_insert is so complex. Moreover, this equation doesn't tell us anything directly about the overall properties of elements t for a given tree t. Nonetheless, we can proceed with a rather ugly verification.

Exercise: 3 stars, standard, optional (kvs_insert_split) Lemma kvs_insert_split :

```
  ∀ (V : Type) (v v0 : V) (e1 e2 : list (key × V)) (k k0 : key),
    Forall (fun '(k',_) ⇒ k' < k0) e1 →
    Forall (fun '(k',_) ⇒ k' > k0) e2 →
    kvs_insert k v (e1 ++ (k0,v0):: e2) =
    if k <? k0 then
      (kvs_insert k v e1) ++ (k0,v0)::e2
    else if k >? k0 then
           e1 ++ (k0,v0)::(kvs_insert k v e2)
         else
           e1 ++ (k,v)::e2.
Proof.
   Admitted.
```

□

Exercise: 3 stars, standard, optional (kvs_insert_elements) Lemma kvs_insert_elements

```
: ∀ (V : Type) (t : tree V),
    BST t →
    ∀ (k : key) (v : V),
      elements (insert k v t) = kvs_insert k v (elements t).
Proof.
   Admitted.
```

□

9.9 Model-based Specifications

At the outset, we mentioned studying two techniques for specifying the correctness of BST operations in this chapter. The first was algebraic specification.

Another approach to proving correctness of search trees is to relate them to our existing implementation of functional partial maps, as developed in Maps. To prove the correctness of a search-tree algorithm, we can prove:

- Any search tree corresponds to some functional partial map, using a function or relation that we write down.
- The lookup operation on trees gives the same result as the find operation on the corresponding map.

- Given a tree and corresponding map, if we insert on the tree and update the map with the same key and value, the resulting tree and map are in correspondence.

This approach is sometimes called *model-based specification*: we show that our implementation of a data type corresponds to a more more abstract *model* type that we already understand. To reason about programs that use the implementation, it suffices to reason about the behavior of the abstract type, which may be significantly easier. For example, we can take advantage of laws that we proved for the abstract type, like update_eq for functional maps, without having to prove them again for the concrete tree type.

We also need to be careful here, because the type of functional maps as defined in Maps do not actually behave quite like our tree-based maps. For one thing, functional maps can be defined on an infinite number of keys, and there is no mechanism for enumerating over the key set. To maintain correspondence with our finite trees, we need to make sure that we consider only functional maps built by finitely many applications of constructor functions (empty and update). Also, thanks to functional extensionality, functional maps obey stronger equality laws than our trees do (as we investigated in the *direct_equalities* exercise above), so we should not be misled into thinking that every fact we can prove about abstract maps necessarily holds for concrete ones.

Compared to the algebraic-specification approach described earlier in this chapter, the model-based approach can save some proof effort, especially if we already have a well-developed theory for the abstract model type. On the other hand, we have to give an explicit *abstraction* relation between trees and maps, and show that it is maintained by all operations. In the end, about the same amount of work is needed to show correctness, though the work shows up in different places depending on how the abstraction relation is defined.

We now give a model-based specification for trees in terms of functional partial maps. It is based on a simple abstraction relation that builds a functional map element by element.

```
Fixpoint map_of_list {V : Type} (el : list (key × V)) : partial_map V :=
  match el with
  | [] ⇒ empty
  | (k, v) :: el' ⇒ update (map_of_list el') k v
  end.

Definition Abs {V : Type} (t : tree V) : partial_map V :=
  map_of_list (elements t).
```

In general, model-based specifications may use an abstraction relation, allowing each concrete value to be related to multiple abstract values. But in this case a simple abstraction *function* will do, assigning a unique abstract value to each concrete one.

One small difference between trees and functional maps is that applying the latter returns an **option** V which might be None, whereas lookup returns a default value if key is not bound lookup fails. We can easily provide a function on functional partial maps having the latter behavior.

```
Definition find {V : Type} (d : V) (k : key) (m : partial_map V) : V :=
```

```
match m k with
| Some v ⇒ v
| None ⇒ d
end.
```

We also need a bound operation on maps.

```
Definition map_bound {V : Type} (k : key) (m : partial_map V) : bool :=
  match m k with
  | Some _ ⇒ true
  | None ⇒ false
  end.
```

We now proceed to prove that each operation preserves (or establishes) the abstraction relationship in an appropriate way:

concrete abstract

——— empty_tree empty bound map_bound lookup find insert update

The following lemmas will be useful, though you are not required to prove them. They can all be proved by induction on the list.

Exercise: 2 stars, standard, optional (in_fst)

```
Lemma in_fst : ∀ (X Y : Type) (lst : list (X × Y)) (x : X) (y : Y),
    In (x , y) lst → In x (map fst lst).
Proof.
  Admitted.
```

□

Exercise: 2 stars, standard, optional (in_map_of_list)

```
Lemma in_map_of_list : ∀ (V : Type) (el : list (key × V)) (k : key) (v : V),
    NoDup (map fst el) →
    In (k , v) el → (map_of_list el) k = Some v.
Proof.
  Admitted.
```

□

Exercise: 2 stars, standard, optional (not_in_map_of_list)

```
Lemma not_in_map_of_list : ∀ (V : Type) (el : list (key × V)) (k : key),
    ¬ In k (map fst el) → (map_of_list el) k = None.
Proof.
  Admitted.
```

□

```
Lemma empty_relate : ∀ (V : Type),
    @Abs V empty_tree = empty.
```

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

Exercise: 3 stars, standard, optional (bound_relate) Theorem bound_relate : ∀ (V : Type) (t : **tree** V) (k : key),
BST t →
map_bound k (Abs t) = bound k t.
Proof.
Admitted.

□

Exercise: 3 stars, standard, optional (lookup_relate) Lemma lookup_relate : ∀ (V : Type) (t : **tree** V) (d : V) (k : key),
BST t → find d k (Abs t) = lookup d k t.
Proof.
Admitted.
□

Exercise: 3 stars, standard, optional (insert_relate) Lemma insert_relate : ∀ (V : Type) (t : **tree** V) (k : key) (v : V),
BST t → Abs (insert k v t) = update (Abs t) k v.
Proof.
unfold Abs.
intros.
rewrite *kvs_insert_elements*; auto.
remember (elements t) as l.
clear -l. *Admitted.*
□

The previous three lemmas are in essence saying that the following diagrams commute.

bound k t ——————-+ | | Abs | | V V m —————> b map_bound k

lookup d k t —————> v | | Abs | | Some V V m —————> Some v find d k

insert k v t —————> t' | | Abs | | Abs V V m —————> m' update' k v

Where we define:

update' k v m = update m k v

Functional partial maps lack a way to extract or iterate over their elements, so we cannot give an analogous abstract operation for elements. Instead, we can prove this trivial little lemma.

Lemma elements_relate : ∀ (V : Type) (t : **tree** V),
BST t →
map_of_list (elements t) = Abs t.

```
Proof.
  unfold Abs. intros. reflexivity.
Qed.
```

9.10 An Alternative Abstraction Relation (Optional, Advanced)

There is often more than one way to specify a suitable abstraction relation between given concrete and abstract datatypes. The following exercises explore another way to relate search trees to functional partial maps without using elements as an intermediate step.

We extend our definition of functional partial maps by adding a new primitive for combining two partial maps, which we call union. Our intention is that it only be used to combine maps with disjoint key sets; to keep the operation symmetric, we make the result be undefined on any key they have in common.

```
Definition union {X} (m1 m2: partial_map X) : partial_map X :=
  fun k ⇒
    match (m1 k, m2 k) with
    | (None, None) ⇒ None
    | (None, Some v) ⇒ Some v
    | (Some v, None) ⇒ Some v
    | (Some _, Some _) ⇒ None
    end.
```

We can prove some simple properties of lookup and update on unions, which will prove useful later.

Exercise: 2 stars, standard, optional (union_collapse)

```
Lemma union_left : ∀ {X} (m1 m2: partial_map X) k,
    m2 k = None → union m1 m2 k = m1 k.
Proof.
   Admitted.

Lemma union_right : ∀ {X} (m1 m2: partial_map X) k,
    m1 k = None →
    union m1 m2 k = m2 k.
Proof.
   Admitted.

Lemma union_both : ∀ {X} (m1 m2 : partial_map X) k v1 v2,
    m1 k = Some v1 →
    m2 k = Some v2 →
    union m1 m2 k = None.
Proof.
```

Admitted.

□

Exercise: 3 stars, standard, optional (union_update) Lemma union_update_right : ∀ {X} (m1 m2: partial_map X) k v,

```
    m1 k = None →
    update (union m1 m2) k v = union m1 (update m2 k v).
Proof.
  Admitted.
```

```
Lemma union_update_left : ∀ {X} (m1 m2: partial_map X) k v,
    m2 k = None →
    update (union m1 m2) k v = union (update m1 k v) m2.
Proof.
  Admitted.
```

□

We can now write a direct conversion function from trees to maps based on the structure of the tree, and prove a basic property preservation result.

```
Fixpoint map_of_tree {V : Type} (t: tree V) : partial_map V :=
  match t with
  | E ⇒ empty
  | T l k v r ⇒ update (union (map_of_tree l) (map_of_tree r)) k v
  end.
```

Exercise: 3 stars, advanced, optional (map_of_tree_prop) Lemma map_of_tree_prop : ∀ (V : Type) (P : key → V → Prop) (t : tree V),

```
    ForallT P t →
    ∀ k v, (map_of_tree t) k = Some v →
           P k v.
Proof.
  Admitted.
```

□

Finally, we define our new abstraction function, and prove the same lemmas as before.

```
Definition Abs' {V : Type} (t: tree V) : partial_map V :=
  map_of_tree t.
```

```
Lemma empty_relate' : ∀ (V : Type),
    @Abs' V empty_tree = empty.
Proof.
  reflexivity.
Qed.
```

Exercise: 3 stars, advanced, optional (bound_relate') Theorem bound_relate' : ∀ (V : Type) (t : **tree** V) (k : key),
  BST t →
  map_bound k (Abs' t) = bound k t.
Proof.
  Admitted.
  □

Exercise: 3 stars, advanced, optional (lookup_relate') Lemma lookup_relate' : ∀ (V : Type) (d : V) (t : **tree** V) (k : key),
  BST t → find d k (Abs' t) = lookup d k t.
Proof.
  Admitted.
  □

Exercise: 4 stars, advanced, optional (insert_relate') Lemma insert_relate' : ∀ (V : Type) (k : key) (v : V) (t : **tree** V),
  BST t → Abs' (insert k v t) = update (Abs' t) k v.
Proof.
  Admitted.
  □

The elements_relate lemma, which was trivial for our previous Abs function, is considerably harder this time. We suggest starting with an auxiliary lemma.

Exercise: 3 stars, advanced, optional (map_of_list_app) Lemma map_of_list_app : ∀ (V : Type) ($el1$ $el2$: **list** (key × V)),
  disjoint (map fst $el1$) (map fst $el2$) →
  map_of_list ($el1$ ++ $el2$) = union (map_of_list $el1$) (map_of_list $el2$).
Proof.
  Admitted.
  □

Exercise: 4 stars, advanced, optional (elements_relate') Lemma elements_relate' : ∀ (V : Type) (t : **tree** V),
  BST t →
  map_of_list (elements t) = Abs' t.
Proof.
  Admitted.
  □

9.11 Efficiency of Search Trees

All the theory we've developed so far has been about correctness. But the reason we use binary search trees is that they are efficient. That is, if there are N elements in a (reasonably well balanced) BST, each insertion or lookup takes about *log* N 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 that ensures the trees stay balanced. We'll do that in 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 used **nat** in this chapter because it's something easy to work with.

3. There's no notion of running 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 Redblack.
- SOLUTION 3: Apply bleeding-edge frameworks for reasoning about run-time of programs represented in Coq.

4. Our functions in Coq are models of implementations in "real" programming languages. What if the real implementations differ from the Coq models?

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

Chapter 10

Library VFA.ADT

10.1 ADT: Abstract Data Types

An *abstract data type* (ADT) can be defined as a set of values together with some operations. From the perspective of formal verification, the specifications of those operations should also be a part of an ADT.

When we implemented BSTs in SearchTree, we saw that maintaining a representation invariant was important to rule out trees that could never be constructed through operations of the data structure. That invariant became a precondition for many operations, though there was no need for *clients* of the data structure to know any details of the invariant.

In this chapter we'll study a little of Coq's *module system*, which enables hiding some details of the implementation of an ADT, while also exposing the formal specification of the ADT.

Some prior knowledge of an ML-style module system, especially OCaml's, will be helpful. Here are some sources that cover it:

- *Introduction to Objective Caml*, chapters 12 and 13. Jason Hickey, 2008. Available from https://ocaml.org/learn/books.html.
- The OCaml System Manual, chapter 2. Xavier Leroy et al., 2020. Available from http://caml.inria.fr/pub/docs/manual-ocaml/.

```
From Coq Require Import String. From VFA Require Import Perm.
From VFA Require Import Maps.
From VFA Require Import SearchTree.
```

10.2 The Table ADT

An *association table* is an ADT that binds keys to values. There are many other names for this concept, including *map*. We've already used that name before, though, for a specific data structure using higher-order functions. So for clarity in this chapter we use *table*.

Below is a Coq `Module Type` that declares an *interface* for the table ADT. A *parameter* is like a definition, except it declares only the type of an identifier, not the value to which it is bound. An *axiom* is similarly like a theorem, except no proof is provided.

`Module Type` TABLE.

A table is a binding from keys to values. It is *total*, meaning it binds all keys. `Parameter` *table* : `Type`.

Keys are natural numbers. `Definition` key := **nat**.

Values are an arbitrary type V. `Parameter` *V* : `Type`.

The default value to which keys are bound. `Parameter` *default* : *V*.

empty is the table that binds all keys to the default value. `Parameter` *empty* : *table*.

get k t is the value v to which k is bound in t. `Parameter` *get* : key → *table* → *V*.

`set` k v `t` is the table that binds k to v and otherwise has the same bindings as `t`. `Parameter` *set* : key → *V* → *table* → *table*.

The following three axioms are an equational specification for the table ADT.

```
Axiom get_empty_default : ∀ (k : key),
    get k empty = default.
Axiom get_set_same : ∀ (k : key) (v : V) (t : table),
    get k (set k v t) = v.
Axiom get_set_other : ∀ (k k' : key) (v : V) (t : table),
    k ≠ k' → get k' (set k v t) = get k' t.
```

`End` TABLE.

10.3 Implementing Table with Functions

We can *implement* TABLE with a `Module` that is parameterized on the type of values – or, rather, parameterized on a module that contains such a type. A parameterized module is also called a *functor*.

Module type VALTYPE says that there must be a type named V, with a default value provided:

```
Module Type ValType.
  Parameter V : Type.
  Parameter default : V.
End ValType.
```

Functor FUNTABLE takes as input a module of type VALTYPE, which must therefore contain a type V. As output, FUNTABLE produces a module of type TABLE. By writing <: TABLE, below, we ask Coq to check that the output module contains all the components required by TABLE.

```
Module FunTable (VT : ValType) <: Table.
  Definition V := VT.V.
  Definition default := VT.default.
  Definition key := nat.
```

A table is a function from keys to values.

```
  Definition table := key → V.
  Definition empty : table :=
    fun _ ⇒ default.
  Definition get (k : key) (t : table) : V :=
    t k.
  Definition set (k : key) (v : V) (t : table) : table :=
    fun k' ⇒ if k =? k' then v else t k'.
```

The implementation must prove the theorems that the interface specified as axioms.

```
  Theorem get_empty_default: ∀ (k : key),
      get k empty = default.
  Proof. intros. unfold get, empty. reflexivity. Qed.
  Theorem get_set_same: ∀ (k : key) (v : V) (t : table),
      get k (set k v t) = v.
  Proof. intros. unfold get, set. bdall. Qed.
  Theorem get_set_other: ∀ (k k' : key) (v : V) (t : table),
      k ≠ k' → get k' (set k v t) = get k' t.
  Proof. intros. unfold get, set. bdall. Qed.
End FunTable.
```

As an example, let's instantiate FunTable with strings as values.

```
Module StringVal.
  Definition V := string.
  Definition default := ""%string.
End StringVal.
Module FunTableExamples.
  Module StringFunTable := FunTable StringVal.
  Import StringFunTable.
  Open Scope string_scope.
  Example ex1 : get 0 empty = "".
  Proof. reflexivity. Qed.
  Example ex2 : get 0 (set 0 "A" empty) = "A".
  Proof. reflexivity. Qed.
  Example ex3 : get 1 (set 0 "A" empty) = "".
  Proof. reflexivity. Qed.
End FunTableExamples.
```

Exercise: 2 stars, standard, optional (NatFunTableExamples) Define a module that uses FunTable to implement a table mapping keys to values, where the values have type **nat**, with a default of 0. Write unit tests to check the operation of `get` and `set`.

```
Module NatFunTableExamples.
End NatFunTableExamples.
```

□

10.4 Implementing Table with Lists

Exercise: 3 stars, standard (lists_table) Use association lists to implement Table.

```
Module ListsTable (VT : ValType) <: Table.
  Definition V := VT.V.
  Definition default := VT.default.
  Definition key := nat.
  Definition table := list (key × V).

  Definition empty : table := [].

  Fixpoint get (k : key) (t : table) : V
    . Admitted.

  Definition set (k : key) (v : V) (t : table) : table
    . Admitted.

  Theorem get_empty_default: ∀ (k : key),
      get k empty = default.
  Proof.
   Admitted.

  Theorem get_set_same: ∀ (k : key) (v : V) (t : table),
      get k (set k v t) = v.
  Proof.
   Admitted.

  Theorem get_set_other: ∀ (k k' : key) (v : V) (t : table),
      k ≠ k' → get k' (set k v t) = get k' t.
  Proof.
   Admitted.
End ListsTable.
```

Instantiate your table and prove the following facts.

```
Module StringListsTableExamples.
  Module StringListsTable := ListsTable StringVal.
  Import StringListsTable.
  Open Scope string_scope.
```

```
Example ex1 : get 0 empty = "".
Proof.
  Admitted.

Example ex2 : get 0 (set 0 "A" empty) = "A".
Proof.
  Admitted.

Example ex3 : get 1 (set 0 "A" empty) = "".
Proof.
  Admitted.

End StringListsTableExamples.
```

□

10.5 Implementing Table with BSTs

Tables implemented with functions and association lists are, of course, inefficient. For a more efficient implementation, we can use BSTs.

```
Module TreeTable (VT : ValType) <: Table.
  Definition V := VT.V.
  Definition default := VT.default.
  Definition key := nat.
  Definition table := tree V.

  Definition empty : table :=
    @empty_tree V.

  Definition get (k : key) (t: table) : V :=
    lookup default k t.

  Definition set (k : key) (v : V) (t : table) : table :=
    insert k v t.
```

The three basic equations we proved about **tree** in SearchTree make short work of the theorems we need to prove for Table.

```
  Theorem get_empty_default: ∀ (k : key),
      get k empty = default.
  Proof.
    apply lookup_empty.
  Qed.

  Theorem get_set_same: ∀ (k : key) (v : V) (t : table),
      get k (set k v t) = v.
  Proof.
    intros. unfold get, set. apply lookup_insert_eq.
```

```
  Qed.
  Theorem get_set_other: ∀ (k k' : key) (v : V) (t : table),
      k ≠ k' → get k' (set k v t) = get k' t.
  Proof.
    intros. unfold get, set. apply lookup_insert_neq. assumption.
  Qed.
End TreeTable.
```

10.6 Tables with an elements Operation

Now let's consider a richer interface ETable for Tables that support bound and elements operation.

