Chapter 14

Modifying Arrays

The many array programs we studied in the previous chapter have in common that they don't change the contents of the arrays. That's about to, ahem, change.

Because heap-allocated storage is accessed via references, the storage itself is not passed as parameters to functions and methods. Specifications nevertheless need to be able to identify the relevant pieces of the heap. For this purpose, we use a *frame*.

For now, our frames are going to be very simple. So simple that I'll postpone a general presentation of frames until Chapter 16. In Section 14.0, I'll tell you what you need to know about frames in this chapter and the next. In Section 14.1, we'll practice writing simple programs that mutate arrays.

14.0.Simple Frames

This section gives a quick guide to working with frames for simple array programs.

Copyright 2023. The MIT Press.
All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 6/22/2023 5:12 AM via UNIVERSITTSBIBLIOTHEK MARBURG
AN: 3320892 ; K. Rustan M. Leino.; Program Proofs
Account: s4375181.main.ehost

14.0.0.Modifies clauses

Consider the `Min` method in Section 13.3.0. Our intention was that `Min` would return the minimum of the *given* array elements. But what if the method implementation were to *change* the value of every array element to, say, `1330` and then return the value `1330`? Upon return from such an implementation, the expression given in `Min`'s **`ensures`** clause would hold. That is, when the method returns, the minimum of the array really *is* `1330`!  Does our specification really allow such an implementation?

Luckily, no. Part of what a method specification does is describe what the method is allowed to modify and what it must leave unchanged. This is expressed by the method's *write frame*, which you specify using a **`modifies`** clause. Because the `Min` method has no **`modifies`** clause (stated differently, it has an *empty* **`modifies`** clause), it is not allowed to make modifications to the given array.

The use of frames is governed by a set of rules that I will refer to as *frame bylaws*. Here is the first frame bylaw:

> If a method changes the elements of an array `a` given as a parameter, its specification must include **`modifies`** `a`.

Here is an example:

```
method SetEndPoints(a: array<int>, left: int, right: int)
 requires a.Length != 0
 modifies a
{
 a[0] := left;
 a[a.Length - 1] := right;
}
```

If no **`modifies`** clause were given, then the body of this method would not live up to its contract. The verifier would complain that the method modifies the contents of an array that is not disclosed in the method's **`modifies`** clause.

It's the array referenced by `a` that is allowed to be changed, according to the specification **`modifies`** `a`. For example,

```
method Aliases(a: array<int>, b: array<int>)
 requires 100 <= a.Length
 modifies a
{
 a[0] := 10;
 var c := a;
 if b == a {
```

EBSCOhost - printed on 6/22/2023 5:12 AM via UNIVERSITTSBIBLIOTHEK MARBURG. All use subject to https://www.ebsco.com/terms-of-use

```
  b[10] := b[0] + 1;
 }
 c[20] := a[14] + 2;
}
```

is correct, because at the time `b[10]` and `c[20]` are updated, it is known that these refer to elements of array `a`. Without the **`if`** `b == a` test, however, the assignment to `b[10]` would give rise to a verification error.

> **Exercise 14.0.**
>
> Try adding the postcondition
>
> **`ensures`** `a[0] == left && a[a.Length - 1] == right`
>
> to method `SetEndPoints`. Adjust the method's precondition (as little as possible) to make the method verify.

> **Exercise 14.1.**
>
> Add **`modifies`** `a` to the specification of method `Min` from Section 13.3.0 and then implement the method as the "make it `1330`!" solution I sketched above.

14.0.1.Old values

Suppose you want to write the postcondition for a method that increments the elements of a given array. You must then express that the values of the elements on return from the method are larger than the values of the elements on entry to the method. In other words, the postcondition needs to refer to both the pre-state and post-state of the method. For this reason, a method postcondition is a *two-state predicate*.

