Part 1

Functional Programs

In the functional-programming style, programs are structured as mathematical functions that return values. Data is immutable and tends to be organized into data structures defined by inductive datatypes.

This Part of the book considers functional programs and how to reason about such programs. We have already seen the basic elements of functional programs in the previous chapters. Now, we put the elements to work to build and prove programs.

I will introduce the concepts of intrinsic versus extrinsic specifications, data-structure invariants, abstraction functions, and modules.

Chapter 6

Lists

Lists are the most common data structure in functional programs. It is therefore appropriate to devote this first chapter in this Part to programs on lists.

In Chapter 5, where we were learning to write proofs, we turned off Dafny's automatic induction. From now on, we're going to allow Dafny to apply its automatic induction. This will not eliminate the need for manually written proofs, but it will reduce our work.

6.0.List Definition

Throughout this chapter, I will use the following definition of a list datatype:

```
datatype List<T> = Nil | Cons(head: T, tail: List<T>)
```

The value `Nil` represents the empty list, whereas `Cons(x, xs)` represents the list that starts with the element `x` and then continues with the elements of list `xs`. The list type is parameterized by a type `T`, which is the type of every element in the list.

Here are four examples of lists. To the left is how you might write each list on a piece of paper. In the middle is the corresponding expression of type `List`, and to the right is a type for the list value.

informal notation	`List` expression	possible type
`2, 5, 3`	`Cons(2, Cons(5, Cons(3, Nil)))`	`List<nat>`
`true, true`	`Cons(true, Cons(true, Nil))`	`List<bool>`
	`Nil`	`List<int>`
`"gimmie", "lists"`	`Cons("gimmie", Cons("lists", Nil))`	`List<string>`

The element `Nil` can have type `List<T>` for any type `T`. Note that all elements in a list have the same type. It is not legal write

```
Cons(true, Cons(3, Nil)) // error: elements must have common type
```

6.1.Length

Here is a function that returns the length of a list:

```
function Length<T>(xs: List<T>): nat {

match xs

case Nil => 0

case Cons(_, tail) => 1 + Length(tail)

}
```

It says that an empty list has length `0` and other lists are `1` longer than their tail.

Here are two little reminders from previous chapters.

As a first reminder, the result type of `Length` is **`nat`**, which denotes the natural numbers, that is, the non-negative integers.

As a second reminder, the definition of `Length` uses a **`match`** expression to distinguish the two variants of lists. An alternative definition would use an **`if`**-**`then`**-**`else`** expression:

```
if xs == Nil then 0 else 1 + Length(xs.tail)
```

The choice between various control structures, like **`match`** versus **`if`**, is simply a matter of taste, but **`match`** is the typical choice when there are many constructors. Note that `tail` in the first definition is a bound variable introduced as part of the **`match`** case, whereas `tail` in the second definition names the destructor that gives us the second component of a `Cons`, as defined in the datatype declaration for `List`.

Exercise 6.0.

Define `Length` as above with a **`match`** and define a function `Length′` with the **`if`** above (but with the recursive call to `Length′`, not `Length`). Declare and prove a lemma that says that `Length` and `Length′` always return the same value.

The constructor Cons makes a list by putting an element in front of another list. Sometimes, we want just the opposite—to make a list by putting an element after another list, an operation commonly named Snoc. Because inductive lists are accessed from front to back, Snoc is computationally more expensive than Cons. Here is a definition:

```
function Snoc<T>(xs: List<T>, y: T): List<T> {

match xs

case Nil => Cons(y, Nil)

case Cons(x, tail) => Cons(x, Snoc(tail, y))

}
```

To gain more confidence that we have defined a function as intended, it is always a good idea to check that it has some properties we'd expect. We can do that by stating and proving lemmas. For example, we expect Snoc(xs, x) to be a list 1 longer than xs:

```
lemma LengthSnoc<T>(xs: List<T>, x: T)

ensures Length(Snoc(xs, x)) == Length(xs) + 1

{

}
```

Exercise 6.1.

Turn off automatic induction and write a proof for LengthSnoc.

6.2.Intrinsic versus Extrinsic Specifications

We are about to encounter an important distinction between two styles of writing specifications.

Let's define a function Append that returns the concatenation of two lists:

```
function Append<T>(xs: List<T>, ys: List<T>): List<T> {

match xs

case Nil => ys

case Cons(x, tail) => Cons(x, Append(tail, ys))

}
```

There is a relation between Append and Length. Let's capture that relation in a lemma, which is proved automatically:

```
lemma LengthAppend<T>(xs: List<T>, ys: List<T>)

ensures Length(Append(xs, ys)) == Length(xs) + Length(ys)

{

}
```

As we have seen before, introducing a lemma like this is commonly how we state and prove properties of functions. The property stated by the lemma is said to be *extrinsic* to the function definition of `Append`, because the lemma is not part of the function declaration itself. This is not the only way to state and prove properties of functions. Another way is to define the function with a postcondition, like we do for methods. A property given in a function postcondition is called *intrinsic*, because it is part of the function declaration and is verified as part of checking the well-formedness of the function.

Here is an alternative definition of `Append` that intrinsically states the property that lemma `LengthAppend` states extrinsically:

```
function Append<T>(xs: List<T>, ys: List<T>): List<T>

ensures Length(Append(xs, ys)) == Length(xs) + Length(ys)

{

match xs

case Nil => ys

case Cons(x, tail) => Cons(x, Append(tail, ys))

}
```

To refer to the value returned by the function, the postcondition simply mentions the function invoked on its parameters, namely `Append(xs, ys)`.

Methods in Dafny are always opaque, so the only properties that are known about them are those stated in the method's specification. In other words, properties about methods are always intrinsic, never extrinsic. Functions, on the other hand, are transparent, so we have the choice of stating properties about them intrinsically or extrinsically.

One advantage of intrinsic specifications is that they get used with every application of the function. In contrast, an extrinsic lemma needs to be applied explicitly in order to make use of its property during verification. Having specifications get used automatically might seem like it would always be an advantage, but it is not. When verification conditions get large, having too much information around may overwhelm the verifier, resulting in long verification times or even in failed verifications (because of reaching some internal resource bounds in the verifier). Therefore, common practice for functions is to state and prove properties extrinsically.

In some cases, writing a property intrinsically is not even an option. This occurs when the property mentions the function several times. For example, it would not be possible to state the property `Mirror(Mirror(t)) == t` in the postcondition of function `Mirror` (Section 5.7). The reason is that the outer call to `Mirror` is a recursive call, so one would have to prove that its argument, namely `Mirror(t)`, is smaller than `t`. This is not possible for all `t`: Consider some particular `t` and `t′` where `t′ == Mirror(t)`, which means that `t == Mirror(t′)`. We certainly cannot argue both that `t` is smaller than `t′` and that `t′` is smaller than `t`. Other examples of properties ill-suited for intrinsic specifications are commutativity and transitivity.

The one situation where intrinsic specifications make sense and can be recommended occurs when the property is likely to be of concern for all clients of the function. It will be convenient in this chapter to think of the length of `Append` as falling in that category, so I will assume the definition of `Append` with the postcondition

```
Length(Append(xs, ys)) == Length(xs) + Length(ys)
```

Exercise 6.2.

Prove that `Snoc` is just a special case of `Append`, that is, that

```
Snoc(xs, y) == Append(xs, Cons(y, Nil))
```

6.2.0.Other properties of Append

It is often useful to know various algebraic properties about functions we define. For example, consider a binary function , which here I will write as an infix operator. A value *L* is called a *left unit element* of if, for all *x*, *L* *x* = *x*, and a value *R* is called a *right unit element* of if, for all *x*, *x* *R* = *x*.

By the definition of `Append`, we see immediately that `Nil` is a left unit, that is, `Append(Nil, x) == x`. We can also prove that `Nil` is a right unit element of `Append`:

```
lemma AppendNil<T>(xs: List<T>)

ensures Append(xs, Nil) == xs

{

}
```

Another useful algebraic property of a function is *associativity*. It says that it doesn't matter how you parenthesize it. For any *x*, *y*, and *z*,

(*x* *y*) *z* = *x* (*y* *z*)

Function `Append` is associative:

```
lemma AppendAssociative<T>(xs: List<T>, ys: List<T>, zs: List<T>)

ensures Append(Append(xs, ys), zs) == Append(xs, Append(ys, zs))

{

}
```

Exercise 6.3.

Mark `AppendNil` with the attribute `{:induction false}` and give a manual proof of the lemma.

Exercise 6.4.

Mark `AppendAssociative` with the attribute `{:induction false}` and give a manual proof of the lemma.

Exercise 6.5.

We saw one connection between `Append` and `Length` above, and we chose to specify it intrinsically. Here's another property of those two functions, which is best given extrinsically.

```
lemma AppendDecomposition<T>(a: List<T>, b: List<T>,
c: List<T>, d: List<T>)

requires Length(a) == Length(c)

requires Append(a, b) == Append(c, d)

ensures a == c && b == d
```

Prove this lemma, with and without automatic induction.

Exercise 6.6.

For an operator on the integers with a left unit `L` and a right unit `R`, prove that `L` and `R` are equal. We can set this up in Dafny as follows:

```
function F(x: int, y: int): int



const L: int

const R: int



lemma LeftUnit(x: int)

ensures F(L, x) == x



lemma RightUnit(x: int)

ensures F(x, R) == x
```

Here, the body-less function `F(x, y)` denotes an arbitrary operation (which I had written as *x y* above). I declared the names `L` and `R` as arbitrary constants; an alternative would be to declare them as body-less nullary functions. The two lemmas, which state properties about `F`, `L`, and `R`, are given as axioms, so go ahead and use them but don't try to prove them. State and prove a lemma that `L == R`.

Exercise 6.7.

A value *L* is called a *left zero element* of an operator if, for all *x*, *L x* = *L*, and a value *R* is called a *right zero element* of if, for all *x*, *x R* = *R*. If an operator has a left zero element *L* and a right zero element *R*, then these two elements are equal. State and prove this property for an arbitrary operator on the integers. Hint: See Exercise 6.6 for how to set this up.

6.3. `Take` and `Drop`

Function `Append` makes two lists into one. The functions `Take` and `Drop` do the reverse, they break a list into two parts.

```
function Take<T>(xs: List<T>, n: nat): List<T>

requires n <= Length(xs)

{

if n == 0 then Nil else Cons(xs.head, Take(xs.tail, n - 1))

}



function Drop<T>(xs: List<T>, n: nat): List<T>

requires n <= Length(xs)

{

if n == 0 then xs else Drop(xs.tail, n - 1)

}
```

Defined with the precondition `n <= Length(xs)` like this, these functions are rather picky about how many elements can be taken or dropped. The precondition allows the function bodies to access `xs.head` and `xs.tail` (which requires `xs.Cons?`) in the branch where `n != 0`.

Exercise 6.8.

Write more liberal versions of `Take` and `Drop` that do not have any precondition. Asked to take more elements than the length of the list, the liberal version returns the entire list. Asked to drop more elements than the length of the list, the liberal version return the empty list. Prove that if `n <= Length(xs)`, then the liberal versions return the same value as the respective strict versions above.

Exercise 6.9.

With automatic induction turned off, prove the lemma in Exercise 6.8.

It is easy to prove the correspondence between `Append`, `Take`, and `Drop`:

```
lemma AppendTakeDrop<T>(xs: List<T>, n: nat)

requires n <= Length(xs)

ensures Append(Take(xs, n), Drop(xs, n)) == xs

{

}



lemma TakeDropAppend<T>(xs: List<T>, ys: List<T>)

ensures Take(Append(xs, ys), Length(xs)) == xs

ensures Drop(Append(xs, ys), Length(xs)) == ys

{
}
```

The proof of lemma `TakeDropAppend` makes use of the intrinsic property given by the postcondition of function `Append`.

> **Exercise 6.10.**
>
> Disable automatic induction and prove lemma `AppendTakeDrop`.

> **Exercise 6.11.**
>
> Disable automatic induction and prove lemma `TakeDropAppend`.

6.4.At

The elements of a list are ordered. We can ask for an element at a given index into the list.

```
function At<T>(xs: List<T>, i: nat): T

requires i < Length(xs)

{

if i == 0 then xs.head else At(xs.tail, i - 1)

}
```

The precondition of `At` requires `i` to be a proper index into the list `xs`. Note that this implies `xs.Cons?`, which justifies our use of the destructors `xs.head` and `xs.tail` in the body.

The element at position `i` in a list `xs` is the element preceded by `i` elements. So, this element is the head of the list you obtain by dropping `i` elements from `xs`. Let's try to formulate this property as a lemma:

```
lemma AtDropHead<T>(xs: List<T>, i: nat)

requires i < Length(xs)

ensures At(xs, i) == Drop(xs, i).head // error: .head

// requires Cons?
```

It's clear we need a precondition for the lemma, because to talk about `At(xs, i)`, we must have `i < Length(xs)`. But trying to state the lemma as I did here produces a complaint that we might be access the `head` member of a list that does not have a `head`. In order for `Drop(xs, i).head` to be well-defined, we need to know that `Drop(xs, i)` returns a list of the `Cons` variant (that is, a non-`Nil` list). There are several ways out of this situation. The simplest, which is appropriate here, is to also include `Drop(xs, i).Cons?` as a proof goal of the lemma. This gives us

```
lemma AtDropHead<T>(xs: List<T>, i: nat)

requires i < Length(xs)

ensures Drop(xs, i).Cons? && At(xs, i) == Drop(xs, i).head
```

```
{
}
```

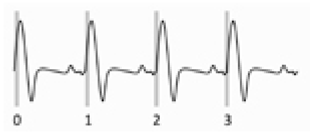
which is handled by Dafny's automatic induction.

We also connect `At` and `Append` in a lemma:

```
lemma AtAppend<T>(xs: List<T>, ys: List<T>, i: nat)
requires i < Length(Append(xs, ys))
ensures At(Append(xs, ys), i)
== if i < Length(xs) then
At(xs, i)
else
At(ys, i - Length(xs))
{
}
```

Sidebar 6.0

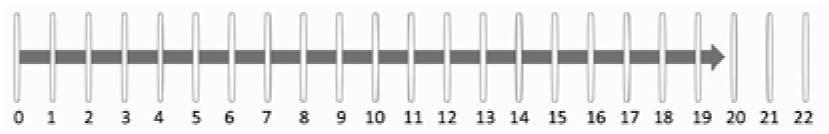
When we're talking about how many steps we have taken on a “journey”, it's clear what the numbers mean. For example, when you take your pulse, you start a 6-second interval and count “zero, one, two, . . . ”



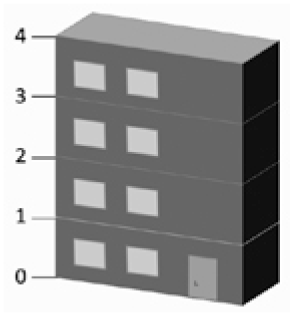
and then you multiply the final number you said by `10` to get your beats per minute.

But what if we want to name the steps (or, in this case, heartbeats) themselves? For that, the world has more than one convention.

When you have lived for `19` years and someone asks your age, you say “`19`”. During the year following your birth, you're living your “year `0`”.



If you live in an apartment `2` floors above the ground floor (street level), then in some countries (e.g., most of Europe), you'd press `2` in the elevator to get to your floor, whereas in other countries (e.g., the US), you'd press `3`.



Here and elsewhere, the names given to elements of a list, sequence, or array start with `0`, like heartbeats, ages, and floors in certain countries. Function `At(xs, i)` therefore has the precondition `0 <= i < Length(xs)`.

Exercise 6.12.

Turn off automatic induction and write a proof for `AtAppend`. Hint: Use a proof of the form

```
match xs
case Nil =>
case Cons(x, tail) =>
if i == 0 {
// calc…
} else {
// calc…
}
```

(Are you happy about automatic induction?)

6.5.`Find`

We define a function `Find` that returns the position of an element in a list. If the element occurs more than once, the index to the first element is returned. If the element does not occur in the list, the length of the list is returned. Stated differently and more concisely, `Find(xs, x)` returns the length of the longest prefix of `xs` that does not contain `x`.

```
function Find<T(==)>(xs: List<T>, y: T): nat
ensures Find(xs, y) <= Length(xs)
{
match xs
case Nil => 0
case Cons(x, tail) =>
if x == y then 0 else 1 + Find(tail, y)

}
```

Whereas our previous operations have been entirely parametric in the element type, function `Find` only makes sense if the element type is one whose values can be compared, that is, a type on which equality is defined. In Dafny, every type has an equality operation in ghost contexts, so we can write `Find` as a ghost function. But not every type offers an equality comparison in compiled contexts. For example, you cannot compare two functions for equality in compiled code, for that may take infinite time to evaluate. If we want `Find` to be a compiled function, we must therefore restrict it to type parameters that support a compiled version of equality. This is done by decorating the type parameter with the sux `(==)`, as I did above (and I'll say more about it in Section 9.4).

Exercise 6.13.

Remove the sux `(==)` from `Find`'s type parameter. What error message do you get?

`Find(xs, x)` always returns a value between `0` and `Length(xs)`, inclusive, which is a property that most clients of `Find` are going to need. Therefore, we declare this property intrinsically, the lower bound by using result type **`nat`** and the upper bound by the declared **`ensures`** clause.

Many properties can be proved about the various list functions we have seen so far. Here are but four examples, where `xs` and `ys` have type `List<T>` for some type `T`, `x` has type `T`, and `i` has type **`nat`**:

```
lemma AtFind<T>(xs: List<T>, y: T)

ensures Find(xs, y) == Length(xs) || At(xs, Find(xs, y)) == y



lemma BeforeFind<T>(xs: List<T>, y: T, i: nat)

ensures i < Find(xs, y) ==> At(xs, i) != y



lemma FindAppend<T>(xs: List<T>, ys: List<T>, y: T)

ensures Find(xs, y) == Length(xs)

|| Find(Append(xs, ys), y) == Find(xs, y)



lemma FindDrop<T>(xs: List<T>, y: T, i: nat)

ensures i <= Find(xs, y) ==> Find(xs, y) == Find(Drop(xs, i), y) + i
```

Exercise 6.14.

The proof goals of lemmas `AtFind` and `BeforeFind` call function `At`, which has a precondition. Explain for each of those calls how the precondition is met.

Exercise 6.15.

Prove lemma `AtFind`, with and without automatic induction.

Exercise 6.16.

Prove lemma `BeforeFind`, with and without automatic induction.

Exercise 6.17.

Prove lemma `FindAppend`, with and without automatic induction.

Exercise 6.18.

Prove lemma `FindDrop`, with and without automatic induction.

6.6.List Reversal

As a final list operation, we consider reversing the elements of a list. Here is one function that reverses a list:

```
function SlowReverse<T>(xs: List<T>): List<T> {

match xs

case Nil => Nil

case Cons(x, tail) => Snoc(SlowReverse(tail), x)

}
```

This version may be easy to understand, but it would execute slowly. To be more precise, `SlowReverse(xs)` requires a number of steps proportional to the square of the length of `xs`. We will write a more ecient version and prove that it outputs the same value as this slow version. But first, let us at least prove that `SlowReverse(xs)` has the same length as `xs`.

```
lemma LengthSlowReverse<T>(xs: List<T>)

ensures Length(SlowReverse(xs)) == Length(xs)
```

Let's see if we can predict what is needed for its proof. Looking at the definition of `SlowReverse`, we expect the `Nil` case to immediately satisfy the lemma. In the `Cons` case, the induction hypothesis will give us that `SlowReverse(tail)` has the same length as `tail`, so if we only have the fact that `Snoc` increases the list length by `1`, then we would be done. Let's try that:

```
{

match xs

case Nil =>

case Cons(x, tail) =>

LengthSnoc(SlowReverse(tail), x);

}
```

Indeed, the verifier is able to fill in the rest of the details.

The trick to writing a more ecient version of list reversal is to add what is called an *accumulator* parameter that keeps track of the part that has been reversed so far. More precisely, the idea for our auxiliary function `ReverseAux(xs, acc)` is that it return `Append(xs′, acc)`, where `xs′` is the list `xs` reversed. Here is the definition:

```
function ReverseAux<T>(xs: List<T>, acc: List<T>): List<T>

{

match xs

case Nil => acc

case Cons(x, tail) => ReverseAux(tail, Cons(x, acc))

}
```

To reverse `Cons(x, tail)` and append `acc` to it, we reverse tail and append `Cons(x, acc)` to it.