10.6.1 A First Attempt at ETable

```
Module Type ETable_first_attempt.
```

Include, as the name suggests, includes all the declarations from Table. Not only does that save keystrokes, it also means if we ever update Table to have new operations or new types, they automatically get included here, too. `Include Table.`

```
  Parameter bound : key → table → bool.
  Parameter elements : table → list (key × V).
  Axiom elements_complete : ∀ (k : key) (v : V) (t : table),
      bound k t = true →
      get k t = v →
      In (k, v) (elements t).
  Axiom elements_correct : ∀ (k : key) (v : V) (t : table),
      In (k, v) (elements t) →
      bound k t = true ∧ get k t = v.
End ETable_first_attempt.
```

We proved in SearchTree that the BST elements operation is correct and complete. So we ought to be able to implement ETable with BSTs. Let's try.

```
Module TreeETable_first_attempt (VT : ValType) <: ETable_first_attempt.
```

Include all the definitions from TreeTable, instantiated on VT. `Include TreeTable VT.`

Thanks to the Include, we now have all the tree operations defined "for free" in this module. For example: `Check get : key → table → V.`

```
  Definition bound (k : key) (t : table) : bool :=
    SearchTree.bound k t.
```

```
Definition elements (t : table) : list (key × V) :=
  SearchTree.elements t.

Theorem elements_complete : ∀ (k : key) (v : V) (t : table),
    bound k t = true →
    get k t = v →
    In (k, v) (elements t).
Proof.
  intros k v t Hbound Hlookup. unfold get in Hlookup.
```
pose *proof* t as *H* is equivalent to assert (*H* : ...*the* type of t...). { apply t. } but saves some keystrokes.

```
  pose proof (SearchTree.elements_complete) as Hcomplete.
  unfold elements_complete_spec in Hcomplete.
  apply Hcomplete with default.
```
- Stuck! We don't know that t satisfies the BST invariant. *Admitted.*

```
Theorem elements_correct : ∀ (k : key) (v : V) (t : table),
    In (k, v) (elements t) →
    bound k t = true ∧ get k t = v.
Proof.
  intros k v t Hin.
  pose proof (SearchTree.elements_correct) as Hcorrect.
  unfold elements_correct_spec in Hcorrect.
  apply Hcorrect.
```
- Again stuck because of the BST invariant. *Admitted.*

```
End TreeETable_first_attempt.
```

10.6.2 A Revised ETable

To prove that elements is correct, we need to know that trees satisfy the BST invariant. So, we declare a function rep_ok in the interface, and use it as a precondition for values of type table. We also add axioms to state that the ADT operations produce values that satisfy the representation invariant, i.e., it is a postcondition. And, we add a specification of bound.

```
Module Type ETable.

  Include Table.

  Parameter rep_ok : table → Prop.
  Parameter bound : key → table → bool.
  Parameter elements : table → list (key × V).
```

empty and set produce valid representations.

```
  Axiom empty_ok : rep_ok empty.

  Axiom set_ok : ∀ (k : key) (v : V) (t : table),
      rep_ok t → rep_ok (set k v t).
```

The specification of bound:

```
  Axiom bound_empty : ∀ (k : key),
      bound k empty = false.
  Axiom bound_set_same : ∀ (k : key) (v : V) (t : table),
      bound k (set k v t) = true.
  Axiom bound_set_other : ∀ (k k' : key) (v : V) (t : table),
      k ≠ k' → bound k' (set k v t) = bound k' t.
```

The specification of elements:

```
  Axiom elements_complete : ∀ (k : key) (v : V) (t : table),
      rep_ok t →
      bound k t = true →
      get k t = v →
      In (k, v) (elements t).
  Axiom elements_correct : ∀ (k : key) (v : V) (t : table),
      rep_ok t →
      In (k, v) (elements t) →
      bound k t = true ∧ get k t = v.
End ETable.
Module TreeETable (VT : ValType) <: ETable.
  Include TreeTable VT.
  Definition rep_ok (t : table) : Prop :=
    BST t.
  Definition bound (k : key) (t : table) : bool :=
    SearchTree.bound k t.
  Definition elements (t : table) : list (key × V) :=
    SearchTree.elements t.
  Theorem empty_ok : rep_ok empty.
  Proof.
    apply empty_tree_BST.
  Qed.
  Theorem set_ok : ∀ (k : key) (v : V) (t : table),
      rep_ok t → rep_ok (set k v t).
  Proof.
    apply insert_BST.
  Qed.
  Theorem bound_empty : ∀ (k : key),
      bound k empty = false.
  Proof.
    reflexivity.
```

```
  Qed.

  Theorem bound_set_same : ∀ (k : key) (v : V) (t : table),
      bound k (set k v t) = true.
  Proof.
    intros k v t. unfold bound, set. induction t; bdall.
  Qed.

  Theorem bound_set_other : ∀ (k k' : key) (v : V) (t : table),
      k ≠ k' → bound k' (set k v t) = bound k' t.
  Proof.
    intros k k' v t Hneq. unfold bound, set. induction t; bdall.
  Qed.

  Theorem elements_complete : ∀ (k : key) (v : V) (t : table),
      rep_ok t →
      bound k t = true →
      get k t = v →
      In (k, v) (elements t).
  Proof.
    intros k v t Hbound Hlookup.
    pose proof SearchTree.elements_complete as Hcomplete.
    unfold elements_complete_spec in Hcomplete.
    apply Hcomplete; assumption.
  Qed.

  Theorem elements_correct : ∀ (k : key) (v : V) (t : table),
      rep_ok t →
      In (k, v) (elements t) →
      bound k t = true ∧ get k t = v.
  Proof.
    intros k v t. simpl. intros Hin.
    pose proof SearchTree.elements_correct as Hcorrect.
    unfold elements_correct_spec in Hcorrect.
    apply Hcorrect; assumption.
  Qed.
End TreeETable.
```

Now we can use the table.

```
Module StringTreeETableExample.

  Module StringTreeETable := TreeETable StringVal.
  Import StringTreeETable.
  Open Scope string_scope.

  Example ex1 :
    In (0, "A") (elements (set 0 "A" (set 1 "B" empty))).
```

```
Proof.
  apply elements_complete;
    auto using empty_ok, set_ok, bound_set_same, get_set_same.
Qed.
End StringTreeETableExample.
```

10.7 Encapsulation with the Coq Module System (Advanced)

Functor TreeETable, above, reveals internal implementation details that your data structures class probably taught you are better kept private. For example, the constructors E and T of trees and the implementation of get are publicly exposed:

```
Example exposed_trees_ex :
  (StringTreeETableExample.StringTreeETable.get 0 (T E 0 "A" E) = "A")%string.
Proof.
  unfold StringTreeETableExample.StringTreeETable.get.   reflexivity.
Qed.
```

Like many languages, Coq makes it possible to *encapsulate* these details, meaning that they become hidden outside the module. It's a simple matter of changing the <: syntax to : in the functor's type. The only change in the implementation below is that one character.

The : syntax makes the module type *opaque*: only what is revealed in the type is available for code outside the module to use. The <: syntax, however, makes the module type *transparent*: the module must conform to the type, but everything about the module is still revealed.

```
Module TreeETableFullyEncapsulated (VT : ValType) : ETable.
  Include TreeTable VT.

  Definition rep_ok (t : table) : Prop :=
    BST t.

  Definition bound (k : key) (t : table) : bool :=
    SearchTree.bound k t.

  Definition elements (t : table) : list (key × V) :=
    SearchTree.elements t.

  Theorem empty_ok : rep_ok empty.
  Proof.
    apply empty_tree_BST.
  Qed.

  Theorem set_ok : ∀ (k : key) (v : V) (t : table),
      rep_ok t → rep_ok (set k v t).
  Proof.
```

```
    apply insert_BST.
  Qed.
  Theorem bound_empty : ∀ (k : key),
      bound k empty = false.
  Proof.
    reflexivity.
  Qed.
  Theorem bound_set_same : ∀ (k : key) (v : V) (t : table),
      bound k (set k v t) = true.
  Proof.
    intros k v t. unfold bound, set. induction t; bdall.
  Qed.
  Theorem bound_set_other : ∀ (k k' : key) (v : V) (t : table),
      k ≠ k' → bound k' (set k v t) = bound k' t.
  Proof.
    intros k k' v t Hneq. unfold bound, set. induction t; bdall.
  Qed.
  Theorem elements_complete : ∀ (k : key) (v : V) (t : table),
      rep_ok t →
      bound k t = true →
      get k t = v →
      In (k, v) (elements t).
  Proof.
    intros k v t Hbound Hlookup.
    pose proof SearchTree.elements_complete as Hcomplete.
    unfold elements_complete_spec in Hcomplete.
    apply Hcomplete; assumption.
  Qed.
  Theorem elements_correct : ∀ (k : key) (v : V) (t : table),
      rep_ok t →
      In (k, v) (elements t) →
      bound k t = true ∧ get k t = v.
  Proof.
    intros k v t. simpl. intros Hin.
    pose proof SearchTree.elements_correct as Hcorrect.
    unfold elements_correct_spec in Hcorrect.
    apply Hcorrect; assumption.
  Qed.
End TreeETableFullyEncapsulated.
```

Unfortunately, the module is now too encapsulated to be useful:

```
Module OverlyEncapsulatedExample.
```

```
  Module StringTreeETableFullyEncapsulated := TreeETableFullyEncapsulated StringVal.
  Import StringTreeETableFullyEncapsulated.
  Open Scope string_scope.
  Fail Example ex1 : get 0 empty = "".
End OverlyEncapsulatedExample.
```

The problem is that the module now hides not only the internal implementation, but also the value type V. The functor was instantiated on StringVal, which defines that type as **string**. But the output of the functor has type ETable, which doesn't reveal anything about V except that it is a `Type`. Note in the output of the following command how V is printed like an axiom would be, indicating that nothing is known (externally) about its implementation:

```
Print OverlyEncapsulatedExample.StringTreeETableFullyEncapsulated.V.
```

We need to selectively expose certain implementation details. We've just seen that V should be exposed. Along with it, we'll also want *default* to be exposed. We can accomplish that with an advanced feature of Coq's module system. The Coq manual doesn't give this feature a name, but OCaml (in which Coq is implemented) calls it a *sharing constraint*, so we'll use that term here. A sharing constraint enables us to constrain definitions shared by modules to be the same.

```
Module Type SimpleTable.
  Parameter key : Type.
  Parameter V : Type.
  Parameter default : V.
  Parameter table : Type.
End SimpleTable.

Module SimpleStringTable1 : SimpleTable.
  Definition key := nat.
  Definition V := string.
  Definition default : string := "".
  Definition table := tree V.
End SimpleStringTable1.

Print SimpleStringTable1.V.

Module Type SimpleTable2 := SimpleTable with Definition V := string.

Module SimpleStringTable2 : SimpleTable2.
  Definition key := nat.
  Definition V := string.
  Definition default : string := "".
  Definition table := tree V.
End SimpleStringTable2.

Print SimpleStringTable2.V.
```

```
Print SimpleStringTable2.default.

Module Type SimpleTable3 := SimpleTable
  with Definition V := string
  with Definition default := ""%string.

Module SimpleStringTable3 : SimpleTable3.
  Definition key := nat.
  Definition V := string.
  Definition default : string := "".
  Definition table := tree V.
End SimpleStringTable3.

Print SimpleStringTable3.V.
Print SimpleStringTable3.default.
```

Putting sharing constraints to use, let's expose V and *default* in our implementation of tree-based tables.

```
Module TreeETableEncapsulated (VT : ValType) : (ETable with Definition V
:= VT.V with Definition default := VT.default).

  Include TreeTable VT.

  Definition rep_ok (t : table) : Prop :=
    BST t.

  Definition bound (k : key) (t : table) : bool :=
    SearchTree.bound k t.

  Definition elements (t : table) : list (key × V) :=
    SearchTree.elements t.

  Theorem empty_ok : rep_ok empty.
  Proof.
    apply empty_tree_BST.
  Qed.

  Theorem set_ok : ∀ (k : key) (v : V) (t : table),
      rep_ok t → rep_ok (set k v t).
  Proof.
    apply insert_BST.
  Qed.

  Theorem bound_empty : ∀ (k : key),
      bound k empty = false.
  Proof.
    reflexivity.
  Qed.

  Theorem bound_set_same : ∀ (k : key) (v : V) (t : table),
      bound k (set k v t) = true.
```

```
  Proof.
    intros k v t. unfold bound, set. induction t; bdall.
  Qed.
  Theorem bound_set_other : ∀ (k k' : key) (v : V) (t : table),
      k ≠ k' → bound k' (set k v t) = bound k' t.
  Proof.
    intros k k' v t Hneq. unfold bound, set. induction t; bdall.
  Qed.
  Theorem elements_complete : ∀ (k : key) (v : V) (t : table),
      rep_ok t →
      bound k t = true →
      get k t = v →
      In (k, v) (elements t).
  Proof.
    intros k v t Hbound Hlookup.
    pose proof SearchTree.elements_complete as Hcomplete.
    unfold elements_complete_spec in Hcomplete.
    apply Hcomplete; assumption.
  Qed.
  Theorem elements_correct : ∀ (k : key) (v : V) (t : table),
      rep_ok t →
      In (k, v) (elements t) →
      bound k t = true ∧ get k t = v.
  Proof.
    intros k v t. simpl. intros Hin.
    pose proof SearchTree.elements_correct as Hcorrect.
    unfold elements_correct_spec in Hcorrect.
    apply Hcorrect; assumption.
  Qed.
End TreeETableEncapsulated.
Module NicelyEncapsulatedExample.
  Module StringTreeETableEncapsulated := TreeETableEncapsulated StringVal.
  Import StringTreeETableEncapsulated.
  Open Scope string_scope.
  Example ex1 : get 0 empty = "".
  Proof.
    Fail reflexivity.
    apply get_empty_default.
  Qed.
  Fail Example ex2 : get 0 (T E 0 "A" E) = "A".
```

End NicelyEncapsulatedExample.

Note what we've happily achieved: we can reason about the behavior of an ADT entirely from its specification, rather than depending on the implementation code. This is what specification comments in interfaces attempt to achieve in real-world code.

Exercise: 4 stars, advanced, optional (elements_spec) Develop a nicely-encapsulated interface and implementation of tree-based tables that exposes the rest of the specification of elements from SearchTree, including the inverses of correctness and completenesss, sortedness, and non-duplication. Send us your solution, so we can include it!

□

10.8 Model-based Specification

The interfaces above have been based on equational specification of tables. Let's consider model-based specifications. Recall from SearchTree that in this style of specification, we

- introduce an abstraction function (or relation) that associates a *concrete value* of the ADT implementation with an *abstract value* in an already well-understood type; and
- show that the concrete and abstract operations are related in a sensible way.

We begin by defining a new map operation, bound, and redefining existing operations to make them more clear in the table specification we are about to write.

```
Definition map_update {V : Type}
           (k : key) (v : V) (m : partial_map V) : partial_map V :=
  update m k v.
Definition map_find {V : Type} := @find V.
Definition empty_map {V : Type} := @Maps.empty V.
```

Now we can define an interface for tables that includes an abstraction function Abs, and specifications written in terms of it.

```
Module Type ETableAbs.
  Parameter table : Type.
  Definition key := nat.
  Parameter V : Type.
  Parameter default : V.
  Parameter empty : table.
  Parameter get : key → table → V.
  Parameter set : key → V → table → table.
  Parameter bound : key → table → bool.
  Parameter elements : table → list (key × V).
```

```
  Parameter Abs : table → partial_map V.
  Parameter rep_ok : table → Prop.

  Axiom empty_ok :
    rep_ok empty.

  Axiom set_ok : ∀ (k : key) (v : V) (t : table),
    rep_ok t → rep_ok (set k v t).

  Axiom empty_relate :
    Abs empty = empty_map.

  Axiom bound_relate : ∀ (t : table) (k : key),
    rep_ok t →
    map_bound k (Abs t) = bound k t.

  Axiom lookup_relate : ∀ (t : table) (k : key),
    rep_ok t →
    map_find default k (Abs t) = get k t.

  Axiom insert_relate : ∀ (t : table) (k : key) (v : V),
    rep_ok t →
    map_update k v (Abs t) = Abs (set k v t).

  Axiom elements_relate : ∀ (t : table),
    rep_ok t →
    Abs t = map_of_list (elements t).

End ETableAbs.
```

Exercise: 4 stars, standard (list_etable_abs) Implement ETableAbs using association lists as the representation type.

```
Module ListETableAbs (VT : ValType) <: ETableAbs.

  Definition V := VT.V.
  Definition default := VT.default.
  Definition key := nat.
  Definition table := list (key × V).

  Definition empty : table
    . Admitted.

  Fixpoint get (k : key) (t : table) : V
    . Admitted.

  Definition set (k : key) (v : V) (t : table) : table
    . Admitted.

  Fixpoint bound (k : key) (t : table) : bool
    . Admitted.

  Definition elements (t : table) : list (key × V)
```

```
. Admitted.
Definition Abs (t : table) : partial_map V
  . Admitted.
Definition rep_ok (t : table) : Prop
  . Admitted.
Theorem empty_ok : rep_ok empty.
Proof.
 Admitted.
Theorem set_ok : ∀ (k : key) (v : V) (t : table),
    rep_ok t → rep_ok (set k v t).
Proof.
 Admitted.
Theorem empty_relate :
  Abs empty = empty_map.
Proof.
 Admitted.
Theorem bound_relate : ∀ (t : table) (k : key),
    rep_ok t →
    map_bound k (Abs t) = bound k t.
Proof.
 Admitted.
Theorem lookup_relate : ∀ (t : table) (k : key),
    rep_ok t →
    map_find default k (Abs t) = get k t.
Proof.
 Admitted.
Theorem insert_relate : ∀ (t : table) (k : key) (v : V),
    rep_ok t →
    map_update k v (Abs t) = Abs (set k v t).
Proof.
 Admitted.
Theorem elements_relate : ∀ (t : table),
    rep_ok t →
    Abs t = map_of_list (elements t).
Proof.
 Admitted.
End ListETableAbs.
Module StringListETableAbs := ListETableAbs StringVal.
```

□

Exercise: 3 stars, standard, optional (TreeTableModel) Give an implementation of ETableAbs using the abstraction function Abs from SearchTree. All the proofs of the *relate* axioms should be simple applications of the lemmas already proved as exercises in that chapter.

```
Definition manual_grade_for_TreeTableModel : option (nat×string) := None.
```

☐

Exercise: 2 stars, advanced, optional (TreeTableModel') Repeat the previous exercise, this time using the alternative **Abs'** function from SearchTree. Hint: Just tweak your solution to the previous exercise.

☐

10.9 Summary of ADT Verification

With equational specifications:

- Define a *representation invariant* to characterize which values of the *representation type* are legal. Prove that each operation on the representation type *preserves* the representation invariant.
- Using the representation invariant, verify the equational specification.

With model-based specifications:

- Define and verify preservation of the the representation invariant.
- Define an *abstraction function* that relates the representation type to another type that is easier to reason about.
- Prove that operations of the abstract and concrete types commute with the abstraction function.

10.10 Another ADT: Queue

Here is an interface and algebraic specification for FIFO queues. To ensure totality, peek takes a default value and deq returns the empty queue when applied to the empty queue.

```
Module Type QUEUE.
  Parameter V : Type.
  Parameter queue : Type.
  Parameter empty: queue.
  Parameter is_empty : queue → bool.
```

```
  Parameter enq : queue → V → queue.
  Parameter deq : queue → queue.
  Parameter peek : V → queue → V.
  Axiom is_empty_empty : is_empty empty = true.
  Axiom is_empty_nonempty : ∀ q v, is_empty (enq q v) = false.
  Axiom peek_empty : ∀ d, peek d empty = d.
  Axiom peek_nonempty : ∀ d q v, peek d (enq q v) = peek v q.
  Axiom deq_empty : deq empty = empty.
  Axiom deq_nonempty : ∀ q v, deq (enq q v) = if is_empty q then q else enq (deq q) v.
End Queue.
```

Exercise: 3 stars, standard (list_queue) Implement that interface and verify your implementation. As the representation type, use list V. At least one operation will have to be linear time; we recommend that it be enq. All the proofs should be quite easy.

```
Module ListQueue <: Queue.
  Definition V := nat.    Definition queue := list V.

  Definition empty : queue
    . Admitted.

  Definition is_empty (q : queue) : bool
    . Admitted.

  Definition enq (q : queue) (v : V) : queue
    . Admitted.

  Definition deq (q : queue) : queue
    . Admitted.

  Definition peek (default : V) (q : queue) : V
    . Admitted.

  Theorem is_empty_empty : is_empty empty = true.
  Proof.
   Admitted.

  Theorem is_empty_nonempty : ∀ q v,
      is_empty (enq q v) = false.
  Proof.
   Admitted.

  Theorem peek_empty : ∀ d,
      peek d empty = d.
  Proof.
   Admitted.

  Theorem peek_nonempty : ∀ d q v,
      peek d (enq q v) = peek v q.
```

```
  Proof.
    Admitted.
  Theorem deq_empty :
    deq empty = empty.
  Proof.
    Admitted.
  Theorem deq_nonempty : ∀ q v,
      deq (enq q v) = if is_empty q then q else enq (deq q) v.
  Proof.
    Admitted.
End ListQueue.
```

□

Here is an interface and model-based specification for queues. We omit rep_ok from it, because our intended implementation isn't going to require a representation invariant.

```
Module Type QueueAbs.
  Parameter V : Type.
  Parameter queue : Type.
  Parameter empty : queue.
  Parameter is_empty : queue → bool.
  Parameter enq : queue → V → queue.
  Parameter deq : queue → queue.
  Parameter peek : V → queue → V.
  Parameter Abs : queue → list V.
  Axiom empty_relate : Abs empty = [].
  Axiom enq_relate : ∀ q v, Abs (enq q v) = (Abs q) ++ [v].
  Axiom peek_relate : ∀ d q, peek d q = hd d (Abs q).
  Axiom deq_relate : ∀ q, Abs (deq q) = tl (Abs q).
End QueueAbs.
```

Exercise: 3 stars, standard (two_list_queue) Below is an implementation of QueueAbs using two lists. It achieves amortized constant-time performance, improving on the single-list implementation. Verify this implementation.

```
Module TwoListQueueAbs <: QueueAbs.
  Definition V := nat.
  Definition queue : Type := list V × list V.
  Definition Abs '((f, b) : queue) : list V :=
    f ++ (rev b).
  Definition empty : queue :=
    ([], []).
  Definition is_empty (q: queue) :=
```

```
  match q with
  | ([], []) ⇒ true
  | _ ⇒ false
  end.
Definition enq '((f, b) : queue) (v : V) :=
  (f, v :: b).
Definition deq (q : queue) :=
  match q with
  | ([], []) ⇒ ([], [])
  | ([], b) ⇒
    match rev b with
    | [] ⇒ ([], [])
    | _ :: f ⇒ (f, [])
    end
  | (_ :: f, b) ⇒ (f, b)
  end.
Definition peek (d : V) (q : queue) :=
  match q with
  | ([], []) ⇒ d
  | ([], b) ⇒
    match rev b with
    | [] ⇒ d
    | v :: _ ⇒ v
    end
  | (v :: _, _) ⇒ v
  end.
Theorem empty_relate : Abs empty = [].
Proof.
 Admitted.
Theorem enq_relate : ∀ q v,
    Abs (enq q v) = (Abs q) ++ [v].
Proof.
 Admitted.
Theorem peek_relate : ∀ d q,
    peek d q = hd d (Abs q).
Proof.
 Admitted.
Theorem deq_relate : ∀ q,
    Abs (deq q) = tl (Abs q).
Proof.
 Admitted.
```

End TwoListQueueAbs.