An expression `E` in a method's **`ensures`** clause ordinarily refers to the value of `E` in the method's post-state. To refer to the value of `E` in the method's pre-state, you write **`old`**`(E)`. For example, the **`modifies`** clause of the following method says that the array's elements may be modified, but the postcondition restricts those modifications to ones that increment `a[4]`, do not increment `a[6]`, and leave `a[8]` unchanged:

```
method UpdateElements(a: array<int>)
 requires a.Length == 10
 modifies a
 ensures old(a[4]) < a[4]
 ensures a[6] <= old(a[6])
 ensures a[8] == old(a[8])
{
 a[4], a[8] := a[4] + 3, a[8] + 1;
 a[7], a[8] := 516, a[8] - 1;
```

EBSCOhost - printed on 6/22/2023 5:12 AM via UNIVERSITTSBIBLIOTHEK MARBURG. All use subject to https://www.ebsco.com/terms-of-use

```
}
```

To be more precise than I was above, **old** affects only the heap dereferences in its argument. For example, in

```
method OldVsParameters(a: array<int>, i: int) returns (y: int)
 requires 0 <= i < a.Length
 modifies a
 ensures old(a[i] + y) == 25
```

the variables a, i, and y are unaffected by the enclosing **old**(_). That is, in-parameters a and i denote their values on entry to the method and out-parameter y denotes its value on exit from the method, just as if **old** were not used. Only the heap dereference denoted by the square brackets is interpreted in the pre-state of the method.

A common mistake is to use **old** with an unintended argument. The postcondition of the following method illustrates:

```
method Increment(a: array<int>, i: int)
requires 0 <= i < a.Length
modifies a
 ensures a[i] == old(a)[i] + 1 // common mistake
{
a[i] := a[i] + 1; // error: postcondition violation
}
```

Since there are no heap dereferences in the argument to **old**, the attempted expression **old**(a) would mean the same thing as just a.

> **Exercise 14.2.**
>
> Move the parentheses of **old** in the specification of Increment, so that its body will establish the postcondition.

> **Exercise 14.3.**
>
> Write an implementation for OldVsParameters that satisfies the specification.

By the way, only the method's **ensures** clause is a two-state predicate. The **requires**, **modifies**, and **decreases** clauses are always interpreted in the method's pre-state.

14.0.2.New arrays

The purpose of a **modifies** clause for a method M is to make it clear which pieces of the caller's state may be changed by M. This does not apply to arrays that are allocated by M, since a caller has no access to such arrays before calling M.

EBSCOhost - printed on 6/22/2023 5:12 AM via UNIVERSITTSBIBLIOTHEK MARBURG. All use subject to https://www.ebsco.com/terms-of-use

A method is allowed to allocate a new array and change the elements of that array without mentioning this array in the **`modifies`** clause.

For example,

```
method NewArray() returns (a: array<int>)
 ensures a.Length == 20
{
 a := new int[20];
 var b := new int[30];
 a[6] := 216;
 b[7] := 343;
}
```

is correct, even without a **`modifies`** clause.

14.0.3.Fresh arrays

Consider a caller of `NewArray()` above:

```
method Caller() {
 var a := NewArray();
 a[8] := 512; // error: modification of a's elements not allowed
}
```

In order for `Caller` to modify the elements of `a`, it must prove that `a` is in `Caller`'s **`modifies`** clause or prove that `a` has been allocated since the time of entry to `Caller`. We could only hope to do the latter. To do that, we must strengthen the specification of `NewArray` to promise that the array returned by `NewArray` is allocated inside `NewArray`. Knowing that, it is then clear that the array is allocated on behalf of `Caller`.

Such a specification is written using the **`fresh`** predicate, as follows:

```
method NewArray() returns (a: array<int>)
 ensures fresh(a) && a.Length == 20
```

With this specification of `NewArray`, method `Caller` verifies.

14.0.4.Reads clauses

A function cannot modify anything, so it does not have a write frame. However, a function has a *read frame*, which announces the function's dependencies on the heap and is specified using a **`reads`** clause. This dependency information is used to determine if various mutations of the heap affect the value of the function (which is especially important if the body of the function is not available or if the function is recursive).

EBSCOhost - printed on 6/22/2023 5:12 AM via UNIVERSITTSBIBLIOTHEK MARBURG. All use subject to https://www.ebsco.com/terms-of-use

If a function accesses the elements of an array `a`, its specification must include **`reads`** `a`.

The following function depends on the values of the given array `a`, so it must include the **`reads`** `a` specification:

```
predicate IsZeroArray(a: array<int>, lo: int, hi: int)
 requires 0 <= lo <= hi <= a.Length
 reads a
 decreases hi - lo
{
 lo == hi || (a[lo] == 0 && IsZeroArray(a, lo + 1, hi))
}
```

Note that **`reads`** clauses express which parts of the mutable heap a function depends on. Thus, the elements of a sequence and the properties of a datatype value can be inspected by a function without the need for a **`reads`** clause (which also explains why we never saw any **`reads`** clauses in Parts 0 and 1 of the book). For example, the sequence version of `IsZeroArray` does not need a **`reads`** clause:

```
predicate IsZeroSeq(a: seq<int>, lo: int, hi: int)
 requires 0 <= lo <= hi <= |a|
 decreases hi - lo
{
 lo == hi || (a[lo] == 0 && IsZeroSeq(a, lo + 1, hi))
}
```

14.1.Basic Array Modification

Let's consider some illustrative examples that modify arrays in simple ways.