If we were writing a method, we would probably have recorded the intended effect of `ReverseAux` in an **`ensures`** clause, as an intrinsic specification. We could do that for `ReverseAux`. For one, we expect only one other caller of the auxiliary function, so our guiding principle of using an intrinsic specification applies: every caller is interested in this property. However, to verify the property, we need to provide some manual guidance. I will show how to do that in Section 6.7. For now, let's state and prove the property extrinsically.

```
lemma ReverseAuxSlowCorrect<T>(xs: List<T>, acc: List<T>)

ensures ReverseAux(xs, acc) == Append(SlowReverse(xs), acc)

{

match xs

case Nil =>

case Cons(x, tail) =>

calc {

Append(SlowReverse(xs), acc);

==  // def. SlowReverse

Append(Snoc(SlowReverse(tail), x), acc);
```

In the proof, we start with the right-hand side, since it is more complicated and may therefore give us better guidance. After applying the definition of `SlowReverse`, we may hope to apply the definition of `Snoc`. Since we are not in a situation where we know which of the two cases of `Snoc` applies, we would need to bring in the entire definition of `Snoc` into the calculation. Sometimes, this is what we have to do. In the current situation, however, we have the alternative of rewriting `Snoc` in terms of `Append`.

```
==  { SnocAppend(SlowReverse(tail), x); }

Append(Append(SlowReverse(tail), Cons(x, Nil)), acc);
```

Here, we can apply the associativity of `Append`.

```
==  { AppendAssociative(SlowReverse(tail), Cons(x, Nil), acc); }

Append(SlowReverse(tail), Append(Cons(x, Nil), acc));
```

Appending `acc` to the singleton list `Cons(x, Nil)` will give a list that starts with `x` and ends with `acc`. We don't have a named lemma for this property, but it follows from two unrollings of the definition of `Append`. To document this proof step, we use an assertion in the hint.

```
==  { assert Append(Cons(x, Nil), acc) == Cons(x, acc); }

Append(SlowReverse(tail), Cons(x, acc));
```

The formula now has a shape like that in the lemma we are trying to prove, so we invoke the induction hypothesis.

```
==  { ReverseAuxSlowCorrect(tail, Cons(x, acc)); }

ReverseAux(tail, Cons(x, acc));
```

The verifier is now happy with the proof. Actually, the verifier was happy even before our explicit invocation of the induction hypothesis, because that was taken care of by automatic induction. But to finish up a readable proof, we take one more step:

```
==  // def. ReverseAux

ReverseAux(xs, acc);

}

}
```

We have now proved that `ReverseAux` does what we intended. What remains is to use it in a function `Reverse` and to prove that the end result is the same as applying `SlowReverse`.

```
function Reverse<T>(xs: List<T>): List<T> {

ReverseAux(xs, Nil)

}
```

The proof of correctness makes use of the fact that `Nil` is a right unit element of `Append`, but is otherwise straightforward.

```
lemma ReverseCorrect<T>(xs: List<T>)

ensures Reverse(xs) == SlowReverse(xs)

{

calc {

Reverse(xs);

== // def. Reverse

ReverseAux(xs, Nil);

==  { ReverseAuxSlowCorrect(xs, Nil); }

Append(SlowReverse(xs), Nil);

==  { AppendNil(SlowReverse(xs)); }
SlowReverse(xs);

}

}
```

Since `Reverse` is more ecient than `SlowReverse`, we prefer it for compiled code. To make note of this in the source code, you can declare `SlowReverse` to be a ghost function.

The fact that `Reverse` and `SlowReverse` are the same, except for their execution time, means that they have the same properties. For example, we can now state the correctness lemma about `ReverseAux` in terms of `Reverse` instead of `SlowReverse`.

```
lemma ReverseAuxCorrect<T>(xs: List<T>, acc: List<T>)

ensures ReverseAux(xs, acc) == Append(Reverse(xs), acc)

{

ReverseCorrect(xs);

ReverseAuxSlowCorrect(xs, acc);

}
```

Similarly, the fact that the reversal of a list does not change its length follows from the connection between `Reverse` and `SlowReverse` and the lemma we already proved about `SlowReverse` preserving length.

```
lemma LengthReverse<T>(xs: List<T>)

ensures Length(Reverse(xs)) == Length(xs)

{

ReverseCorrect(xs);

LengthSlowReverse(xs);

}
```

Exercise 6.19.

Prove `Length(ReverseAux(xs, acc)) == Length(xs) + Length(acc)`, for any lists `xs` and `acc`.

Exercise 6.20.

Write an alternative proof for `LengthReverse` that uses the lemma in Exercise 6.19.

An interesting property to consider is the reversal of the concatenation of two lists. We start with a property that relates `ReverseAux` and `Append`.

```
lemma ReverseAuxAppend<T>(xs: List<T>, ys: List<T>, acc: List<T>)

ensures ReverseAux(Append(xs, ys), acc)

== Append(Reverse(ys), ReverseAux(xs, acc))
```

```
{
match xs
case Nil =>
ReverseAuxCorrect(ys, acc);
case Cons(x, tail) =>
}
```

Unusually, this proof requires a manual step in the `Nil` case and goes through automatically in the `Cons` case.

> **Exercise 6.21.**
>
> Mark lemma `ReverseAuxAppend` with `{:induction false}` and write a calculational proof for the `Cons` case.

> **Exercise 6.22.**
>
> Prove the following lemma:
>
> ```
> lemma ReverseAppend<T>(xs: List<T>, ys: List<T>)
> ensures Reverse(Append(xs, ys))
> == Append(Reverse(ys), Reverse(xs))
> ```

> **Exercise 6.23.**
>
> Prove that `Reverse` is an involution. That is, prove that
>
> ```
> Reverse(Reverse(xs)) == xs
> ```
>
> Hint: Use lemmas from earlier in this section and from Exercise 6.22.

6.7.Lemmas in Expressions

We have used three proof-authoring constructs: inline assertions, proof calculations, and lemma calls. These are statements, and we used them in method bodies and lemma bodies to help discharge proof obligations. There are also proof obligations to establish the well-formedness of functions. So far, all such proof obligations for functions have gone through automatically, but there are plenty of functions where this is not the case.

Although general statements cannot be used in expressions in Dafny, proof-authoring statements can be. If `S` is an inline assertion, proof calculation, or lemma call, and `E` is an expression, then `S E` is also an expression. For example, **`assert`** `0 < x; 100 / x` is an expression.

The value of an expression `S E` is simply `E`. The statement `S` does not affect the value of `E`. In fact, the `S` part is always ghost, so it is erased during compilation and is not present at run time. The proof-authoring statement `S` is used only to help establish any proof obligations that arise in `E` (and downstream of `E` in any enclosing expression). For example, the value of **`assert`** `0 < x; 100 / x` is `100 / x`, but the proof obligation that the divisor `x` is non-zero follows from the assertion **`assert`** `0 <`

`x;`. Of course, the `S` part may itself entail proof obligations (like `0 < x` when the statement is **`assert`** `0 < x`) and these need to be established as well.

The next two examples make use of proof-authoring constructs in expressions.

6.7.0.Intrinsic specification of `ReverseAux`

In Section 6.6, we proved a lemma that `ReverseAux` correctly implements our intent. I argued that this property is a good candidate to be stated intrinsically, that is, as an **`ensures`** clause of the function. To do that, we declare `ReverseAux` as follows:

```
function ReverseAux<T>(xs: List<T>, acc: List<T>): List<T>

ensures ReverseAux(xs, acc) == Append(SlowReverse(xs), acc)

{

match xs

case Nil => acc

case Cons(x, tail) =>

ReverseAux(tail, Cons(x, acc)) // error: cannot prove postcondition

}
```

Alas, the verifier is unable to prove this postcondition automatically from the given function body. More precisely, the verifier is unable to show that `ReverseAux(xs, acc)`, which in the **`ensures`** clause denotes the result value of the function invocation, equals

```
Append(SlowReverse(xs), acc)
```

To help the verifier along, let us insert a proof calculation before the recursive call, showing that this expression equals

```
Append(SlowReverse(xs), acc)

function ReverseAux<T>(xs: List<T>, acc: List<T>): List<T>

ensures ReverseAux(xs, acc) == Append(SlowReverse(xs), acc)

{

match xs

case Nil => acc

case Cons(x, tail) =>

calc {

Append(SlowReverse(xs), acc);

==  // def. SlowReverse

Append(Snoc(SlowReverse(tail), x), acc);
==  { SnocAppend(SlowReverse(tail), x); }

Append(Append(SlowReverse(tail), Cons(x, Nil)), acc);

==  { AppendAssociative(SlowReverse(tail), Cons(x, Nil), acc); }

Append(SlowReverse(tail), Append(Cons(x, Nil), acc));

==  { assert Append(Cons(x, Nil), acc) == Cons(x, acc); }

Append(SlowReverse(tail), Cons(x, acc));

==  // postcondition of ReverseAux

ReverseAux(tail, Cons(x, acc));

}

ReverseAux(tail, Cons(x, acc))

}
```

The proof calculation itself is identical to the one we wrote in the extrinsic lemma `ReverseAuxSlowCorrect` in Section 6.6, except that we do not include the last line of the previous calculation that appealed to the definition of `ReverseAux`.

Note that the proof calculation goes *before* the final expression of `ReverseAux`, even though we're using it to help prove the postcondition of the enclosing function. This is because the syntax is always `S E`, where `S` is a proof-authoring statement and `E` is an expression. If you prefer to see it in the other order, you can use a let expression:

```
var r := ReverseAux(tail, Cons(x, acc));

calc {

// proof calculation goes here

}

r
```

Regardless of which of these formulations you prefer, having the proof calculation in the body of function `ReverseAux` allows us to give the `ReverseAux` correctness property intrinsically. On the downside, it clutters up the body of the function. (Again, this does not affect the run-time behavior of the function, but it makes the function more dicult to read for a human). So, yet another alternative is to refactor the proof calculation into a separate lemma and to call this lemma from the function body of `ReverseAux`:

```
function ReverseAux<T>(xs: List<T>, acc: List<T>): List<T>

ensures ReverseAux(xs, acc) == Append(SlowReverse(xs), acc)

{

match xs

case Nil => acc

case Cons(x, tail) =>

ReverseAuxHelper(xs, acc);
ReverseAux(tail, Cons(x, acc))

}



lemma ReverseAuxHelper<T>(xs: List<T>, acc: List<T>)

requires xs.Cons?

ensures ReverseAux(xs.tail, Cons(xs.head, acc))

== Append(SlowReverse(xs), acc)

decreases xs, acc, 0

{

var x, tail := xs.head, xs.tail;

calc {

Append(SlowReverse(xs), acc);

==  // def. SlowReverse

Append(Snoc(SlowReverse(tail), x), acc);

==  { SnocAppend(SlowReverse(tail), x); }

Append(Append(SlowReverse(tail), Cons(x, Nil)), acc);

==  { AppendAssociative(SlowReverse(tail), Cons(x, Nil), acc); }

Append(SlowReverse(tail), Append(Cons(x, Nil), acc));

==  { assert Append(Cons(x, Nil), acc) == Cons(x, acc); }

Append(SlowReverse(tail), Cons(x, acc));

==  // postcondition of ReverseAux

ReverseAux(tail, Cons(x, acc));

}

}
```

Note that this transformation makes the function `ReverseAux` mutually recursive with `ReverseAuxHelper`—the function calls the lemma and the lemma uses the function (in both its postcondition and its body). This means that we need to think about termination. Since `ReverseAux` does not have an explicit **`decreases`** clause, Dafny by default uses the lexicographic tuple `xs, acc` (that is, the parameters of the function). It would do the same for the lemma, but then nothing is decreased when the function calls the lemma. Instead, we manually write a **`decreases`** clause for the lemma. It suces to take any lexicographic triple that starts with `xs, acc`, since Dafny orders the longer tuple strictly below the shorter one (see Section 3.3.3).

Exercise 6.24.

The `acc` component of the **`decreases`** clauses of function `ReverseAux` and lemma `ReverseAuxHelper` is never used. Write explicit **`decreases`** clauses for `ReverseAux` and `ReverseAuxHelper` that prove termination and do not mention acc.

6.7.1.Lemma about `At` and `Reverse`

Here is one more example where the well-formedness of expressions requires a proof. We consider a lemma that states that the first-but-`i` element of a list `xs` is the same as the last-but-`i` element of `Reverse(xs)`:

```
lemma AtReverse<T>(xs: List<T>, i: nat)

requires i < Length(xs)

ensures At(xs, i) == At(Reverse(xs), Length(xs) - 1 - i)

// possible precondition violation on previous line
```

As written, the statement of this lemma is not well-defined. The problem is that `At` requires a proper index. The precondition of the lemma tells us that `i` is a proper index into `xs`, but without knowing more about the length of `Reverse(xs)`, it is not clear that `Length(xs) - 1 - i` is a proper index into `Reverse(xs)`. We could add this as another postcondition:

```
lemma AtReverse<T>(xs: List<T>, i: nat)

requires i < Length(xs)

ensures Length(xs) - 1 - i < Length(Reverse(xs))

ensures At(xs, i) == At(Reverse(xs), Length(xs) - 1 - i)
```

akin to the extra postcondition conjunct we added to lemma `AtDropHead` in Section 6.4. Here, however, this extra conjunct only clutters up the statement of the lemma. Instead, to convince Dafny that the call to `At` does satisfy its precondition, we invoke the lemma `LengthReverse` and we do so in the postcondition expression itself!

```
lemma AtReverse<T>(xs: List<T>, i: nat)

requires i < Length(xs)

ensures (LengthReverse(xs);

At(xs, i) == At(Reverse(xs), Length(xs) - 1 - i))
```

(For backward compatibility with specification clauses that end with a semi-colon, Dafny allows an optional semi-colon at the end of specification clauses. An unfortunate consequence of this support is that we need to wrap parentheses around a specification clause that calls a lemma, like this postcondition.)

In the body of this lemma, we would like to use a proof calculation that starts with `At(Reverse(xs), Length(xs) - 1 - i)`. Again, this bites us with a complaint that we may be passing an improper index to `At`. Our new call to `LengthReverse` in the postcondition gives information about `Length(Reverse(xs))`, which is used in the rest of the postcondition, but this does not make the information available in the body. Instead, we need to invoke the lemma in the body as well.

Soon, it will also be convenient to have names for the head and tail of `xs`, so we introduce local variables to hold these. As I remarked with the introduction of function `At` in Section 6.4, note that the precondition `i < Length(xs)` implies `xs.Cons?`. Okay, so we start the body of `AtReverse` like this:

```
{
```

```
var x, tail := xs.head, xs.tail;

LengthReverse(xs);

calc {

At(Reverse(xs), Length(xs) - 1 - i);

==  // def. Reverse

At(ReverseAux(xs, Nil), Length(xs) - 1 - i);

==  // def. ReverseAux

At(ReverseAux(tail, Cons(x, Nil)), Length(xs) - 1 - i);

==  { ReverseAuxSlowCorrect(tail, Cons(x, Nil)); }

At(Append(SlowReverse(tail), Cons(x, Nil)), Length(xs) - 1 - i);

==  { ReverseCorrect(tail); }

At(Append(Reverse(tail), Cons(x, Nil)), Length(xs) - 1 - i);
```

These steps in the calculation are fairly straightforward. With all this talk about the well-formedness of expressions, you may be wondering why Dafny does not complain about the precondition of `At` in the second and subsequent lines of the calculation. This is because when Dafny checks the well-formedness of a line in a calculation, it assumes the well-formedness of the previous line. This usually works so smoothly that you probably did not even think about that it had to be checked.

The current line in our proof performs an `At` of an `Append`. The `AtAppend` lemma tells us that such an expression can be rewritten into an `At` of either the first or second list argument to `Append`. However, at this stage in the proof, we cannot tell which part it will be, so we must keep the **`if-then-else`** expression:

```
==  { AtAppend(Reverse(tail), Cons(x, Nil), Length(xs) - 1 - i); }

if Length(xs) - 1 - i < Length(Reverse(tail)) then

At(Reverse(tail), Length(xs) - 1 - i)

else

At(Cons(x, Nil), Length(xs) - 1 - i-Length(Reverse(tail)));
```

Let's not be discouraged by the length of this expression. Instead, let us simplify parts of it to make it easier to understand:

```
==  { LengthReverse(tail); }

if Length(xs) - 1 - i < Length(tail) then

At(Reverse(tail), Length(xs) - 1 - i)

else

At(Cons(x, Nil), Length(xs) - 1 - i - Length(tail));

==  // arithmetic, using Length(xs) == Length(tail) + 1 and 0 <= i

if 0 < i then
```

```
At(Reverse(tail), Length(tail) - 1 - (i - 1))

else

At(Cons(x, Nil), 0);
```

Alright, now things would get messier if we just continued carrying the **`if`** forward in the calculation. Instead, let us close the calculation and break into two cases, one of which is proved automatically by the verifier:

```
}

if 0 < i {
```

Guarded by this condition, we would like to continue our calculation from the line `At(Reverse(tail), Length(tail) - 1 - (i - 1))`. A new calculation will not remember the well-formedness of steps of some other calculation, so we need to invoke the `LengthReverse` lemma once more, this time on `tail`.

```
LengthReverse(tail);

calc {

At(Reverse(tail), Length(tail) - 1 - (i - 1));
```

Well, the good news is that we are almost done. This expression has the form of the lemma we are trying to prove, so we invoke the induction hypothesis, and that just about wraps ’er up.

```
==  { AtReverse(tail, i - 1); }

At(tail, i - 1);

==  // def. At

At(xs, i);

}

}

}
```

In stating and proving this lemma, we had to make several appeals to lemma `LengthReverse`. We could have avoided the need for these lemma calls if we had instead written the `LengthReverse` property as a postcondition of function `Reverse`. Writing such intrinsic specifications is tempting, and sometimes they are a good idea. They do simplify our proof task by automatically introducing more facts for the verifier. However, as I have mentioned, in more complex situations, these additional facts may just overwhelm the verifier. So, use intrinsic specifications sparingly.

Exercise 6.25.

In Section 6.4, we came across an initial problem with stating lemma `AtDropHead`, because the expression in the proof goal was not well-defined by itself. State a lemma

```
lemma DropLessThanEverything<T>(xs: List<T>, i: nat)

requires i < Length(xs)

ensures Drop(xs, i).Cons?
```

and use this lemma in the proof goal of the original attempt at writing lemma `AtDropHead`. Then, prove both `DropLessThanEverything` and your new `AtDropHead`.

6.8.Eliding Type Arguments

In this section, we have seen numerous examples of functions and lemmas about lists. In all of these examples, the list element type was abstract. That is, our functions and lemmas were parameterized by a type, which in this chapter I happened to always name `T`. Since the specific type of this payload is irrelevant, you may have the feeling that type signatures like

```
function ReverseAux<T>(xs: List<T>, acc: List<T>): List<T>
```

are overly verbose or cluttered. When you don't have any need for listing the type `T` by itself, Dafny has an elision rule that saves you from mentioning `T` at all. I will use two forms of this elision rule in other parts of the book. The elision rule appears to be unique to Dafny, so I will explain those forms here.

Recall that, in Dafny, a parameterized type like `List` must always be instantiated with some type argument; for example, `List<int>`, or `List<T>` where `T` is type parameter. In the type signature of a function, method, or lemma, any parameterized type that is mentioned without any type arguments (that is, without the angle brackets where such type arguments would be given) gets filled in automatically with the list of type parameters of the enclosing function, method, or lemma. This form of the elision rule means that the type signature of function `ReverseAux` above can be given as

```
function ReverseAux<T>(xs: List, acc: List): List
```

Also, the type signature of function `Length` (Section 6.1) can be given as

```
function Length<T>(xs: List): nat
```

and the type signature of function `At` (Section 6.4) can be given as

```
function At<T>(xs: List, i: nat): T
```

The other form of the elision rule is that, if the function, method, or lemma's type parameter is not mentioned anywhere else, then it, too, can be elided. So, the type signatures of `ReverseAux` and `Length` can be abbreviated further as

```
function ReverseAux(xs: List, acc: List): List
```

and

```
function Length(xs: List): nat
```

However, this second elision rule is of no further help for function `At`, because `At` needs to mention the type parameter as its result type.

Exercise 6.26.

Make use of the elision rules to shorten the signatures of `Append`, `Take`, `AtAppend`, `Find`, `AtDropHead`, and `Reverse`.

6.9.Summary

In this chapter, I introduced the datatype `List`, which is used throughout functional programming. Since operations on `List` are defined as recursive functions, lemmas that state properties of the operations are a good match for inductive proofs.

A property of a function that is declared as a postcondition of the function is called an *intrinsic* specification. It is an integral part of the function and has to be proved as part of showing the well-formedness of the function.

A property of a function that is declared separately from the function, as a lemma, is called an *extrinsic* specification. It is proved by looking at the definition of the function and can easily state properties that involve multiple invocations of the function, such as associativity.

