Appendix C

Answers to Select Exercises

Answer to Exercise 2.34

Yes, provided it also crashes your program. For example, by the equation

```
assert true, CandyCrushScore == 50 =
CandyCrushScore == 50
```

we know that *if* we start with a score of `50` and *if* the program terminates without crashing, then the score remains `50`. However, the equation says nothing about the post-state if the statement crashes and does not tell us when a statement crashes.

Answer to Exercise 2.35

```
x := E, P  =
true
[assert E, P]  =
P ==> E
S;T, P  =
S, P && T, S, P
if B { S } else { T }, P  =
S, P && B && T, P && !B
```

Answer to Exercise 3.10

a) `x x - 2`
b) `x - 2 < x && 0 <= x`

Answer to Exercise 3.13

For `Outer`, use **`decreases`** `a`.

For `Inner`, use **`decreases`** `a, b`.

Answer to Exercise 4.1

```
function ReverseColors(t: BYTree): BYTree {

match t

case BlueLeaf => YellowLeaf

case YellowLeaf => BlueLeaf

case Node(left, right) => Node(ReverseColors(left), ReverseColors(right))

}
```

Answer to Exercise 4.5

The precondition of the destructor can be met using either an **`if-then-else`** expression:

```
if t.Node? then t.left == u else false
```

or using the short-circuit boolean operator &&:

```
t.Node? && t.left == u
```

Note that `t.Node?` must be checked before using the destructor; it would be an error to write

```
t.left == u && t.Node? // error
```

Answer to Exercise 5.6

```
lemma Ack1(n: nat)
ensures Ack(1, n) == n + 2
{
if n == 0 {
// trivial
} else {
calc {
Ack(1, n);
==  // def. Ack
Ack(0, Ack(1, n - 1));
==  // def. Ack(0, _)
Ack(1, n - 1) + 1;
==  { Ack1(n - 1); } // induction hypothesis
(n - 1) + 2 + 1;
==  // arithmetic
n + 2;
}
}
}
```

Answer to Exercise 5.13

The following proof spells out each case in detail:

```
lemma {:induction false} OceanizeIdempotent(t: BYTree)
```

```
ensures Oceanize(Oceanize(t)) == Oceanize(t)
{
match t
case BlueLeaf =>
calc {
Oceanize(Oceanize(BlueLeaf));
== // def. Oceanize
Oceanize(BlueLeaf);
}
case YellowLeaf =>
calc {
Oceanize(Oceanize(YellowLeaf));
== // def. Oceanize
Oceanize(BlueLeaf);
== // def. Oceanize
BlueLeaf;
== // def. Oceanize
Oceanize(YellowLeaf);
}
case Node(left, right) =>
calc {
Oceanize(Oceanize(Node(left, right)));
== // def. Oceanize
Oceanize(Node(Oceanize(left), Oceanize(right)));
== // def. Oceanize
Node(Oceanize(Oceanize(left)), Oceanize(Oceanize(right)));
== { OceanizeIdempotent(left); }
Node(Oceanize(left), Oceanize(Oceanize(right)));
== { OceanizeIdempotent(right); }
Node(Oceanize(left), Oceanize(right));
== // def. Oceanize
Oceanize(Node(left, right));
}
```

```
}
```

Answer to Exercise 5.14

```
lemma {:induction false} OceanizeUpsBlueCount(t: BYTree)
ensures BlueCount(t) <= BlueCount(Oceanize(t))
{
match t
case BlueLeaf =>
case YellowLeaf =>
case Node(left, right) =>
calc {
BlueCount(Node(left, right));
==  // def. BlueCount
BlueCount(left) + BlueCount(right);
<= { OceanizeUpsBlueCount(left); }
BlueCount(Oceanize(left)) + BlueCount(right);
<= { OceanizeUpsBlueCount(right); }
BlueCount(Oceanize(left)) + BlueCount(Oceanize(right));
==  // def. BlueCount
BlueCount(Node(Oceanize(left), Oceanize(right)));
==  // def. Oceanize
BlueCount(Oceanize(Node(left, right)));
}
}
```