14.1.0.Initializing an array

We'll write a method that sets all array elements to a given value. Here is the method specification:

```
method InitArray<T>(a: array<T>, d: T)
 modifies a
 ensures forall i :: 0 <= i < a.Length ==> a[i] == d
```

To implement this method, we specify a loop, obtaining the invariant by replacing the constant `a.Length` in the postcondition by the loop index for our loop:

```
{
var n := 0;
```

EBSCOhost - printed on 6/22/2023 5:12 AM via UNIVERSITTSBIBLIOTHEK MARBURG. All use subject to https://www.ebsco.com/terms-of-use

```
while n != a.Length
   invariant 0 <= n <= a.Length
   invariant forall i :: 0 <= i < n ==> a[i] == d
}
```

It's been a while since we practiced working backward from the loop invariant over the loop-index update `n := n + 1` (see Chapters 2, 11, and 12). Let's do it here for this simple program.

```
{{ (forall i :: 0 <= i < n ==> a[i] == d) && a[n] == d }}
{{ forall i :: 0 <= i < n + 1 ==> a[i] == d }}
n := n + 1
{{ forall i :: 0 <= i < n ==> a[i] == d }}
```

This calculation immediately tells us that we need to set `a[n]` to `d` before the update to `n`. So, here is the loop body that completes our method implementation:

```
{
 a[n] := d;
 n := n + 1;
}
```

14.1.1.Initializing a matrix

Let's repeat the `InitArray` method, but this time for a matrix:

```
method InitMatrix<T>(a: array2<T>, d: T)
 modifies a
 ensures forall i, j ::
  0 <= i < a.Length0 && 0 <= j < a.Length1 ==> a[i,j] == d
```

The postcondition quantifies over both dimensions of indices of `a`. This suggests that we need two loop indices. We've seen many programs like that in the previous chapter, but in those examples, we were lucky enough to get away with a single loop that, in some way, traversed through a path of index values. Here, there are no shortcuts—we must go through each of the quadratically many index pairs. That suggests using two loops, one nested inside the other.

Our outer loop will have the effect of initializing a whole row with each iteration. In other words, after `m` iterations, `m` rows will be initialized. This is reflected in the loop invariant, which we obtain by replacing the constant `a.Length0` in the postcondition with the loop index `m` (Loop Design Technique 12.0):

```
{
 var m := 0;
 while m != a.Length0
  invariant 0 <= m <= a.Length0
```

EBSCOhost - printed on 6/22/2023 5:12 AM via UNIVERSITTSBIBLIOTHEK MARBURG. All use subject to https://www.ebsco.com/terms-of-use

```
  invariant forall i, j ::
   0 <= i < m && 0 <= j < a.Length1 ==> a[i,j] == d
}
```

For this program, too, let's work backward from the loop invariant over the update of the loop index:

```
{{ (forall i, j :: 0 <= i < m && 0 <= j < a.Length1 ==> a[i,j] == d) &&
(forall j :: 0 <= j < a.Length1 ==> a[m,j] == d) }}
// apply One-Point Rule on second quantifier
{{ (forall i, j :: 0 <= i < m && 0 <= j < a.Length1 ==> a[i,j] == d) &&
(forall i, j :: i == m && 0 <= j < a.Length1 ==> a[i,j] == d) }}
// apply Range Split
{{ forall i, j :: 0 <= i < m + 1 && 0 <= j < a.Length1 ==> a[i,j] == d }}
m := m + 1
{{ forall i, j :: 0 <= i < m && 0 <= j < a.Length1 ==> a[i,j] == d }}
```

The top annotation shows two quantifiers, which thus become the postcondition of our inner loop. It's easy to forget that the inner loop needs to establish *both* of these quantifiers, not just the `forall j` quantifier. If we only mention row `m` in the invariant of the inner loop, then that loop specification would allow any modification whatsoever of the other rows.