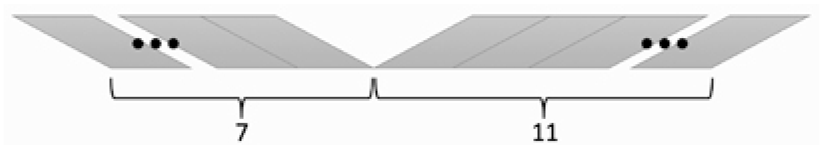
In the chapter, I also showed the use of an accumulator to obtain a more ecient implementation of a function; the use of the type characteristic `(==)`, which says a type supports compiled equality operations (more about that in Section 9.4); various techniques for making sure expressions are well-defined, including the use of proof-authoring statements in expressions; and Dafny's rule for eliding type parameters in the signatures of functions, methods, and lemmas.

Notes

In Dafny, the well-definedness of a function or method's postcondition must follow from the precondition, without regard to the body of the function or method. This is done differently in SPARK [43] and OpenJML [105], where for a function or method with a body, the well-definedness of the postcondition is checked on exit from the body.

Sidebar 6.1

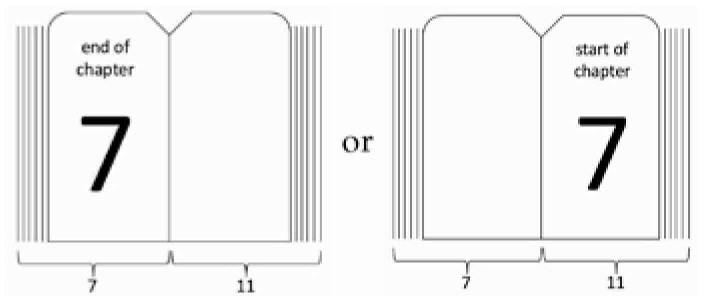
Suppose you have a book with `18` chapters and you open it so that `7` chapters fall to the left and the other `11` chapters fall to the right, as depicted here:



If you have read to the place where you have opened the book, then you have read `7` chapters.

Let's use numbers to name the chapters. What shall we name the chapters visible in this spread? Certainly, it will be convenient to use “`7`” to name either the chapter on the left or the chapter on the right. But which one?

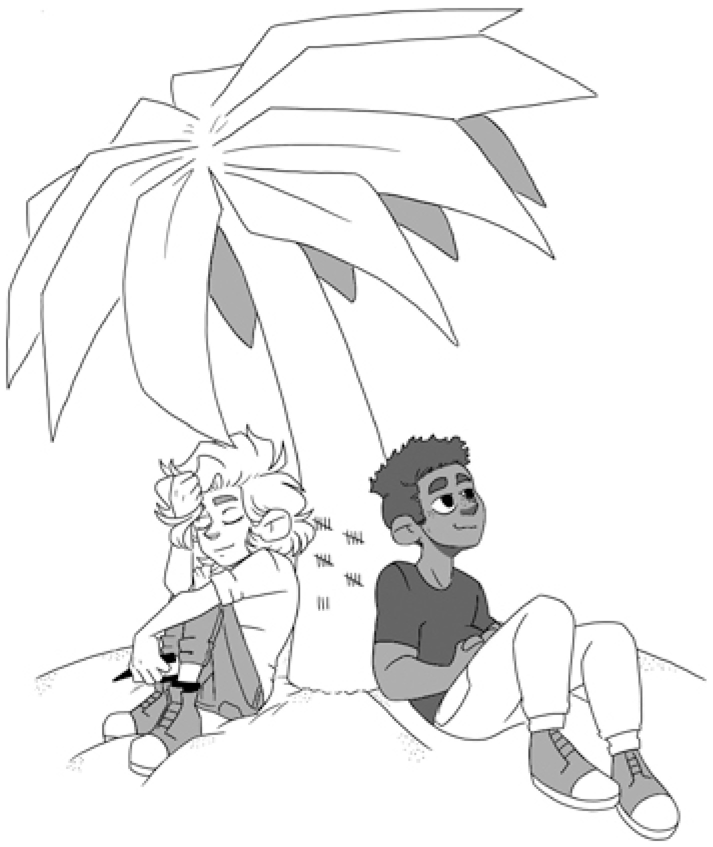
If we print “`7`” in this spread, then the two choices come down to writing



As for me, I like seeing the name of the chapter the moment I start reading it, not the moment I have finished it. Therefore, in this book, when you're about to start chapter `7`, you know there are `7` chapters before it.

Chapter 7

Unary Numbers



In the days of cave dwellers, the need for doing large calculations was not big. Therefore, some lines drawn on a wall or some pine cones collected in a pile could address the demand for recording numbers. For even slightly more sophisticated applications, we find problems with such unary numbering systems. With our improved state today, we would see a problem not only in the diculty to distinguish recording the number zero from not having recorded anything in particular, but also in the inecient ways of doing arithmetic on such numbers. Nevertheless, in this chapter I will take us back to the primitive unary numbers to define various functions and prove some lemmas.

7.0.Basic Definitions

The unary numbers are defined inductively:

```
datatype Unary = Zero | Suc(pred: Unary)
```

As you can see, a unary number is either zero or the successor of some other unary number. When we define a new datatype, we don't always have the luxury to link it with another equivalent representation. Instead, we often have to prove that what we have defined has the properties we want. But in the case of the unary numbers, we can directly give conversion functions to and from the natural numbers:

```
function UnaryToNat(x: Unary): nat {

match x

case Zero => 0

case Suc(x’) => 1 + UnaryToNat(x’)

}



function NatToUnary(n: nat): Unary {

if n == 0 then Zero else Suc(NatToUnary(n-1))

}
```

From these definitions, it is easy to prove that the unary numbers and natural numbers are in one-to-one correspondence with each other. That is, we prove that the conversion function in each direction is injective.

```
lemma NatUnaryCorrespondence(n: nat, x: Unary)

ensures UnaryToNat(NatToUnary(n)) == n

ensures NatToUnary(UnaryToNat(x)) == x

{

}
```

> **Exercise 7.0.**
>
> Nonsensically change the bodies of `UnaryToNat` and `NatToUnary` in a way that breaks injectivity, that is, that renders lemma `NatUnaryCorrespondence` false.

In this chapter, I will use these functions to prove that the operations we define on unary numbers do indeed correspond to the analogous operations on the natural numbers, for example, that `Add` corresponds to `+`. These properties will be given in lemmas with names of the form `…Correct`. The whole point of this chapter is to get to work with `Unary` rather than Dafny's built-in **`nat`**, so let's not use these conversion functions or the `NatUnaryCorrespondence` lemma for anything else.

7.1.Comparisons

Our first operation on unary numbers is a comparison.

```
predicate Less(x: Unary, y: Unary) {

y != Zero && (x.Suc? ==> Less(x.pred, y.pred))

}
```

This definition says that `x` is (strictly) less than `y` exactly when `y` is non-zero and, furthermore, if `x` is also non-zero, then the predecessor of `x` is less than the predecessor of `y`. As mentioned in Section 4.2, of course you could also define `Less` using a **`match`** expression with two cases. To my taste, I find the definition I gave here to be more straightforward.

Exercise 7.1.

Using a **`match`** expression, define a predicate `Less'`. Prove that it is equivalent to `Less` above.

Exercise 7.2.

Define `Less` without a **`match`**, without using the discriminator `Suc?`, and without using the implication operator, ==>.

To prove to ourselves that we have defined the correct ordering, we write the following lemma, which is proved automatically:

```
lemma LessCorrect(x: Unary, y: Unary)

ensures Less(x, y) <==> UnaryToNat(x) < UnaryToNat(y)

{

}
```

Note that the lemma postcondition uses the equivalence operator on booleans ("if and only if", whose binding power is lower than that of any other binary operator). Alternatively, we could have used the equality operator of booleans:

```
ensures Less(x, y) == (UnaryToNat(x) < UnaryToNat(y))
```

However, unlike ==, operator <==> has lower operator precedence than <, so we save a pair of parentheses in the first formulation.

Next, we show that `Less` has the property of being *transitive*. Here is one way to state and prove the property:

```
lemma LessTransitive(x: Unary, y: Unary, z: Unary)

requires Less(x, y) && Less(y, z)

ensures Less(x, z)

{

}
```

Often when a lemma has an antecedent, like `Less(x, y) && Less(y, z)` in this case, it is better to write the antecedent in a **`requires`** clause. This way, if a client fails to establish the precondition when calling the lemma, the call gets flagged with an error. However, for the transitivity property, we may want to allow a client to use the property in a different direction, for example to establish `!Less(y, z)` from `Less(x, y)` and `!Less(x, z)`. Therefore, it is better to state this lemma without a precondition and instead with an implication in the postcondition.

```
lemma LessTransitive(x: Unary, y: Unary, z: Unary)

ensures Less(x, y) && Less(y, z) ==> Less(x, z)
```

Sidebar 7.0

For many operators in Dafny, you can tell its relative operator precedence from its width: a wider operator has lower precedence. The 4-characters-wide operator <==> has the lowest precedence. The 3-characters-wide operators = => and <== bind more tightly, and the 2-characters-wide operators `&&` and `||` bind more tightly yet. Comparison operators bind even more tightly, though their widths are both 1 and 2 characters: ==, `!=`, <=, <, >, and >=. Single-character operators bind tighter than these, first the additive operators `+` and `-`, then the multiplicative operators `*`, `/`, and `%`, and finally, most strongly, the bitwise operators `&`, `|`, and .

Perhaps surprisingly, this version of the lemma is not proved automatically. It is not necessary to understand why automation fails. Instead, let us just write the proof, which will illustrate good learning points. We are proving `Less(x, z)` under the assumption of

```
Less(x, y) && Less(y, z)
```

To introduce this assumption in our proof, we use a conditional statement:

```
{

if Less(x, y) && Less(y, z) {
```

and write the proof in the then branch. As usual in programming, we get to assume the stated condition to hold upon entering the then branch. In the else branch, there is nothing for us to prove, since the lemma holds trivially if `Less(x, y) && Less(y, z)` does not hold.

To prove `Less(x, z)` inside the then branch, we note that the definition of `Less(x, z)` consists of two conjuncts. So, to prove `Less(x, z)`, we'll have to prove both of those conjuncts. The first conjunct is `z != Zero`, which follows directly from the same conjunct in the definition of `Less(y, z)` . The other conjunct is an implication, so we continue the proof with yet another **`if`** statement:

```
if x.Suc? {
```

By the definition of `Less`, the outer `if` guard implies that `y` and `z` are non-zero, and the inner `if` guard gives us the additional assumption that `x` is non-zero. This means we can call the lemma recursively on the predecessors of `x`, `y`, and `z`. In other words, we invoke the induction hypothesis on these predecessors:

```
LessTransitive(x.pred, y.pred, z.pred);

}

}

}
```

Evidently, Dafny can fill in the remaining proof glue, so this completes the proof.

Exercise 7.3.

State and prove a lemma that `Less` is trichotomous, that is, that *exactly one* of the following holds: `Less(x, y), x == y, Less(y, x)`. (Remember here and elsewhere in this chapter, do not let your proof depend on the conversion of `Unary` to **nat**.)

Exercise 7.4.

Define a predicate `Below(x, y)` as `Less(x, y) || x == y`. State and prove that `Below` is a *total order*, that is, that it is reflexive, transitive, antisymmetric (`Below(x, y) && Below(y, x)` implies `x == y`), and total (`Below(x, y) || Below(y, x)`).

7.2.Addition and Subtraction

The addition of two unary numbers is defined by a recursive function `Add`:

```
function Add(x: Unary, y: Unary): Unary {

match y

case Zero => x

case Suc(y’) => Suc(Add(x, y’))

}
```

This function corresponds to addition on natural numbers, as the following lemma shows.

```
lemma AddCorrect(x: Unary, y: Unary)

ensures UnaryToNat(Add(x, y)) == UnaryToNat(x) + UnaryToNat(y)

{

}
```

The definition of `Add(x, y)` proceeds by considering different cases for `y`. This immediately gives us the property `Suc(Add(x, y)) == Add(x, Suc(y))`. In other words, `Suc` distributes over the second argument of `Add`. `Suc` also distributes over the first argument of `Add`, but this property requires an inductive proof:

```
lemma SucAdd(x: Unary, y: Unary)

ensures Suc(Add(x, y)) == Add(Suc(x), y)

{

}
```

We could instead have chosen to define `Add(x, y)` by considering different cases for `x`. We would then get the distribution of `Suc` over the first argument of `Add` for free, and we would need to give an inductive proof for the property that `Suc` distributes over the first argument to `Add`.

The definition of `Add` immediately gives us that `Zero` is a right unit element of `Add` (that is, `Add(x, Zero) == x` for any `x`). Here is a proof that `Zero` is also a left unit element:

```
lemma AddZero(x: Unary)

ensures Add(Zero, x) == x

{

}
```

Exercise 7.5.

Turn off automatic induction and write proofs for (a) `AddCorrect`, (b) `SucAdd`, and `(c) AddZero`.

Exercise 7.6.

Prove that `Add` is associative:

```
Add(Add(x, y), z) == Add(x, Add(y, z))
```

Exercise 7.7.

Prove that `Add` is commutative: `Add(x, y) == Add(y, x)`.

Exercise 7.8.

Prove the following lemma, which states a combination of associativity and commutativity:

```
lemma AddCommAssoc(x: Unary, y: Unary, z: Unary)
ensures Add(Add(x, y), z) == Add(Add(x, z), y)
```

We also define a function for subtracting unary numbers. Since we do not have any negative unary numbers, we define `Sub(x, y)` only when `x` is at least as large as `y`.

```
function Sub(x: Unary, y: Unary): Unary

requires !Less(x, y)

{

match y

case Zero => x

case Suc(y’) => Sub(x.pred, y’)

}



lemma SubCorrect(x: Unary, y: Unary)

requires !Less(x, y)

ensures UnaryToNat(Sub(x, y)) == UnaryToNat(x) - UnaryToNat(y)

{

}
```

Exercise 7.9.

Write a suitable precondition for the following lemma. Then prove the lemma.

```
lemma AddSub(x: Unary, y: Unary)
// requires ?
ensures Add(Sub(x, y), y) == x
```

7.3.Multiplication

For unary numbers, multiplication is defined by repeated addition.

```
function Mul(x: Unary, y: Unary): Unary {
match x
case Zero => Zero
case Suc(x’) => Add(Mul(x’, y), y)
}
```

Analogous to what we did for the other operation, we prove that `Mul` corresponds to * on the natural numbers. This time, the proof requires a little assistance from us.

```
lemma MulCorrect(x: Unary, y: Unary)
ensures UnaryToNat(Mul(x, y)) == UnaryToNat(x) * UnaryToNat(y)
{
match x
case Zero =>
case Suc(x’) =>
calc {
UnaryToNat(Mul(x, y));
==  // def. Mul
UnaryToNat(Add(Mul(x’, y), y));
==  { AddCorrect(Mul(x’, y), y); }
UnaryToNat(Mul(x’, y)) + UnaryToNat(y);
// Dafny can take it from here on
}
}
```

7.4.Division and Modulus

Next up, as we define division and modulus (remainder), we encounter several teaching points. For unary numbers, division and modulus are done by repeated subtraction. Since the two operations perform similar computations, we will package them both in one function. The function we define will therefore return a *pair*. For any types `A` and `B`, the type representing pairs with one `A` value and one `B` value is denoted `(A, B)` in Dafny.

```
function DivMod(x: Unary, y: Unary): (Unary, Unary)
```

Our convention will be that the first component of the returned pair is the division of `x` by `y` and the second component is their modulus. The notation for constructing a pair of `d` and `m` is simply `(d, m)`. The destructors for the two respective components are called `0` and `1`, so if `r` is the pair returned by `DivMod(x, y)`, then `r.0` denotes the division of `x` by `y` and `r.1` denotes their modulus.

We define our function only for non-zero divisors:

```
requires y != Zero
```

A straightforward definition of `DivMod` would seem to be the following:

```
{

if Less(x, y) then

(Zero, x)

else

var r := DivMod(Sub(x, y), y); // cannot prove termination

(Suc(r.0), r.1)

}
```

However, Dafny is not automatically able to prove that the recursive call to `DivMod` terminates. Let's look at two different ways to help the verifier along to prove termination.

7.4.0.Termination via natural numbers

To prove termination, we need to use a termination metric that we can show to decrease with every recursive call. We have that `DivMod(x, y)` calls `DivMod(Sub(x, y), y)`, so our only choice is to show that something about the first argument decreases. Interpreted as natural numbers, we have that `UnaryToNat(x) - UnaryToNat(y)` is less than `UnaryToNat(x)`, since `y` is known to be non-zero. To say that we want to use the natural number corresponding to `x` as our termination metric, we declare our own **`decreases`** clause:

```
decreases UnaryToNat(x)
```

To prove termination of the recursive call, we now need to prove

```
UnaryToNat(x)  UnaryToNat(Sub(x, y))
```

Since `UnaryToNat` results in a **`nat`**, this condition is

```
UnaryToNat(Sub(x, y)) < UnaryToNat(x)
```

Alas, this is not done automatically, either.

To help the verifier some more, we can invoke the `SubCorrect` lemma, which says the left-hand side of this inequality is equal to

```
UnaryToNat(x) - UnaryToNat(y)
```

We insert this lemma call just before the let expression:

```
SubCorrect(x, y);
var r := DivMod(Sub(x, y), y);
```

The lemma call fits nicely on this line by itself in the source text. If we wanted to, we could move it even closer to the place where it is needed:

```
var r := (SubCorrect(x, y); DivMod(Sub(x, y), y));
```

Note the need for the extra parentheses in this case, so that the parser does not just take `SubCorrect(x, y)` as the right-hand side of the let binding.

In fact, we can move the lemma call even closer yet. For example:

```
var r := DivMod(SubCorrect(x, y); Sub(x, y), y);
```

The important thing is to give the verifier the lemma's information before it checks the precondition and termination of the call to `DivMod`, which happens after the evaluation of the call's parameters. It is generally desirable to place a lemma close to where it is needed, but there is no reason to go overboard, so I like the placement on a line by itself before the let.

7.4.1.Termination via structural inclusion

Since we happened to have the correspondence with natural numbers in this case, it was easy enough to use `UnaryToNat` and `SubCorrect` in the termination proof. But this is not the only way we can do it. Another way is to argue more directly that the datatype value returned by `Sub(x, y)` is structurally smaller than `x`. Stated differently, if you think of unary numbers as being scribbled as lines on the wall of a cave, then `Sub(x, y)` is a shorter sequence of lines than `x`. Dafny knows that the datatype parameters to a constructor are structurally smaller than what the constructor returns. To get it to understand that this property also holds transitively, we need to prove a lemma:

```
lemma SubStructurallySmaller(x: Unary, y: Unary)

requires !Less(x, y) && y != Zero

ensures Sub(x, y) < x

{

}
```

In Dafny, the < operator is overloaded to work on various argument types. In the post-condition of this lemma, it works on datatype values and thus stands for structural inclusion. This lemma is proved automatically by induction.

All we have to do now is call the lemma before the let expression in `DivMod`. Here is the full definition of `DivMod`:

```
function DivMod(x: Unary, y: Unary): (Unary, Unary)

requires y != Zero

{

if Less(x, y) then

(Zero, x)

else

SubStructurallySmaller(x, y);

var r := DivMod(Sub(x, y), y);

(Suc(r.0), r.1)

}
```

In Section 6.2, I discussed intrinsic versus extrinsic ways of stating and proving properties of functions. Termination of functions (and of methods, too) must always be done intrinsically in Dafny. That is, you are not allowed to declare a function, ignore termination, and try to prove termination later in a separate lemma—termination must be proved in the function declaration itself (possibly with the help of additional lemmas, as illustrated by `SubCorrect` and `SubStructurallySmaller` above).

7.4.2.Let expressions with patterns

While we are tweaking the definition of `DivMod`, this gives me the opportunity to introduce another convenient language feature. The left-hand side of a let expression can be a pattern, much like the cases of a **`match`**. For example, if `F(e)` is a function that is known to return a non-zero unary number, then we can use the pattern `Suc(w)` as the left-hand side of the let. This will bind `w` to the parameter of the `Suc` value returned by `F(e)`. In other words, it will compute the value of `F(e)`, then “shave off” the outermost `Suc` of that value, and then bind what remains to `w`. For this to work, it must be known that `F(e)` does indeed return a `Suc`, not `Zero`. So, after the following two let bindings

```
var r := F(e);
var Suc(w) := F(e);
```

we have `r == Suc(w)` and `r.pred == w`.