□

10.11 Representation Invariants and Subset Types

In specifications thus far, whenever there was a representation invariant that needed to be enforced we had to add it as a precondition and postcondition for operations. For example, the correctness of the table get operation depended on its table input satisfying rep_ok. We had to write propositions like ∀ (t : table), rep_ok t → ..., in which we had a type table with lots of values, and we got rid of some of those values by requiring rep_ok to hold of them.

Coq makes it possible to directly express the requirement that values of a type must satisfy a proposition. The type

{x : A | P}

is the type of all values x of type A that satisfy property P, which itself has type $A \to$ Prop. The notation is deliberately suggestive of set-builder notation used in mathematics. Such types are known as *subset types*.

10.11.1 Example: The Even Naturals

We can define the subset type of even natural numbers using the property *Nat.Even* from the standard library:

```
Definition even_nat := {x : nat | Nat.Even x}.
```

But when we try to say that 2 is an even_nat, Coq rejects the definition:

```
Fail Definition two : even_nat := 2.
```

The problem is that 2 is a **nat**, but we haven't proved *Nat.Even* 2. The proof is easy:

```
Lemma Even2 : Nat.Even 2.
Proof. ∃ 1. reflexivity. Qed.
```

Now we can provide that proof to convince Coq two is an even_nat. We can do that with function exist, which is suggestive of "there exists an x : A such that P x."

```
Check exist : ∀ {A : Type} (P : A → Prop) (x : A), P x → {x : A | P x}.
Definition two : even_nat := exist Nat.Even 2 Even2.
```

Another way of constructing two is to enter Coq's proof scripting mode and use tactics. We saw this briefly in *ProofObjects*.

```
Definition two' : even_nat.
Proof.
  apply exist with (x := 2).
  ∃ 1. reflexivity.
Defined.
```

That technique is often useful with subset types, because it helps us more easily build the proof objects, rather than have to write them ourselves.

A value of type even_nat is like a "package" containing the **nat** and the proof that the **nat** is even. We have to use functions to extract those components from the package.

```
Fail Example plus_two : 1 + two = 3.
Check proj1_sig : ∀ {A : Type} {P : A → Prop} (e : {x : A | P x}), A.
Example plus_two : 1 + proj1_sig two = 3.
Proof. reflexivity. Qed.
Check proj2_sig : ∀ {A : Type} {P : A → Prop} (e : {x : A | P x}), P (proj1_sig e).
Example Even2' : Nat.Even 2 := proj2_sig two.
```

10.11.2 Defining Subset Types

Like nearly everything else we've seen in Coq's logic, subset types are actually defined in the standard library rather than being built-in to the language.

```
Module SigSandbox.
```

Subset types are just a syntactic notation for **sig**:

```
Inductive sig {A : Type} (P : A → Prop) : Type :=
| exist (x : A) : P x → sig P.
Notation "{ x : A | P }" := (sig A (fun x ⇒ P)).
```

The name **sig** is short for the Greek capital letter sigma, because subset types are similar to something known in type theory as *sigma types*, aka *dependent sums*.

Subset types and existential quantification are quite similar. Recall how the latter is defined:

```
Inductive ex {A : Type} (P : A → Prop) : Prop :=
| ex_intro (x : A) : P x → ex P.
```

The only difference is that **sig** creates a Type whereas **ex** creates a Prop. That is, the former is computational in content, whereas the latter is logical. Therefore we can pattern match to recover the witness from a **sig**, but we cannot do the same with an **ex**.

```
Definition proj1_sig {A : Type} {P : A → Prop} (e : sig P) : A :=
  match e with
  | exist _ x _ ⇒ x
  end.
Definition proj2_sig {A : Type} {P : A → Prop} (e : sig P) : P (proj1_sig e) :=
  match e with
  | exist _ _ p ⇒ p
  end.
Fail Definition proj1_ex {A : Type} {P : A → Prop} (e : ex P) : A :=
```

```
    match e with
    | ex_intro _ x _ ⇒ x
    end.
End SigSandbox.
```

10.11.3 Example: Vectors

A *vector* is a list of a known length. Type vector X contains values (xs, n) with a particular representation invariant: n must be the length of xs.

```
Definition vector (X : Type) :=
  { '(xs , n) : list X × nat | length xs = n }.
```

The type itself enforces the representation invariant, because a value of type vector X cannot be constructed without first proving that the invariant holds.

Exercise: 1 star, standard (a_vector) Construct any vector of your choice.

```
Example a_vector : vector nat.
Proof.
   Admitted.
```

☐

Exercise: 2 stars, standard (vector_cons_correct) Define a cons operation on vectors. Remember to end with `Defined` rather than `Qed`.

```
Definition vector_cons {X : Type} (x : X) (v : vector X) : vector X.
Proof.
   Admitted.
```

Prove the correctness of your cons operation.

```
Definition list_of_vector {X : Type} (v : vector X) : list X :=
  fst (proj1_sig v).
Theorem vector_cons_correct : ∀ (X : Type) (x : X) (v : vector X),
    list_of_vector (vector_cons x v) = x :: (list_of_vector v).
Proof.
   Admitted.
```

☐

Exercise: 2 stars, standard (vector_app_correct) Define an append operation on vectors.

```
Definition vector_app {X : Type} (v1 v2 : vector X) : vector X.
Proof.
   Admitted.
```

Prove the correctness of your append operation.

```
Theorem vector_app_correct : ∀ (X : Type) (v1 v2 : vector X),
    list_of_vector (vector_app v1 v2) =
    list_of_vector v1 ++ list_of_vector v2.
Proof.
```

Admitted.

□

10.11.4 Using Subset Types to Enforce the BST Invariant

Let's use subset types to reimplement tree-based tables with an elements operation. Previously we had to add rep_ok to the interface and specifications. With subset types we can eliminate that.

```
Module Type ETableSubset.
  Include Table.
```

Note: no rep_ok anywhere.

```
  Parameter bound : key → table → bool.
  Parameter elements : table → list (key × V).

  Axiom bound_empty : ∀ (k : key),
      bound k empty = false.

  Axiom bound_set_same : ∀ (k : key) (v : V) (t : table),
      bound k (set k v t) = true.

  Axiom bound_set_other : ∀ (k k' : key) (v : V) (t : table),
      k ≠ k' → bound k' (set k v t) = bound k' t.

  Axiom elements_complete : ∀ (k : key) (v : V) (t : table),
      bound k t = true →
      get k t = v →
      In (k, v) (elements t).

  Axiom elements_correct : ∀ (k : key) (v : V) (t : table),
      In (k, v) (elements t) →
      bound k t = true ∧ get k t = v.

End ETableSubset.

Module TreeETableSubset (VT : ValType) <: ETableSubset.
  Definition V := VT.V.
  Definition default := VT.default.
  Definition key := nat.
```

table now is required to enforce BST.

```
  Definition table := { t : tree V | BST t }.
```

Now instead of proving separate theorems that operations return valid representations, the proofs are "baked in" to the operations.

```
Definition empty : table.
  Proof.
    apply (exist _ empty_tree).
    apply empty_tree_BST.
  Defined.
```

Now we insert a projection to get to the tree.

```
Definition get (k : key) (t : table) : V :=
    lookup default k (proj1_sig t).

  Definition set (k : key) (v : V) (t : table) : table.
  Proof.
    destruct t as [t Ht].
    apply (exist _ (insert k v t)).
    apply insert_BST. assumption.
  Defined.

  Definition bound (k : key) (t : table) : bool :=
    SearchTree.bound k (proj1_sig t).

  Definition elements (t : table) : list (key × V) :=
    elements (proj1_sig t).

  Theorem get_empty_default: ∀ (k : key),
      get k empty = default.
  Proof.
    apply lookup_empty.
  Qed.
```

Now the rest of the proofs require minor modifications to destruct the table to get the tree and the representation invariant, and use the latter where needed.

```
  Theorem get_set_same: ∀ (k : key) (v : V) (t : table),
      get k (set k v t) = v.
  Proof.
    intros. unfold get, set.
    destruct t as [t Hbst]. simpl.
    apply lookup_insert_eq.
  Qed.

  Theorem get_set_other: ∀ (k k' : key) (v : V) (t : table),
      k ≠ k' → get k' (set k v t) = get k' t.
  Proof.
    intros. unfold get, set.
    destruct t as [t Hbst]. simpl.
    apply lookup_insert_neq. assumption.
  Qed.
```

```
Theorem bound_empty : ∀ (k : key),
    bound k empty = false.
Proof.
  reflexivity.
Qed.

Theorem bound_set_same : ∀ (k : key) (v : V) (t : table),
    bound k (set k v t) = true.
Proof.
  intros k v t. unfold bound, set.
  destruct t as [t Hbst]. simpl in *.
  induction t; inv Hbst; bdall.
Qed.

Theorem bound_set_other : ∀ (k k' : key) (v : V) (t : table),
    k ≠ k' → bound k' (set k v t) = bound k' t.
Proof.
  intros k k' v t Hneq. unfold bound, set.
  destruct t as [t Hbst]. simpl in *.
  induction t; inv Hbst; bdall.
Qed.

Theorem elements_complete : ∀ (k : key) (v : V) (t : table),
    bound k t = true →
    get k t = v →
    In (k, v) (elements t).
Proof.
  intros k v t Hbound Hlookup.
  pose proof SearchTree.elements_complete as Hcomplete.
  unfold elements_complete_spec in Hcomplete.
  apply Hcomplete with default; try assumption.
  destruct t as [t Hbst]. assumption.
Qed.

Theorem elements_correct : ∀ (k : key) (v : V) (t : table),
    In (k, v) (elements t) →
    bound k t = true ∧ get k t = v.
Proof.
  intros k v t. simpl. intros Hin.
  pose proof SearchTree.elements_correct as Hcorrect.
  unfold elements_correct_spec in Hcorrect.
  apply Hcorrect; try assumption.
  destruct t as [t Hbst]. assumption.
Qed.
End TreeETableSubset.
```

Exercise: 4 stars, advanced (ListsETable) Implement ETableSubset using association lists that are not permitted to contain duplicate keys. Enforce that representation invariant with a subset type. Note that you do not have to keep the lists in a sorted order. The implementation of elements should therefore just be quite easy: just return the association list. The implementation of `set`, though, will have to be a linear-time operation.

```
Definition manual_grade_for_ListsETable : option (nat×string) := None.
```

□

Chapter 11

Library VFA.Extract

11.1 Extract: Running Coq Programs in OCaml

Coq's `Extraction` feature enables you to write a functional program inside Coq, use Coq's logic to prove some correctness properties about it, and translate it into an OCaml program that you can compile with your optimizing OCaml compiler. Haskell is also supported.

The `Extraction` chapter of *Logical Foundations* has a simple example of Coq's program extraction features, but it's not required reading. This chapter starts from scratch and goes deeper.

```
From VFA Require Import Perm.
Require Extraction.
```

11.2 Extraction

As an example, let's extract insertion sort, which we implemented in `Sort`.

```
Fixpoint ins (i : nat) (l : list nat) :=
  match l with
  | [] ⇒ [i]
  | h :: t ⇒ if i <=? h then i :: h :: t else h :: ins i t
  end.

Fixpoint sort (l : list nat) : list nat :=
  match l with
  | [] ⇒ []
  | h :: t ⇒ ins h (sort t)
  end.
```

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

```
Extraction sort.
```

You can see the translation of sort from Coq to OCaml in your IDE. Examine it there, and notice the similarities and differences. To get the whole program, we need `Recursive Extraction`:

```
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, i.e. callable by, ordinary OCaml code. So we want to use OCaml's standard definition of **bool** in place of Coq's inductive definition, **bool**. You'll notice the same issue with lists. The following directive causes Coq to use OCaml's definitions of **bool** and **list** in the extracted code:

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

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 linear time, proportional to the difference in value between n and m.

We could instead use Coq's **Z**, which is a binary representation of integers. But that is logarithmic-time, not constant.

```
Require Import ZArith.
Open Scope Z_scope.

Fixpoint insertZ (i : Z) (l : list Z) :=
  match l with
  | [] ⇒ [i]
  | h :: t ⇒ if i <=? h then i :: h :: t else h :: insertZ i t
  end.

Fixpoint sortZ (l : list Z) : list Z :=
  match l with
  | [] ⇒ []
  | h :: t ⇒ insertZ h (sortZ t)
  end.

Recursive Extraction sortZ.
```

Of course, for that extraction to be meaningful, we would need to prove that sortZ is a sorting algorithm.

Other alternatives include:

- Extract **nat** directly to OCaml *int*. But *int* is finite (2^63 in modern implementations), so there are theorems we could prove in Coq that wouldn't hold in OCaml.

- Use Coq's *Int63*, which faithfully models 63-bit cyclic arithmetic, and extract directly to OCaml *int*. But that's painful.

- Define and axiomatize our own lightweight abstract type of naturals, but extract it to OCaml *int*. But, this is dangerous! If our axioms are inconsistent, we can prove anything at all. If they are not faithful to OCaml, our proofs will be meaningless.

11.3 Lightweight Extraction to *int*

We begin by positing a Coq type *int* that will be extracted to OCaml's *int*:

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

We'll abstract OCaml *int* to Coq **Z**. Every *int* does have a representation as a **Z**, though the other direction cannot hold.

```
Parameter Abs : int → Z.
Axiom Abs_inj: ∀ (n m : int), Abs n = Abs m → n = m.
```

Nothing else is known so far about *int*. Let's add a less-than operators, which are extracted to OCaml's:

```
Parameter ltb: int → int → bool.
Extract Inlined Constant ltb ⇒ "(<)".
Axiom ltb_lt : ∀ (n m : int), ltb n m = true ↔ Abs n < Abs m.
```

```
Parameter leb: int → int → bool.
Extract Inlined Constant leb ⇒ "(<=)".
Axiom leb_le : ∀ (n m : int), leb n m = true ↔ Abs n ≤ Abs m.
```

Those axioms are sound: OCaml's $<$ and $\leq$ are consistent with Coq's on any *int*. Note that we do not give extraction directives for Abs, *ltb_lt*, or *leb_le*. They will not appear in programs, only in proofs –which are not meant to be extracted.

You could imagine doing the same thing we just did with $(+)$, but that would be wrong:

Parameter ocaml_plus : int -> int -> int. Extract Inlined Constant ocaml_plus => "(+)". Axiom ocaml_plus_plus: forall a b c: int, ocaml_plus a b = c <-> Abs a + Abs b = Abs 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,

Abs max_int + Abs max_int = Abs (ocaml_plus max_int max_int)

which is not true in OCaml because of overflow.

In Perm we proved several theorems showing that Boolean operators were reflected in propositions. Below, we do that for *int* and **Z** comparisons.

```
Lemma int_ltb_reflect : ∀ x y, reflect (Abs x < Abs y) (ltb x y).
Proof.
  intros x y.
  apply iff_reflect. symmetry. apply ltb_lt.
```

```
Qed.
Lemma int_leb_reflect : ∀ x y, reflect (Abs x ≤ Abs y) (leb x y).
Proof.
  intros x y.
  apply iff_reflect. symmetry. apply leb_le.
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.
Lemma Z_leb_reflect : ∀ x y, reflect (x ≤ y) (Z.leb x y).
Proof.
  intros x y.
  apply iff_reflect. symmetry. apply Z.leb_le.
Qed.
Lemma Z_gtb_reflect : ∀ x y, reflect (x > y) (Z.gtb x y).
Proof.
  intros x y.
  apply iff_reflect. symmetry. rewrite Z.gtb_ltb. rewrite Z.gt_lt_iff. apply Z.ltb_lt.
Qed.
Lemma Z_geb_reflect : ∀ x y, reflect (x ≥ y) (Z.geb x y).
Proof.
  intros x y.
  apply iff_reflect. symmetry. rewrite Z.geb_leb. rewrite Z.ge_le_iff. apply Z.leb_le.
Qed.
```

Now we upgrade *bdall* to work with **Z** and *int*.

```
Hint Resolve
      int_ltb_reflect int_leb_reflect
      Z_eqb_reflect Z_ltb_reflect Z_leb_reflect Z_gtb_reflect Z_geb_reflect
  : bdestruct.
Ltac bdestruct_guard:=
  match goal with
  | ⊢ context [ if Nat.eqb ?X ?Y then _ else _] ⇒ bdestruct (Nat.eqb X Y)
  | ⊢ context [ if Nat.ltb ?X ?Y then _ else _] ⇒ bdestruct (Nat.ltb X Y)
```

```
  | ⊢ context [ if Nat.leb ?X ?Y then _ else _] ⇒ bdestruct (Nat.leb X Y)
  | ⊢ context [ if Z.eqb ?X ?Y then _ else _] ⇒ bdestruct (Z.eqb X Y)
  | ⊢ context [ if Z.ltb ?X ?Y then _ else _] ⇒ bdestruct (Z.ltb X Y)
  | ⊢ context [ if Z.leb ?X ?Y then _ else _] ⇒ bdestruct (Z.leb X Y)
  | ⊢ context [ if Z.gtb ?X ?Y then _ else _] ⇒ bdestruct (Z.gtb X Y)
  | ⊢ context [ if Z.geb ?X ?Y then _ else _] ⇒ bdestruct (Z.geb X Y)
  | ⊢ context [ if ltb ?X ?Y then _ else _] ⇒ bdestruct (ltb X Y)
  | ⊢ context [ if leb ?X ?Y then _ else _] ⇒ bdestruct (leb X Y)
  end.

Ltac bdall :=
  repeat (simpl; bdestruct_guard; try omega; auto).
```

11.4 Insertion Sort, Extracted

We're ready to state insertion sort with *int*, and to extract it:

```
Fixpoint ins_int (i : int) (l : list int) :=
  match l with
  | [] ⇒ [i]
  | h :: t ⇒ if leb i h then i :: h :: t else h :: ins_int i t
  end.

Fixpoint sort_int (l : list int) : list int :=
  match l with
  | [] ⇒ []
  | h :: t ⇒ ins_int h (sort_int t)
  end.

Recursive Extraction sort_int.
```

Again, for that extraction to be meaningful, we need to prove that sort_int is a sorting algorithm. We can do that with the same techniques we used in `Sort`. In particular, `omega` works with **Z**, so we can enjoy automation without having to do any unnecessary work axiomatizing and proving lemmas about *int*.