Answer to Exercise 5.15

Here is the **calc** statement for the `Cons` case:

```
calc {
EvalList(SubstituteList(args, n, c), op, env);
==  // args == Cons(e, tail)
EvalList(SubstituteList(Cons(e, tail), n, c), op, env);
==  // def. SubstituteList
EvalList(Cons(Substitute(e, n, c),
SubstituteList(tail, n, c)), op, env);
```

```
==  // def. EvalList
var v0, v1 :=
Eval(Substitute(e, n, c), env),
EvalList(SubstituteList(tail, n, c), op, env);
match op
case Add => v0 + v1
case Mul => v0 * v1;
==  { EvalSubstitute(e, n, c, env); }
var v0, v1 :=
Eval(e, env[n := c]),
EvalList(SubstituteList(tail, n, c), op, env);
match op
case Add => v0 + v1
case Mul => v0 * v1;
==  { EvalSubstituteList(tail, op, n, c, env); }
var v0, v1 :=
Eval(e, env[n := c]),
EvalList(tail, op, env[n := c]);
match op
case Add => v0 + v1
case Mul => v0 * v1;
==  // def. EvalList
EvalList(Cons(e, tail), op, env[n := c]);
==  // args == Cons(e, tail)
EvalList(args, op, env[n := c]);
}
```

Answer to Exercise 5.16

```
lemma EvalEnv(e: Expr, n: string, env: map<string, nat>)
requires n in env.Keys
ensures Eval(e, env) == Eval(Substitute(e, n, env[n]), env)
{
EvalSubstitute(e, n, env[n], env);
assert env == env[n := env[n]]; // needed for extensionality
```

```
}
```

Answer to Exercise 6.0

```
function Length'<T>(xs: List<T>): nat {
if xs == Nil then 0 else 1 + Length'(xs.tail)
}


lemma LengthLength'<T>(xs: List<T>)
ensures Length(xs) == Length'(xs)
{
}
```

This lemma is proved automatically by Dafny—you only need to write the empty lemma body: `{}`.

Answer to Exercise 6.3

```
lemma {:induction false} AppendNil<T>(xs: List<T>)
ensures Append(xs, Nil) == xs
{
match xs
case Nil =>
case Cons(x, tail) =>
calc {
Append(xs, Nil);
==  // def. Append
Cons(x, Append(tail, Nil));
==  { AppendNil(tail); }
Cons(x, tail);
==
xs;
}
}
```

Answer to Exercise 6.6

```
lemma UnitsAreTheSame()
ensures L == R
```

```
{
calc {
L;
==  { RightUnit(L); }
F(L, R);
==  { LeftUnit(R); }
R;
}
}
```

Answer to Exercise 6.8

```
function LiberalTake<T>(xs: List<T>, n: nat): List<T>
{
if n == 0 || xs == Nil then
Nil
else
Cons(xs.head, LiberalTake(xs.tail, n - 1))
}


function LiberalDrop<T>(xs: List<T>, n: nat): List<T>
{
if n == 0 || xs == Nil then
xs
else
LiberalDrop(xs.tail, n - 1)
}


lemma TakesDrops<T>(xs: List<T>, n: nat)
requires n <= Length(xs)
ensures Take(xs, n) == LiberalTake(xs, n)
ensures Drop(xs, n) == LiberalDrop(xs, n)
{
}
```

Answer to Exercise 6.14

In the postcondition of `AtFind`, the second argument to `At` is `Find(xs, y)`. The intrinsic specification of `Find` tells us that `Find(xs, y)` does not exceed `Length(xs)`. Moreover, since `||` is a short-circuit operator, `At` is called only if the first disjunct (`Find(xs, y) == Length(xs)`) does not hold. These facts imply the precondition of the call to `At`.

For `BeforeFind`, the second argument to `At` is `i`. Since `==>` is a short-circuit operator, `At` is called only if the antecedent `i < Find(xs, y)` holds. The intrinsic specification of `Find` thus lets us establish the precondition of the call to `At`.