A pair is a built-in datatype with special syntax. In particular, its sole constructor is written with just parentheses. Using a pattern in the let expression, we can therefore write the last two lines of `DivMod` as follows:

```
var (d, m) := DivMod(Sub(x, y), y);
(Suc(d), m)
```

Finally, note the subtle difference between this let, which binds `d` and `m` to the different components of the one pair returned by `DivMod`, and an incorrect expression

```
var d, m := DivMod(Sub(x, y), y); // error
```

where there are *two* left-hand sides (`d` and `m`) and only one right-hand side (the call to `DivMod`).

7.4.3.Correctness of `DivMod`

I'll wrap up this chapter with a proof that `DivMod` is really correct. But instead of proving correctness via the natural numbers, let us prove that division and modulus satisfy the properties we expect. These properties are as follows, for any natural number `a` and positive integer `b`:

```
(a / b) * b + (a % b) == a
(a % b) < b
```

(All of the parentheses here are unnecessary. I added them to emphasize where the division and modulus operations are.) Stated as a lemma about the unary numbers, we have:

```
lemma DivModCorrect(x: Unary, y: Unary)

requires y != Zero

ensures var (d, m) := DivMod(x, y);

Add(Mul(d, y), m) == x &&

Less(m, y)
```

Note how it was convenient to use a let expression in the postcondition of this lemma, so that we can give names to what is returned by `DivMod`.

We start the proof by introducing local variables (with the same names as the bound variables inside the postcondition) to stand for the unary numbers we are going to prove the properties of.

```
{

var (d, m) := DivMod(x, y);
```

Next, we will follow the two cases in the definition of `DivMod`. Although it is not necessary for the proof, we can use an **`assert`** statement to remind ourselves of what `DivMod` returns in this case.

```
if Less(x, y) {

assert d == Zero && m == x; // since (d, m) == DivMod(x, y)
```

Here, we immediately have `Less(m, y)`. The rest of this case is a simple calculation:

```
calc {

Add(Mul(d, y), m) == x;

==  // d, m

Add(Mul(Zero, y), x) == x;

==  // def. Mul

Add(Zero, x) == x;

==  { AddZero(x); }

true;

}
```

For the other case, we start by introducing names for what is returned by the recursive call to `DivMod`, and we use an **`assert`** to remind ourselves of what `DivMod(x, y)` returns in this case.

```
} else {

var (d’, m’) := DivMod(Sub(x, y), y);

assert d == Suc(d’) && m == m’; // since (d, m) == DivMod(x, y)
```

There are several ways to proceed. We usually start a proof calculation with the more complicated side of the proof goal, because this gives us more structure that may help us in figuring out the proof steps. Let us do it differently in this case and start with a formula at the opposite end of the spectrum of structurally rich formulas, the literal **`true`**.

```
calc {

true;
```

This seems rather masochistic, but after one more step we will see where this is going: the next step is to introduce the properties gained by the induction hypothesis.

```
==>  { SubStructurallySmaller(x, y);
DivModCorrect(Sub(x, y), y); }
Add(Mul(d’, y), m) == Sub(x, y) && Less(m, y);
```

Note that to prove termination of the recursive call to `DivModCorrect`, we first invoke the `SubStructurallySmaller` lemma in the calculation hint. You can also tell from the calculation connective in the left margin that we are going for a proof of the form **`true`** `==>` *`proof goal`*. The rest of the proof is straightforward, using the lemmas proved in Exercises 7.9 and 7.8:

```
==>  // add y to both sides

Add(Add(Mul(d’, y), m), y) == Add(Sub(x, y), y) &&

Less(m, y);

==  { AddSub(x, y); }

Add(Add(Mul(d’, y), m), y) == x && Less(m, y);

==  { AddCommAssoc(Mul(d’, y), m, y); }

Add(Add(Mul(d’, y), y), m) == x && Less(m, y);

==  // def. Mul, d

Add(Mul(d, y), m) == x && Less(m, y);

}

}

}
```

7.5.Summary

This chapter has mostly been a playground to practice proving properties of functions on a simple datatype. But we also came across several important points:

For many recursive functions, there's a choice in how to decompose the parameters for the recursive call and how to build the result from the recursive call. This choice will make some properties of the function trivial to prove, while similar properties may require a proof by induction. For example, if you define `Add` by decomposing its first parameter, the definition trivially shows that `Zero` is a left unit of `Add`, whereas you'll need an inductive proof (using a lemma) to show that `Zero` is a right unit. If you instead define `Add` by decomposing its second parameter, you'll need induction for the left-unit property, whereas the right-unit property follows trivially from the definition. It's a good idea to be aware of the trade-offs with this asymmetry. We have also seen this asymmetry with `Append` in Section 6.2.0, and we'll see related issues in Sections 12.2 and 12.3.

In this chapter, we also learned more about termination. While many termination proofs are automatic, even to the degree that specifying the termination metric is automatic, we sometimes have to write termination proofs manually. In Dafny, the termination metric for a function is usually (the lexicographic tuple consisting of) the function's list of parameters, but by manually supplying a **`decreases`** clause, you can make the termination metric be any list of values computed from the parameters. It is common in verification tools (including Dafny) to require the termination of a function to be proved intrinsically; that is, termination must be established as part of the function definition itself.

Functions defined on datatypes often use structural inclusion to prove termination. While direct structural inclusion is automatically considered by the verifier, you may need to prove a lemma to obtain transitive structural inclusion.

To prove a property like `P ==> Q`, you often write a proof of the form

```
if P {

// prove Q here

}
```

Notes

Our `UnaryToNat` and `NatToUnary` functions demonstrate that there is a one-to-one correspondence between the values of type `Unary` and the natural numbers. The idea to represent the natural numbers by an inductive datatype like `Unary`, with constructors `Zero` and `Suc`, follows directly from an *axiomatization* of the natural numbers that was developed by Giuseppe Peano. Therefore, the set of unary numbers—more precisely, the `Zero`/`Suc` representation of the natural numbers—is often known as Peano numbers.

Because the Peano numbers are described by such a simple inductive datatype, they are popular as a subject of formal proofs.

Chapter 8

Sorting

An often occurring theme in computer science is that of sorting. This chapter is concerned with the correctness of two sorting algorithms that operate on lists. The datatype `List` and various functions on lists, such as `Length`, are those from Chapter 6.

In this chapter, I will start to separate the main algorithmic functions, which we eventually want to compile, from the supporting functions used only in specifications. These specification-only functions will be declared as **`ghost`**.

8.0.Specification

We start by considering the specification of sorting. Obviously, we want the output of our sorting routines to be ordered.

8.0.0.Sorted

For a list of integers to be sorted means that the elements are in ascending numerical order. When we work with lists, this property is most easily expressed by saying that any two consecutive elements of the list are ordered.

```
ghost predicate Ordered(xs: List<int>) {

match xs

case Nil => true

case Cons(x, Nil) => true

case Cons(x, Cons(y, _)) => x <= y && Ordered(xs.tail)

}
```

There are three cases. The first two apply to the empty list and to singleton lists, respectively, which are always sorted. The use of the constructor `Nil` inside the `Cons` pattern in the second case is a nested pattern, which matches when `xs` is a `Cons` whose tail is `Nil`. If the case applies, then `x` is bound to the head of `xs` for use in the body of the **`case`** construct. The third case also uses a nested pattern. It matches when both `xs` and the tail of `xs` are `Cons` values, and if the case applies, then `x` and `y` are bound to `xs.head` and `xs.tail.head`, respectively. As usual in patterns, the underscore in the position of `xs.tail.tail` says that we don't care to introduce any bound variable to stand for that subexpression. The body of the third case evaluates to **`true`** when the first two elements of `xs` are ordered and the elements of `xs.tail` are pairwise ordered.

If you feel that a definition, like `Ordered` here, is complicated or non-obvious, it is a good idea to verify that it has some properties that we expect it to have. For a list that satisfies `Ordered`, we expect that any element earlier in the list is below any element later in the list. We can state this precisely using the list function `At` (from Section 6.4), which retrieves a particular element of the list. That leads us to the following lemma:

```
lemma AllOrdered(xs: List<int>, i: nat, j: nat)

requires Ordered(xs) && i <= j < Length(xs)

ensures At(xs, i) <= At(xs, j)
```

The proof of this lemma does not go through automatically, so let's help the verifier along by writing the proof. By the definition of `At`, if `i` is non-zero (which, by `AllOrdered`'s precondition, implies that `j` is non-zero, too), then our proof goal is equivalent to

```
At(xs.tail, i - 1) <= At(xs.tail, j - 1)
```

so we start our proof of `AllOrdered` as follows:

```
{

if i != 0 {

AllOrdered(xs.tail, i - 1, j - 1);
```

If `i` is `0`, then we need to argue that going down the list until we find the element at position `j` gives us an element that is no smaller than the head of `xs`. This is trivial if `j` is also `0`:

```
} else if i == j {
```

If `j` is non-zero, then the property follows from

```
At(xs, 0) <= At(xs, 1) <= At(xs, j)
```

We get the condition `At(xs, 0) <= At(xs, 1)` from the fact that the list is ordered. The condition `At(xs, 1) <= At(xs, j)` is equivalent to `At(xs.tail, 0) <= At(xs.tail, j - 1)`, which means we can obtain it from the induction hypothesis by calling the lemma recursively. Our proof of `AllOrdered` thus comes to an end like this:

```
} else {

AllOrdered(xs.tail, 0, j - 1);

}

}
```

We can now feel reasonably confident that we have stated the property `Ordered` correctly.

8.0.1.Same elements

Not every function that outputs a list of ordered elements is a sorting routine. We also want to describe some relation between the input elements and the output elements. For one, we want the output to consist of exactly the *same elements* as in the input, but arranged in a possibly different order. We define this property by saying: for any element `p`, the number of occurrences of `p` in the input equals the number of occurrences of `p` in the output. In order words, this is saying that the *multiset* of elements in the input equals the multiset of elements in the output. A multiset (sometimes known as a *bag*) is like a *set*, but the multiset keeps track of the multiplicity of each element.

Parenthetically, let me mention that this same-elements property of sorting routines is often described by saying: the output is a *permutation* of the input. If we tried to formalize what it means for one list to be a permutation of another, we would likely get ourselves into a complicated mess. So, while “permutation” mentally suggests a possible reordering, it really just means that the elements are the same.

To specify the same-elements property of a sorting routine, we could define a function that counts the number of occurrences of a given value `p` in a list:

```
ghost function Count(xs: List<int>, p: int): nat {

match xs

case Nil => 0

case Cons(x, tail) =>

(if x == p then 1 else 0) + Count(tail, p)

}
```

Our sorting specification would then include a property like

```
Count(xs, p) == Count(MySortingFunction(xs), p)
```

for every `p`.

However, instead of using `Count`, I will do something slightly more general, as I'll describe next.

Exercise 8.0.

Generalize the definition of `Count` beyond integer lists. What type characteristic is needed of the list's element type?

8.0.2.Stability

A property that a sorting algorithm may or may not have is *stability*. This property is of interest when the elements to be sorted have a key that is not the entire element. For example, if you are sorting a list of (student name, exam score) pairs by their exam score, then a stable sort would output any two students with the same exam score in the same order as these students were listed in the input. In other words, a stable sort preserves the order of records with the same key.

Suppose you wanted to output a list of (student name, exam score) pairs in the order of exam scores, and within each group of the same exam score, by the student names in alphabetical order. You can accomplish this by first sorting the list using the name as the key and then, using a stable sort, sorting by exam score.

In this chapter, I only present sorting for lists of integers, so the sorting key of an element is the element itself. In this case, stability does not matter; for example, you cannot distinguish the list `Cons(4, Cons(4, Nil))` from the list `Cons(4, Cons(4, Nil))`. 🤪 Nevertheless, I will define functions that can easily be adapted for lists where keys are computed from elements and where stability may matter. The following function will pick out the elements equal to (or, think: that have the key) `p`:

```
ghost function Project(xs: List<int>, p: int): List<int> {

match xs

case Nil => Nil

case Cons(x, tail) =>

if x == p then Cons(x, Project(tail, p)) else Project(tail, p)

}
```

Using this `Project` function, we can specify stability by the following *stability equation*: for every `p`,

```
Project(xs, p) == Project(MySortingFunction(xs), p)
```

Note that the equality in this equation is on lists, not numbers like in the similar equation with `Count` above. Importantly, note also that stability (defined in this way) implies the same-elements property (cf. [79]). Therefore, even though stability does not make a difference in this chapter, I will phrase the same-elements property like we would have stated stability, namely as the stability equation above.

Exercise 8.1.

For any `p` and any two lists that, after projection to `p`, are equal, state and prove a lemma that says the two lists have the same occurrence count of `p`.

In the rest of this chapter, I define two classic sorting algorithms, Insertion Sort and Merge Sort. We will prove for each algorithm that its output is sorted and that the output has the same elements as the input. Both Insertion Sort and Merge Sort are in fact stable sorts, but we can't tell since I'm only using lists of integers. Nevertheless, as I mentioned above, I will specify the same-elements property using function `Project`.

8.1.Insertion Sort

The idea of Insertion Sort is simple. To sort a nonempty list, sort its tail and then insert its head into the appropriate position. Here is a function that does that:

```
function InsertionSort(xs: List<int>): List<int> {

match xs

case Nil => Nil

case Cons(x, tail) => Insert(x, InsertionSort(tail))

}
```

As for the insertion into a list, if the list has a head that is smaller than the element to be inserted, insert the element into tail. Otherwise, prepend the element to the list.

```
function Insert(y: int, xs: List<int>): List<int> {

match xs

case Nil => Cons(y, Nil)

case Cons(x, tail) =>

if y < x then Cons(y, xs) else Cons(x, Insert(y, tail))

}
```

That's all there is. Let us verify that we wrote it correctly.

We start by proving the ordered property of the output. As is common, we will have one lemma per function in the implementation. These will look something like this:

```
lemma InsertionSortOrdered(xs: List<int>)
ensures Ordered(InsertionSort(xs))

lemma InsertOrdered(y: int, xs: List<int>)
ensures Ordered(Insert(y, xs))
```

The proof of the first calls the second, just like the `InsertionSort` function calls `Insert`:

```
{

match xs

case Nil =>

case Cons(x, tail) =>

InsertOrdered(x, InsertionSort(tail));

}
```

The verifier is happy with this proof. Since `Insert` is a recursive function, we expect the `InsertOrdered` lemma to be recursive as well. And since the recursion is pretty simple, we may also have good hopes that Dafny will verify `InsertOrdered` automatically. If you supply the empty proof body, `{}`, you will find this is not the case. To debug the situation, let's start filling in the proof manually. Following our standard technique of following the structure of the body of the function we are trying to prove something about, we write the following structure of cases in the lemma:

```
{

match xs

case Nil =>

case Cons(x, tail) =>

if y < x { // cannot prove postcondition on this return path

} else {

}

}
```

This reveals that even when `Insert` returns `Cons(y, xs)`, in the then branch of the conditional, the verifier fails to prove the postcondition. But, of course! There is no reason to think that `Cons(y, xs)` is sorted just because `y < xs.head`—the rest of `xs` may contain elements in any order. We have thus discovered that the lemma needs to know more about the given list, `xs`. We state this property, namely `Sorted(xs)`, as a precondition of the lemma. Now, the lemma holds and, furthermore, Dafny verifies it automatically:

```
lemma InsertOrdered(y: int, xs: List<int>)

requires Ordered(xs)

ensures Ordered(Insert(y, xs))

{

}
```

Note that the function `Insert` itself does not need any precondition. Indeed, `Insert` is happy to insert `y` after the prefix of elements of `xs` that are smaller than `y`. The need for the precondition `Sorted(xs)` arises only when we are trying to prove that `Insert` outputs a sorted list.

One thing remains: proving the same-elements property of the sorting functions. To state this property, we need to talk about any value of the list's element type. We can do that by parameterizing our lemmas with another parameter, call it `p`, like I did in the stability equation in Section 8.0.2. Since we state and prove the lemmas for any such `p`, we obtain a proof of the desired same-elements property.

```
lemma InsertionSortSameElements(xs: List<int>, p: int)

ensures Project(xs, p) == Project(InsertionSort(xs), p)

lemma InsertSameElements(y: int, xs: List<int>, p: int)

ensures Project(Cons(y, xs), p) == Project(Insert(y, xs), p)
```

The first lemma says that an element `p` occurs in `InsertionSort(xs)` exactly as much as it does in the input `xs`, and the second lemma says that `p` occurs in `Insert(y, xs)` exactly as much as it does in `Cons(y, xs)`.

To prove the first lemma, we simply call the second, just like function `InsertionSort` calls

```
Insert:

{

match xs

case Nil =>

case Cons(x, tail) =>

InsertSameElements(x, InsertionSort(tail), p);

}
```

The proof of `InsertSameElements` is automatic:

```
{

}
```

8.2.Merge Sort

The Merge Sort algorithm is faster than the Insertion Sort algorithm for large inputs. Asymptotically, for an input of length *n*, Merge Sort runs in time ($n \log n$), whereas Insertion Sort requires time (n^2). The difference in speed comes from the fact that Insertion Sort reduces its list size by just `1` with each recursive call, whereas Merge Sort divides its input in half with each recursive call. In more detail, Merge Sort first splits the given list into two halves, then sorts each of those halves, and finally merges the two sorted halves.

8.2.0.The implementation

To split the input in two parts whose lengths are as equal as possible, it is convenient to know the length of the input. Rather than computing this length with each recursive call, we will compute it at the beginning and then keep track of the lengths of the halves. The bulk of the work of our Merge Sort will therefore be done in an auxiliary function that takes the length as a parameter. Here is the top-level function:

```
function MergeSort(xs: List<int>): List<int> {

MergeSortAux(xs, Length(xs))

}
```

The auxiliary function is declared as follows:

```
function MergeSortAux(xs: List<int>, len: nat): List<int>

requires len == Length(xs)
```

If the list has length `0` or `1`, it is already sorted, so we just return the input:

```
{

if len < 2 then

xs

else
```

Otherwise, we need two more functions, `Split` and `Merge`. Let's write those functions next and then return to `MergeSortAux`.

Function `Split` splits a list into a prefix and a sux. We let the function take a parameter that indicates the desired length of the prefix.

```
function Split(xs: List, n: nat): (List, List)

requires n <= Length(xs)
```

The result type of `Split` is a pair of lists, as indicated by the type `(List, List)`. (See Section 6.8 if you wonder what happened with the type argument to `List`, which is irrelevant to `Split`.) To give all callers of `Split` information about the two lists returned, we write an intrinsic specification in an **`ensures`** clause of the function:

```
ensures var (left, right) := Split(xs, n);

Length(left) == n &&

Length(right) == Length(xs) - n &&

Append(left, right) == xs
```

This postcondition uses a let expression to give names to the two components of the pair that is being returned. The body of `Split` is defined like this:

```
{

if n == 0 then

(Nil, xs)

else

var (l, r) := Split(xs.tail, n - 1);

(Cons(xs.head, l), r)

}
```

Function `Merge(xs, ys)` is like `Append(xs, ys)` (Section 6.2), except that `Append` always takes its next elements from `xs`, provided `xs` is nonempty. In contrast, `Merge` takes the next element from `ys` if it is smaller. To set this up, we match not on the list `xs`, but on the pair of lists `(xs, ys)`:

```
function Merge(xs: List<int>, ys: List<int>): List<int>

{

match (xs, ys)

case (Nil, Nil) => Nil

case (Cons(_, _), Nil) => xs

case (Nil, Cons(_, _)) => ys

case (Cons(x, xtail), Cons(y, ytail)) =>

if x <= y then

Cons(x, Merge(xtail, ys))
else

Cons(y, Merge(xs, ytail))

}
```

Having defined `Split` and `Merge`, we return to `MergeSortAux`. We fill in the **`else`** branch where we had left off with

```
var (left, right) := Split(xs, len / 2);

Merge(MergeSortAux(left, len / 2),

MergeSortAux(right, len - len / 2))

}
```

This expression splits `xs` into the prefix `left` and the sux `right`, whose lengths are `len / 2` and `len - len / 2`, respectively. Two recursive calls to `MergeSortAux` sort these smaller lists and the sorted lists are then merged.

Here, the verifier complains that it cannot prove termination of either of the recursive calls. We have arranged to make these calls with lists of lengths `len / 2` and `len - len / 2`, each of which is indeed smaller than `len`, since the recursive calls take place on the branch where `2 <= len`. But the termination metric for `MergeSortAux` is the default—the lexicographic pair of the function's arguments: `xs, n`. It is not true that `left` is structurally included in `xs`, so the verifier has good reasons to complain about the first recursive call. (For the second call, `right` is actually structurally included in `xs`, but to communicate that to the verifier, we would have to prove a lemma.) Our way out is simple: we manually change the termination metric to `len`, which we do by adding the following clause after the type signature of `MergeSortAux`:

```
decreases len
```

Exercise 8.2.