```
Inductive sorted : list int → Prop :=
| sorted_nil:
    sorted []
| sorted_1: ∀ x,
    sorted [x]
| sorted_cons: ∀ x y l,
    Abs x ≤ Abs y → sorted (y :: l) → sorted (x :: y :: l).

Hint Constructors sorted.
```

Exercise: 3 stars, standard (sort_int_correct) Prove the correctness of sort_int by adapting your solution to insertion_sort_correct from Sort.

```
Theorem sort_int_correct : ∀ (al : list int),
    Permutation al (sort_int al) ∧ sorted (sort_int al).
Proof.
  Admitted.
```

□

11.5 Binary Search Trees, Extracted

We can reimplement BSTs with *int* keys.

```
Definition key := int.

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

Arguments E {V}.
Arguments T {V}.

Definition empty_tree {V : Type} : tree V := E.

Fixpoint lookup {V : Type} (default : V) (x : key) (t : tree V) : V :=
  match t with
  | E ⇒ default
  | T l k v r ⇒ if ltb x k then lookup default x l
                else if ltb k x then lookup default x r
                     else v
  end.

Fixpoint insert {V : Type} (x : key) (v : V) (t : tree V) : tree V :=
  match t with
  | E ⇒ T E x v E
  | T l y v' r ⇒ if ltb x y then T (insert x v l) y v' r
                 else if ltb y x then T l y v' (insert x v r)
                      else T l x v r
  end.

Fixpoint elements_tr {V : Type}
         (t : tree V) (acc : list (key × V)) : list (key × V) :=
  match t with
  | E ⇒ acc
  | T l k v r ⇒ elements_tr l ((k , v) :: elements_tr r acc)
  end.

Definition elements {V : Type} (t : tree V) : list (key × V) :=
```

elements_tr t [].

Theorem lookup_empty : ∀ (V : Type) ($default$: V) (k : key),
lookup $default$ k empty_tree = $default$.
Proof. auto. Qed.

Exercise: 2 stars, standard (lookup_insert_eq) Theorem lookup_insert_eq :
∀ (V : Type) ($default$: V) (t : **tree** V) (k : key) (v : V),
lookup $default$ k (insert k v t) = v.
Proof.
Admitted.
□

Exercise: 3 stars, standard (lookup_insert_neq) Theorem lookup_insert_neq :
∀ (V : Type) ($default$: V) (t : **tree** V) (k k' : key) (v : V),
$k \neq k'$ → lookup $default$ k' (insert k v t) = lookup $default$ k' t.
Proof.
Admitted.
□

Exercise: 5 stars, standard, optional (int_elements) Port the definition of **BST** and re-prove the properties of elements for *int*-keyed trees. Send us your solution so we can include it!
□

Now see the extraction in your IDE:

Extract Inductive **prod** ⇒ "(*)" ["(,)"]. Recursive Extraction *empty_tree insert lookup elements.*

11.6 Performance Tests

Let's measure the performance of BSTs. First, we extract to an OCaml file:

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

Second, in the same directory as this file (*Extract.v*) you will find the file *test_searchtree.ml*. You can run it using the OCaml toplevel with these commands:

```
# #use "searchtree.ml";;
# #use "test_searchtree.ml";;
```

On a recent machine with a 2.9 GHz Intel Core i9 that prints:

Insert and lookup 1000000 random integers in .889566 seconds. Insert and lookup 20000 random integers in 0.009918 seconds. Insert and lookup 20000 consecutive integers in 2.777335 seconds.

That execution uses the bytecode interpreter. The native compiler will have better performance:

```
$ ocamlopt -c searchtree.mli searchtree.ml
$ ocamlopt searchtree.cmx -open Searchtree test_searchtree.ml -o test_searchtree
$ ./test_searchtree
```

On the same machine that prints,

Insert and lookup 1000000 random integers in 0.488973 seconds. Insert and lookup 20000 random integers in 0.003237 seconds. Insert and lookup 20000 consecutive integers in 0.387535 seconds.

Of course, the reason why the performance is so much worse with consecutive integers is that BSTs exhibit worst-case performance under that workload: linear time instead of logarithmic. We need balanced search trees to achieve logarithmic. Redblack will do that.

Chapter 12

Library VFA.Redblack

12.1 Redblack: Red-Black Trees

Red-black trees are a kind of *balanced* binary search tree (BST). Keeping the tree balanced ensures that the worst-case running time of operations is logarithmic rather than linear.

This chapter uses Okasaki's algorithms for red-black trees. If you don't recall those or haven't seem them in a while, read one of the following:

- Red-Black Trees in a Functional Setting, by Chris Okasaki. *Journal of Functional Programming*, 9(4):471-477, July 1999. Available from https://doi.org/10.1017/S0956796899003494. Archived at https://web.archive.org/web/20070926220746/http://www.eecs.usma.edu/webs/people/

- *Purely Functional Data Structures*, by Chris Okasaki. Section 3.3. Cambridge University Press, 1998.

You can also consult Wikipedia or other standard textbooks, though they are likely to use different, imperative implementations.

This chapter is based on the Coq standard library module *MSetRBT*, which can be found at *https://coq.inria.fr/distrib/current/stdlib/Coq.MSets.MSetRBT.html*. The design decisions for that module are described in the following paper:

- Efficient Verified Red-Black Trees, by Andrew W. Appel, September 2011. Available from http://www.cs.princeton.edu/~appel/papers/redblack.pdf.

```
From Coq Require Import String.
From Coq Require Import Logic.FunctionalExtensionality.
From VFA Require Import Perm.
From VFA Require Import Extract.
Open Scope Z_scope.
```

12.2 Implementation

A `Section` enables declaration of some `Variables`, which are in scope throughout. That saves us from having to write V and *default* as arguments to most of the definitions in the chapter. The variables do get inserted by Coq as arguments after the section is closed.

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

We use the *int* type axiomatized in `Extract` as the key type.

```
  Definition key := int.

  Inductive color := Red | Black.

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

  Definition empty_tree : tree :=
    E.
```

The lookup implementation for red-black trees is exactly the same as the lookup for BSTs, except that the T constructor carries a color component that is ignored.

```
  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.
```

We won't explain the insert algorithm here; read Okasaki's work if you want to understand it. In fact, you'll need very little understanding of it to follow along with the verification below. It uses balance and ins as helpers:

- ins recurses down the tree to find where to insert, and is mostly the same as the BST insert algorithm.
- balance takes care of rebalancing the tree on the way back up.

```
  Definition balance (rb : color) (t1 : tree) (k : key) (vk : V) (t2 : tree) : tree :=
    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.
```

```
Fixpoint ins (x : key) (vx : V) (t : tree) : tree :=
  match t 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 make_black (t : tree) : tree :=
  match t with
  | E ⇒ E
  | T _ a x vx b ⇒ T Black a x vx b
  end.
```

```
Definition insert (x : key) (vx : V) (t : tree) :=
  make_black (ins x vx t).
```

The elements implementation is the same as for BSTs, except that it ignores colors.

```
Fixpoint elements_tr (t : tree) (acc: list (key × V)) : list (key × V) :=
  match t with
  | E ⇒ acc
  | T _ l k v r ⇒ elements_tr l ((k, v) :: elements_tr r acc)
  end.
```

```
Definition elements (t : tree) : list (key × V) :=
  elements_tr t [].
```

12.3 Case-Analysis Automation

Before verifying the correctness of our red-black tree implementation, let's warm up by proving that the result of any insert is a nonempty tree.

```
Lemma ins_not_E : ∀ (x : key) (vx : V) (t : tree),
    ins x vx t ≠ E.
```

```
Proof.
  intros. destruct t; simpl.
  discriminate.

  destruct (ltb x k).
  unfold balance.

  destruct c.
  discriminate.

  destruct (ins x vx t1).

  destruct t2.
  discriminate.

  match goal with
  | ⊢ match ?c with Red ⇒ _ | Black ⇒ _ end ≠ _ ⇒ destruct c
  | ⊢ match ?t with E ⇒ _ | T _ _ _ _ _ _ ⇒ _ end ≠ _⇒ destruct t
  end.

  repeat
    match goal with
    | ⊢ match ?c with Red ⇒ _ | Black ⇒ _ end ≠ _ ⇒ destruct c
    | ⊢ match ?t with E ⇒ _ | T _ _ _ _ _ _ ⇒ _ end ≠ _⇒ destruct t
    end.

  discriminate.

  match goal with
  | ⊢ T _ _ _ _ _ _ ≠ E ⇒ discriminate
  end.

Abort.

Lemma ins_not_E : ∀ (x : key) (vx : V) (t : tree),
    ins x vx t ≠ E.
Proof.
  intros. destruct t; simpl.
  - discriminate.
  - unfold balance.
    repeat
      match goal with
      | ⊢ (if ?x then _ else _) ≠ _ ⇒ destruct x
      | ⊢ match ?c with Red ⇒ _ | Black ⇒ _ end ≠ _⇒ destruct c
      | ⊢ match ?t with E ⇒ _ | T _ _ _ _ _ _ ⇒ _ end ≠ _⇒ destruct t
      | ⊢ T _ _ _ _ _ _ ≠ E ⇒ discriminate
      end.
```

```
Qed.
```

This automation of case analysis will be quite useful in the rest of our development.

12.4 The BST Invariant

The BST invariant is mostly the same for red-black trees as it was for ordinary BSTs as defined in SearchTree. We adapt it by ignoring the color of each node, and changing from **nat** keys to *int*.

ForallT P t holds if $P\ k\ v$ holds for every (k, v) node of tree t.

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

Inductive BST : tree → Prop :=
| ST_E : BST E
| ST_T : ∀ (c : color) (l : tree) (k : key) (v : V) (r : tree),
    ForallT (fun k' _ ⇒ (Abs k') < (Abs k)) l →
    ForallT (fun k' _ ⇒ (Abs k') > (Abs k)) r →
    BST l →
    BST r →
    BST (T c l k v r).

Lemma empty_tree_BST: BST empty_tree.
Proof.
  unfold empty_tree. constructor.
Qed.
```

Let's show that insert preserves the BST invariant, that is:

```
Theorem insert_BST : ∀ t v k,
    BST t →
    BST (insert k v t).
Abort.
```

It will take quite a bit of work, but automation will help.

First, we show that if a non-empty tree would be a BST, then the balanced version of it is also a BST:

```
Lemma balance_BST: ∀ (c : color) (l : tree) (k : key) (v : V) (r : tree),
    ForallT (fun k' _ ⇒ (Abs k') < (Abs k)) l →
    ForallT (fun k' _ ⇒ (Abs k') > (Abs k)) r →
    BST l →
    BST r →
    BST (balance c l k v r).
```

```
Proof.
  intros c l k v r PL PR BL BR. unfold balance.

  repeat
    match goal with
    | ⊢ BST (match ?c with Red ⇒ _ | Black ⇒ _ end) ⇒ destruct c
    | ⊢ BST (match ?t with E ⇒ _ | T _ _ _ _ _ ⇒ _ end) ⇒ destruct t
    end.

  - constructor. assumption. assumption. assumption. assumption.
  - constructor; auto.
  - constructor; auto.
  -
    constructor; auto.
    + simpl in *. repeat split.
      destruct PR as [? _]. omega.
    + simpl in *. repeat split.
      × inv BR. simpl in *. destruct H5 as [? _]. omega.
      × inv BR. simpl in *. destruct H5 as [_ [? _]]. auto.
      × inv BR. simpl in *. destruct H5 as [_ [_ ?]]. auto.
    + constructor; auto.
    + inv BR. inv H7. constructor; auto.
  - constructor; auto.
  -

Abort.
```

Let's use some of what we discovered above to automate. Whenever we have a subgoal of the form

ForallT _(T _)

we can split it. Whenever we have a hypothesis of the form

BST (T _)

we can invert it. And with a hypothesis

ForallT _(T _)

we can simplify then destruct it. Actually, the simplification is optional – Coq will do the destruct without needing the simplification. Anything else seems able to be finished with `constructor`, `auto`, and `omega`. Let's see how far that can take us...

```
Lemma balance_BST: ∀ (c : color) (l : tree) (k : key) (v : V) (r : tree),
    ForallT (fun k' _ ⇒ (Abs k') < (Abs k)) l →
    ForallT (fun k' _ ⇒ (Abs k') > (Abs k)) r →
    BST l →
    BST r →
```

```
      BST (balance c l k v r).
Proof.
  intros. unfold balance.

  repeat
    (match goal with
      | ⊢ BST (match ?c with Red ⇒ _ | Black ⇒ _ end) ⇒ destruct c
      | ⊢ BST (match ?t with E ⇒ _ | T _ _ _ _ _ ⇒ _ end) ⇒ destruct t
      | ⊢ ForallT _ (T _ _ _ _ _) ⇒ repeat split
      | H: ForallT _ (T _ _ _ _ _) ⊢ _ ⇒ destruct H as [? [? ?] ]
      | H: BST (T _ _ _ _ _) ⊢ _ ⇒ inv H
      end;
      (try constructor; auto; try omega)).
```

41 cases remain. It's a little disappointing that we didn't clear more of them. Let's look at why are we stuck.

All the remaining subgoals appear to be about proving an inequality over all the nodes of a subtree. For example, the first subgoal follows from the hypotheses

ForallT (fun (k' : int) (_: V) => Abs k' > Abs k0) r2 Abs k1 < Abs k0

The other goals look similar.

```
Abort.
```

To make progress, we can set up some helper lemmas.

```
Lemma ForallT_imp : ∀ (P Q : int → V → Prop) t,
    ForallT P t →
    (∀ k v, P k v → Q k v) →
    ForallT Q t.
Proof.
  induction t; intros.
  - auto.
  - destruct H as [? [? ?]]. repeat split; auto.
Qed.

Lemma ForallT_greater : ∀ t k k0,
    ForallT (fun k' _ ⇒ Abs k' > Abs k) t →
    Abs k > Abs k0 →
    ForallT (fun k' _ ⇒ Abs k' > Abs k0) t.
Proof.
  intros. eapply ForallT_imp; eauto.
  intros. simpl in H1. omega.
Qed.

Lemma ForallT_less : ∀ t k k0,
    ForallT (fun k' _ ⇒ Abs k' < Abs k) t →
    Abs k < Abs k0 →
```

```
    ForallT (fun k' _ ⇒ Abs k' < Abs k0) t.
Proof.
  intros; eapply ForallT_imp; eauto.
  intros. simpl in H1. omega.
Qed.
```

Now we can return to automating the proof.

```
Lemma balance_BST: ∀ (c : color) (l : tree) (k : key) (v : V) (r : tree),
    ForallT (fun k' _ ⇒ (Abs k') < (Abs k)) l →
    ForallT (fun k' _ ⇒ (Abs k') > (Abs k)) r →
    BST l →
    BST r →
    BST (balance c l k v r).
Proof.
  intros. unfold balance.

  repeat
    (match goal with
     | ⊢ BST (match ?c with Red ⇒ _ | Black ⇒ _ end) ⇒ destruct c
     | ⊢ BST (match ?s with E ⇒ _ | T _ _ _ _ _ ⇒ _ end) ⇒ destruct s
     | ⊢ ForallT _ (T _ _ _ _ _) ⇒ repeat split
     | H: ForallT _ (T _ _ _ _ _) ⊢ _ ⇒ destruct H as [? [? ?] ]
     | H: BST (T _ _ _ _ _) ⊢ _ ⇒ inv H
     end;
     (try constructor; auto; try omega)).

  all: try eapply ForallT_greater; try eapply ForallT_less; eauto; try omega.
Qed.
```

Exercise: 2 stars, standard (balanceP) Prove that balance preserves ForallT P. Use proof automation with `match goal` and/or *all*:.

```
Lemma balanceP : ∀ (P : key → V → Prop) (c : color) (l r : tree) (k : key) (v : V),
    ForallT P l →
    ForallT P r →
    P k v →
    ForallT P (balance c l k v r).
Proof.
  Admitted.
```

□

Exercise: 2 stars, standard (insP) Prove that ins preserves ForallT P. Hint: proceed by induction on t. Use the previous lemma. There's no need for automated case analysis.

```
Lemma insP : ∀ (P : key → V → Prop) (t : tree) (k : key) (v : V),
```

```
    ForallT P t →
    P k v →
    ForallT P (ins k v t).
Proof.
 Admitted.
```

☐

Exercise: 3 stars, standard (ins_BST) Prove that ins maintains **BST**. Proceed by induction on the evidence that t is a BST. You don't need any automated case analysis.

```
Lemma ins_BST : ∀ (t : tree) (k : key) (v : V),
    BST t →
    BST (ins k v t).
Proof.
 Admitted.
```

☐

Exercise: 2 stars, standard (insert_BST) Prove the main theorem: insert preserves **BST**.

```
Theorem insert_BST : ∀ t v k,
    BST t →
    BST (insert k v t).
Proof.
 Admitted.
```

☐

12.5 Verification

We now verify that the equational specification of maps holds for red-black trees:

lookup k empty_tree = default lookup k (insert k v t) = v lookup k' (insert k v t) = lookup k' t if k <> k'

The first equation is trivial to verify.

```
Lemma lookup_empty : ∀ k, lookup k empty_tree = default.
Proof. auto. Qed.
```

The next two equations are more challenging because of balance.

Exercise: 4 stars, standard (balance_lookup) Prove that balance preserves the result of lookup on non-empty trees. Hint: automate the case analysis similarly to balance_BST.

```
Lemma balance_lookup: ∀ (c : color) (k k' : key) (v : V) (l r : tree),
    BST l →
```

```
    BST r →
    ForallT (fun k' _ ⇒ Abs k' < Abs k) l →
    ForallT (fun k' _ ⇒ Abs k' > Abs k) r →
    lookup k' (balance c l k v r) =
    if Abs k' <? Abs k
    then lookup k' l
    else if Abs k' >? Abs k
         then lookup k' r
         else v.
Proof.
  Admitted.
```

☐

Exercise: 3 stars, standard (lookup_ins_eq) Verify the second equation, though for ins rather than insert. Proceed by induction on the evidence that t is a **BST**. Note that precondition **BST** t will be essential in your proof, unlike the ordinary BST's we saw in SearchTree.

Hint: no automation of case analysis is needed; rely on the lemmas we've already proved above about balance and ins.

```
Lemma lookup_ins_eq: ∀ (t : tree) (k : key) (v : V),
    BST t →
    lookup k (ins k v t) = v.
Proof.
  Admitted.
```

☐

Exercise: 3 stars, standard (lookup_ins_neq) Verify the third equation, again for ins instead of insert. The same hints as for the second equation hold.

```
Theorem lookup_ins_neq: ∀ (t : tree) (k k' : key) (v : V),
    BST t →
    k ≠ k' →
    lookup k' (ins k v t) = lookup k' t.
Proof.
  Admitted.
```

☐

Finally, finish verify the second and third equations. The proofs are almost identical.

Exercise: 3 stars, standard (lookup_insert) Theorem lookup_insert_eq : ∀ (t : tree) (k : key) (v : V),

```
    BST t →
```

```
    lookup k (insert k v t) = v.
Proof.
 Admitted.
Theorem lookup_insert_neq: ∀ (t : tree) (k k' : key) (v : V),
    BST t →
    k ≠ k' →
    lookup k' (insert k v t) = lookup k' t.
Proof.
 Admitted.
```

□

That concludes the verification of the map equations for red-black trees. We have proved these main theorems:

```
Check empty_tree_BST : BST empty_tree.
Check insert_BST :
  ∀ (t : tree) (v : V) (k : key),
    BST t → BST (insert k v t).
Check lookup_empty :
  ∀ k : key,
    lookup k empty_tree = default.
Check lookup_insert_eq :
  ∀ (t : tree) (k : key) (v : V),
    BST t → lookup k (insert k v t) = v.
Check lookup_insert_neq :
  ∀ (t : tree) (k k' : key) (v : V),
    BST t →
    k ≠ k' →
    lookup k' (insert k v t) = lookup k' t.
```

We could now proceed to reprove all the facts about elements that we developed in SearchTree. But since elements does not not pay attention to colors, and does not rebalance the tree, these proofs should be a simple copy-paste from that chapter, with only minor edits. This would be an uninteresting exercise, so we don't pursue it here.

12.6 Efficiency

Red-black trees are more efficient than ordinary search trees, because red-black trees stay balanced. The insert operation ensures that these *red-black invariants* hold:

- Local Invariant: No red node has a red child.
    - Global Invariant: Every path from the root to a leaf has the same number of black nodes.