Heap-related loop frames

The root cause of the issue I just mentioned is loop frames, that is, the (usually implicit) part of the loop specification that says what the loop is allowed to modify. I mentioned loop frames before in the context of local variables (Sections 11.0.5 and 11.1.3), but here we also need to consider what a loop specification says about heap-allocated storage, like the elements of arrays.

Loops have the notion of a **`modifies`** clause for heap-allocated storage, just like methods do. However, it is rare that the **`modifies`** clause of a loop is any different than the **`modifies`** clause of the enclosing method (or enclosing loop, in case of nested loops). Therefore, unless a loop mentions an explicit **`modifies`** clause, the default **`modifies`** clause of the loop is that of the enclosing method (or enclosing loop).

The inner loop

Back to our matrix initialization method. Our working-backward calculation above concluded that the inner loop must establish the two quantifiers shown in the top annotation. The first quantification (`forall i, j`) already holds on entry to the inner loop, so all the inner loop needs to do is maintain this condition. We'll apply the *always* Loop Design Technique 13.1 to it. For the second quantification (`forall j`), we'll replace the constant `a.Length1` with a loop index `n`. What we get is:

```
{
   var n := 0;
   while n != a.Length1
```

EBSCOhost - printed on 6/22/2023 5:12 AM via UNIVERSITTSBIBLIOTHEK MARBURG. All use subject to https://www.ebsco.com/terms-of-use

```
    invariant 0 <= n <= a.Length1
    invariant forall i, j ::
     0 <= i < m && 0 <= j < a.Length1 ==> a[i,j] == d
    invariant forall j :: 0 <= j < n ==> a[m,j] == d
```

It's easy to fill in the rest of the method implementation:

```
   {
    a[m,n] := d;
    n := n + 1;
   }
   m := m + 1;
}
```

14.1.2.Incrementing the values in an array

I'll show two more examples at this level of diculty, to make sure you master this basic technique. In this next example, the method increments every element of a given array by `1`. Here is the method specification:

```
method IncrementArray(a: array<int>)
 modifies a
 ensures forall i :: 0 <= i < a.Length ==> a[i] == old(a[i]) + 1
```

Note that the **`old`** expression encloses the expression `a[i]`, which means it denotes element `i` of array `a` on entry to the method. Remember from Section 14.0.1, an expression like **`old`**`(a)[`**`old`**`(i)]` denotes a different value—since **`old`** has no effect on parameters and bound variables, it is the same as just `a[i]`. It is the square brackets that dereference the heap, so it is the square brackets that we want to apply **`old`** to, and we do that by writing **`old`**`(a[i])`.

If we think of replacing the constant `a.Length` with a loop index `n`, we would write the loop as follows:

```
{
 var n := 0;
 while n != a.Length
  invariant 0 <= n <= a.Length
  invariant forall i :: 0 <= i < n ==> a[i] == old(a[i]) + 1
 {
  a[n] := a[n] + 1;
  n := n + 1;
 } // error: second loop invariant not maintained by loop body
```

EBSCOhost - printed on 6/22/2023 5:12 AM via UNIVERSITTSBIBLIOTHEK MARBURG. All use subject to https://www.ebsco.com/terms-of-use

```
}
```

However, in this form, we cannot maintain the invariant. The loop body sets `a[n]` to `a[n] + 1`, and in order to maintain the invariant, we need to know

```
a[n] + 1 == old(a[n]) + 1
```

The loop frame is the same as for the enclosing method, so as far as the loop specification is concerned, the elements of `a` can have any values allowed by the invariant. The invariant above only talks about the array elements `a[..n]`, so there is no information about the relation between the current value of `a[n]` and the value of `a[n]` on entry to the loop.

Debugging the verification

To realize this is the problem, you must remember to look only at the loop specification and not be tempted to draw any other conclusions about what states iterations of the loop body may reach (Chapter 11). If you still don't see what the problem is, I suggest you debug the problem in the following way.

Start by adding an **`assert`** statement with the violated invariant at the place where it is supposed to hold. Your loop body will now look like

```
a[n] := a[n] + 1;

n := n + 1;

assert forall i :: 0 <= i < n ==> a[i] == old(a[i]) + 1; // error
```

This should also have had the effect of trading the error message on the invariant for an error message on the assertion. That's what we'd expect, because if we could prove this condition here, then the invariant would be fine, too.