Answer to Exercise 7.2

```
predicate Less(x: Unary, y: Unary) {
y != Zero && (x == Zero || Less(x.pred, y.pred))
}
```

Answer to Exercise 7.3

```
lemma LessTrichotomous(x: Unary, y: Unary)
// 1 or 3 of them are true:
ensures Less(x, y) <==> x == y <==> Less(y, x)
// not all 3 are true:
ensures !(Less(x, y) && x == y && Less(y, x))
{
}
```

Note that this solution uses the fact that <==> is associative, not chaining. That is,

```
Less(x, y) <==> x == y <==> Less(y, x)
```

can for example be parenthesized as

```
Less(x, y) <==> (x == y <==> Less(y, x))
```

Answer to Exercise 7.5

```
lemma {:induction false} AddCorrect(x: Unary, y: Unary)
ensures UnaryToNat(Add(x, y)) == UnaryToNat(x) + UnaryToNat(y)
{
match y
case Zero =>
case Suc(y') =>
calc {
```

```
UnaryToNat(Add(x, y));
==  // y == Suc(y’)
UnaryToNat(Add(x, Suc(y’)));
==  // def. Add
UnaryToNat(Suc(Add(x, y’)));
==  // def. UnaryToNat
1 + UnaryToNat(Add(x, y’));
==  { AddCorrect(x, y’); }
1 + UnaryToNat(x) + UnaryToNat(y’);
==  // def. UnaryToNat
UnaryToNat(x) + UnaryToNat(Suc(y’));
==  // y == Suc(y’)
UnaryToNat(x) + UnaryToNat(y);
}
}


lemma {:induction false} SucAdd(x: Unary, y: Unary)
ensures Suc(Add(x, y)) == Add(Suc(x), y)
{
match y
case Zero =>
case Suc(y’) =>
calc {
Suc(Add(x, Suc(y’)));
==  // def. Add
Suc(Suc(Add(x, y’)));
==  { SucAdd(x, y’); }
Suc(Add(Suc(x), y’));
==  // def. Add
Add(Suc(x), Suc(y’));
}
}
```

```
lemma {:induction false} AddZero(x: Unary)
ensures Add(Zero, x) == x
{
match x
case Zero =>
case Suc(x’) =>
calc {
Add(Zero, Suc(x’));
==  // def. Add
Suc(Add(Zero, x’));
==  { AddZero(x’); }
Suc(x’);
}
}
```

Answer to Exercise 9.0

```
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<A>()

ghost function Length(q: Queue): nat
lemma EmptyCorrect<A>()
ensures Length(Empty<A>()) == 0
lemma EnqueueCorrect<A>(q: Queue, x: A)
ensures Length(Enqueue(q, x)) == Length(q) + 1
lemma DequeueCorrect(q: Queue)
requires q != Empty()
```

```
ensures Length(Dequeue(q).1) == Length(q) - 1

}
```

Answer to Exercise 9.5

In `Client()`, we wrote that equality in an **`assert`**, which is a ghost statement. In ghost contexts, Dafny supports equality for all types.

Answer to Exercise 9.6

```
module QueueExtender {

import IQ = ImmutableQueue

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

{

if IQ.IsEmpty(q) then (default, q) else IQ.Dequeue(q)

}

}
```

Answer to Exercise 10.0

To prove this program correct, we need an auxiliary assertion that reminds the verifier of what it knows about `Elements(pq)` after the two insertions.

```
module PriorityQueueTestHarness {

import PQ = PriorityQueue

method Test(x: int, y: int) {

PQ.EmptyCorrect(); var pq := PQ.Empty();

PQ.InsertCorrect(pq, x);  pq := PQ.Insert(pq, x);

PQ.InsertCorrect(pq, y);  pq := PQ.Insert(pq, y);

assert PQ.Elements(pq) == multiset{x,y};

PQ.IsEmptyCorrect(pq); PQ.RemoveMinCorrect(pq);

var (a, pq’) := PQ.RemoveMin(pq);

PQ.IsEmptyCorrect(pq’); PQ.RemoveMinCorrect(pq’);

var (b, pq”) := PQ.RemoveMin(pq’);

assert {a,b} == {x,y} && a <= b;

}

}
```

With all these lemmas, we get the work done, but the result ain't pretty. We'll work on prettifying the situation in Section 10.3.