Together these invariants guarantee that no leaf is more than twice as deep as another leaf, a property that we will here call *approximately balanced*. The maximum depth of a node is therefore 2 *log* N, so the running-time of insert and lookup is O(*log* N), where N is the number of nodes in the tree.

Coq does not have a formal time–cost model for its execution, so we cannot verify that logarithmic running time in Coq. But we can prove that the trees are approximately balanced.

These ensure that the tree remains approximately balanced.

Relation **RB**, below, formalizes the red-black invariants. Proposition **RB** t c n holds when t satisfies the red-black invariants, assuming that c is the color of t's parent, and n is the black height that t is supposed to have.

If t happens to have no parent (i.e., it is the entire tree), then it will be colored black by insert, so it won't actually matter what color its (non-existent) parent might purportedly have: whether red or black, it can't violate the local invariant.

If t is a leaf, then it likewise won't matter what its parent color is, and its black height must be zero.

```
Inductive RB : tree → color → nat → Prop :=
| RB_leaf: ∀ (c : color), RB E c 0
| RB_r: ∀ (l r : tree) (k : key) (v : V) (n : nat),
    RB l Red n →
    RB r Red n →
    RB (T Red l k v r) Black n
| RB_b: ∀ (c : color) (l r : tree) (k : key) (v : V) (n : nat),
    RB l Black n →
    RB r Black n →
    RB (T Black l k v r) c (S n).
```

Exercise: 2 stars, standard (RB_blacken_parent) Prove that blackening a parent would preserve the red-black invariants.

```
Lemma RB_blacken_parent : ∀ (t : tree) (n : nat),
    RB t Red n → RB t Black n.
Proof.
  Admitted.
```

☐

Exercise: 2 stars, standard (RB_blacken_root) Prove that blackening a subtree root (whose hypothetical parent is black) would preserve the red-black invariants, though the black height of the subtree might change (and the color of the parent would need to become red).

```
Lemma RB_blacken_root : ∀ (t : tree) (n : nat),
    RB t Black n →
```

```
    ∃ (n' : nat) , RB (make_black t) Red n'.
Proof.
 Admitted.
```

☐

Relation **NearlyRB** expresses, "the tree is a red-black tree, except that it's nonempty and it is permitted to have two consecutive red nodes at the root only."

```
Inductive NearlyRB : tree → nat → Prop :=
| NearlyRB_r : ∀ (l r : tree) (k : key) (v : V) (n : nat),
    RB l Black n →
    RB r Black n →
    NearlyRB (T Red l k v r) n
| NearlyRB_b : ∀ (l r : tree) (k : key) (v : V) (n : nat),
    RB l Black n →
    RB r Black n →
    NearlyRB (T Black l k v r) (S n).
```

Exercise: 5 stars, standard (ins_RB) Prove that ins creates a tree that is either red-black or nearly so, depending on what the parent's color was. You will need significant case-analysis automation in a similar style to the proofs of ins_not_E and balance_lookup.

```
Lemma ins_RB : ∀ (k : key) (v : V) (t : tree) (n : nat),
    (RB t Black n → NearlyRB (ins k v t) n) ∧
    (RB t Red n → RB (ins k v t) Black n).
Proof.
  induction t; intro n; simpl; split; intros; inv H; repeat constructor; auto.
  × destruct (IHt1 n); clear IHt1.
    destruct (IHt2 n); clear IHt2.
    specialize (H0 H6).
    specialize (H2 H7).
    clear H H1.
    unfold balance.
 Admitted.
```

☐

Therefore, ins produces a red-black tree when given one as input – though the parent color changes. `Corollary ins_red : ∀ (t : tree) (k : key) (v : V) (n : nat),`

```
    (RB t Red n → RB (ins k v t) Black n).
Proof.
  intros. apply ins_RB. assumption.
Qed.
```

Exercise: 2 stars, standard (insert_RB) Prove that insert produces a red-black tree when given one as input. This can be done entirely with lemmas already proved.

```
Lemma insert_RB : ∀ (t : tree) (k : key) (v : V) (n : nat),
    RB t Red n →
    ∃ (n' : nat) , RB (insert k v t) Red n'.
Proof.
 Admitted.
```

□

Exercise: 4 stars, advanced (redblack_bound) To confirm that red-black trees are approximately balanced, define functions to compute the height (i.e., maximum depth) and minimum depth of a red-black tree, and prove that the height is bounded by twice the minimum depth, possibly plus 1. Hints:

- Prove two auxiliary lemmas, one about height and the other about mindepth, and then combine them to get the result. The lemma about height will need a slightly complicated induction hypothesis for the proof to go through.
- Depending on how you defined height and mindepth, the tactic *zify* (defined in the standard library *Coq.omega.PreOmega*) may be useful as a preliminary to using omega when proving these lemmas.

```
Fixpoint height (t : tree) : nat
. Admitted.

Fixpoint mindepth (t : tree) : nat
. Admitted.

Lemma redblack_balanced : ∀ t c n,
    RB t c n →
    (height t ≤ 2 × mindepth t + 1)%nat.
Proof.
 Admitted.
Definition manual_grade_for_redblack_bound : option (nat×string) := None.
```

□

```
End ValueType.
```

12.7 Performance of Extracted Code

We can extract the red-black tree implementation:

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

Run it in the OCaml top level with these commands:
use "test_searchtree.ml";;

On a recent machine with a 2.9 GHz Intel Core i9 that prints:

Insert and lookup 1000000 random integers in 0.860663 seconds. Insert and lookup 20000 random integers in 0.007908 seconds. Insert and lookup 20000 consecutive integers in 0.004668 seconds.

That execution uses the bytecode interpreter. The native compiler will have better performance:

```
$ ocamlopt -c redblack.mli redblack.ml
$ ocamlopt redblack.cmx -open Redblack test_searchtree.ml -o test_redblack
$ ./test_redblack
```

On the same machine that prints,

Insert and lookup 1000000 random integers in 0.475669 seconds. Insert and lookup 20000 random integers in 0.00312 seconds. Insert and lookup 20000 consecutive integers in 0.001183 seconds.

The benchmark measurements above (and in `Extract`) demonstrate the following:

- On random insertions, red-black trees are about the same as ordinary BSTs.
- On consecutive insertions, red-black trees are *much* faster than ordinary BSTs.
- Red-black trees are about as fast on consecutive insertions as on random.

Chapter 13

Library VFA.Trie

13.1 Trie: Number Representations and Efficient Lookup Tables

13.2 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.

13.3 A Simple Program That's Waaaaay Too Slow.

```
From Coq Require Import 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 eqb.
```

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

`End` VERYSLOW.

13.4 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, standard (succ_correct)

```
Lemma succ_correct: ∀ p,
   positive2nat (succ p) = S (positive2nat p).
Proof.
   Admitted.
```

□

Exercise: 3 stars, standard (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, standard (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.

13.4.1 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.
```

13.4.2 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.
```

13.4.3 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 $N*(logN)\hat{}2$.

13.5 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.
```

13.5.1 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, standard (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** (**nat**×**string**) := None.
□

13.6 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.

13.6.1 Lemmas About the Relation Between lookup and insert

Exercise: 1 star, standard (look_leaf) Lemma look_leaf:
$\forall$ A $(a{:}A)$ j, look a j Leaf = a.
Admitted.
□

Exercise: 2 stars, standard (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, standard (look_ins_same) Induction on j? Induction on t? Do you feel lucky?

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

13.6.2 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, standard (pos2nat_bijective) Lemma pos2nat_injective: ∀ p q, pos2nat p = pos2nat q → p=q.
Admitted.

Lemma nat2pos_injective: ∀ i j, nat2pos i = nat2pos j → i=j.
Admitted.
☐

13.6.3 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, standard (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, standard (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, standard (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, standard (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.
  □
```

13.6.4 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.
```

13.7 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 14

Library VFA.Priqueue

14.1 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

14.2 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.

14.3 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!

14.3.1 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, standard (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.
```

□

14.3.2 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.
```

14.4 Predicates on Priority Queues

14.4.1 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'.
```

14.4.2 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.
```

14.4.3 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, standard (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 15

Library VFA.Binom

15.1 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.)

15.2 Required Reading

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

Binomial Queues https://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.

15.3 The Program

```
From Coq Require Import 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 $2\hat{}i$ 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.
```

15.4 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.
```

15.5 Proof of Algorithm Correctness

15.5.1 Various Functions Preserve the Representation Invariant

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

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

Admitted.

□

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

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

Admitted.

□

Exercise: 3 stars, standard (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, standard, optional (insert_valid) Theorem insert_priq: ∀ x q, priq q → priq (insert x q).
Admitted.
□

Exercise: 3 stars, standard, 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, standard, optional (delete_max_Some_priq) Theorem delete_max_Some_priq:
∀ p q k, priq p → delete_max p = Some(k,q) → priq q.
Admitted.
□

15.5.2 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, standard (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 (nat×string) := None.
```

□

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

15.5.3 Sanity Checks on the Abstraction Relation

Exercise: 2 stars, standard (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, standard (tree_perm) Theorem tree_perm: ∀ t $e1$ $e2$,
  tree_elems t $e1$ → tree_elems t $e2$ → Permutation $e1$ $e2$.
  Admitted.
  □

Exercise: 2 stars, standard (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, standard (abs_perm) Theorem abs_perm: ∀ p al bl,
  priq p → Abs p al → Abs p bl → Permutation al bl.
Proof.
  Admitted.
  □

Exercise: 2 stars, standard (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.
  □

15.5.4 Various Functions Preserve the Abstraction Relation

Exercise: 1 star, standard (empty_relate) `Theorem` empty_relate: Abs empty nil.
`Proof`.
  Admitted.
  □

Exercise: 3 stars, standard (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.
  □

15.5.5 Optional Exercises

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

Exercise: 4 stars, standard, optional (carry_elems) `Theorem` carry_elems:
  $\forall$ n q, priq' n q $\rightarrow$
  $\forall$ t, (t=Leaf $\vee$ 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, standard, 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, standard, optional (join_elems) `Theorem` join_elems:
  $\forall$ p q c n,
    priq' n p $\rightarrow$
    priq' n q $\rightarrow$
    (c=Leaf $\vee$ 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, standard, optional (merge_relate) `Theorem` merge_relate:

$\forall$ 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.

☐

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

$\forall$ p, priq p → (Abs p nil ↔ delete_max p = None).

Admitted.

☐

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

$\forall$ (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.

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.

15.6 Measurement.

Exercise: 5 stars, standard, 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 16

Library VFA.Decide

16.1 Decide: Programming with Decision Procedures

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

16.2 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 ltb_reflect theorem.

```
Print reflect.
```

```
Check ltb_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 ltb_reflect theorem. The result of `Check ltb_reflect`, above, says that for any x,y, there does exist a value (ltb_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 := ltb_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 := ltb_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` (ltb_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.

16.3 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 ltb_reflect.

```
Check ltb_reflect.
```

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

```
Definition lt_dec (a: nat) (b: nat) : {a<b}+{~(a<b)} :=
match ltb_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 (ltb_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.
```

16.3.1 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 ltb_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 leb_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, standard (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.

□

16.4 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 $\forall\ P,\ P \lor \neg P$. 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 $P \lor \neg P$ 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 ltb_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.
```

16.5 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 ltb_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 ltb_reflect, then we will have to rebuild ltb_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, standard (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!

16.6 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 17

Library VFA.Color

17.1 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.

17.2 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 Setoid. 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.
```

17.3 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.
```

17.3.1 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.
```

17.3.2 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.
```

17.3.3 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, standard (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.
```

□

17.3.4 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, standard (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, standard (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:
  ∀ A (al: list (positive×A)),
    Sorted (@M.lt_key A) al ↔ Sorted E.lt (map (@fst positive A) al).
Proof.
```

Admitted.

☐

17.3.5 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, standard (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, standard (Sremove_cardinal_less) Lemma Sremove_cardinal_less: $\forall$ i s,

```
      S.In i s → 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, standard (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, standard (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, standard (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, standard (Sin_domain) Lemma Sin_domain: ∀ A n (g: M.t A), S.In n (Mdomain g) ↔ M.In n g.

To reason about *M.fold*, used in the definition of Mdomain, a useful theorem is *WP.fold_rec_bis*.

Admitted.

☐

17.4 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) :=
   ∀ i j, S.In j (adj g i) → S.In i (adj g j).

Definition no_selfloop (g: graph) := ∀ i, ¬ S.In i (adj g i).

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

Definition subset_nodes
                    (P: node → nodeset → bool)
                    (g: graph) :=
   M.fold (fun n adj s ⇒ 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).
```

17.4.1 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, standard (subset_nodes_sub) `Lemma subset_nodes_sub: ∀ P g, S.Subset (subset_nodes P g) (nodes g).`

Admitted.

☐

Exercise: 3 stars, standard (select_terminates) `Lemma select_terminates:`

```
  ∀ (K: nat) (g : graph) (n : S.elt),
   S.choose (subset_nodes (low_deg K) g) = Some n →
   M.cardinal (remove_node n g) < M.cardinal g.
```

Admitted.

☐

17.4.2 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).
```

17.5 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, standard (adj_ext) Lemma adj_ext: ∀ g i j, E.eq i j → S.eq (adj g i) (adj g j).

Admitted.

□

Exercise: 3 stars, standard (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, standard (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.

17.6 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 18

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 (_ ?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 "------------------- eqb_idP --------------------".
```

```
idtac " ".
idtac "#> eqb_idP".
idtac "Possible points: 2".
check_type @eqb_idP ((∀ x y : nat, Bool.reflect (x = y) (PeanoNat.Nat.eqb x y))).
idtac "Assumptions:".
Abort.
Print Assumptions eqb_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".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "functional_extensionality_dep".
```

```
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "int".
idtac "Abs".
idtac "Abs_inj".
idtac "ltb".
idtac "ltb_lt".
idtac "leb".
idtac "leb_le".
idtac "Extract.int".
idtac "Extract.Abs".
idtac "Extract.Abs_inj".
idtac "Extract.ltb".
idtac "Extract.ltb_lt".
idtac "Extract.leb".
idtac "Extract.leb_le".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "---------- eqb_idP ---------".
Print Assumptions eqb_idP.
idtac "---------- t_update_same ---------".
Print Assumptions t_update_same.
idtac "---------- t_update_permute ---------".
Print Assumptions t_update_permute.
idtac "".
idtac "********** Advanced **********".
Abort.
```

Chapter 19

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 (_ ?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".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "functional_extensionality_dep".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "int".
idtac "Abs".
idtac "Abs_inj".
idtac "ltb".
idtac "ltb_lt".
idtac "leb".
idtac "leb_le".
idtac "Extract.int".
idtac "Extract.Abs".
idtac "Extract.Abs_inj".
idtac "Extract.ltb".
idtac "Extract.ltb_lt".
idtac "Extract.leb".
idtac "Extract.leb_le".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "".
idtac "********** Advanced **********".
Abort.
```

Chapter 20

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 (_ ?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: Permutation_properties".
idtac "Possible points: 2".
print_manual_grade manual_grade_for_Permutation_properties.
idtac " ".
idtac "-------------------  permut_example ---------------------".
idtac " ".
idtac "#> permut_example".
idtac "Possible points: 3".
check_type @permut_example (
(∀ a b : list nat,
 @Permutation nat (5 :: 6 :: a ++ b) ((5 :: b) ++ 6 :: a ++ []))).
idtac "Assumptions:".
Abort.
Print Assumptions permut_example.
Goal True.
idtac " ".
idtac "-------------------  not_a_permutation ---------------------".
idtac " ".
idtac "#> not_a_permutation".
idtac "Possible points: 2".
check_type @not_a_permutation ((¬ @Permutation nat [1; 1] [1; 2])).
idtac "Assumptions:".
Abort.
Print Assumptions not_a_permutation.
Goal True.
idtac " ".
idtac "-------------------  Forall_perm ---------------------".
idtac " ".
idtac "#> Forall_perm".
idtac "Possible points: 3".
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: 10".
idtac "Max points - advanced: 10".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "functional_extensionality_dep".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "int".
idtac "Abs".
idtac "Abs_inj".
idtac "ltb".
idtac "ltb_lt".
idtac "leb".
idtac "leb_le".
idtac "Extract.int".
idtac "Extract.Abs".
idtac "Extract.Abs_inj".
idtac "Extract.ltb".
idtac "Extract.ltb_lt".
idtac "Extract.leb".
idtac "Extract.leb_le".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "---------- Permutation_properties ---------".
idtac "MANUAL".
idtac "---------- permut_example ---------".
Print Assumptions permut_example.
idtac "---------- not_a_permutation ---------".
Print Assumptions not_a_permutation.
idtac "---------- Forall_perm ---------".
Print Assumptions Forall_perm.
```

```
idtac "".
idtac "********** Advanced **********".
Abort.
```

Chapter 21

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 (_ ?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_sorted ——————".
```

```
idtac " ".
idtac "#> insert_sorted".
idtac "Possible points: 3".
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 "-------------------  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 "-------------------- insertion_sort_correct --------------------".
idtac " ".
idtac "#> insertion_sort_correct".
idtac "Possible points: 1".
check_type @insertion_sort_correct ((is_a_sorting_algorithm sort)).
idtac "Assumptions:".
Abort.
Print Assumptions insertion_sort_correct.
Goal True.
idtac " ".
idtac "-------------------- sorted_sorted' --------------------".
idtac " ".
idtac "#> sorted_sorted'".
idtac "Advanced".
idtac "Possible points: 6".
check_type @sorted_sorted' ((∀ al : list nat, sorted al → sorted' al)).
idtac "Assumptions:".
Abort.
Print Assumptions sorted_sorted'.
Goal True.
idtac " ".
idtac "-------------------- sorted'_sorted --------------------".
idtac " ".
idtac "#> sorted'_sorted".
idtac "Advanced".
idtac "Possible points: 3".
check_type @sorted'_sorted ((∀ al : list nat, sorted' al → sorted al)).
idtac "Assumptions:".
Abort.
Print Assumptions sorted'_sorted.
Goal True.
idtac " ".
idtac " ".
idtac "Max points - standard: 12".
idtac "Max points - advanced: 21".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "functional_extensionality_dep".
```

```
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "int".
idtac "Abs".
idtac "Abs_inj".
idtac "ltb".
idtac "ltb_lt".
idtac "leb".
idtac "leb_le".
idtac "Extract.int".
idtac "Extract.Abs".
idtac "Extract.Abs_inj".
idtac "Extract.ltb".
idtac "Extract.ltb_lt".
idtac "Extract.leb".
idtac "Extract.leb_le".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "---------- insert_sorted ---------".
Print Assumptions insert_sorted.
idtac "---------- sort_sorted ---------".
Print Assumptions sort_sorted.
idtac "---------- insert_perm ---------".
Print Assumptions insert_perm.
idtac "---------- sort_perm ---------".
Print Assumptions sort_perm.
idtac "---------- insertion_sort_correct ---------".
Print Assumptions insertion_sort_correct.
idtac "".
idtac "********** Advanced **********".
idtac "---------- sorted_sorted' ---------".
Print Assumptions sorted_sorted'.
```

```
idtac "------- sorted'_sorted ------".
Print Assumptions sorted'_sorted.
Abort.
```

Chapter 22

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 (_ ?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 "-------------------  union_swap --------------------".
idtac " ".
idtac "#> union_swap".
idtac "Possible points: 2".
check_type @union_swap (
(∀ a b c : multiset, union a (union b c) = union b (union a c))).
idtac "Assumptions:".
Abort.
Print Assumptions union_swap.
Goal True.
idtac " ".
idtac "-------------------  insert_contents --------------------".
idtac " ".
idtac "#> insert_contents".
idtac "Possible points: 3".
check_type @insert_contents (
(∀ (x : nat) (l : list nat),
 contents (Sort.insert x l) = contents (x :: l))).
idtac "Assumptions:".
Abort.
Print Assumptions insert_contents.
```

```
Goal True.
idtac " ".
idtac "—————————- sort_contents ——————————".
idtac " ".
idtac "#> sort_contents".
idtac "Possible points: 2".
check_type @sort_contents ((∀ l : list value, contents l = contents (Sort.sort l))).
idtac "Assumptions:".
Abort.
Print Assumptions sort_contents.
Goal True.
idtac " ".
idtac "—————————- insertion_sort_correct ——————————".
idtac " ".
idtac "#> insertion_sort_correct".
idtac "Possible points: 1".
check_type @insertion_sort_correct ((is_a_sorting_algorithm' Sort.sort)).
idtac "Assumptions:".
Abort.
Print Assumptions insertion_sort_correct.
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 "-------------------  contents_nil_inv  ---------------------".
idtac " ".