The integer parameter to `Split` makes it easy to see what is happening. To use this `Split` function, we must start by computing the length of the original list (as we did when calling `MergeSortAux` from `MergeSort`). However, it is not necessary to use such an integer, because we can instead use the elements of a list to do the counting. Implement a function with the following specification:

```
function Split’(xs: List, nn: List): (List, List)

requires Length(nn) <= Length(xs)

ensures var (left, right) := Split’(xs, nn);

var n := Length(nn) / 2;

Length(left) == n &&

Length(right) == Length(xs) - n &&

Append(left, right) == xs
```

(The last three lines are the same as from the postcondition of `Split`.)

▪

Exercise 8.3.

Write a new function `MergeSort'(xs: List<int>)` that is implemented directly in terms of `Split'` (from Exercise 8.2) and `Merge`. Prove `MergeSort(xs) == Merge- Sort'(xs)`. Hint: Lemma `AppendDecomposition` in Exercise 6.5 will be useful for the proof.

8.2.1.Ordered correctness

We prove the correctness of Merge Sort in two steps, just as for Insertion Sort. We start by showing that the output of `MergeSort` is ordered. Our implementation has four functions (`MergeSort`, `MergeSortAux`, `Split`, and `Merge`), so it is reasonable to expect that we will have four lemmas as well. As it turns out, we will need only three, because we captured the salient properties of `Split` by its intrinsic specification.

The first two lemmas mimic the structure of the corresponding functions:

```
lemma MergeSortOrdered(xs: List<int>)

ensures Ordered(MergeSort(xs))

{

MergeSortAuxOrdered(xs, Length(xs));

}



lemma MergeSortAuxOrdered(xs: List<int>, len: nat)

requires len == Length(xs)

ensures Ordered(MergeSortAux(xs, len))

decreases len

{

if 2 <= len {

var (left, right) := Split(xs, len / 2);

MergeOrdered(MergeSortAux(left, len / 2),

MergeSortAux(right, len - len / 2));

}

}
```

The lemma that shows the output of `Merge` to be ordered holds only if the inputs of `Merge` are already sorted, so the lemma also needs a precondition. The proof is done automatically by Dafny:

```
lemma MergeOrdered(xs: List<int>, ys: List<int>)

requires Ordered(xs) && Ordered(ys)

ensures Ordered(Merge(xs, ys))

{
}
```

Exercise 8.4.

We had to write part of the proof of `MergeSortAuxOrdered` above, but we didn't explicitly write calls to the induction hypothesis. Mark the lemma with `{:induction false}` and manually fill in the necessary calls.

Exercise 8.5.

Mark `MergeOrdered` with `{:induction false}` and write its proof without relying on Dafny's automatic induction.

8.2.2.Same-elements correctness

Our final proof obligation—which, recall, has the form of the stability equation in Section 8.0.2—requires more work. It starts off straightforwardly, following the structure of `MergeSort`, which calls the auxiliary function:

```
lemma MergeSortSameElements(xs: List<int>, p: int)

ensures Project(xs, p) == Project(MergeSort(xs), p)

{

MergeSortAuxSameElements(xs, Length(xs), p);

}
```

```
lemma MergeSortAuxSameElements(xs: List<int>, len: nat, p: int)

requires len == Length(xs)

ensures Project(xs, p) == Project(MergeSortAux(xs, len), p)

decreases len
```

We need to help the verifier along for this proof, so we start off like we usually do, following the structure of the corresponding function `MergeSortAux` and starting a calculation from the more complicated side of the equality we are trying to prove:

```
{

if 2 <= len {

var (left, right) := Split(xs, len / 2);

calc {

Project(MergeSortAux(xs, len), p);

== // def. MergeSortAux

Project(Merge(MergeSortAux(left, len / 2),
MergeSortAux(right, len - len / 2)), p);
```

(The next step is the most dicult in our calculation, so prepare yourself for a longer discourse. The steps after that will be easier again.)

To simplify this `Project(…, p)` expression, we need some property about `Project(Merge(…), p)`. Let us switch our attention to such a lemma. What *can* we say about the `p` elements of a merge? Those elements come from the `p` elements of the two lists being merged. Let's call the two sorted lists being merged `xs` and `ys` and let's think about what `Merge(xs, ys)` does. The merge proceeds by repeatedly picking an element from either `xs` or `ys` and adding it to the result. It does this for all elements that are less than `p` before it does it for any element that is equal to or greater than `p`. Moreover, it does this for all elements that are equal to `p` before it does it for any element that is greater than `p`. Therefore, the effect of projecting the result of the merge to the elements that are equal to `p` is the same as if we ignore the non-`p` elements of the two lists. In other words, we expect `Project` to distribute over `Merge`:

```
Project(Merge(xs, ys), p) ==
Merge(Project(xs, p), Project(ys, p))
```

Furthermore, merging two lists of `p`'s is the same as appending them, which leads us to state the following lemma:

```
lemma MergeSameElements(xs: List<int>, ys: List<int>, p: int)

requires Ordered(xs) && Ordered(ys)

ensures Project(Merge(xs, ys), p)

== Append(Project(xs, p), Project(ys, p))

{

}
```

The proof of this lemma goes through automatically.

Next, we would like to apply this lemma in the calculation in lemma `MergeSortAuxSameElements` where we left off. That is, we would like to do the following step:

```
Project(Merge(MergeSortAux(left, len / 2),

MergeSortAux(right, len - len / 2)),

p);

==  { MergeSameElements(

MergeSortAux(left, len / 2),

MergeSortAux(right, len - len / 2),

p);

}

Append(

Project(MergeSortAux(left, len / 2), p),

Project(MergeSortAux(right, len - len / 2), p));
```

But the verifier complains, saying we can call `MergeSameElements` only on sorted lists. We can obtain this needed information by calling the lemma `MergeSortAuxOrdered`, which we proved above in Section 8.2.1. We make the two calls to this lemma inside the hint, before calling `MergeSameElements` in the hint. So, our long-awaited next step in the `MergeSortAuxSameElements` calculation is

```
Project(Merge(MergeSortAux(left, len / 2),

MergeSortAux(right, len - len / 2)),

p);

==  { MergeSortAuxOrdered(left, len / 2);

MergeSortAuxOrdered(right, len - len / 2);

MergeSameElements(

MergeSortAux(left, len / 2),

MergeSortAux(right, len - len / 2),

p);

}

Append(

Project(MergeSortAux(left, len / 2), p),

Project(MergeSortAux(right, len - len / 2), p));
```

Here, we have two subexpressions that project the result of `MergeSortAux`, so we can apply the induction hypothesis:

```
==  { MergeSortAuxSameElements(left, len / 2, p);

MergeSortAuxSameElements(right, len - len / 2, p); }

Append(Project(left, p), Project(right, p));
```

First projecting `left` and `right` on `p` and then appending them gives the same result as first appending `left` and `right` and then projecting on `p`, which we prove separately by another lemma:

```
lemma AppendProject(xs: List<int>, ys: List<int>, p: int)

ensures Append(Project(xs, p), Project(ys, p))

== Project(Append(xs, ys), p)

{

}
```

With this lemma, the proof calculation of `MergeSortAuxSameElements` is now complete:

```
== { AppendProject(left, right, p); }

Project(Append(left, right), p);

==

Project(xs, p);
```

```
}
}
}
```

Whew! We have now proved the correctness of our `MergeSort` function.

> **Exercise 8.6.**
>
> It is common for implementations of (n log n) sorting algorithms to defer to a simpler (n^2) sorting algorithm when the input is small. This can lead to better performance, because (n log n) may incur greater overhead before setting up its recursive calls. Change the Merge Sort function above to call Insertion Sort on small inputs. More specifically, change **`if`** `len < 2` **`then`** `xs` in `MergeSortAux` to
>
> ```
> if len < 8 then InsertionSort(xs)
> ```
>
> Verify that the new Merge Sort program outputs the same elements as in the input, but ordered.

8.3.Summary

In this chapter, we considered the task of sorting lists. We first wrote general specifications for sorting algorithms and then applied these to Insertion Sort and Merge Sort.

The specification of sorting has two parts: the output is sorted, and the output has the same elements as the input. A *stable* sort additionally says that elements with equals keys are arranged in the same order in the input and output.

It is critical in engineering to get the specifications right. With program proofs and an automated verifier, you can check that an implementation meets its specification. How you formulate a specification affects how easy it is for a human to read and understand the specification. How you formulate the specification also affects the complexity of proofs, and for an automated verifier, the formulation may affect the degree of automation. In this chapter, I formulated the “same elements” specification in terms of preserving the multiset of the list's elements. Not only do I find that specification more straightforward to understand, but it also avoids the complexity of (the common approach of) trying to define what a permutation is.

Notes

The gallery of verified programs available from the Why3 home page [20] contains many algorithms that sort a given `List`. (It also contains programs that sort mutable arrays, which in this book we'll get to in Chapter 15.) The literature contains many “proof pearls” (short articles that demonstrate something interesting about proofs) of sorting algorithms. For example, the Natural Merge Sort algorithm, which is used in the Haskell standard library, has been verified in Isabelle/HOL [119] and in Dafny [79].

For imperative sorting algorithms in this book, see Chapter 15, which contains additional pointers and historical notes.

Chapter 9

Abstraction

Abstraction and information hiding are critical to good program design. They allow us to manage the complexity of interacting parts of a program. The chapters so far used small examples to illustrate how to reason about algorithms and how to write proofs. In those examples, there was no concern about (or need for, really) abstraction. In this chapter, I will show how to collect a type and various operations on that type into a *module*. One focus in this endeavor is how to draw the line between what clients of the module are allowed to know and what will remain as private details of the implementation. The chapter illustrates these organizational principles by a module that implements a queue.

9.0.Grouping Declarations into Modules

One basic role of a module is to group together related declarations. For example, in Chapter 6, we defined a type `List` along with operations on lists and lemmas about those operations. We can place these declarations in a module, call it `ListLibrary`.

```
module ListLibrary {

datatype List<T> = Nil | Cons(head: T, tail: List<T>)
function Append(xs: List, ys: List): List

// …



lemma AppendAssociative(xs: List, ys: List, zs: List)

ensures Append(Append(xs, ys), zs) == Append(xs, Append(ys, zs))

// …



// many other declarations go here…

}
```

Not only does a module group together related declarations, but it also divides up the program's *namespace*. This means that the same identifier can be introduced in multiple modules. For example, a module `StackLibrary` can also declare a function (or type or lemma or whatever) called `Append`. To qualify which `Append` you are referring to, you prefix it with the name of the module and a “`.`”. For example, to refer to the `ListLibrary` module's `Append` function from outside the `ListLibrary` module, you write `ListLibrary.Append`.

Two declarations placed alongside each other—that is, in the same *scope*—are known as *siblings*. Suppose there is a method `Main` that is a sibling of the module `ListLibrary`. The method can then use the name `ListLibrary`. Furthermore (and more interestingly), the method can refer to `ListLibrary` declarations by mentioning `ListLibrary` in qualified names. The following example illustrates:

```
method Main() {

var xs, i := ListLibrary.Nil, 0;

while i < 5 {

xs, i := ListLibrary.Cons(i, xs), i + 1;

}

print xs, "\n";

}
```

9.1.Module Imports

A module cannot refer to another module willy-nilly. More precisely, the names of sibling modules are not automatically included in other modules. For example, the following program is not legal:

```
module ModuleA {

function Plus(x: int, y: int): int {

x + y

}

}
```

```
module ModuleB {

function Double(x: int): int {

ModuleA.Plus(x, x) // error: unresolved identifier 'ModuleA'

}

}
```

To include the module name `ModuleA` inside the sibling module `ModuleB`, the latter must *import* the former. This is achieved by an **`import`** declaration inside `ModuleB`:

```
module ModuleB {

import ModuleA

function Double(x: int): int {

ModuleA.Plus(x, x)

}

}
```

More generally, the **`import`** declaration gives the importing module the ability to indicate a different local name for referring to the imported module. For example, in

```
module ModuleB {

import A = ModuleA

function Double(x: int): int {

A.Plus(x, x)

}

}
```

the **`import`** declaration introduces the local name `A` inside `ModuleB` and binds this name to the module `ModuleA`. In fact, the simpler declaration **`import`** `ModuleA` is just a shorthand for **`import`** `ModuleA = ModuleA`. (Note that the right-hand side of the "=" in the **`import`** declaration is rather special, because it has the ability to refer to names that are not in scope inside the module.)

The module imports in a program are not allowed to be cyclic. In other words, the import relation in a legal program forms a hierarchy of modules. For example, if `ModuleB` imports `ModuleA`, then `ModuleA` is not allowed to import `ModuleB`.

9.2.Export Sets

Modules do more than divide up the namespace of a program. They facilitate *information hiding*. This means that certain declarations, or certain parts of declarations, are not accessible outside the module. By hiding some portions of its declarations, a module gains the ability to make private implementation decisions. Such private implementation decisions can freely be changed in future versions of the module without disturbing any client of the module.

In Dafny, which parts of a module are available outside the module is controlled by *export sets*.

These are introduced by the **`export`** declaration. If a module does not include any **`export`** declaration, then everything in the module becomes available to importers of the module. For example, the `ListLibrary` module in Section 9.0 and `ModuleA` in Section 9.1 default to exporting everything.

Most every declaration consists of some kind of *signature* and some kind of *body*. The signature consists of the name introduced by the declaration, its type or parameters, and any specification clauses. The body is the rest of the declaration and typically consists of an expression or statement inside curly braces or some definition that follows an `=`.

For example, consider the following two declarations:

```
datatype Color = Blue | Yellow | Green | Red

function Double(x: int): nat

requires 0 <= x

ensures Double(x) % 2 == 0

{

if x == 0 then 0 else 2 + Double(x - 1)

}
```

The signature of `Color` is just the name `Color` and the signature of `Double` is its type signature, precondition, and postcondition. The body of `Color` is what follows the "`=`" and the body of `Double` is the expression in curly braces.

The export mechanism in Dafny gives control over the decision to export both the signature and body of a declaration, just the signature, or neither. In more detail, the **`provides`** clause of the **`export`** declaration lists the names of those declarations whose signature is to be exported and the **`reveals`** clause of the **`export`** declaration lists the names of those declarations whose signature and body are to be exported.

For illustration, consider the following modules:

```
module ModuleC {

export

provides Color

reveals Double

datatype Color = Blue | Yellow | Green | Red

function Double(x: int): nat

requires 0 <= x

ensures Double(x) % 2 == 0

{

if x == 0 then 0 else 2 + Double(x - 1)

}

}
```

```
module ModuleD {

import ModuleC

method Test() {

var c: ModuleC.Color;

c := ModuleC.Yellow; // error: unresolved identifier ‘Yellow’

assert ModuleC.Double(3) == 6;

}

}
```

The export set of `ModuleC` provides the signature of the type `Color` and reveals both the signature and body of the function `Double`. Since `ModuleD` imports `ModuleC`, the type `ModuleC.Color` can be used in `ModuleD`. The example uses this type in the declaration of local variable `c`. However, since `Color` is only provided by `ModuleC`, not revealed, the constructors of `Color` are not known in `ModuleD`, so the particular assignment statement in `ModuleD` is not allowed. In fact, `ModuleD` does not even get to know that `Color` is an inductive datatype—all it can tell is that `Color` is a type. On the other hand, the export set in `ModuleC` reveals `Double`, so `ModuleD` can both refer to the function and reason about its value. In particular, the assertion in `ModuleD` can be verified.

An export set must provide a self-contained view of the module: if you delete what is not exported, then what remains must be a program that still type checks. For one, all identifiers mentioned must have an available declaration. To illustrate, consider the following module:

```
module ModuleE {

datatype Parity = Even | Odd

function F(x: int): Parity

{

if x % 2 == 0 then Even else Odd

}

}
```

Suppose the export set of this module were to just provide the function `F`, which would be declared as follows:

```
export provides F // inconsistent export

// set for ModuleE
```

This would be illegal, because after deleting what is not exported, what remains is

```
function F(x: int): Parity
```

where `Parity` would be an unknown type. In other words, to a module that imports `ModuleE`, the declaration of `F` would seem malformed, because it mentions `Parity`, whose declaration is not known to the importer. Similarly, an export declaration

```
export provides Parity reveals F // inconsistent export

// set for ModuleE
```

would not be legal either. Here, what is made available by the export set is:

```
type Parity

function F(x: int): Parity

{

if x % 2 == 0 then Even else Odd

}
```

where `Even` and `Odd` would be unresolved identifiers.

An example of a consistent export set for `ModuleE` is

```
export

provides Parity, F
```

It makes the following declarations available:

```
type Parity

function F(x: int): Parity
```

Another legal export set is:

```
export

reveals Parity, F
```

which makes everything about type `Parity` and function `F` available to importers. Since those are the only two declarations in this module, the export declaration can also be given as

```
export

reveals *
```

which, as we have seen before, is also the default export set if there is no **`export`** declaration at all.

9.3.Modular Specification of a Queue

Let's try our hand at applying good information hiding to the implementation of an immutable queue. A *queue* is a list of elements with three operations: *initialization* creates an empty queue, *enqueue* appends an element to the queue, and *dequeue* removes and returns an elements from the head of the queue. The queues we consider here are *immutable*, so each operation returns a queue value rather than modifying some existing queue.

9.3.0.A basic queue interface

The most basic module describing a queue is the following:

```
module ImmutableQueue {

type Queue<A>

function Empty(): Queue

function Enqueue<A>(q: Queue, a: A): Queue

function Dequeue<A>(q: Queue): (A, Queue)

requires q != Empty()

}
```

This is not yet a complete module—let me build it up one step at a time. What I have shown here is how a *client* of the module (that is, a user of the module) would see it. Such a view of a module is called the module *interface,* because it describes how the module's clients and implementation interface with one another.

This module interface declares a type `Queue`, parameterized by some element type `A`. The module does not reveal anything about this type other than its name and number of type parameters. This is good information-hiding practice, because it gives the implementation freedom in how to represent values of the type. We say it *abstracts* over the implementation, which simply means that we can speak of queues in higher-level, more abstract terms.

The three operations are declared as functions. Since the dequeue operation cannot be applied to an empty queue, it has a precondition.

9.3.1.Abstraction functions

The basic queue interface above is too simple, because it does not give a client enough information to verify its use of the queue. For example, a client cannot tell from the basic interface whether or not the `Enqueue` operation may return an empty queue, which makes ever calling `Dequeue` dicult. To describe the operations in more detail, it would be helpful to be able to speak about what a queue really is.

A queue consists of a list of elements. If we could speak about that list, it would make for useful and intelligible specifications. It's that easy! Let's introduce a function from a queue to its list of elements:

```
ghost function Elements(q: Queue): List
```

The function is declared as ghost, because we intend to use this function only in specifications. It could be that the queue is implemented as a list, but just because we have this function does not constrain our implementation of the queue to be a list. We say that this function *abstracts* over the implementation and that the function is called an *abstraction function,* because it lets us think of the queue at a level that is higher than its eventual implementation.

Whatever module you are designing, it is a good idea to think about what abstraction the module provides. Doing so gives you clarity of mind when you dip into the implementation details.

Using the abstraction function `Elements`, we can specify the queue operations in terms of lists of elements. The specifications can be given intrinsically (in the postcondition of the functions) or extrinsically (in separate lemmas). Let us do it extrinsically. Here is our revised module interface:

```
module ImmutableQueue {
```

```
import LL = ListLibrary


type Queue<A>

function Empty(): Queue

function Enqueue<A>(q: Queue, a: A): Queue

function Dequeue<A>(q: Queue): (A, Queue)

requires q != Empty()


ghost function Elements(q: Queue): LL.List

lemma EmptyCorrect<A>()

ensures Elements(Empty<A>()) == LL.Nil

lemma EnqueueCorrect<A>(q: Queue, x: A)

ensures Elements(Enqueue(q, x)) == LL.Snoc(Elements(q), x)

lemma DequeueCorrect(q: Queue)

requires q != Empty()

ensures var (a, q’) := Dequeue(q);

LL.Cons(a, Elements(q’)) == Elements(q)

}
```

In order for the module to talk about lists, it imports the module `ListLibrary`. We are going to refer to the declarations in `ListLibrary` many times, so it will be convenient to have a shorter name to use in qualifications. For this reason, the **`import`** declaration introduces the local name `LL` to stand for the imported library.

For module `ImmutableQueue` to refer to declarations inside module `ListLibrary`, it prefixes them with “`LL.`”. For example, inside module `ListLibrary`, the datatype constructor `Cons` can be referred to as just `Cons` (since the name does not clash with other declarations in `ListLibrary`) or as `List.Cons` or even `List<B>.Cons` for any type `B`. To refer to this constructor from `ImmutableQueue`, we can therefore write `LL.Cons`, `LL.List.Cons`, or `LL.List<B>.Cons`.