Answer to Exercise 10.2

```
lemma {:induction false} BinaryHeapStoresMinimum(pq: PQueue, y: int)
requires IsBinaryHeap(pq) && y in Elements(pq)
ensures pq.x <= y
{
// By the definition of Elements, we consider the three
// cases that "y in Elements(pq)" gives rise to.
if
case y == pq.x =>
// trivial
case y in Elements(pq.left) =>
calc {
pq.x;
<=  // def. IsBinaryHeap
pq.left.x;
<=  { BinaryHeapStoresMinimum(pq.left, y); }
y;
}
case y in Elements(pq.right) =>
calc {
pq.x;
<=  // def. IsBinaryHeap
pq.right.x;
<=  { BinaryHeapStoresMinimum(pq.right, y); }
y;
}
}
```

Answer to Exercise 10.3

See Section 10.3.0.

Answer to Exercise 10.4

```
lemma BalanceConsequence(pq: PQueue)
```

```
requires !IsEmpty(pq) && IsBalanced(pq)
ensures pq.left == Leaf ==> pq.right == Leaf
{
}
```

Answer to Exercise 11.7

```
UpWhileLess:
invariant i <= N
decreases N - i


UpWhileNotEqual:
invariant i <= N
decreases N - i


DownWhileNotEqual:
invariant 0 <= i
decreases i


DownWhileGreater:
invariant 0 <= i
decreases i
```

Answer to Exercise 12.5

```
method FastExp(b: nat, n: nat) returns (p: nat)
ensures p == Exp(b, n)
{
p := 1;
var d, k := b, n;
while k != 0
invariant p * Exp(d, k) == Exp(b, n)
{
calc {
Exp(d, k);
```

```
== { assert k == 2 * (k / 2) + k % 2; }
Exp(d, 2 * (k / 2) + k % 2);
== { ExpAddExponent(d, k / 2 * 2, k % 2); }
Exp(d, 2 * (k / 2)) * Exp(d, k % 2);
== { ExpSquareBase(d, k / 2); }
Exp(d * d, k / 2) * Exp(d, k % 2);
}
if k % 2 == 1 {
assert Exp(d, k % 2) == d;
p := p * d;
}
d, k := d * d, k / 2;
}
}
```

Answer to Exercise 12.8

```
lemma {:induction false} AppendSumUp(lo: int, hi: int)
requires lo < hi
ensures SumUp(lo, hi - 1) + F(hi - 1) == SumUp(lo, hi)
decreases hi - lo
{
if lo == hi - 1 {
} else {
AppendSumUp(lo + 1, hi);
}
}
```

Answer to Exercise 13.3

```
forall i, j :: 0 <= i < j < a.Length ==> a[i] < a[j]
```

Answer to Exercise 13.14

Add a case:

```
case a[m] == b[n] =>
return 0;
```

and (optionally) change the <= in each of the other guards to <.

Answer to Exercise 13.19

Because `lo` and `hi` are known to be at least `0`, you can write

```
mid := lo + (hi - lo) / 2;
```

Answer to Exercise 14.0

Add

```
requires a.Length == 1 ==> left == right
```

Answer to Exercise 14.2

The following postcondition makes the method verify:

```
ensures a[i] == old(a[i]) + 1
```

Answer to Exercise 14.3

The following assignment implements the specification of `OldVsParameters`:

```
y := 25 - a[i];
```

Answer to Exercise 14.8

Hint: Method `DoubleArray` is like `CopyArray`, with two differences. One difference is to insert `2 *` in front of the `src` term in the method specification, loop specification, and loop body. The other difference is that the invariant that talks about elements of `src` being unchanged must only quantify over the indices in the range `n <= i < src.Length`.