idtac "#> contents_nil_inv".
idtac "Advanced".
idtac "Possible points: 2".
check_type @contents_nil_inv (
(∀ l : list value,
 (∀ x : value, 0 = contents l x) → l = @nil value)).
idtac "Assumptions:".
Abort.
Print Assumptions contents_nil_inv.
Goal True.
idtac " ".

idtac "-------------------  contents_cons_inv  ---------------------".
idtac " ".

idtac "#> contents_cons_inv".
idtac "Advanced".
idtac "Possible points: 3".
check_type @contents_cons_inv (
(∀ (l : list value) (x : value) (n : nat),
 S n = contents l x →
 ∃ l1 l2 : list value,
   l = (l1 ++ x :: l2)%list ∧ contents (l1 ++ l2) x = n)).
idtac "Assumptions:".
Abort.
Print Assumptions contents_cons_inv.
Goal True.
idtac " ".

idtac "-------------------  contents_insert_other  ---------------------".
idtac " ".

idtac "#> contents_insert_other".
idtac "Advanced".
idtac "Possible points: 2".
check_type @contents_insert_other (
(∀ (l1 l2 : list value) (x y : value),
 y ≠ x → contents (l1 ++ x :: l2) y = contents (l1 ++ l2) y)).
idtac "Assumptions:".
Abort.
Print Assumptions contents_insert_other.
Goal True.
idtac " ".
```

```
idtac "-------------- contents_perm --------------".
idtac " ".
idtac "#> contents_perm".
idtac "Advanced".
idtac "Possible points: 3".
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 "-------------- same_contents_iff_perm --------------".
idtac " ".
idtac "#> same_contents_iff_perm".
idtac "Possible points: 1".
check_type @same_contents_iff_perm (
(∀ al bl : list value,
 contents al = contents bl ↔ @Permutation.Permutation value al bl)).
idtac "Assumptions:".
Abort.
Print Assumptions same_contents_iff_perm.
Goal True.
idtac " ".
idtac "-------------- sort_specifications_equivalent --------------".
idtac " ".
idtac "#> sort_specifications_equivalent".
idtac "Possible points: 2".
check_type @sort_specifications_equivalent (
(∀ sort : list nat → list nat,
 Sort.is_a_sorting_algorithm sort ↔ is_a_sorting_algorithm' sort)).
idtac "Assumptions:".
Abort.
Print Assumptions sort_specifications_equivalent.
Goal True.
idtac " ".
idtac " ".
idtac "Max points - standard: 17".
idtac "Max points - advanced: 27".
idtac "".
```

```
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "functional_extensionality_dep".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "int".
idtac "Abs".
idtac "Abs_inj".
idtac "ltb".
idtac "ltb_lt".
idtac "leb".
idtac "leb_le".
idtac "Extract.int".
idtac "Extract.Abs".
idtac "Extract.Abs_inj".
idtac "Extract.ltb".
idtac "Extract.ltb_lt".
idtac "Extract.leb".
idtac "Extract.leb_le".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "---------- union_assoc ---------".
Print Assumptions union_assoc.
idtac "---------- union_comm ---------".
Print Assumptions union_comm.
idtac "---------- union_swap ---------".
Print Assumptions union_swap.
idtac "---------- insert_contents ---------".
Print Assumptions insert_contents.
idtac "---------- sort_contents ---------".
Print Assumptions sort_contents.
idtac "---------- insertion_sort_correct ---------".
```

```
Print Assumptions insertion_sort_correct.
idtac "-------- permutations_vs_multiset --------".
idtac "MANUAL".
idtac "-------- perm_contents --------".
Print Assumptions perm_contents.
idtac "-------- same_contents_iff_perm --------".
Print Assumptions same_contents_iff_perm.
idtac "-------- sort_specifications_equivalent --------".
Print Assumptions sort_specifications_equivalent.
idtac "".
idtac "********** Advanced **********".
idtac "-------- contents_nil_inv --------".
Print Assumptions contents_nil_inv.
idtac "-------- contents_cons_inv --------".
Print Assumptions contents_cons_inv.
idtac "-------- contents_insert_other --------".
Print Assumptions contents_insert_other.
idtac "-------- contents_perm --------".
Print Assumptions contents_perm.
Abort.
```

Chapter 23

Library VFA.BagPermTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From VFA Require Import BagPerm.

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 (_ ?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 BagPerm.
Import Check.

Goal True.

idtac "------------------- bag_eqv_properties --------------------".
```

```
idtac " ".
idtac "#> bag_eqv_refl".
idtac "Possible points: 0.5".
check_type @bag_eqv_refl ((∀ b : bag, bag_eqv b b)).
idtac "Assumptions:".
Abort.
Print Assumptions bag_eqv_refl.
Goal True.
idtac " ".
idtac "#> bag_eqv_sym".
idtac "Possible points: 0.5".
check_type @bag_eqv_sym ((∀ b1 b2 : bag, bag_eqv b1 b2 → bag_eqv b2 b1)).
idtac "Assumptions:".
Abort.
Print Assumptions bag_eqv_sym.
Goal True.
idtac " ".
idtac "#> bag_eqv_trans".
idtac "Possible points: 0.5".
check_type @bag_eqv_trans (
(∀ b1 b2 b3 : bag, bag_eqv b1 b2 → bag_eqv b2 b3 → bag_eqv b1 b3)).
idtac "Assumptions:".
Abort.
Print Assumptions bag_eqv_trans.
Goal True.
idtac " ".
idtac "#> bag_eqv_cons".
idtac "Possible points: 0.5".
check_type @bag_eqv_cons (
(∀ (x : nat) (b1 b2 : bag),
 bag_eqv b1 b2 → bag_eqv (x :: b1)%list (x :: b2)%list)).
idtac "Assumptions:".
Abort.
Print Assumptions bag_eqv_cons.
Goal True.
idtac " ".
idtac "-------------------- insert_bag --------------------".
idtac " ".
idtac "#> insert_bag".
idtac "Possible points: 3".
check_type @insert_bag (
```

```
(∀ (x : nat) (l : list nat), bag_eqv (x :: l)%list (Sort.insert x l))).
idtac "Assumptions:".
Abort.
Print Assumptions insert_bag.
Goal True.
idtac " ".

idtac "-------------------  sort_bag  --------------------".
idtac " ".

idtac "#> sort_bag".
idtac "Possible points: 2".
check_type @sort_bag ((∀ l : bag, bag_eqv l (Sort.sort l))).
idtac "Assumptions:".
Abort.
Print Assumptions sort_bag.
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_bag  --------------------".
idtac " ".

idtac "#> perm_bag".
idtac "Possible points: 3".
check_type @perm_bag (
(∀ al bl : list nat, @Permutation.Permutation nat al bl → bag_eqv al bl)).
idtac "Assumptions:".
Abort.
Print Assumptions perm_bag.
Goal True.
idtac " ".

idtac "-------------------  bag_nil_inv  --------------------".
idtac " ".

idtac "#> bag_nil_inv".
idtac "Advanced".
idtac "Possible points: 2".
check_type @bag_nil_inv ((∀ b : bag, bag_eqv (@nil nat) b → b = @nil nat)).
idtac "Assumptions:".
```

```
Abort.
Print Assumptions bag_nil_inv.
Goal True.
idtac " ".

idtac "-------------------  bag_cons_inv --------------------".
idtac " ".

idtac "#> bag_cons_inv".
idtac "Advanced".
idtac "Possible points: 3".
check_type @bag_cons_inv (
(∀ (l : bag) (x n : nat),
 S n = count x l →
 ∃ l1 l2 : list nat,
   l = (l1 ++ x :: l2)%list ∧ count x (l1 ++ l2)%list = n)).
idtac "Assumptions:".
Abort.
Print Assumptions bag_cons_inv.
Goal True.
idtac " ".

idtac "-------------------  count_insert_other --------------------".
idtac " ".

idtac "#> count_insert_other".
idtac "Advanced".
idtac "Possible points: 2".
check_type @count_insert_other (
(∀ (l1 l2 : list nat) (x y : nat),
 y ≠ x → count y (l1 ++ x :: l2)%list = count y (l1 ++ l2)%list)).
idtac "Assumptions:".
Abort.
Print Assumptions count_insert_other.
Goal True.
idtac " ".

idtac "-------------------  bag_perm --------------------".
idtac " ".

idtac "#> bag_perm".
idtac "Advanced".
idtac "Possible points: 3".
check_type @bag_perm (
(∀ al bl : bag, bag_eqv al bl → @Permutation.Permutation nat al bl)).
idtac "Assumptions:".
Abort.
```

```
Print Assumptions bag_perm.
Goal True.
idtac " ".
idtac " ".
idtac "Max points - standard: 11".
idtac "Max points - advanced: 21".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "functional_extensionality_dep".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "int".
idtac "Abs".
idtac "Abs_inj".
idtac "ltb".
idtac "ltb_lt".
idtac "leb".
idtac "leb_le".
idtac "Extract.int".
idtac "Extract.Abs".
idtac "Extract.Abs_inj".
idtac "Extract.ltb".
idtac "Extract.ltb_lt".
idtac "Extract.leb".
idtac "Extract.leb_le".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "---------- bag_eqv_refl ---------".
Print Assumptions bag_eqv_refl.
idtac "---------- bag_eqv_sym ---------".
Print Assumptions bag_eqv_sym.
```

```
idtac "——- bag_eqv_trans ——".
Print Assumptions bag_eqv_trans.
idtac "——- bag_eqv_cons ——".
Print Assumptions bag_eqv_cons.
idtac "——- insert_bag ——".
Print Assumptions insert_bag.
idtac "——- sort_bag ——".
Print Assumptions sort_bag.
idtac "——- permutations_vs_multiset ——".
idtac "MANUAL".
idtac "——- perm_bag ——".
Print Assumptions perm_bag.
idtac "".
idtac "********** Advanced **********".
idtac "——- bag_nil_inv ——".
Print Assumptions bag_nil_inv.
idtac "——- bag_cons_inv ——".
Print Assumptions bag_cons_inv.
idtac "——- count_insert_other ——".
Print Assumptions count_insert_other.
idtac "——- bag_perm ——".
Print Assumptions bag_perm.
Abort.
```

Chapter 24

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 (_ ?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) (y : nat) (r : list nat),
 (y , r) = select x l → @Permutation.Permutation nat (x :: l) (y :: r))).
idtac "Assumptions:".
Abort.
Print Assumptions select_perm.
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 n))).
idtac "Assumptions:".
Abort.
Print Assumptions selsort_perm.
Goal True.
idtac " ".
idtac "———————- selection_sort_perm ———————".
idtac " ".
idtac "#> selection_sort_perm".
idtac "Possible points: 1".
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_rest_length ———————".
idtac " ".
idtac "#> select_rest_length".
idtac "Possible points: 2".
check_type @select_rest_length (
(∀ (x : nat) (l : list nat) (y : nat) (r : list nat),
 select x l = (y , r) → @length nat l = @length nat r)).
```

```
idtac "Assumptions:".
Abort.
Print Assumptions select_rest_length.
Goal True.
idtac " ".

idtac "-------------------  select_fst_leq ------------------".
idtac " ".

idtac "#> select_fst_leq".
idtac "Possible points: 3".
check_type @select_fst_leq (
(∀ (al bl : list nat) (x y : nat), select x al = (y, bl) → y ≤ x)).
idtac "Assumptions:".
Abort.
Print Assumptions select_fst_leq.
Goal True.
idtac " ".

idtac "-------------------  select_smallest ------------------".
idtac " ".

idtac "#> select_smallest".
idtac "Possible points: 3".
check_type @select_smallest (
(∀ (al bl : list nat) (x y : nat), select x al = (y, bl) → y <=* bl)).
idtac "Assumptions:".
Abort.
Print Assumptions select_smallest.
Goal True.
idtac " ".

idtac "-------------------  select_in ------------------".
idtac " ".

idtac "#> select_in".
idtac "Possible points: 3".
check_type @select_in (
(∀ (al bl : list nat) (x y : nat),
 select x al = (y, bl) → @In nat y (x :: al))).
idtac "Assumptions:".
Abort.
Print Assumptions select_in.
Goal True.
idtac " ".

idtac "-------------------  cons_of_small_maintains_sort ------------------".
idtac " ".
```

```
idtac "#> cons_of_small_maintains_sort".
idtac "Possible points: 3".
check_type @cons_of_small_maintains_sort (
(∀ (bl : list nat) (y n : nat),
 n = @length nat bl →
 y <=* bl → sorted (selsort bl n) → sorted (y :: selsort bl n))).
idtac "Assumptions:".
Abort.
Print Assumptions cons_of_small_maintains_sort.
Goal True.
idtac " ".

idtac "—————————- selsort_sorted —————————".
idtac " ".

idtac "#> selsort_sorted".
idtac "Possible points: 3".
check_type @selsort_sorted (
(∀ (n : nat) (al : list nat),
 @length nat al = n → sorted (selsort al n))).
idtac "Assumptions:".
Abort.
Print Assumptions selsort_sorted.
Goal True.
idtac " ".

idtac "—————————- selection_sort_sorted —————————".
idtac " ".

idtac "#> selection_sort_sorted".
idtac "Possible points: 1".
check_type @selection_sort_sorted ((∀ al : list nat, sorted (selection_sort al))).
idtac "Assumptions:".
Abort.
Print Assumptions selection_sort_sorted.
Goal True.
idtac " ".

idtac "—————————- selection_sort_is_correct —————————".
idtac " ".

idtac "#> selection_sort_is_correct".
idtac "Possible points: 1".
check_type @selection_sort_is_correct ((is_a_sorting_algorithm selection_sort)).
idtac "Assumptions:".
Abort.
Print Assumptions selection_sort_is_correct.
```

```
Goal True.
idtac " ".
idtac "—————————- selsort'_perm ——————————".
idtac " ".
idtac "#> selsort'_perm".
idtac "Possible points: 2".
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: 28".
idtac "Max points - advanced: 28".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "functional_extensionality_dep".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "int".
idtac "Abs".
idtac "Abs_inj".
idtac "ltb".
idtac "ltb_lt".
idtac "leb".
idtac "leb_le".
idtac "Extract.int".
idtac "Extract.Abs".
idtac "Extract.Abs_inj".
idtac "Extract.ltb".
idtac "Extract.ltb_lt".
idtac "Extract.leb".
idtac "Extract.leb_le".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
```

```
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "------ select_perm ------".
Print Assumptions select_perm.
idtac "------ selsort_perm ------".
Print Assumptions selsort_perm.
idtac "------ selection_sort_perm ------".
Print Assumptions selection_sort_perm.
idtac "------ select_rest_length ------".
Print Assumptions select_rest_length.
idtac "------ select_fst_leq ------".
Print Assumptions select_fst_leq.
idtac "------ select_smallest ------".
Print Assumptions select_smallest.
idtac "------ select_in ------".
Print Assumptions select_in.
idtac "------ cons_of_small_maintains_sort ------".
Print Assumptions cons_of_small_maintains_sort.
idtac "------ selsort_sorted ------".
Print Assumptions selsort_sorted.
idtac "------ selection_sort_sorted ------".
Print Assumptions selection_sort_sorted.
idtac "------ selection_sort_is_correct ------".
Print Assumptions selection_sort_is_correct.
idtac "------ selsort'_perm ------".
Print Assumptions selsort'_perm.
idtac "".
idtac "********** Advanced **********".
Abort.
```

Chapter 25

Library VFA.MergeTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From VFA Require Import Merge.

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 (_ ?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 Merge.
Import Check.

Goal True.

idtac "——————- split_perm ——————–".
```

```
idtac " ".
idtac "#> split_perm".
idtac "Possible points: 3".
check_type @split_perm (
(∀ (X : Type) (l l1 l2 : list X),
 @split X l = (l1 , l2) → @Permutation.Permutation X l (l1 ++ l2))).
idtac "Assumptions:".
Abort.
Print Assumptions split_perm.
Goal True.
idtac " ".
idtac "———————- sorted_merge1 ———————-".
idtac " ".
idtac "#> sorted_merge1".
idtac "Possible points: 2".
check_type @sorted_merge1 (
(∀ (x x1 : nat) (l1 : list nat) (x2 : nat) (l2 : list nat),
 x ≤ x1 →
 x ≤ x2 →
 Sort.sorted (merge (x1 :: l1) (x2 :: l2)) →
 Sort.sorted (x :: merge (x1 :: l1) (x2 :: l2)))).
idtac "Assumptions:".
Abort.
Print Assumptions sorted_merge1.
Goal True.
idtac " ".
idtac "———————- sorted_merge ———————-".
idtac " ".
idtac "#> sorted_merge".
idtac "Possible points: 6".
check_type @sorted_merge (
(∀ l1 : list nat,
 Sort.sorted l1 →
 ∀ l2 : list nat, Sort.sorted l2 → Sort.sorted (merge l1 l2))).
idtac "Assumptions:".
Abort.
Print Assumptions sorted_merge.
Goal True.
idtac " ".
idtac "———————- mergesort_sorts ———————-".
idtac " ".
```

```
idtac "#> mergesort_sorts".
idtac "Possible points: 2".
check_type @mergesort_sorts ((∀ l : list nat, Sort.sorted (mergesort l))).
idtac "Assumptions:".
Abort.
Print Assumptions mergesort_sorts.
Goal True.
idtac " ".

idtac "—————————- merge_perm ——————————".
idtac " ".

idtac "#> merge_perm".
idtac "Advanced".
idtac "Possible points: 3".
check_type @merge_perm (
(∀ l1 l2 : list nat,
 @Permutation.Permutation nat (l1 ++ l2) (merge l1 l2))).
idtac "Assumptions:".
Abort.
Print Assumptions merge_perm.
Goal True.
idtac " ".

idtac "—————————- mergesort_perm ——————————".
idtac " ".

idtac "#> mergesort_perm".
idtac "Advanced".
idtac "Possible points: 3".
check_type @mergesort_perm (
(∀ l : list nat, @Permutation.Permutation nat l (mergesort l))).
idtac "Assumptions:".
Abort.
Print Assumptions mergesort_perm.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 13".
idtac "Max points - advanced: 19".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "functional_extensionality_dep".
idtac "FunctionalExtensionality.functional_extensionality_dep".
```

```
idtac "int".
idtac "Abs".
idtac "Abs_inj".
idtac "ltb".
idtac "ltb_lt".
idtac "leb".
idtac "leb_le".
idtac "Extract.int".
idtac "Extract.Abs".
idtac "Extract.Abs_inj".
idtac "Extract.ltb".
idtac "Extract.ltb_lt".
idtac "Extract.leb".
idtac "Extract.leb_le".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "---------- split_perm ---------".
Print Assumptions split_perm.
idtac "---------- sorted_merge1 ---------".
Print Assumptions sorted_merge1.
idtac "---------- sorted_merge ---------".
Print Assumptions sorted_merge.
idtac "---------- mergesort_sorts ---------".
Print Assumptions mergesort_sorts.
idtac "".
idtac "********** Advanced **********".
idtac "---------- merge_perm ---------".
Print Assumptions merge_perm.
idtac "---------- mergesort_perm ---------".
Print Assumptions mergesort_perm.
Abort.
```

Chapter 26

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 (_ ?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 "------------------- empty_tree_BST --------------------".
```

```
idtac " ".
idtac "#> empty_tree_BST".
idtac "Possible points: 1".
check_type @empty_tree_BST ((∀ V : Type, @BST V (@empty_tree V))).
idtac "Assumptions:".
Abort.
Print Assumptions empty_tree_BST.
Goal True.
idtac " ".
idtac "——————- insert_BST ——————".
idtac " ".
idtac "#> ForallT_insert".
idtac "Possible points: 3".
check_type @ForallT_insert (
(∀ (V : Type) (P : key → V → Prop) (t : tree V),
 @ForallT V P t →
 ∀ (k : key) (v : V), P k v → @ForallT V P (@insert V k v t))).
idtac "Assumptions:".
Abort.
Print Assumptions ForallT_insert.
Goal True.
idtac " ".
idtac "#> insert_BST".
idtac "Possible points: 3".
check_type @insert_BST (
(∀ (V : Type) (k : key) (v : V) (t : tree V),
 @BST V t → @BST V (@insert V k v t))).
idtac "Assumptions:".
Abort.
Print Assumptions insert_BST.
Goal True.
idtac " ".
idtac "——————- elements_complete ——————".
idtac " ".
idtac "#> elements_complete".
idtac "Possible points: 3".
check_type @elements_complete (elements_complete_spec).
idtac "Assumptions:".
Abort.
Print Assumptions elements_complete.
Goal True.
```