I chose to declare `Elements` as a ghost function, because we will use it only in specifications. This is common for abstraction functions. But if the function's body is compilable and you want to make it available at run time, just remove the keyword **`ghost`** from the function's declaration.

Exercise 9.0.

Function `Elements` provides an abstraction of the queue implementation. Another, more coarse-grained, abstraction would have been to instead introduce a function

```
ghost function Length(q: Queue): nat
```

that returns the number of elements in the queue. Write a module interface that uses `Length` instead of `Elements`.

9.3.2.Declaring an export set

We did not yet explicitly declare an export set in our module-under-development. By default, Dafny will therefore reveal everything. This would be fine for the body-less declarations we have written so far. But as we complete the module to fully define the type, functions, and lemmas, we do not want to reveal those definitions. By writing an export set explicitly, we get to control the clients' view of the module.

We use `provides` clauses in an `export` declaration to list the names of declarations we wish to make available outside the module. To make the export set self-contained, we must make sure to provide clients with the `ListLibrary` as well. We accomplish this by listing its local name, `LL`, in the export declaration.

Here is the export declaration we add to the module:

```
export

provides Queue, Empty, Enqueue, Dequeue

provides LL, Elements

provides EmptyCorrect, EnqueueCorrect, DequeueCorrect
```

Since these are all the names declared in our module so far, we could have written

```
export provides *
```

However, this would also provide other helper functions and lemmas that we might introduce later.

One more remark about exporting the imported module `ListLibrary`. Through the exported name `LL`, a client module that imports `ImmutableQueue` under the name `IQ` can refer to the list type as `IQ.LL.List<B>` for any type `B`. A client can also declare its own import of the `ListLibrary` module, say

```
import MyOwnLLImport = ListLibrary
```

and then use `IQ.LL.List<B>` and `MyOwnLLImport.List<B>` synonymously.

9.3.3.A sample client

Now that we have a module interface, let us test it to make sure it can be used by a simple client. Our client will create an empty queue, enqueue an element, and then dequeue that element:

```
module QueueClient {

import IQ = ImmutableQueue

method Client() {

var q := IQ.Empty();

q := IQ.Enqueue(q, 20);

var (a, q’) := IQ.Dequeue(q); // precondition violation

}
}
```

The verifier complains that the call to `Dequeue` may violate its precondition. We had anticipated this before, which is why we introduced the lemmas. Let us make use of the lemmas—one for each function call (phew!)—and let us also add a couple of assertions at the end to check that we are getting the end results we'd expect:

```
module QueueClient {

import IQ = ImmutableQueue

method Client() {

IQ.EmptyCorrect<int>();var q := IQ.Empty();

IQ.EnqueueCorrect(q, 20);q := IQ.Enqueue(q, 20);

IQ.DequeueCorrect(q);  var (a, q’) := IQ.Dequeue(q);

assert a == 20;

assert q’ == IQ.Empty(); // assertion cannot be verified

}

}
```

The precondition of `Dequeue` can now be verified, so we can see that our lemmas are useful. But they don't do everything we would like, because the verifier complains about the last assertion.

To debug this situation, we can try to precede the failing assertion with the following assertion:

```
assert IQ.Elements(q’) == IQ.LL.Nil;
```

This assertion goes through the verifier! So, if we know that `q’` represents the empty list of elements, then how come we cannot verify that it equals `IQ.Empty()`, which (as `EmptyCorrect()` tells us) also represents the empty list of elements?

The problem is that there may be several values of the type `Queue<int>` that represent the empty list of elements. Indeed, depending on what data structure the implementation of `ImmutableQueue` uses, there may be many `Queue<int>` values that represent the same list of elements. In general, it can be computationally very expensive to make sure that all values of a type are canonical, that is, that each one represents a different abstract value.

We have a decision to make. At one extreme, we can decide that a client should never compare a `Queue` value directly, but should always resort to comparisons via the abstraction function `Elements`. At the other extreme, we can decide that the implementation must make it possible to compare `Queue` values directly, which essentially comes down to making sure function `Elements` is *injective* (that is, that `Elements` is a one-to-one function). We will settle for a good choice in between these two extremes, namely to make `Elements` injective for the empty queue. Stated differently, we will make `Empty()` the only value of `Queue` that represents the empty list of elements. We accomplish this by adding another lemma in `ImmutableQueue`:

```
lemma EmptyUnique(q: Queue)

ensures Elements(q) == LL.Nil ==> q == Empty()
```

and (don't forget!) adding `EmptyUnique` to the **`provides`** clause of the module's export clause. As usual, there are several ways to express the property of this lemma (see the discussion of `LessTransitive` in Section 7.1). Here, I chose to use an implication in the postcondition, rather than writing the antecedent in a precondition. This makes it possible to use the contrapositive of the lemma. That is, by calling the lemma with a `q` that is known not to be `Empty()`, a caller is able to conclude `Elements(q) != LL.Nil`.

Finally, our simple client can be verified:

```
module QueueClient {

import IQ = ImmutableQueue

method Client() {

IQ.EmptyCorrect<int>();var q := IQ.Empty();

IQ.EnqueueCorrect(q, 20);q := IQ.Enqueue(q, 20);

IQ.DequeueCorrect(q);  var (a, q’) := IQ.Dequeue(q);

assert a == 20;

IQ.EmptyUnique(q’);

assert q’ == IQ.Empty();

}

}
```

This gets the job done, but it sure looks cluttered. We'll work on reducing clutter like this in Section 10.3. Next, we are ready to take on the implementation of `ImmutableQueue`.

> **Exercise 9.1.**
>
> Write a client module (like `QueueClient`) to test the module interface in Exercise 9.0. Adjust the queue interface so that your client module verifies.

9.3.4.Queue implementation

We return to module `ImmutableQueue`. To make it complete, we need to define the type `Queue`, the four functions, and the four lemmas.

One simple implementation for our queue would be to use a list, the same list that `Elements` returns. However, the cost of appending an element to the list, as the `Enqueue` operation would do, is proportional to the length of the queue. The key idea behind getting a better implementation is to use two lists, call them `front` and `rear`. A portion of the elements at the head of the queue are stored in `front`, where they can be dequeued in constant time. The rest of the elements are stored in `rear` in *reverse order*, which lets them be enqueued in constant time. When the `Dequeue` operation finds `front` empty, it copies *and reverses* the elements from `rear` into `front`. Consequently, each element dequeued will have undergone two constant-time operations and been part of one reversal. As we have seen in Section 6.6, list reversal requires linear time, so `Dequeue` will have an *amortized* running time that is constant. This means that as a queue evolves, the sum of the running times of `Dequeue` operations that created the queue is a constant multiple of the number of `Dequeue` invocations.

We start our implementation by defining the type of the data structure. We replace the declaration **`type`** `Queue<A>` with the following:

```
datatype Queue<A> = FQ(front: LL.List<A>, rear: LL.List<A>)
```

Note, since our module only *provides*, not *reveals*, the type `Queue`, the fact that `Queue` is defined as a datatype remains a private implementation decision in the module.

The key idea of this queue implementation is captured in the definition of `Elements`:

```
ghost function Elements(q: Queue): LL.List {
LL.Append(q.front, LL.Reverse(q.rear))
}
```

The other functions are also easy to write:

```
function Empty(): Queue {
FQ(LL.Nil, LL.Nil)
}


function Enqueue<A>(q: Queue, x: A): Queue {
FQ(q.front, LL.Cons(x, q.rear))
}


function Dequeue<A>(q: Queue): (A, Queue)
requires q != Empty()
{
match q.front
case Cons(x, front’) =>
(x, FQ(front’, q.rear))
case Nil =>
var front := LL.Reverse(q.rear);
(front.head, FQ(front.tail, LL.Nil))
}
```

Lemmas `EmptyCorrect` and `EmptyUnique` are done automatically—you need only supply the empty body, `{}`, so I won't show them here. The proofs of the other two lemmas make for good exercises, so I won't show them here, either. 😜

Exercise 9.2.

Write the proof for `EnqueueCorrect`.

Exercise 9.3.

Write the proof for `DequeueCorrect`.

Exercise 9.4.

(a) Write a module `BlockChain` declaring a type `Ledger` with three operations that have the following type signatures:

```
function Init(): Ledger

function Deposit(ledger: Ledger, n: nat): Ledger

function Withdraw(ledger: Ledger, n: nat): Ledger
```

Write an abstraction function

```
ghost function Balance(ledger: Ledger): int
```

that reports the balance of a ledger, and make `n <= Balance(ledger)` be a precondition of `Withdraw`. Declare an export set for the module.

(b) Write intrinsic specifications for the functions in terms of `Balance`. Write a simple client to test that they behave as you'd expect.

(c) Implement the module with a list of integers. Specifically, define `Ledger` to be a type synonym:

```
type Ledger = List<int>
```

where `List` is our usual list type. The implementations of `Deposit` and `Withdraw` prepend (a non-negative or non-positive number, respectively) to the list.

(Note that parameter `n` to `Deposit` and `Withdraw` is **`nat`**, whereas `Balance` returns an **`int`**. In order to declare `Balance` to return a **`nat`**, we need a data-structure invariant, which I will cover in Chapter 10, see Exercise 10.5.)

9.4.Equality-Supporting Types

There is one more operation that clients of our immutable queue may want to use and that we thus need to export: equality. If you don't feel too concerned about this, you may want to skip this section, because it goes into a number of picky details of the Dafny language.

9.4.0.Equality

In Dafny, every type has a built-in equality operator, written ==. For example, equality on integers (**int**) is the expected arithmetic equality, and equality on sets of integers (**set**<**int**>) is mathematical set equality where the elements are compared using equality on integers. Equality can always be used in ghost contexts in Dafny, but equality cannot always be used in compiled contexts. This is because the values of some types may not be computable in finite time or at all. For example, to compare possibly infinite sets of integers (**iset**<**int**>) requires an infinite number of membership tests, and to compare functions on integers (**int -> int**) requires invoking the function on an infinite number of values.

For this reason, Dafny keeps track of which types are known to have a *compilable* equality operation. These are referred to as “equality-supporting types”—a slight misnomer, since all types support equality in ghost contexts. When the definition of a type is available, Dafny uses the definition to figure out whether or not the type supports equality. This is not possible for type parameters, opaque types, or types that are just *provided* by an export set. In such cases, we can explicitly declare that the type refers to an equality-supporting type. This is done by writing the *type characteristic* (==) after the name of the type in its declaration. (We saw an instance of this already, for function `Find` in Section 6.5.)

Consider a client of module `ImmutableQueue` that defines a variation of `Dequeue`, one that can be applied to any queue and that returns a default value when given an empty queue. Here is such a client module:

```
module QueueExtender {

import IQ = ImmutableQueue

function TryDequeue<A>(q: IQ.Queue, default: A): (A, IQ.Queue)

{

if q == IQ.Empty() then (default, q) else IQ.Dequeue(q)

}

}
```

With `ImmutableQueue` as we defined it, Dafny complains that function `TryDequeue` uses == to try to compare values of type `IQ.Queue<A>`. In module `QueueExtender`, it is not known whether or not `IQ.Queue<A>` supports equality. We will consider two possible ways out of this quagmire.

9.4.1.Explicating equality support

One way out of the quagmire is to export, from module `ImmutableQueue`, that type `Queue<A>` does indeed support equality. You might think this would be done by adding a `(==)` in the **datatype** declaration:

```
datatype Queue(==)<A> = // syntax error

FQ(front: LL.List<A>, rear: LL.List<A>)
```

However, the `(==)` mark can be used (for type parameters, as we saw for `Find` in Section 6.5 or) only in **type** declarations. That's not a problem—we simply rename the datatype and use a **type** declaration to define a *type synonym* that we export:

```
type Queue(==)<A> = // claimed equality support is in error Queue′<A>

datatype Queue′<A> = FQ(front: LL.List<A>, rear: LL.List<A>)
```

This almost works. The problem is that the datatype `Queue′<A>` does not always support equality! It supports equality only if the type parameter `A` does. So, to claim that `Queue<A>` supports equality, we must restrict the definition to `A`'s that support equality. This is done by marking the type parameter with `(==)` as well:

```
type Queue(==)<A(==)> = Queue′<A>
datatype Queue′<A> = FQ(front: LL.List<A>, rear: LL.List<A>)
```

Exercise 9.5.

How come we didn't need `(==)` already when we compared `q′ == IQ.Empty()` in method `Client()` in Section 9.3.3?

9.4.2.Exporting an emptiness predicate

Another way out of the quagmire is for `ImmutableQueue` to export not the queue == operation itself but only a test that compares a queue with `Empty()`. After all, as I remarked in Section 9.3.3 where I motivated the introduction of the `EmptyUnique` lemma, equality on `Queue` values is not so useful, because there are several `Queue` values that represent the same list of elements. But equality *is* useful for the empty queue, because `ImmutableQueue` is using a unique representation for the empty queue. Actually, you could also argue that using the precondition `q != Empty()`—back in Section 9.3.0 —was a mistake, because it already assumes a unique representation for the empty queue.

Under this quagmire exit strategy, we keep the **`datatype`** declaration of `Queue` as we had it before, and we add to module `ImmutableQueue` the following compiled predicate:

```
predicate IsEmpty(q: Queue)

ensures IsEmpty(q) <==> q == Empty()

{

q == FQ(LL.Nil, LL.Nil)

}
```

We also add `IsEmpty` in a **`provides`** clause of the **`export`** declaration in `ImmutableQueue`.

Looking at the body of `IsEmpty`, you may wonder how it can compare `q` using equality, since `q` is of a type that supports equality only if the element type of `q` supports equality. The explanation is that, as a special case, Dafny allows comparisons with expressions that consist of just literals, like `FQ(LL.Nil, LL.Nil)`. After all, such a comparison could also be done with **`match`** expressions or with the datatype discriminators `FQ?` and `Nil?`. (As a reminder, the comparison `q == Empty()` in the postcondition is also allowed, because Dafny always supports equality in ghost contexts.)

Exercise 9.6.

Without using `(==)`, rewrite module `QueueExtender` to use `IsEmpty`.

Exercise 9.7.

Export two new members of module `ImmutableQueue`: a compiled function `Length` that returns the number of elements in a given queue, and a correctness lemma about `Length(q)` that says it returns the length of the list `Elements(q)`. From these, it follows that the test `Length(q) == 0` gives the same result as would `q == Empty()`. Rewrite function `TryDequeue` in module `QueueExtender` to use `Length`.

Exercise 9.8.

The implementation of `TryDequeue` as suggested in Exercise 9.7 is not very ecient. Why?

9.5.Summary

Modular design is crucial in software engineering. It leads to a program decomposition where implementation decisions can be hidden inside modules. David Parnas pointed this out, in what can be summarized in the slogan “every module has one secret” [108]. To make it possible to write specifications and do verification in the presence of such information hiding, we abstract over the details. In this chapter, I showed how that can be done using an abstraction function (`Elements`, in the example) that maps the values we are implementing to values of other, known, simpler types.

Declarations are divided up into modules, and a module imports the modules it relies on. By declaring an export set, a module decides which declarations are to be available to clients. For each declaration in an export set, the export set can choose to include just the signature of the declaration (using **`provides`** clauses) or both the signature and body of the declaration (using **`reveals`** clauses).

Notes

Many fun and useful implementations of functional data structures, such as the queue implementation in Section 9.3.4, are found in Chris Okasaki's book on *Purely Functional Data Structures* [104].

To support information hiding, programming languages provide mechanisms of *access control* that limit the visibility of declarations. As we have seen, access control for the declarations in a module in Dafny is defined via export sets. Many class-based object-oriented languages, including Java and C#, allow members of a **`class`** to be declared with access modifiers like **`private`** (meaning, visible only in the implementation of the class) and **`public`** (meaning, usable in any context that has access to the class). C++ and Eiffel additionally allow members of one class to declare “friend” classes that get privileged access to the members. Other languages, including SPARK [117], Oberon-2, and Go, use a private/public mechanism at the module level.

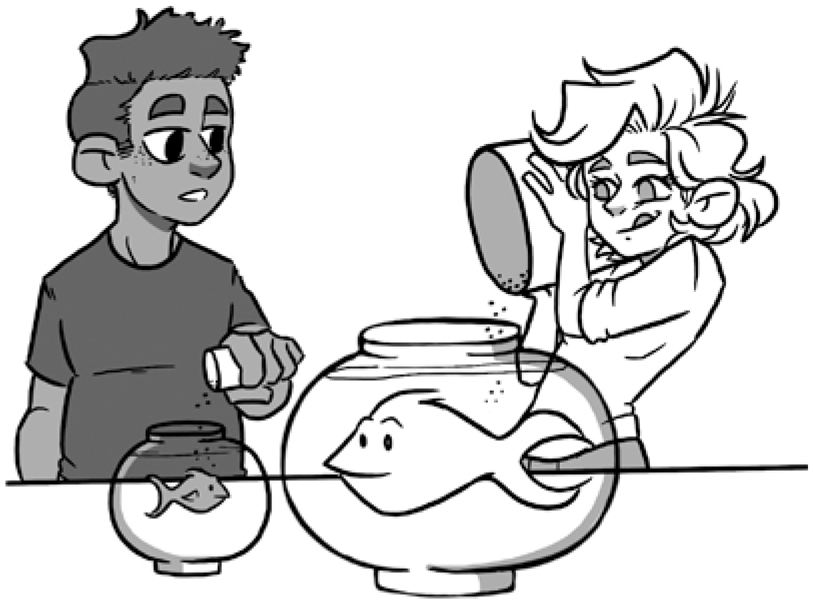
In many programming languages, access-control mechanisms govern only whether or not a client has access to a given declaration. This is sucient for compilation. But when specifications and proofs are involved, it is also important to be able to restrict how much of the *meaning* of a declaration is visible to a client. For this reason, Dafny offers two levels of exporting declarations (**`provides`** and **`reveals`**) [80], and F* [53] allows its module interfaces to contain semantic information.

Each module in this book has only one export set, which is the common case. In other cases, a module may need to present different clients with different views of the module. For example, a module may want to reveal some detail of its representation to some “friend” module. For this purpose, Dafny allows a module to declare several export sets [80]. This kind of flexibility is also provided by the modules and interfaces in Modula-3 [100].

Abstraction can be achieved in various ways. In this chapter, we used abstraction functions—ghost functions that speak of some values without constraining the implementation to any particular representation of those values. Abstraction functions are common in verification tools. In Chapters 16 and 17, we'll achieve abstraction by using ghost fields in combination with a validity predicate. The SPARK language [117] has special support for declaring abstract state (`Abstract_State`) and the concrete counterpart (`Refined_State`). There are many great texts that teach techniques for abstracting over data; to learn from true masters of abstractions, scope out Morgan using specification statements [93], Abrial using Event-B [0], and Lamport using TLA$^+$ [73].

Chapter 10

Data-Structure Invariants



Thank goodness for types! They let us describe the skeleton of a data structure and they give a simple way for a program analysis to check that a program maintains that skeleton. This means we can rely on, for example, the fact that a value of the type `Tree` defined in Section 4.5 is either a `Leaf` or a `Node`.

For most data structures, there is more to the design than just the skeleton. Every value of the data structure matches the skeleton, but not every value that matches the skeleton is a value that the data-structure operations can ever produce. For example, a *binary search tree* is a data structure whose skeleton is a tree and whose payload values are arranged in sorted order. Constraints like these that go beyond the skeleton are called *data-structure invariants*. Invariants are critical to good program design and program correctness.

To illustrate data-structure invariants in this chapter, I will use a *priority queue* as the running example. Whereas an ordinary queue is a collection of elements that are accessed in a first-in first-out fashion (as we saw in Section 9.3), a priority queue gives access only to a minimal element of the collection. We start with the specification and then go into the implementation.

10.0.Priority-Queue Specification

Other than creating and checking for an empty priority queue, the operations on a priority queue are insertion of an element and removal of the minimal element. We will consider priority queues that store integers, so “minimal” is defined in terms of arithmetic comparisons. We start our specification by declaring the following simple (and incomplete) module:

```
module PriorityQueue {

type PQueue
function Empty(): PQueue

predicate IsEmpty(pq: PQueue)

function Insert(pq: PQueue, y: int): PQueue

function RemoveMin(pq: PQueue): (int, PQueue)

requires !IsEmpty(pq)

}
```

So far, the module looks a lot like our starting point for ordinary queues in Section 9.3. Here, I have decided right away that we don't want to commit to a unique representation of values, not even the empty priority queue, and therefore the module introduces an `IsEmpty` predicate (as described in Section 9.4.2).