Answer to Exercise 14.13

```
method CopyMatrix<T>(src: array2<T>, dst: array2<T>)

requires src.Length0 == dst.Length0

requires src.Length1 == dst.Length1

modifies dst

ensures forall i, j :: 0 <= i < dst.Length0 && 0 <= j < dst.Length1 ==>

dst[i, j] == old(src[i, j])

{

forall i, j | 0 <= i < dst.Length0 && 0 <= j < dst.Length1 {

dst[i, j] := src[i, j];

}
```

```
}


method TestHarness() {
var m := new int[2, 1];
m[0, 0], m[1, 0] := 5, 7;
CopyMatrix(m, m);
// the following assertion will not hold if you forget
// ’old’ in the specification of CopyMatrix
assert m[1, 0] == 7;
var n := new int[2, 1];
CopyMatrix(m, n);
assert m[1, 0] == n[1, 0] == 7;
}
```

Answer to Exercise 15.5

Insert the following code before the assignment to `pivot`:

```
var p0, p1, p2 := a[lo], a[(lo + hi) / 2], a[hi - 1];
if {
case p0 <= p1 <= p2 || p2 <= p1 <= p0 =>
a[(lo + hi) / 2], a[lo] := a[lo], a[(lo + hi) / 2];
case p1 <= p2 <= p0 || p0 <= p2 <= p1 =>
a[hi - 1], a[lo] := a[lo], a[hi - 1];
case p2 <= p0 <= p1 || p1 <= p0 <= p2 =>
// nothing to do
}
```

Answer to Exercise 15.7

Here are three hints. First, here is a good specification:

```
requires IsSorted(a, 0, a.Length) && IsSorted(b, 0, b.Length)
ensures fresh(c) && c.Length == a.Length + b.Length
ensures IsSorted(c, 0, c.Length)
ensures multiset(c[..]) == multiset(a[..]) + multiset(b[..])
```

You'll have to define the predicate `IsSorted`. The postcondition **`fresh`**`(c)` lets the caller know that the returned array is independent of any array that the caller may have already had. The condition `c.Length == a.Length + b.Length` follows from the last postcondition, but the proof requires induction, so callers will probably appreciate the condition being stated directly in the postcondition.

Second, for the implementation, use the *replace constant by variable* Loop Design Technique 12.0 to replace the implicit upper-bound constants `a.Length`, `b.Length`, and `c.Length` in `a[..]`, `b[..]`, and `c[..]` by variables, say, `i`, `j`, and `k`. As the last line of your implementation, you will need to help the verifier with the hint

```
assert a[..i] == a[..] && b[..j] == b[..] && c[..k] == c[..];
```

Third, in your loop invariant, you will need a condition similar to the `SplitPoint` predicate we used for Selection Sort and Quicksort. I suggest something like

```
predicate Below(a: seq<int>, b: seq<int>) {

forall i, j :: 0 <= i < |a| && 0 <= j < |b| ==> a[i] <= b[j]

}
```

Good luck!

Answer to Exercise 16.8

```
method RemoveGrinder() returns (grinder: Grinder)

requires Valid()

modifies Repr

ensures Valid() && fresh(Repr - old(Repr))

ensures grinder.Valid() && grinder in old(Repr) - Repr

{

grinder := g;

g := new Grinder();

Repr := Repr + {g} - {grinder};

}
```

Answer to Exercise 17.0

Here is a good export set for the module:

```
export

reveals LazyArray

provides LazyArray.Elements, LazyArray.Repr

provides LazyArray.Valid

provides LazyArray.Get, LazyArray.Update
```

Answer to Exercise 17.3

The body of the constructor is

```
M, Repr, root := map[], {this}, null;
```

Answer to Exercise 17.7

The body of `BinarySearchTree.Add` is

```
if root == null {
root := new Node(key, value);
} else {
root.Add(key, value);
}
M := root.M;
Repr := Repr + root.Repr;
```

Answer to Exercise 17.10

Without knowing the node is valid, nothing would be known about the relation between the fields of **`this`** (which is needed, for example, to prove the postcondition `k in M.Keys`), or the relation between the fields of **`this`** and the fields of `left` (which is needed, for example, to prove termination of the recursive call).

References