```
idtac " ".
idtac "-------------------- elements_preserves_forall ---------------------".
idtac " ".
idtac "#> elements_preserves_forall".
idtac "Possible points: 2".
check_type @elements_preserves_forall (
(∀ (V : Type) (P : key → V → Prop) (t : tree V),
 @ForallT V P t →
 @List.Forall (key × V) (@uncurry key V Prop P) (@elements V t))).
idtac "Assumptions:".
Abort.
Print Assumptions elements_preserves_forall.
Goal True.
idtac " ".
idtac "-------------------- elements_preserves_relation ---------------------".
idtac " ".
idtac "#> elements_preserves_relation".
idtac "Possible points: 2".
check_type @elements_preserves_relation (
(∀ (V : Type) (k k' : key) (v : V) (t : tree V) (R : key → key → Prop),
 @ForallT V (fun (y : key) (_ : V) ⇒ R y k') t →
 @List.In (key × V) (k, v) (@elements V t) → R k k')).
idtac "Assumptions:".
Abort.
Print Assumptions elements_preserves_relation.
Goal True.
idtac " ".
idtac "-------------------- elements_correct ---------------------".
idtac " ".
idtac "#> elements_correct".
idtac "Possible points: 6".
check_type @elements_correct (elements_correct_spec).
idtac "Assumptions:".
Abort.
Print Assumptions elements_correct.
Goal True.
idtac " ".
idtac "-------------------- elements_complete_inverse ---------------------".
idtac " ".
idtac "#> elements_complete_inverse".
```

```
idtac "Advanced".
idtac "Possible points: 2".
check_type @elements_complete_inverse (
(∀ (V : Type) (k : key) (v : V) (t : tree V),
 @BST V t →
 @bound V k t = false → ¬ @List.In (key × V) (k , v) (@elements V t))).
idtac "Assumptions:".
Abort.
Print Assumptions elements_complete_inverse.
Goal True.
idtac " ".

idtac "-------------------  elements_correct_inverse ---------------------".
idtac " ".

idtac "#> bound_value".
idtac "Advanced".
idtac "Possible points: 3".
check_type @bound_value (
(∀ (V : Type) (k : key) (t : tree V),
 @bound V k t = true → ∃ v : V , ∀ d : V, @lookup V d k t = v)).
idtac "Assumptions:".
Abort.
Print Assumptions bound_value.
Goal True.
idtac " ".

idtac "#> elements_complete_inverse".
idtac "Advanced".
idtac "Possible points: 3".
check_type @elements_complete_inverse (
(∀ (V : Type) (k : key) (v : V) (t : tree V),
 @BST V t →
 @bound V k t = false → ¬ @List.In (key × V) (k , v) (@elements V t))).
idtac "Assumptions:".
Abort.
Print Assumptions elements_complete_inverse.
Goal True.
idtac " ".

idtac "-------------------  sorted_app ---------------------".
idtac " ".

idtac "#> sorted_app".
idtac "Advanced".
idtac "Possible points: 3".
```

```
check_type @sorted_app (
(∀ (l1 l2 : list nat) (x : nat),
 Sort.sorted l1 →
 Sort.sorted l2 →
 @List.Forall nat (fun n : nat ⇒ n < x) l1 →
 @List.Forall nat (fun n : nat ⇒ n > x) l2 → Sort.sorted (l1 ++ x :: l2))).
idtac "Assumptions:".
Abort.
Print Assumptions sorted_app.
Goal True.
idtac " ".

idtac "-------------------  sorted_elements  --------------------".
idtac " ".

idtac "#> sorted_elements".
idtac "Advanced".
idtac "Possible points: 6".
check_type @sorted_elements (
(∀ (V : Type) (t : tree V),
 @BST V t → Sort.sorted (@list_keys V (@elements V t)))).
idtac "Assumptions:".
Abort.
Print Assumptions sorted_elements.
Goal True.
idtac " ".

idtac "-------------------  fast_elements_eq_elements  --------------------".
idtac " ".

idtac "#> fast_elements_tr_helper".
idtac "Possible points: 2".
check_type @fast_elements_tr_helper (
(∀ (V : Type) (t : tree V) (lst : list (key × V)),
 @fast_elements_tr V t lst = (@elements V t ++ lst)%list)).
idtac "Assumptions:".
Abort.
Print Assumptions fast_elements_tr_helper.
Goal True.
idtac " ".

idtac "#> fast_elements_eq_elements".
idtac "Possible points: 1".
check_type @fast_elements_eq_elements (
(∀ (V : Type) (t : tree V), @fast_elements V t = @elements V t)).
idtac "Assumptions:".
```

```
Abort.
Print Assumptions fast_elements_eq_elements.
Goal True.
idtac " ".
idtac " ".
idtac "Max points - standard: 23".
idtac "Max points - advanced: 40".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "functional_extensionality_dep".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "int".
idtac "Abs".
idtac "Abs_inj".
idtac "ltb".
idtac "ltb_lt".
idtac "leb".
idtac "leb_le".
idtac "Extract.int".
idtac "Extract.Abs".
idtac "Extract.Abs_inj".
idtac "Extract.ltb".
idtac "Extract.ltb_lt".
idtac "Extract.leb".
idtac "Extract.leb_le".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "---------- empty_tree_BST ---------".
Print Assumptions empty_tree_BST.
idtac "---------- ForallT_insert ---------".
```

```
Print Assumptions ForallT_insert.
idtac "---------- insert_BST ----------".
Print Assumptions insert_BST.
idtac "---------- elements_complete ----------".
Print Assumptions elements_complete.
idtac "---------- elements_preserves_forall ----------".
Print Assumptions elements_preserves_forall.
idtac "---------- elements_preserves_relation ----------".
Print Assumptions elements_preserves_relation.
idtac "---------- elements_correct ----------".
Print Assumptions elements_correct.
idtac "---------- fast_elements_tr_helper ----------".
Print Assumptions fast_elements_tr_helper.
idtac "---------- fast_elements_eq_elements ----------".
Print Assumptions fast_elements_eq_elements.
idtac "".
idtac "********** Advanced **********".
idtac "---------- elements_complete_inverse ----------".
Print Assumptions elements_complete_inverse.
idtac "---------- bound_value ----------".
Print Assumptions bound_value.
idtac "---------- elements_complete_inverse ----------".
Print Assumptions elements_complete_inverse.
idtac "---------- sorted_app ----------".
Print Assumptions sorted_app.
idtac "---------- sorted_elements ----------".
Print Assumptions sorted_elements.
Abort.
```

Chapter 27

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 (_ ?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 "------------------- lists_table --------------------".
```

```
idtac " ".
idtac "#> StringListsTableExamples.StringListsTable.get_empty_default".
idtac "Possible points: 0.5".
check_type @StringListsTableExamples.StringListsTable.get_empty_default (
(∀ k : StringListsTableExamples.StringListsTable.key,
 StringListsTableExamples.StringListsTable.get k
   StringListsTableExamples.StringListsTable.empty =
 StringListsTableExamples.StringListsTable.default)).
idtac "Assumptions:".
Abort.
Print Assumptions StringListsTableExamples.StringListsTable.get_empty_default.
Goal True.
idtac " ".
idtac "#> StringListsTableExamples.StringListsTable.get_set_same".
idtac "Possible points: 0.5".
check_type @StringListsTableExamples.StringListsTable.get_set_same (
(∀ (k : StringListsTableExamples.StringListsTable.key)
   (v : StringListsTableExamples.StringListsTable.V)
   (t : StringListsTableExamples.StringListsTable.table),
 StringListsTableExamples.StringListsTable.get k
   (StringListsTableExamples.StringListsTable.set k v t) = v)).
idtac "Assumptions:".
Abort.
Print Assumptions StringListsTableExamples.StringListsTable.get_set_same.
Goal True.
idtac " ".
idtac "#> StringListsTableExamples.StringListsTable.get_set_other".
idtac "Possible points: 0.5".
check_type @StringListsTableExamples.StringListsTable.get_set_other (
(∀ (k k' : StringListsTableExamples.StringListsTable.key)
   (v : StringListsTableExamples.StringListsTable.V)
   (t : StringListsTableExamples.StringListsTable.table),
 k ≠ k' →
 StringListsTableExamples.StringListsTable.get k'
   (StringListsTableExamples.StringListsTable.set k v t) =
 StringListsTableExamples.StringListsTable.get k' t)).
idtac "Assumptions:".
Abort.
Print Assumptions StringListsTableExamples.StringListsTable.get_set_other.
Goal True.
idtac " ".
```

```
idtac "#> StringListsTableExamples.ex1".
idtac "Possible points: 0.5".
check_type @StringListsTableExamples.ex1 (
(StringListsTableExamples.StringListsTable.get 0
   StringListsTableExamples.StringListsTable.empty = String.EmptyString)).
idtac "Assumptions:".
Abort.
Print Assumptions StringListsTableExamples.ex1.
Goal True.
idtac " ".

idtac "#> StringListsTableExamples.ex2".
idtac "Possible points: 0.5".
check_type @StringListsTableExamples.ex2 (
(StringListsTableExamples.StringListsTable.get 0
   (StringListsTableExamples.StringListsTable.set 0
      (String.String
         (Ascii.Ascii true false false false false false true false)
         String.EmptyString) StringListsTableExamples.StringListsTable.empty) =
 String.String (Ascii.Ascii true false false false false false true false)
   String.EmptyString)).
idtac "Assumptions:".
Abort.
Print Assumptions StringListsTableExamples.ex2.
Goal True.
idtac " ".

idtac "#> StringListsTableExamples.ex3".
idtac "Possible points: 0.5".
check_type @StringListsTableExamples.ex3 (
(StringListsTableExamples.StringListsTable.get 1
   (StringListsTableExamples.StringListsTable.set 0
      (String.String
         (Ascii.Ascii true false false false false false true false)
         String.EmptyString) StringListsTableExamples.StringListsTable.empty) =
 String.EmptyString)).
idtac "Assumptions:".
Abort.
Print Assumptions StringListsTableExamples.ex3.
Goal True.
idtac " ".

idtac "-------------------  list_etable_abs  --------------------".
idtac " ".
```

```
idtac "#> StringListETableAbs.empty_ok".
idtac "Possible points: 0.5".
check_type @StringListETableAbs.empty_ok (
(StringListETableAbs.rep_ok StringListETableAbs.empty)).
idtac "Assumptions:".
Abort.
Print Assumptions StringListETableAbs.empty_ok.
Goal True.
idtac " ".

idtac "#> StringListETableAbs.set_ok".
idtac "Possible points: 0.5".
check_type @StringListETableAbs.set_ok (
(∀ (k : StringListETableAbs.key) (v : StringListETableAbs.V)
   (t : StringListETableAbs.table),
 StringListETableAbs.rep_ok t →
 StringListETableAbs.rep_ok (StringListETableAbs.set k v t))).
idtac "Assumptions:".
Abort.
Print Assumptions StringListETableAbs.set_ok.
Goal True.
idtac " ".

idtac "#> StringListETableAbs.empty_relate".
idtac "Possible points: 0.5".
check_type @StringListETableAbs.empty_relate (
(StringListETableAbs.Abs StringListETableAbs.empty =
 @empty_map StringListETableAbs.V)).
idtac "Assumptions:".
Abort.
Print Assumptions StringListETableAbs.empty_relate.
Goal True.
idtac " ".

idtac "#> StringListETableAbs.bound_relate".
idtac "Possible points: 0.5".
check_type @StringListETableAbs.bound_relate (
(∀ (t : StringListETableAbs.table) (k : StringListETableAbs.key),
 StringListETableAbs.rep_ok t →
 @SearchTree.map_bound StringListETableAbs.V k (StringListETableAbs.Abs t) =
 StringListETableAbs.bound k t)).
idtac "Assumptions:".
Abort.
Print Assumptions StringListETableAbs.bound_relate.
Goal True.
```

```
idtac " ".
idtac "#> StringListETableAbs.lookup_relate".
idtac "Possible points: 1.5".
check_type @StringListETableAbs.lookup_relate (
(∀ (t : StringListETableAbs.table) (k : StringListETableAbs.key),
 StringListETableAbs.rep_ok t →
 @map_find StringVal.V StringListETableAbs.default k
   (StringListETableAbs.Abs t) = StringListETableAbs.get k t)).
idtac "Assumptions:".
Abort.
Print Assumptions StringListETableAbs.lookup_relate.
Goal True.
idtac " ".
idtac "#> StringListETableAbs.insert_relate".
idtac "Possible points: 1.5".
check_type @StringListETableAbs.insert_relate (
(∀ (t : StringListETableAbs.table) (k : StringListETableAbs.key)
   (v : StringListETableAbs.V),
 StringListETableAbs.rep_ok t →
 @map_update StringListETableAbs.V k v (StringListETableAbs.Abs t) =
 StringListETableAbs.Abs (StringListETableAbs.set k v t))).
idtac "Assumptions:".
Abort.
Print Assumptions StringListETableAbs.insert_relate.
Goal True.
idtac " ".
idtac "#> StringListETableAbs.elements_relate".
idtac "Possible points: 1".
check_type @StringListETableAbs.elements_relate (
(∀ t : StringListETableAbs.table,
 StringListETableAbs.rep_ok t →
 StringListETableAbs.Abs t =
 @SearchTree.map_of_list StringListETableAbs.V
   (StringListETableAbs.elements t))).
idtac "Assumptions:".
Abort.
Print Assumptions StringListETableAbs.elements_relate.
Goal True.
idtac " ".
idtac "------------- list_queue -------------".
idtac " ".
```

```
idtac "#> ListQueue.is_empty_empty".
idtac "Possible points: 0.5".
check_type @ListQueue.is_empty_empty ((ListQueue.is_empty ListQueue.empty = true)).
idtac "Assumptions:".
Abort.
Print Assumptions ListQueue.is_empty_empty.
Goal True.
idtac " ".

idtac "#> ListQueue.is_empty_nonempty".
idtac "Possible points: 0.5".
check_type @ListQueue.is_empty_nonempty (
(∀ (q : ListQueue.queue) (v : ListQueue.V),
 ListQueue.is_empty (ListQueue.enq q v) = false)).
idtac "Assumptions:".
Abort.
Print Assumptions ListQueue.is_empty_nonempty.
Goal True.
idtac " ".

idtac "#> ListQueue.peek_empty".
idtac "Possible points: 0.5".
check_type @ListQueue.peek_empty (
(∀ d : ListQueue.V, ListQueue.peek d ListQueue.empty = d)).
idtac "Assumptions:".
Abort.
Print Assumptions ListQueue.peek_empty.
Goal True.
idtac " ".

idtac "#> ListQueue.peek_nonempty".
idtac "Possible points: 0.5".
check_type @ListQueue.peek_nonempty (
(∀ (d : ListQueue.V) (q : ListQueue.queue) (v : ListQueue.V),
 ListQueue.peek d (ListQueue.enq q v) = ListQueue.peek v q)).
idtac "Assumptions:".
Abort.
Print Assumptions ListQueue.peek_nonempty.
Goal True.
idtac " ".

idtac "#> ListQueue.deq_empty".
idtac "Possible points: 0.5".
check_type @ListQueue.deq_empty ((ListQueue.deq ListQueue.empty = ListQueue.empty)).
idtac "Assumptions:".
```

```
Abort.
Print Assumptions ListQueue.deq_empty.
Goal True.
idtac " ".

idtac "#> ListQueue.deq_nonempty".
idtac "Possible points: 0.5".
check_type @ListQueue.deq_nonempty (
(∀ (q : ListQueue.queue) (v : ListQueue.V),
 ListQueue.deq (ListQueue.enq q v) =
 (if ListQueue.is_empty q then q else ListQueue.enq (ListQueue.deq q) v))).
idtac "Assumptions:".
Abort.
Print Assumptions ListQueue.deq_nonempty.
Goal True.
idtac " ".

idtac "-------------------  two_list_queue  ----------------------".
idtac " ".

idtac "#> TwoListQueueAbs.empty_relate".
idtac "Possible points: 0.5".
check_type @TwoListQueueAbs.empty_relate (
(TwoListQueueAbs.Abs TwoListQueueAbs.empty = @nil TwoListQueueAbs.V)).
idtac "Assumptions:".
Abort.
Print Assumptions TwoListQueueAbs.empty_relate.
Goal True.
idtac " ".

idtac "#> TwoListQueueAbs.enq_relate".
idtac "Possible points: 0.5".
check_type @TwoListQueueAbs.enq_relate (
(∀ (q : TwoListQueueAbs.queue) (v : TwoListQueueAbs.V),
 TwoListQueueAbs.Abs (TwoListQueueAbs.enq q v) =
 (TwoListQueueAbs.Abs q ++ v :: @nil TwoListQueueAbs.V)%list)).
idtac "Assumptions:".
Abort.
Print Assumptions TwoListQueueAbs.enq_relate.
Goal True.
idtac " ".

idtac "#> TwoListQueueAbs.peek_relate".
idtac "Possible points: 1".
check_type @TwoListQueueAbs.peek_relate (
(∀ (d : TwoListQueueAbs.V) (q : TwoListQueueAbs.queue),
```

```
 TwoListQueueAbs.peek d q =
 @List.hd TwoListQueueAbs.V d (TwoListQueueAbs.Abs q))).
idtac "Assumptions:".
Abort.
Print Assumptions TwoListQueueAbs.peek_relate.
Goal True.
idtac " ".

idtac "#> TwoListQueueAbs.deq_relate".
idtac "Possible points: 1".
check_type @TwoListQueueAbs.deq_relate (
(∀ q : TwoListQueueAbs.queue,
 TwoListQueueAbs.Abs (TwoListQueueAbs.deq q) =
 @List.tl TwoListQueueAbs.V (TwoListQueueAbs.Abs q))).
idtac "Assumptions:".
Abort.
Print Assumptions TwoListQueueAbs.deq_relate.
Goal True.
idtac " ".

idtac "——————————- a_vector ——————————".
idtac " ".

idtac "#> a_vector".
idtac "Possible points: 1".
check_type @a_vector ((vector nat)).
idtac "Assumptions:".
Abort.
Print Assumptions a_vector.
Goal True.
idtac " ".

idtac "——————————- vector_cons_correct ——————————".
idtac " ".

idtac "#> vector_cons_correct".
idtac "Possible points: 2".
check_type @vector_cons_correct (
(∀ (X : Type) (x : X) (v : vector X),
 @list_of_vector X (@vector_cons X x v) = (x :: @list_of_vector X v)%list)).
idtac "Assumptions:".
Abort.
Print Assumptions vector_cons_correct.
Goal True.
idtac " ".

idtac "——————————- vector_app_correct ——————————".
```

```
idtac " ".
idtac "#> vector_app_correct".
idtac "Possible points: 2".
check_type @vector_app_correct (
(∀ (X : Type) (v1 v2 : vector X),
 @list_of_vector X (@vector_app X v1 v2) =
 (@list_of_vector X v1 ++ @list_of_vector X v2)%list)).
idtac "Assumptions:".
Abort.
Print Assumptions vector_app_correct.
Goal True.
idtac " ".
idtac "-------------------  ListsETable ---------------------".
idtac " ".
idtac "#> Manually graded: ListsETable".
idtac "Advanced".
idtac "Possible points: 6".
print_manual_grade manual_grade_for_ListsETable.
idtac " ".
idtac " ".
idtac "Max points - standard: 20".
idtac "Max points - advanced: 26".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "functional_extensionality_dep".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "int".
idtac "Abs".
idtac "Abs_inj".
idtac "ltb".
idtac "ltb_lt".
idtac "leb".
idtac "leb_le".
idtac "Extract.int".
idtac "Extract.Abs".
idtac "Extract.Abs_inj".
idtac "Extract.ltb".
idtac "Extract.ltb_lt".
idtac "Extract.leb".
idtac "Extract.leb_le".
```

```
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "---------- StringListsTableExamples.StringListsTable.get_empty_default ---------".
Print Assumptions StringListsTableExamples.StringListsTable.get_empty_default.
idtac "---------- StringListsTableExamples.StringListsTable.get_set_same ---------".
Print Assumptions StringListsTableExamples.StringListsTable.get_set_same.
idtac "---------- StringListsTableExamples.StringListsTable.get_set_other ---------".
Print Assumptions StringListsTableExamples.StringListsTable.get_set_other.
idtac "---------- StringListsTableExamples.ex1 ---------".
Print Assumptions StringListsTableExamples.ex1.
idtac "---------- StringListsTableExamples.ex2 ---------".
Print Assumptions StringListsTableExamples.ex2.
idtac "---------- StringListsTableExamples.ex3 ---------".
Print Assumptions StringListsTableExamples.ex3.
idtac "---------- StringListETableAbs.empty_ok ---------".
Print Assumptions StringListETableAbs.empty_ok.
idtac "---------- StringListETableAbs.set_ok ---------".
Print Assumptions StringListETableAbs.set_ok.
idtac "---------- StringListETableAbs.empty_relate ---------".
Print Assumptions StringListETableAbs.empty_relate.
idtac "---------- StringListETableAbs.bound_relate ---------".
Print Assumptions StringListETableAbs.bound_relate.
idtac "---------- StringListETableAbs.lookup_relate ---------".
Print Assumptions StringListETableAbs.lookup_relate.
idtac "---------- StringListETableAbs.insert_relate ---------".
Print Assumptions StringListETableAbs.insert_relate.
idtac "---------- StringListETableAbs.elements_relate ---------".
Print Assumptions StringListETableAbs.elements_relate.
idtac "---------- ListQueue.is_empty_empty ---------".
Print Assumptions ListQueue.is_empty_empty.
idtac "---------- ListQueue.is_empty_nonempty ---------".
```

```
Print Assumptions ListQueue.is_empty_nonempty.
idtac "-------- ListQueue.peek_empty ---------".
Print Assumptions ListQueue.peek_empty.
idtac "-------- ListQueue.peek_nonempty ---------".
Print Assumptions ListQueue.peek_nonempty.
idtac "-------- ListQueue.deq_empty ---------".
Print Assumptions ListQueue.deq_empty.
idtac "-------- ListQueue.deq_nonempty ---------".
Print Assumptions ListQueue.deq_nonempty.
idtac "-------- TwoListQueueAbs.empty_relate ---------".
Print Assumptions TwoListQueueAbs.empty_relate.
idtac "-------- TwoListQueueAbs.enq_relate ---------".
Print Assumptions TwoListQueueAbs.enq_relate.
idtac "-------- TwoListQueueAbs.peek_relate ---------".
Print Assumptions TwoListQueueAbs.peek_relate.
idtac "-------- TwoListQueueAbs.deq_relate ---------".
Print Assumptions TwoListQueueAbs.deq_relate.
idtac "-------- a_vector ---------".
Print Assumptions a_vector.
idtac "-------- vector_cons_correct ---------".
Print Assumptions vector_cons_correct.
idtac "-------- vector_app_correct ---------".
Print Assumptions vector_app_correct.
idtac "".
idtac "********** Advanced **********".
idtac "-------- ListsETable ---------".
idtac "MANUAL".
Abort.
```