10.0.0.Abstraction

To say more about what is returned by these functions, we need some abstraction of what each `PQueue` value represents. As in Section 9.3, we introduce for this purpose an abstraction function that maps a `PQueue` to its elements. Here, we have a choice. One possibility is to abstract the elements into a sorted sequence (of type **`seq`**`<`**`int`**`>`). This will make it easy to write the specification for `RemoveMin`, but will complicate the specification of `Insert`. Another possibility (and this is what we will do) is to abstract the elements into a multiset (of type **`multiset`**`<`**`int`**`>`). This makes the specification of `Insert` simple, but complicates (only slightly) the specification of `RemoveMin`.

We introduce such a (specification-only) abstraction function `Elements` in our module. Using it, we write specifications for our functions extrinsically, as separate lemmas:

```
ghost function Elements(pq: PQueue): multiset<int>

lemma EmptyCorrect()

ensures Elements(Empty()) == multiset{}

lemma IsEmptyCorrect(pq: PQueue)

ensures IsEmpty(pq) <==> Elements(pq) == multiset{}

lemma InsertCorrect(pq: PQueue, y: int)

ensures Elements(Insert(pq, y))

== Elements(pq) + multiset{y}

lemma RemoveMinCorrect(pq: PQueue)

requires !IsEmpty(pq)

ensures var (y, pq’) := RemoveMin(pq);

IsMin(y, Elements(pq)) &&

Elements(pq’) + multiset{y} == Elements(pq)
```

The lemma about `RemoveMin` uses a predicate `IsMin`, which says that `y` is a minimal element in the multiset `Elements(pq)`. It's possible to define `IsMin` recursively, but a much more straightforward way is to use a *universal quantifier* (**`forall`**, written in mathematical texts) that can say something about every elements in the multiset. I will illustrate quantifiers in greater detail in Chapter 13; for our purposes here, we only need a head-nodding understanding of that `IsMin` is defined correctly.

```
ghost predicate IsMin(y: int, s: multiset<int>) {

y in s && forall x :: x in s ==> y <= x

}
```

In words, `IsMin(y, s)` says that `y` is an element of the multiset `s` and that for each element `x` in `s`, element `y` is below element `x`.

Our module so far looks pretty good from the point of view of clients.

Exercise 10.0.

Write a small test harness in a client module (cf. `QueueClient` in Section 9.3.3). Make the test harness insert two elements into a newly created priority queue and then use `RemoveMin` twice to retrieve them. Verify that the two elements are returned in the correct order.

Hint: To convince the verifier of this, you will need to use some lemmas. Of course, you will need to invoke the lemmas declared in `PriorityQueue`, but you will also need some lemmas to help the verifier reason about multisets. In particular, you will need to write some inline assertions about what is known about the elements of the priority at intermediate program points in test harness.

10.0.1.Export set

As we continue the development of the `PriorityQueue` module, let us be clear about what clients get to see and what remains private implementation details. We do that by adding an explicit **`export`** declaration. The module provides everything we have declared so far, except for `IsMin`, which it reveals in full.

```
export

provides PQueue, Empty, IsEmpty, Insert, RemoveMin

provides Elements

provides EmptyCorrect, IsEmptyCorrect

provides InsertCorrect, RemoveMinCorrect

reveals IsMin
```

This export set uses a mixture of **`provides`** and **`reveals`** clauses, which I introduced in Section 9.2.

10.1.Designing the Data Structure

To implement the priority queue, we will use a *binary heap*. A binary heap is a binary tree with two properties. First, a skeletal property: unlike the tree in Section 9.3, which placed the tree's payload in the leaves, the binary heap places its payload in the interior nodes of the tree and the leaves are empty. Second, a property beyond the skeleton, a data-structure invariant: every node in the tree satisfies what is called the *heap property*—the value stored in a node is minimal among all the values stored in the subtree rooted at that node.

The heap property makes it possible to obtain good performance: methods `Insert` and `RemoveMin` can each operate in time (log *n*) (that is, in time proportional to the logarithm of the size of the priority queue), *provided* the tree is balanced. A tree is *balanced* if every left subtree has roughly the same size as its neighboring right subtree.

An elegant data structure that ensures that the binary-heap tree is balanced is a *Braun tree*. This is what we are going to implement. The central idea of a Braun tree is that either the left and right subtrees of a node have the same size or the left subtree has one more element than the right subtree.

So, we have three properties altogether. There's the skeletal property that places values in interior nodes, there's the heap property, and there's the Braun-tree balance property. We call the latter two properties *data-structure invariants,* since they go beyond the skeletal property.

To ensure the skeletal property of the binary heap, we define type `PQueue` to be the following datatype:

```
type PQueue = BraunTree

datatype BraunTree =

| Leaf

| Node(x: int, left: BraunTree, right: BraunTree)
```

(Dafny allows a `|` before the first constructor name, which looks nice when you declare the constructors on separate lines, as I did here.) To ensure the data-structure invariant of Braun trees, we need to involve some logical expressions.

10.1.0.Valid trees

The correctness of our priority queue relies on the heap property. Thus, for verification, we need to distinguish those trees that have the heap property from other trees that (in principle) could be built from the constructors `Leaf` and `Node`. We do this by introducing a predicate that we shall call `Valid`. The idea is that a tree satisfies `Valid` if and only if it's a tree that may be returned from our `PriorityQueue` module.

We provide the `Valid` predicate to clients so it can be used in specifications, but we will keep the details private to the module.

```
export // …

provides Valid


ghost predicate Valid(pq: PQueue)
```

Next, we have a choice to make: `Valid` can be used intrinsically or extrinsically. That is, we can either use `Valid` in the specifications of the priority-queue operations themselves (intrinsically) or we can use `Valid` only in the correctness lemmas (extrinsically). If the crash-free functioning of the tree operations will rely on the validity condition, then we *have* to use it intrinsically, but otherwise it is a choice like other extrinsic-versus-intrinsic choices we have seen before (see Sections 6.2). For now, I will use `Valid` extrinsically, but I will return to this issue in Section 10.3.

Using `Valid`, we update the definitions of the lemmas as follows:

```
lemma EmptyCorrect()

ensures var pq := Empty();

Valid(pq) &&

Elements(pq) == multiset{}

lemma IsEmptyCorrect(pq: PQueue)

requires Valid(pq)

ensures IsEmpty(pq) <==> Elements(pq) == multiset{}

lemma InsertCorrect(pq: PQueue, y: int)

requires Valid(pq)

ensures var pq’ := Insert(pq, y);

Valid(pq’) &&

Elements(pq’) == Elements(pq) + multiset{y}

lemma RemoveMinCorrect(pq: PQueue)

requires Valid(pq) && !IsEmpty(pq)

ensures var (y, pq’) := RemoveMin(pq);

Valid(pq’) &&

IsMin(y, Elements(pq)) &&

Elements(pq’) + multiset{y} == Elements(pq)
```

Exercise 10.1.

Revisit Exercise 10.0 to make sure that the test harness still works, now that the correctness lemmas take into consideration the data-structure invariant.

10.2.Implementation

10.2.0.Defining the invariant

We define `Valid` to include all the properties we want to make sure are invariants of our data structure. In addition to the heap property, our invariant says that the data structure satisfies the Braun-tree balance property. Here are those definitions:

```
ghost predicate Valid(pq: PQueue) {

IsBinaryHeap(pq) && IsBalanced(pq)

}



ghost predicate IsBinaryHeap(pq: PQueue) {

match pq

case Leaf => true

case Node(x, left, right) =>

IsBinaryHeap(left) && IsBinaryHeap(right) &&

(left == Leaf || x <= left.x) &&

(right == Leaf || x <= right.x)

}



ghost predicate IsBalanced(pq: PQueue) {

match pq

case Leaf => true

case Node(_, left, right) =>

IsBalanced(left) && IsBalanced(right) &&

var L, R := |Elements(left)|, |Elements(right)|;

L == R || L == R + 1

}
```

Since predicate `IsBinaryHeap` is defined to hold recursively for the subtrees `left` and `right`, it suces to state for each node the heap property between the node's value and the value stored in its direct children (see Exercise 10.2 after the definition of `Elements` below).

As in the queue example in Section 9.3, abstraction function `Elements` is central to all our descriptions of correctness. It abstracts over the implementation's data structure to present clients with a simple mental model of the priority queue's operation. We define the function as the multiset union of all the values stored in the tree:

```
ghost function Elements(pq: PQueue): multiset<int> {

match pq

case Leaf => multiset{}

case Node(x, left, right) =>

multiset{x} + Elements(left) + Elements(right)

}
```

Exercise 10.2.

Predicate `IsBinaryHeap` is defined to hold recursively for the subtrees `left` and `right`; the property “a node holds the minimum value of the node's entire subtree” is stated only as comparisons between the node's value and the values stored in its direct children. Prove the definition correct. In other words, prove that the value stored in any node that satisfies `IsBinaryHeap` is minimal in the tree rooted at that node. That is, prove the following lemma:

```
lemma {:induction false} BinaryHeapStoresMinimum(pq: PQueue, y: int)

requires IsBinaryHeap(pq) && y in Elements(pq)

ensures pq.x <= y
```

The statement of the lemma itself is curious. The postcondition selects `.x`, so you might expect the precondition to have a conjunct `pq.Node?`. Why is the statement of the lemma still okay?

The proof of this lemma is also curious. (Spoiler alert!) The proof is by induction over the structure of `pq`. Most such proofs break into the structural cases. Not this one. Here, `pq` is required to be a `Node`. The proof's cases instead break down like the 3-part union in the definition of `Elements`.

Next, we write the implementation and correctness proofs for emptiness, insertion, and removal.

10.2.1.Emptiness

An empty priority queue is represented as a leaf. We define the body of `Empty` as follows:

```
function Empty(): PQueue {

Leaf

}
```

In fact, in our implementation, the empty priority queue has a unique representation. So, we define predicate `IsEmpty` as an equality comparison with `Leaf`:

```
predicate IsEmpty(pq: PQueue) {

pq == Leaf

}
```

Next, we prove the correctness of `Empty` and `IsEmpty` by filling in the bodies of lemmas

`EmptyCorrect` and `IsEmptyCorrect`:

```
lemma EmptyCorrect()

ensures var pq := Empty();

Valid(pq) &&

Elements(pq) == multiset{}

{

}

lemma IsEmptyCorrect(pq: PQueue)

requires Valid(pq)

ensures IsEmpty(pq) <==> Elements(pq) == multiset{}

{

}
```

Both of these lemmas are verified automatically.

10.2.2.Insertion

To insert an element into an empty binary heap is just a matter of returning a tree storing the element in a node with empty subtrees:

```
function Insert(pq: PQueue, y: int): PQueue {

match pq

case Leaf => Node(y, Leaf, Leaf)
```

To insert an element into a `Node` takes some more work. Recall that a binary heap is a tree where the element stored in each node compares below the elements stored in the node's subtrees. Thus, if the element stored in the node, call it `x`, is below the one we want to insert, `y`, then the insertion operation proceeds by creating a node that stores `x` and inserting `y` into a subtree. Otherwise, if `y` is below `x`, then the insertion operation does it the other way around: creating a node that stores `y` and inserting `x` into a subtree.

When `x` or `y` is inserted into a subtree, which subtree do we want to insert it into? As for maintaining the heap property, it does not matter. We can do the recursive insertion in either the left or right subtree. As for maintaining the balance property, however, it matters. We want to do the insertion into whichever subtree is smaller.

How do we determine which of the two subtrees is smaller? Here is where the key insight behind Braun trees comes into play. In a Braun tree, the right subtree is never larger than the left, so we want to insert it there. But if the two subtrees start off having the same size, then inserting into the right subtree would make it bigger. The beautiful observation in Braun trees is that we can maintain the Braun-tree invariant by (inserting into the right subtree and then) swapping the left and right subtrees. This will result in a tree where, again, either the left and right subtrees have the same size or the left subtree is one larger.

Here is the rest of the code in the `Insert` method:

```
case Node(x, left, right) =>

if y < x then

Node(y, Insert(right, x), left)

else

Node(x, Insert(right, y), left)

}
```

The correctness proof of `Insert` is by induction, following the cases in the function. In fact, it is so simple that the verifier handles it automatically:

```
lemma InsertCorrect(pq: PQueue, y: int)

requires Valid(pq)

ensures var pq’ := Insert(pq, y);

Valid(pq’) &&

Elements(pq’) == Elements(pq) + multiset{y}

{

}
```

10.2.3.Removal of the minimum

Because of the precondition `!IsEmpty(pq)`, function `RemoveMin(pq)` always applies to a `Node`, never to a `Leaf`. The element of that node is returned in the first component of `RemoveMin`'s return tuple, because a binary-heap stores its minimal element in the root node (see also Exercise 10.2). The other component of the return tuple is `pq` but with the minimum deleted. Let's write an auxiliary recursive function, `DeleteMin`, to do the latter. Then, we implement `RemoveMin` as follows:

```
function RemoveMin(pq: PQueue): (int, PQueue)

requires !IsEmpty(pq)

{

(pq.x, DeleteMin(pq))

}



function DeleteMin(pq: PQueue): PQueue

requires !IsEmpty(pq)
```

We define a correctness lemma for `DeleteMin`. This will make sure that we are clear about what `DeleteMin` is supposed to do. Plus, it lets us prove the correctness of `RemoveMin`.

```
lemma RemoveMinCorrect(pq: PQueue)

requires Valid(pq) && !IsEmpty(pq)
```

```
ensures var (y, pq’) := RemoveMin(pq);

Valid(pq’) &&

IsMin(y, Elements(pq)) &&

Elements(pq’) + multiset{y} == Elements(pq)

{

DeleteMinCorrect(pq);

}



lemma DeleteMinCorrect(pq: PQueue)

requires Valid(pq) && pq != Leaf

ensures var pq’ := DeleteMin(pq);

Valid(pq’) &&

Elements(pq’) + multiset{pq.x} == Elements(pq)
```

It seems magical that the proof of `RemoveMinCorrect` only calls lemma `DeleteMinCorrect`, since the latter does not mention anything about `IsMin`. The magic comes from Dafny's automatic induction, which lets the verifier discharge the proof obligation about `IsMin` without further guidance from us. (To confirm that induction is used, you can temporarily disable automatic induction for `RemoveMinCorrect` by marking it with the attribute `{:induction false}`. This will cause the verifier to complain that it no longer can prove the lemma.) I cannot expect you to write the proof about `IsMin` yourself at this time, because I have not yet said anything about how to write a manual proof of the universal quantifier in `IsMin`. So, if you want to understand in more detail why `y` is the minimum among the values in `Elements(pq)`, I instead recommend you do Exercise 10.2.

10.2.4.Auxiliary function `DeleteMin`

For a small tree where one or both subtrees are empty, the deletion of the root results in just the left subtree:

```
function DeleteMin(pq: PQueue): PQueue

requires !IsEmpty(pq)

{

if pq.left == Leaf || pq.right == Leaf then

pq.left
```

Consider a node with two nonempty subtrees. To delete its minimum, we put in its place the root of one of the subtrees. We want to be sure that the element we “promote” in this way is the smaller of the roots of the two subtrees, since this is what is needed to maintain the heap property. So we do something like this:

```
else if pq.left.x <= pq.right.x then

Node(pq.left.x, …)
```

```
else

Node(pq.right.x, …)
```

In the first of these two cases, we need to delete the element we promoted from the left subtree, so we call `DeleteMin(pq.left)` recursively. This will maintain the heap property of the Braun tree. To correctly maintain the balance condition of the Braun tree, we also swap the two subtrees, essentially reversing what the `Insert` operation does. This makes the first of the two cases into:

```
else if pq.left.x <= pq.right.x then

Node(pq.left.x, pq.right, DeleteMin(pq.left))
```

The second case is not symmetric to the first, because if we called function `DeleteMin(pq.right)`, we would produce a subtree that could potentially contain two fewer elements than `pq.left`, which would not satisfy the balance property of Braun trees. Instead, we first swap the roots of the left and right subtrees and then proceed as in the previous case. We do the swap with yet one more auxiliary method, `ReplaceRoot`:

```
else

Node(pq.right.x, ReplaceRoot(pq.right, pq.left.x),

DeleteMin(pq.left))

}



function ReplaceRoot(pq: PQueue, y: int): PQueue

requires !IsEmpty(pq)



lemma ReplaceRootCorrect(pq: PQueue, y: int)

requires Valid(pq) && !IsEmpty(pq)

ensures var pq’ := ReplaceRoot(pq, y);

Valid(pq’) &&

Elements(pq) + multiset{y} ==

Elements(pq’) + multiset{pq.x}
```

Exercise 10.3.

The very first case of `DeleteMin` handles the situation when one or both subtrees are empty. In the **`if`** condition, I wrote the test as a disjunction (an “or”):

```
pq.left == Leaf || pq.right == Leaf
```

If the tree is balanced, then the condition `pq.left == Leaf` implies `pq.right == Leaf`, so the disjunction is equivalent to the simpler test `pq.right == Leaf`. Why didn't I write it that way? What happens if you change the condition to the simpler one?

We start the proof of `DeleteMinCorrect` off with a usual case study, following the cases of the

`DeleteMin` function:

```
lemma DeleteMinCorrect(pq: PQueue)

requires Valid(pq) && pq != Leaf

ensures var pq’ := DeleteMin(pq);

Valid(pq’) &&

Elements(pq’) + multiset{pq.x} == Elements(pq)

{

if pq.left == Leaf || pq.right == Leaf {

} else if pq.left.x <= pq.right.x {

DeleteMinCorrect(pq.left);
```

The first case is handled automatically. For the second case, the definition of function `DeleteMin` makes a recursive call on `pq.left`, so we expect that the lemma will need an analogous recursive call. Indeed, that does the trick.

The third case is more involved. Let's start off by giving names to subexpressions we'll use:

```
} else {

var left, right :=

ReplaceRoot(pq.right, pq.left.x), DeleteMin(pq.left);

var pq’ := Node(pq.right.x, left, right);
```

That's a lot of symbols. To make sure we didn't introduce any mistakes, we let the verifier check that `pq’` is indeed the result of `DeleteMin`:

```
assert pq’ == DeleteMin(pq);
```

We expect to need the correctness properties of `left` and `right`, so we add calls to the relevant lemmas:

```
ReplaceRootCorrect(pq.right, pq.left.x);
DeleteMinCorrect(pq.left);
```

If we were lucky, the verifier would now finish the proof. It doesn't, so we have more work to do. There are various ways to proceed. If you were on your own, you would probably (like I did) try proving each of the different parts of the lemma's conclusion. When I tried it, I then ended up with a long and ugly proof. After looking at various pieces of the proof, refactoring it, and trying many different ways to prove each part, I ended up with a reasonable proof. Here, I will only show my final proof.

It starts off by a proof calculation that shows the `Elements` correctness property of `DeleteMin`. It is a property that shows the equality of two multisets. For such proofs, the verifier typically requires a fair amount of hand-holding. To understand how to approach such a proof, I have two pieces of advice.

First piece of advice for proofs that use multisets

The verifier may be able to prove the equality of two multisets, but it is not very creative when it comes to figuring out which equalities it should try to prove. Therefore, if you have an expression like `…A…` where `A` denotes a multiset and you want to show that it is equal to an expression `…B…` for some other multiset `B`, then you typically have to structure your proof calculation like this:

```
…A…;

== { assert A == B; }

…B…;
```

In the hint, the assertion will tell the verifier to prove the equality of multisets `A` and `B`. The verifier does this by trying to prove the multisets have the same elements (more precisely, that the multisets have the same multiplicity of every element). Once it has verified that `A` and `B` are indeed equal, it will use that equality to check the step in the proof calculation.

There's a name for the kind of equality in the assertion above: *extensionality*. By asking the verifier to check `A == B`, it does so by checking that `A` and `B` have the same elements, which it often can do, even if it didn't think of the need for the equality `A == B` by itself.