Chapter 28

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 (_ ?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 "------------------- sort_int_correct --------------------".
```

```
idtac " ".
idtac "#> sort_int_correct".
idtac "Possible points: 3".
check_type @sort_int_correct (
(∀ al : list int,
 @Permutation.Permutation int al (sort_int al) ∧ sorted (sort_int al))).
idtac "Assumptions:".
Abort.
Print Assumptions sort_int_correct.
Goal True.
idtac " ".
idtac "-------------------  lookup_insert_eq --------------------".
idtac " ".
idtac "#> lookup_insert_eq".
idtac "Possible points: 2".
check_type @lookup_insert_eq (
(∀ (V : Type) (default : V) (t : tree V) (k : key) (v : V),
 @lookup V default k (@insert V k v t) = v)).
idtac "Assumptions:".
Abort.
Print Assumptions lookup_insert_eq.
Goal True.
idtac " ".
idtac "-------------------  lookup_insert_neq --------------------".
idtac " ".
idtac "#> lookup_insert_neq".
idtac "Possible points: 3".
check_type @lookup_insert_neq (
(∀ (V : Type) (default : V) (t : tree V) (k k' : key) (v : V),
 k ≠ k' → @lookup V default k' (@insert V k v t) = @lookup V default k' t)).
idtac "Assumptions:".
Abort.
Print Assumptions lookup_insert_neq.
Goal True.
idtac " ".
idtac " ".
idtac "Max points - standard: 8".
idtac "Max points - advanced: 8".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
```

```
idtac "functional_extensionality_dep".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "int".
idtac "Abs".
idtac "Abs_inj".
idtac "ltb".
idtac "ltb_lt".
idtac "leb".
idtac "leb_le".
idtac "Extract.int".
idtac "Extract.Abs".
idtac "Extract.Abs_inj".
idtac "Extract.ltb".
idtac "Extract.ltb_lt".
idtac "Extract.leb".
idtac "Extract.leb_le".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "---------- sort_int_correct ---------".
Print Assumptions sort_int_correct.
idtac "---------- lookup_insert_eq ---------".
Print Assumptions lookup_insert_eq.
idtac "---------- lookup_insert_neq ---------".
Print Assumptions lookup_insert_neq.
idtac "".
idtac "********** Advanced **********".
Abort.
```

Chapter 29

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 (_ ?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 "————————- balanceP ————————".
```

```
idtac " ".
idtac "#> balanceP".
idtac "Possible points: 2".
check_type @balanceP (
(∀ (V : Type) (P : key → V → Prop) (c : color)
   (l r : tree V) (k : key) (v : V),
 ForallT V P l → ForallT V P r → P k v → ForallT V P (balance V c l k v r))).
idtac "Assumptions:".
Abort.
Print Assumptions balanceP.
Goal True.
idtac " ".
idtac "—————————- insP —————————".
idtac " ".
idtac "#> insP".
idtac "Possible points: 2".
check_type @insP (
(∀ (V : Type) (P : key → V → Prop) (t : tree V) (k : key) (v : V),
 ForallT V P t → P k v → ForallT V P (ins V k v t))).
idtac "Assumptions:".
Abort.
Print Assumptions insP.
Goal True.
idtac " ".
idtac "—————————- ins_BST —————————".
idtac " ".
idtac "#> ins_BST".
idtac "Possible points: 3".
check_type @ins_BST (
(∀ (V : Type) (t : tree V) (k : key) (v : V),
 BST V t → BST V (ins V k v t))).
idtac "Assumptions:".
Abort.
Print Assumptions ins_BST.
Goal True.
idtac " ".
idtac "—————————- insert_BST —————————".
idtac " ".
idtac "#> insert_BST".
idtac "Possible points: 2".
check_type @insert_BST (
```

```
(∀ (V : Type) (t : tree V) (v : V) (k : key),
 BST V t → BST V (insert V k v t))).
idtac "Assumptions:".
Abort.
Print Assumptions insert_BST.
Goal True.
idtac " ".

idtac "-------------------  balance_lookup --------------------".
idtac " ".

idtac "#> balance_lookup".
idtac "Possible points: 6".
check_type @balance_lookup (
(∀ (V : Type) (default : V) (c : color) (k k' : key)
   (v : V) (l r : tree V),
 BST V l →
 BST V r →
 ForallT V
   (fun (k'0 : Extract.int) (_ : V) ⇒
    BinInt.Z.lt (Extract.Abs k'0) (Extract.Abs k)) l →
 ForallT V
   (fun (k'0 : Extract.int) (_ : V) ⇒
    BinInt.Z.gt (Extract.Abs k'0) (Extract.Abs k)) r →
 lookup V default k' (balance V c l k v r) =
 (if BinInt.Z.ltb (Extract.Abs k') (Extract.Abs k)
  then lookup V default k' l
  else
   if BinInt.Z.gtb (Extract.Abs k') (Extract.Abs k)
   then lookup V default k' r
   else v))).
idtac "Assumptions:".
Abort.
Print Assumptions balance_lookup.
Goal True.
idtac " ".

idtac "-------------------  lookup_ins_eq --------------------".
idtac " ".

idtac "#> lookup_ins_eq".
idtac "Possible points: 3".
check_type @lookup_ins_eq (
(∀ (V : Type) (default : V) (t : tree V) (k : key) (v : V),
 BST V t → lookup V default k (ins V k v t) = v)).
```

```
idtac "Assumptions:".
Abort.
Print Assumptions lookup_ins_eq.
Goal True.
idtac " ".

idtac "------------------- lookup_ins_neq --------------------".
idtac " ".

idtac "#> lookup_ins_neq".
idtac "Possible points: 3".
check_type @lookup_ins_neq (
(∀ (V : Type) (default : V) (t : tree V) (k k' : key) (v : V),
 BST V t →
 k ≠ k' → lookup V default k' (ins V k v t) = lookup V default k' t)).
idtac "Assumptions:".
Abort.
Print Assumptions lookup_ins_neq.
Goal True.
idtac " ".

idtac "------------------- lookup_insert --------------------".
idtac " ".

idtac "#> lookup_insert_eq".
idtac "Possible points: 2".
check_type @lookup_insert_eq (
(∀ (V : Type) (default : V) (t : tree V) (k : key) (v : V),
 BST V t → lookup V default k (insert V k v t) = v)).
idtac "Assumptions:".
Abort.
Print Assumptions lookup_insert_eq.
Goal True.
idtac " ".

idtac "#> lookup_insert_neq".
idtac "Possible points: 1".
check_type @lookup_insert_neq (
(∀ (V : Type) (default : V) (t : tree V) (k k' : key) (v : V),
 BST V t →
 k ≠ k' → lookup V default k' (insert V k v t) = lookup V default k' t)).
idtac "Assumptions:".
Abort.
Print Assumptions lookup_insert_neq.
Goal True.
idtac " ".
```

```
idtac "------------- RB_blacken_parent -------------".
idtac " ".
idtac "#> RB_blacken_parent".
idtac "Possible points: 2".
check_type @RB_blacken_parent (
(∀ (V : Type) (t : tree V) (n : nat), RB V t Red n → RB V t Black n)).
idtac "Assumptions:".
Abort.
Print Assumptions RB_blacken_parent.
Goal True.
idtac " ".
idtac "------------- RB_blacken_root -------------".
idtac " ".
idtac "#> RB_blacken_root".
idtac "Possible points: 2".
check_type @RB_blacken_root (
(∀ (V : Type) (t : tree V) (n : nat),
 RB V t Black n → ∃ n' : nat , RB V (make_black V t) Red n')).
idtac "Assumptions:".
Abort.
Print Assumptions RB_blacken_root.
Goal True.
idtac " ".
idtac "------------- ins_RB -------------".
idtac " ".
idtac "#> ins_RB".
idtac "Possible points: 10".
check_type @ins_RB (
(∀ (V : Type) (k : key) (v : V) (t : tree V) (n : nat),
 (RB V t Black n → NearlyRB V (ins V k v t) n) ∧
 (RB V t Red n → RB V (ins V k v t) Black n))).
idtac "Assumptions:".
Abort.
Print Assumptions ins_RB.
Goal True.
idtac " ".
idtac "------------- insert_RB -------------".
idtac " ".
idtac "#> insert_RB".
idtac "Possible points: 2".
check_type @insert_RB (
```

```
(∀ (V : Type) (t : tree V) (k : key) (v : V) (n : nat),
 RB V t Red n → ∃ n' : nat, RB V (insert V k v t) Red n')).
idtac "Assumptions:".
Abort.
Print Assumptions insert_RB.
Goal True.
idtac " ".
idtac "-------------------- redblack_bound --------------------".
idtac " ".
idtac "#> Manually graded: redblack_bound".
idtac "Advanced".
idtac "Possible points: 6".
print_manual_grade manual_grade_for_redblack_bound.
idtac " ".
idtac " ".
idtac "Max points - standard: 40".
idtac "Max points - advanced: 46".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "functional_extensionality_dep".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "int".
idtac "Abs".
idtac "Abs_inj".
idtac "ltb".
idtac "ltb_lt".
idtac "leb".
idtac "leb_le".
idtac "Extract.int".
idtac "Extract.Abs".
idtac "Extract.Abs_inj".
idtac "Extract.ltb".
idtac "Extract.ltb_lt".
idtac "Extract.leb".
idtac "Extract.leb_le".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
```

```
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "---------- balanceP ---------".
Print Assumptions balanceP.
idtac "---------- insP ---------".
Print Assumptions insP.
idtac "---------- ins_BST ---------".
Print Assumptions ins_BST.
idtac "---------- insert_BST ---------".
Print Assumptions insert_BST.
idtac "---------- balance_lookup ---------".
Print Assumptions balance_lookup.
idtac "---------- lookup_ins_eq ---------".
Print Assumptions lookup_ins_eq.
idtac "---------- lookup_ins_neq ---------".
Print Assumptions lookup_ins_neq.
idtac "---------- lookup_insert_eq ---------".
Print Assumptions lookup_insert_eq.
idtac "---------- lookup_insert_neq ---------".
Print Assumptions lookup_insert_neq.
idtac "---------- RB_blacken_parent ---------".
Print Assumptions RB_blacken_parent.
idtac "---------- RB_blacken_root ---------".
Print Assumptions RB_blacken_root.
idtac "---------- ins_RB ---------".
Print Assumptions ins_RB.
idtac "---------- insert_RB ---------".
Print Assumptions insert_RB.
idtac "".
idtac "********** Advanced **********".
idtac "---------- redblack_bound ---------".
idtac "MANUAL".
Abort.
```

Chapter 30

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 (_ ?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: 10".
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 (
(∀ (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 (
(∀ (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 (
(∀ (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: 34".
idtac "Max points - advanced: 34".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "functional_extensionality_dep".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "int".
idtac "Abs".
idtac "Abs_inj".
idtac "ltb".
idtac "ltb_lt".
idtac "leb".
```

```
idtac "leb_le".
idtac "Extract.int".
idtac "Extract.Abs".
idtac "Extract.Abs_inj".
idtac "Extract.ltb".
idtac "Extract.ltb_lt".
idtac "Extract.leb".
idtac "Extract.leb_le".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "---------- Integers.succ_correct ---------".
Print Assumptions Integers.succ_correct.
idtac "---------- Integers.addc_correct ---------".
Print Assumptions Integers.addc_correct.
idtac "---------- Integers.compare_correct ---------".
Print Assumptions Integers.compare_correct.
idtac "---------- successor_of_Z_constant_time ---------".
idtac "MANUAL".
idtac "---------- look_leaf ---------".
Print Assumptions look_leaf.
idtac "---------- look_ins_same ---------".
Print Assumptions look_ins_same.
idtac "---------- look_ins_same ---------".
Print Assumptions look_ins_same.
idtac "---------- pos2nat_injective ---------".
Print Assumptions pos2nat_injective.
idtac "---------- nat2pos_injective ---------".
Print Assumptions nat2pos_injective.
idtac "---------- is_trie ---------".
Print Assumptions is_trie.
idtac "---------- empty_relate ---------".
```

```
Print Assumptions empty_relate.
idtac "-------- lookup_relate --------".
Print Assumptions lookup_relate.
idtac "-------- insert_relate --------".
Print Assumptions insert_relate.
idtac "".
idtac "********** Advanced **********".
Abort.
```

Chapter 31

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 (_ ?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".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "functional_extensionality_dep".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "int".
idtac "Abs".
idtac "Abs_inj".
idtac "ltb".
idtac "ltb_lt".
idtac "leb".
idtac "leb_le".
idtac "Extract.int".
idtac "Extract.Abs".
idtac "Extract.Abs_inj".
idtac "Extract.ltb".
idtac "Extract.ltb_lt".
idtac "Extract.leb".
idtac "Extract.leb_le".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "---------- List_Priqueue.select_perm ---------".
Print Assumptions List_Priqueue.select_perm.
idtac "---------- List_Priqueue.select_biggest_aux ---------".
Print Assumptions List_Priqueue.select_biggest_aux.
idtac "---------- List_Priqueue.select_biggest ---------".
Print Assumptions List_Priqueue.select_biggest.
idtac "---------- List_Priqueue.delete_max_None_relate ---------".
Print Assumptions List_Priqueue.delete_max_None_relate.
idtac "---------- List_Priqueue.delete_max_Some_relate ---------".
```

```
Print Assumptions List_Priqueue.delete_max_Some_relate.
idtac "-------- List_Priqueue.delete_max_Some_relate --------".
Print Assumptions List_Priqueue.delete_max_Some_relate.
idtac "".
idtac "********** Advanced **********".
Abort.
```

Chapter 32

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 (_ ?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 (
(forall (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 (
(forall (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 (
(forall (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".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "functional_extensionality_dep".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "int".
idtac "Abs".
idtac "Abs_inj".
idtac "ltb".
idtac "ltb_lt".
idtac "leb".
idtac "leb_le".
idtac "Extract.int".
idtac "Extract.Abs".
idtac "Extract.Abs_inj".
idtac "Extract.ltb".
idtac "Extract.ltb_lt".
idtac "Extract.leb".
idtac "Extract.leb_le".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
```

```
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "---------- BinomQueue.empty_priq ---------".
Print Assumptions BinomQueue.empty_priq.
idtac "---------- BinomQueue.smash_valid ---------".
Print Assumptions BinomQueue.smash_valid.
idtac "---------- BinomQueue.carry_valid ---------".
Print Assumptions BinomQueue.carry_valid.
idtac "---------- priqueue_elems ---------".
idtac "MANUAL".
idtac "---------- BinomQueue.tree_elems_ext ---------".
Print Assumptions BinomQueue.tree_elems_ext.
idtac "---------- BinomQueue.tree_perm ---------".
Print Assumptions BinomQueue.tree_perm.
idtac "---------- BinomQueue.priqueue_elems_ext ---------".
Print Assumptions BinomQueue.priqueue_elems_ext.
idtac "---------- BinomQueue.abs_perm ---------".
Print Assumptions BinomQueue.abs_perm.
idtac "---------- BinomQueue.can_relate ---------".
Print Assumptions BinomQueue.can_relate.
idtac "---------- BinomQueue.empty_relate ---------".
Print Assumptions BinomQueue.empty_relate.
idtac "---------- BinomQueue.smash_elems ---------".
Print Assumptions BinomQueue.smash_elems.
idtac "".
idtac "********** Advanced **********".
Abort.
```

Chapter 33

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 (_ ?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".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "functional_extensionality_dep".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "int".
idtac "Abs".
idtac "Abs_inj".
idtac "ltb".
idtac "ltb_lt".
idtac "leb".
idtac "leb_le".
idtac "Extract.int".
idtac "Extract.Abs".
idtac "Extract.Abs_inj".
```

```
idtac "Extract.ltb".
idtac "Extract.ltb_lt".
idtac "Extract.leb".
idtac "Extract.leb_le".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "---------- ScratchPad2.insert_sorted ---------".
Print Assumptions ScratchPad2.insert_sorted.
idtac "---------- list_nat_in ---------".
Print Assumptions list_nat_in.
idtac "".
idtac "********** Advanced **********".
Abort.
```

Chapter 34

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 (_ ?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: 6".
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: 6".
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: 6".
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 (
(∀ (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 (
(∀ (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 ((∀ 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 (
```

```
(∀ (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: 6".
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: 51".
idtac "Max points - advanced: 51".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "functional_extensionality_dep".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "int".
idtac "Abs".
idtac "Abs_inj".
idtac "ltb".
idtac "ltb_lt".
idtac "leb".
```

```
idtac "leb_le".
idtac "Extract.int".
idtac "Extract.Abs".
idtac "Extract.Abs_inj".
idtac "Extract.ltb".
idtac "Extract.ltb_lt".
idtac "Extract.leb".
idtac "Extract.leb_le".
idtac "".
idtac "".
idtac "********** Summary **********".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "********** Standard **********".
idtac "---------- Sremove_elements ---------".
Print Assumptions Sremove_elements.
idtac "---------- InA_map_fst_key ---------".
Print Assumptions InA_map_fst_key.
idtac "---------- Sorted_lt_key ---------".
Print Assumptions Sorted_lt_key.
idtac "---------- cardinal_map ---------".
Print Assumptions cardinal_map.
idtac "---------- Sremove_cardinal_less ---------".
Print Assumptions Sremove_cardinal_less.
idtac "---------- Mremove_elements ---------".
Print Assumptions Mremove_elements.
idtac "---------- Mremove_cardinal_less ---------".
Print Assumptions Mremove_cardinal_less.
idtac "---------- fold_right_rev_left ---------".
Print Assumptions fold_right_rev_left.
idtac "---------- Snot_in_empty ---------".
Print Assumptions Snot_in_empty.
idtac "---------- Sin_domain ---------".
Print Assumptions Sin_domain.
idtac "---------- subset_nodes_sub ---------".
```

```
Print Assumptions subset_nodes_sub.
idtac "------- select_terminates -------".
Print Assumptions select_terminates.
idtac "------- adj_ext -------".
Print Assumptions adj_ext.
idtac "------- in_colors_of_1 -------".
Print Assumptions in_colors_of_1.
idtac "------- color_correct -------".
Print Assumptions color_correct.
idtac "".
idtac "********** Advanced **********".
Abort.
```