Second piece of advice for proofs that use multisets

Just like addition for arithmetic, multiset union is associative and commutative. For example, `A + B + C` is the same multiset as `A + C + B`. The verifier can check this, but you may have to guide it by bringing the equality to its attention, like I showed above. When it substitutes equals for equals, the verifier treats multiset union as being left associative and not necessarily commutative. Consequently, suppose a subexpression `A + B + C` appears in a formula and you want to show that the formula is unchanged if you replace that subexpression with `A + D + E`, where you have some way to proving `B + C == D + E`. If you tried a proof like

```
…A + B + C…;

== { assert B + C == D + E; } // I'm supposing you have

// some reason for this to hold

…A + D + E…; // you get an error that the proof

// step to here does not hold
```

then the verifier would still complain, because what it sees is:

```
…(A + B) + C…;

== { assert B + C == D + E; }

…(A + D) + E…; // you get an error that the proof

// step to here does not hold
```

Instead, you have to manually rearrange the parts of the formula. This is most comfortably done in separate steps.

```
…A + B + C…;
```

```
== { assert A + B + C == A + (B + C); }

…A + (B + C)…;

== { assert B + C == D + E; } // again, I'm supposing

// you have some reason for why this holds

…A + (D + E)…;

== { assert A + (D + E) == A + D + E; }

…A + D + E…;
```

Enough advice, let's get back to the proof

With these two pieces of advice in mind, we can now do the proof calculation for `DeleteMin`. In the steps without hints, all I have done is to rearrange the operands of the multiset union.

```
calc {

Elements(pq’) + multiset{pq.x};

== // def. Elements, since pq’ is a Node

multiset{pq.right.x} + Elements(left) +

Elements(right) + multiset{pq.x};

==

Elements(left) + multiset{pq.right.x} +

Elements(right) + multiset{pq.x};

== { assert Elements(left) + multiset{pq.right.x}

== Elements(pq.right) + multiset{pq.left.x}; }

Elements(pq.right) + multiset{pq.left.x} +

Elements(right) + multiset{pq.x};

==

Elements(right) + multiset{pq.left.x} +

Elements(pq.right) + multiset{pq.x};

== { assert Elements(right) + multiset{pq.left.x}

== Elements(pq.left); }

Elements(pq.left) + Elements(pq.right) +

multiset{pq.x};

==

multiset{pq.x} + Elements(pq.left) +

Elements(pq.right);

== // def. Elements, since pq is a Node
Elements(pq);

}
```

To finish the proof of `DeleteMin`, we need some information about the sizes of `pq'.left` and `pq'.right`. For `pq'` to be balanced, the size of `pq'.left` should be equal to or one more than the size of `pq'.right`. Since `pq` is balanced, the property we need follows from the fact that the recursive call to `DeleteMin` reduces the size by one and the call to `ReplaceRoot` does not change the size. We can prove these properties from the postconditions we wrote for lemmas `DeleteMinCorrect` and `ReplaceRootCorrect`. However, it will be more convenient if these lemmas had postconditions that mentioned these size properties explicitly. So, let's change the postcondition of `DeleteMinCorrect` to add the conjunct `|Elements(pq')| == |Elements(pq)| - 1` and change the postcondition of `ReplaceRootCorrect` to add the conjunct `|Elements(pq')| == |Elements(pq)|`.

The proof of `DeleteMinCorrect` now goes through without further interaction:

```
}

}
```

So, after strengthening the lemmas to explicitly mention how `DeleteMin` and `ReplaceRoot` affect the size of a tree, the only non-boilerplate proof we had to supply was the proof calculation about `Elements` in the third branch of `DeleteMin`.

10.2.5.Auxiliary function `ReplaceRoot`

The finish line is in sight—we have just one function left. Function `ReplaceRoot(pq, y)` deletes the minimum element of `pq` and inserts `y`. If neither subtree of `pq` has an element that is smaller than `y`, then we return a node like `pq` but with `y` as its root. This happens in several cases, depending on how many nonempty subtrees `pq` has:

```
function ReplaceRoot(pq: PQueue, y: int): PQueue

requires !IsEmpty(pq)

{

if pq.left == Leaf ||

(y <= pq.left.x && (pq.right == Leaf || y <= pq.right.x))

then

Node(y, pq.left, pq.right)
```

Of the small-tree cases, only one remains, namely when pq has one nonempty subtree with an element that is smaller than `y`. In that case, we promote that element and place `y` in a subtree by itself:

```
else if pq.right == Leaf then

Node(pq.left.x, Node(y, Leaf, Leaf), Leaf)
```

In the remaining cases, we promote the root of whichever subtree has the minimal element and then replace its minimum with `y`:

```
else if pq.left.x < pq.right.x then

Node(pq.left.x, ReplaceRoot(pq.left, y), pq.right)
```

```
else

Node(pq.right.x, pq.left, ReplaceRoot(pq.right, y))

}
```

The proof of `ReplaceRoot` starts off in the usual way, following the cases of the function. The first two cases are handled automatically. (Recall that the proof of `DeleteMinCorrect` compelled us to add to `ReplaceRootCorrect` the conjunct that says `ReplaceRoot` does not change the size of the tree.)

```
lemma ReplaceRootCorrect(pq: PQueue, y: int)

requires Valid(pq) && !IsEmpty(pq)

ensures var pq' := ReplaceRoot(pq, y);

Valid(pq') &&

Elements(pq) + multiset{y} == Elements(pq') + multiset{pq.x} &&

|Elements(pq')| == |Elements(pq)|

{

if pq.left == Leaf ||

(y <= pq.left.x && (pq.right == Leaf || y <= pq.right.x))

{

} else if pq.right == Leaf {
```

We start the third case by introducing names for the subexpressions of the result of `ReplaceRoot`, as we also did for the third case of `DeleteMin` in Section 10.2.4.

```
} else if pq.left.x < pq.right.x {

var left := ReplaceRoot(pq.left, y);

var pq' := Node(pq.left.x, left, pq.right);
```

We follow this up with a check that we wrote the expressions correctly:

```
assert pq' == ReplaceRoot(pq, y);
```

The proof is surely going to use the induction hypothesis, suitably parameterized, so we add that call, too.

```
ReplaceRootCorrect(pq.left, y);
```

Next, we prove the lemma's conclusion about `Elements`. This is another example of where the verifier can check multiset properties, but we have to lead it through which properties to check.

```
calc {

Elements(pq) + multiset{y};

== // def. Elements, since pq is a Node

multiset{pq.x} + Elements(pq.left) +
Elements(pq.right) + multiset{y};
==
Elements(pq.left) + multiset{y} +
Elements(pq.right) + multiset{pq.x};
== { assert Elements(pq.left) + multiset{y}
== Elements(left) + multiset{pq.left.x}; } // I.H.
multiset{pq.left.x} + Elements(left) +
Elements(pq.right) + multiset{pq.x};
== // def. Elements, since pq’ is a Node
Elements(pq’) + multiset{pq.x};
}
```

The fourth and final case is similar:

```
} else {
var right := ReplaceRoot(pq.right, y);
var pq’ := Node(pq.right.x, pq.left, right);
assert pq’ == ReplaceRoot(pq, y);
ReplaceRootCorrect(pq.right, y);
calc {
Elements(pq) + multiset{y};
== // def. Elements, since pq is a Node
multiset{pq.x} + Elements(pq.left) +
Elements(pq.right) + multiset{y};
==
Elements(pq.right) + multiset{y} +
Elements(pq.left) + multiset{pq.x};
== { assert Elements(pq.right) + multiset{y}
== Elements(right) + multiset{pq.right.x}; }
multiset{pq.right.x} + Elements(pq.left) +
Elements(right) + multiset{pq.x};
== // def. Elements, since pq’ is a Node
Elements(pq’) + multiset{pq.x};
}
}
```

```
}
```

Well, it took us a while, but we have now implemented all the functions in module `PriorityQueue`, and we have proved it all to be correct. The code is subtle, and there are more cases to consider than a human reviewer or test suite could confidently check. In contrast, the relatively simple specification and our accompanying proof give us high assurance that we have written the code correctly.

Sidebar 10.0

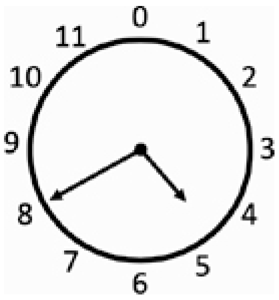
In the 12-hour system of clocks, many people get 12am and 12pm confused. The confusing thing is that an hour after 11am is 12pm (noon). If instead of using

```
1 <= hour <= 12
```

you replace the `12` on the clock by `0`, so you get

```
0 <= hour < 12
```

then it would be easy to remember that 0pm (noon) is an hour before 1pm.



Choose your invariants wisely and it will simplify your design.

10.3.Making Intrinsic from Extrinsic

In Section 10.0.0, we made the decision in the development of our `PriorityQueue` module that all specifications of our functions, except the non-emptiness precondition of `RemoveMin`, would be extrinsic. This brings the benefit of not having to clutter up the function bodies with proof elements. A drawback is that the functions cannot rely on the data-structure invariant. For some functions, this is not a big deal. Indeed, all the functions in the queue example in Chapter 9 work without spelling out a data-structure invariant at all. For the Braun tree example, it makes a difference. Let's see how.

10.3.0.Optimizing `DeleteMin`

We started off our function `DeleteMin` like this:

```
function DeleteMin(pq: PQueue): PQueue

requires !IsEmpty(pq)
```

```
{
if pq.left == Leaf || pq.right == Leaf then
pq.left
```

For a tree that satisfies the Braun-tree balance condition, this **`if`** test is a bit redundant. Any Braun tree satisfies:

```
pq.left == Leaf ==> pq.right == Leaf
```

so the disjunction in the **`if`** test can be simplified to just `pq.right == Leaf`. In other words, there is no case where only the left tree is a leaf, so the code might as well just inspect the right tree.

> **Exercise 10.4.**
>
> Formulate and prove a lemma that shows the Braun-tree balance condition to entail the implication above.

It is now tempting to change our `DeleteMin` implementation to instead start off

```
function DeleteMin(pq: PQueue): PQueue
requires !IsEmpty(pq)
{
if pq.right == Leaf then
pq.left
else if pq.left.x <= pq.right.x then // error: .x requires pq.left
// to be a Node
```

but that does not work, because of the next line of the function, which I included here. Since this `DeleteMin` function could, in principle, be called on any nonempty tree, even trees that do not satisfy the Braun-tree balance property, the verifier complains that the code tries to access member `x` of `pq.left`. (Well, there! I just gave you the answer to Exercise 10.3.)

To optimize our code for Braun trees, we need to give it a precondition that restricts it to Braun trees. That's what our decision of making the specifications extrinsic did not consider. Let's change that decision, at least partially, and add a precondition to `DeleteMin`.

```
function DeleteMin(pq: PQueue): PQueue
requires IsBalanced(pq) && !IsEmpty(pq)
{
if pq.right == Leaf then
pq.left
else if pq.left.x <= pq.right.x then
Node(pq.left.x, pq.right, DeleteMin(pq.left))
else
Node(pq.right.x, ReplaceRoot(pq.right, pq.left.x),

DeleteMin(pq.left))

}
```

With this change of specifications, the function body verifies even with the simplified **`if`** test. Good. But the verifier complains about the call to `DeleteMin` in `RemoveMin`. Let's change its specification accordingly. We might try:

```
function RemoveMin(pq: PQueue): (int, PQueue)

requires IsBalanced(pq) && !IsEmpty(pq)
```

This does not work, because our module exports `RemoveMin`, but not `IsBalanced`. We could consider exporting `IsBalanced` as well, but that would burden priority-queue clients with yet another concept. Instead, let's strengthen the precondition further:

```
function RemoveMin(pq: PQueue): (int, PQueue)

requires Valid(pq) && !IsEmpty(pq)
```

This does the trick. We have now optimized the code in `DeleteMin`, strengthened its precondition, and propagated its change in specifications to its callers.

10.3.1.The intrinsic-extrinsic spectrum

To apply the optimization in `DeleteMin`, we had to strengthen some specifications. We imposed as few additional preconditions as possible, culminating in the addition of `Valid(pq)` to the precondition of the exported function `RemoveMin`. Aesthetically, this leaves something to be desired, since `RemoveMin` is now the only one of the exported functions to mention `Valid`. In other words, we have gotten ourselves into a situation where the module interface uses a mix between intrinsic and extrinsic specifications.

We could consider making all of the specifications fully intrinsic. In other words, instead of writing the function specifications in separate correctness lemmas, we could write the specifications on the functions directly. In this approach, function `RemoveMin` would be declared as

```
function RemoveMin(pq: PQueue): (int, PQueue)

requires Valid(pq) && !IsEmpty(pq)

ensures var (y, pq’) := RemoveMin(pq);

Valid(pq’) &&

IsMin(y, Elements(pq)) &&

Elements(pq’) + multiset{y} == Elements(pq)
```

The choice between intrinsic and extrinsic specifications is a spectrum. An intermediate point in the design space is to always include the data-structure invariant, `Valid`, in the intrinsic specifications, but, to the extent possible, omit other functional-correctness specifications. In this approach, function `RemoveMin` would be declared as

```
function RemoveMin(pq: PQueue): (int, PQueue)

requires Valid(pq) && !IsEmpty(pq)
```

```
ensures var (y, pq’) := RemoveMin(pq);

Valid(pq’)
```

10.3.2.Externally intrinsic, internally extrinsic

Regardless of which point of the intrinsic-extrinsic spectrum we choose, as soon as a function includes a nontrivial postcondition, we face the extrinsic-specification disadvantage of having to pollute the function body with additional lemma calls. But there is a division of labor that gives us a simple intrinsic (or as-intrinsic-as-we-want) module interface and yet lets us separate function definitions from their correctness proofs. It is to layer a suite of intrinsically specified functions on top of the extrinsically specified functions. A clean way to do this is to write the two layers as separate modules.

First, let us change the name of the `PriorityQueue` module that we have written so far to `PriorityQueueExtrinsic`.

```
module PriorityQueueExtrinsic {

export

provides PQueue, Empty, IsEmpty, Insert, RemoveMin

provides Valid, Elements

reveals IsMin

provides EmptyCorrect, IsEmptyCorrect, InsertCorrect

provides RemoveMinCorrect



// the rest of the module as before…

}
```

Next, we define another module as a layer above `PriorityQueueExtrinsic`, where types and functions are defined directly in terms of the extrinsic counterparts. This module does not contain the correctness lemmas, because we are about to write the specifications as part of the functions. We give this module the straightforward name `PriorityQueue`, because we expect it to be the module that clients will use. Here, you can see how it works:

```
module PriorityQueue {

export

provides PQueue, Empty, IsEmpty, Insert, RemoveMin

provides Valid, Elements

reveals IsMin

provides Extrinsic



import Extrinsic = PriorityQueueExtrinsic
```

```
type PQueue = Extrinsic.PQueue


ghost predicate Valid(pq: PQueue) {
Extrinsic.Valid(pq)
}


ghost function Elements(pq: PQueue): multiset<int> {
Extrinsic.Elements(pq)
}


ghost predicate IsMin(m: int, s: multiset<int>) {
Extrinsic.IsMin(m, s)
}


function Empty(): PQueue {
Extrinsic.Empty()
}


predicate IsEmpty(pq: PQueue) {
Extrinsic.IsEmpty(pq)
}


function Insert(pq: PQueue, y: int): PQueue {
Extrinsic.Insert(pq, y)
}


function RemoveMin(pq: PQueue): (int, PQueue)
requires Valid(pq) && !IsEmpty(pq)
{
Extrinsic.RemoveMin(pq)
}

}
```

Note that the module export set provides the symbol `Extrinsic`. This is necessary if we want to, like in `PriorityQueueExtrinsic`, reveal function `IsMin`. (By providing `Extrinsic`, we also make it possible to reveal, not just provide, type `PQueue`. This can be useful in some advanced situations where we expect clients to want to use both modules.)

Sidebar 10.1

As I've shown them here, modules `PriorityQueue` and `PriorityQueueExtrinsic` are sibling declarations. An alternative organization is to make `PriorityQueueExtrinsic` a nested module, which you do by writing it inside module `PriorityQueue`. You can then also remove the **`import`** declaration in `PriorityQueue` that introduced the local name `Extrinsic` and give the inner module the name `Extrinsic`.

Finally, we add the specifications and proofs. This involves copying each correctness-lemma specification from `PriorityQueueExtrinsic` to the respective function in `PriorityQueue` and adding a call from the function to the lemma.

```
function Empty(): PQueue

ensures var pq := Empty();

Valid(pq) &&

Elements(pq) == multiset{}

{

Extrinsic.EmptyCorrect();

Extrinsic.Empty()

}



predicate IsEmpty(pq: PQueue)

requires Valid(pq)

ensures IsEmpty(pq) <==> Elements(pq) == multiset{}

{

Extrinsic.IsEmptyCorrect(pq);

Extrinsic.IsEmpty(pq)

}
```

```
function Insert(pq: PQueue, y: int): PQueue

requires Valid(pq)

ensures var pq' := Insert(pq, y);

Valid(pq') &&

Elements(pq') == multiset{y} + Elements(pq)

{

Extrinsic.InsertCorrect(pq, y);

Extrinsic.Insert(pq, y)

}



function RemoveMin(pq: PQueue): (int, PQueue)

requires Valid(pq) && !IsEmpty(pq)

ensures var (y, pq') := RemoveMin(pq);

Valid(pq') &&

IsMin(y, Elements(pq)) &&

Elements(pq') + multiset{y} == Elements(pq)

{

Extrinsic.RemoveMinCorrect(pq);

Extrinsic.RemoveMin(pq)

}
```

In this example, where I showed how to layer an intrinsic-specification module on top of an extrinsic-specification module, I used the full extrinsic specifications. If you want a different point on the intrinsic-extrinsic spectrum, it would follow the same form.

10.4.Summary

An *invariant* is a condition that always holds. In this chapter, the invariant of interest described the consistency condition of Braun trees: the heap property and the Braun-tree balance property. We used this invariant to prove the functional correctness of priorityqueue operations, and also to prove that the operations produce only balanced trees. We encoded the invariant as a predicate `Valid`. More precisely, `Valid` is a predicate that holds when a given skeleton value satisfies the desired data-structure invariant. We then used `Valid()` in pre- and postconditions (for the correctness lemmas in Section 10.1.0 and for the intrinsically specified functions in Section 10.3.2).

By using `Valid()` in the pre- and postcondition of the priority-queue operations, clients of our `PriorityQueue` module are made aware of the existence of a validity condition. By only providing, not revealing, `Valid()` in the module interface, we abstract over the details of the invariant. In effect, this means that clients cannot fabricate their own tree values from scratch, because the client doesn't know what `Valid()` entails. Instead, clients can obtain priority-queue values only by calling the operations we provided in the module interface. In this way, we have created a verified interface with desirable information hiding.

If there's a property you think always holds of your data structure, you can use the invariant to check it. To do that, write and prove a lemma that the data-structure invariant (`Valid()`) implies the property of interest (see Exercise 10.4).

From the perspective of program structure and specification, what the example I showed has in common with so many other modules that provide immutable data structures is that it captures the data-structure invariant in a predicate `Valid` and uses some number (here, one) of abstraction functions (like `Elements`) to describe the effect of the various operations of the module. We'll see a similar, slightly more complicated pattern when we get to mutable data structures in the next Part of the book.

Exercise 10.5.

Revisit the `BlockChain` module of Exercise 9.4 and change the result type of the `Balance` function from the previous **`int`** to **`nat`**. To prove that the function does indeed return a non-negative number, you need to use a data-structure invariant. Add a predicate

```
ghost predicate Valid(ledger: Ledger)
```

to the module, like we have done in this chapter. Use this validity predicate in the specifications of the `Init`, `Deposit`, `Withdraw`, and `Balance` functions, and prove that the implementations of the functions meet their specifications. As usual, write and verify a test harness to make sure your specifications are usable in the way you'd expect.

Notes

It was Aaron Stump's delightful introductory book on verification in Agda [120] that introduced me to Braun trees [24].

The validity predicates we used in this chapter (and those we'll see in Chapters 16 and 17) let us describe the expected state of a data structure without divulging details of that state to clients. This achieves information hiding. But couldn't we hide even more? The way we created our module interface, a client will never be able to obtain a non-valid skeleton. So, then, is it really necessary to bother clients with the existence of a validity condition? The general answer to this question depends on if there are ways for clients to detect mutations in or interactions between parts of the implementation. Since we wrote our module using functional programming, where values are immutable, the answer is that we can in fact hide the existence of `Valid` as well. Many verifiers (including Dafny) support *subset types* (aka *predicate subtypes* or *refinement types*) for this purpose, but I won't cover them in this book.

Some verification languages and proof assistants, including Coq [16], PVS [107], Agda [22], F* [53], and LiquidHaskell [123], are primarily based around such subset types. In those languages, it is customary (or even necessary) to formulate datastructure invariants as parts of types. Advanced forms of such programming and verification are treated in Adam Chlipala's book on *Certified Programming* [27].

Part 2

Imperative Programs

In imperative programming, the value of a variable can be changed during the execution of the program. This gives us a natural way to represent evolving real-life state, like your social-media contact list or the speed of a locomotive. It also means that we don't have to immediately produce the desired value of a computation, but we can proceed in steps, gradually changing the values of variables until we obtain what we want. Imperative programming is sometimes hailed as achieving eciency, because a step of a computation can update just what needs to be changed rather than making a copy of all pieces of the state that do not change.

In this Part, I will show you how to reason about three kinds of imperative state: local variables, arrays of values, and dynamically allocated, evolving data structures. Common to reasoning about each of these is some form of *invariant*, like the datastructure invariants we already saw in Chapter 10.