Next, move—or copy, if you wish—this assertion to before the assignment to `n`, and in doing so, replace `n` by `n + 1`, as when you're computing the weakest precondition. You'll then have

```
a[n] := a[n] + 1;

assert forall i :: 0 <= i < n + 1 ==> a[i] == old(a[i]) + 1; // error

n := n + 1;
```

This assertion before the assignment to `n` is the same as the assertion we had after the assignment to `n` a moment ago. Therefore, the verifier should now be complaining about this assertion and nothing else. (If you find otherwise, then you should focus your debugging on why that is.)

Next, let's manually split this quantifier. The range is `0 <= i < n + 1`, so we split it into the disjunction of `0 <= i < n` and `i == n`. We apply the One-Point Rule to the latter. (See Section 13.1.5 or Appendix B if you need a reminder about these quantifier rules.) Showing the result as two additional assertions, we now have

```
a[n] := a[n] + 1;

assert forall i :: 0 <= i < n ==> a[i] == old(a[i]) + 1;

assert a[n] == old(a[n]) + 1; // error

assert forall i :: 0 <= i < n + 1 ==> a[i] == old(a[i]) + 1;
```

EBSCOhost - printed on 6/22/2023 5:12 AM via UNIVERSITTSBIBLIOTHEK MARBURG. All use subject to https://www.ebsco.com/terms-of-use

```
n := n + 1;
```

The verifier now shows the middle assertion as the one it cannot prove. Since the preceding assignment statement just gave a new value to `a[n]`, we expect `a[n]` in the violated assertion to have the value that the right-hand side of the assignment (that is, `a[n] + 1`) had just before the assignment. Let's add an assertion to that effect:

```
assert a[n] + 1 == old(a[n]) + 1; // error

a[n] := a[n] + 1;

assert forall i :: 0 <= i < n ==> a[i] == old(a[i]) + 1;

assert a[n] == old(a[n]) + 1;

assert forall i :: 0 <= i < n + 1 ==> a[i] == old(a[i]) + 1;

n := n + 1;
```

The verifier now complains about the first of the assertions, but about nothing else. This tells us that if we can only establish that condition at the point of the assertion, then the verification goes through.

I think the failing assertion we ended up with clearly points out the missing information, namely the need to know `a[n] == old(a[n])` at the beginning of the loop. Now, we think about the whole loop for a minute and realize that array elements at index `n` and higher have not been changed by the loop. To add this fact to the verification process, we incorporate it into the invariant:

```
invariant forall i :: n <= i < a.Length ==> a[i] == old(a[i])
```

This causes all the verifier's complaints to go away. In other words, the new invariant is checked to hold on entry to the loop, all of the assertions are checked to hold, and all the invariants can now be proved to hold at the end of the loop body. Since the **`assert`** statements were added just to help us diagnose the situation, they have served their purpose, so we can remove them.

> **Exercise 14.4.**
>
> Specify and implement a method that increases every element of a matrix by 1.

14.1.3.Copying an array

Let's copy the elements of one array into another.

```
method CopyArray(src: array, dst: array)
 requires src.Length == dst.Length
 modifies dst
 ensures forall i :: 0 <= i < src.Length ==> dst[i] == old(src[i])
```

This method takes two arrays, one of which is listed in the **`modifies`** clause. Clearly, this gives the method license to modify the elements of `dst`. More precisely, the elements of the array referenced by `dst` can be modified. If `src` happens to reference the same array as `dst`—that is, if `src == dst`—then it is also true that the method can modify the elements of the array reference by `src`. As I remarked before (in Section 13.0.2), a **`modifies`** clause says which arrays are allowed to be modified, but does not care what expression you use to reference those arrays.

EBSCOhost - printed on 6/22/2023 5:12 AM via UNIVERSITTSBIBLIOTHEK MARBURG. All use subject to https://www.ebsco.com/terms-of-use

In the event that `src` and `dst` reference the same array, a postcondition like

```
ensures forall i :: 0 <= i < src.Length ==> dst[i] == src[i]
```

would trivially hold even if the method completely altered the contents of the array, because this postcondition would simply say that the post-value of each element (of the one array we're dealing with) is equal to itself. That's why we write the specification to say **old**`(src[i])`. (Alternatively, we could have added the precondition `src != dst`.)

Having settled on that method specification, the loop specification and loop body follow patterns we've seen before.

Note that, akin to the situation we debugged in Section 14.1.2, we need to be sure to include the loop invariant that says the elements of `src` are unchanged in order to handle the situation where `src == dst`.

```
{
 var n := 0;
 while n != src.Length
  invariant 0 <= n <= src.Length
  invariant forall i :: 0 <= i < n ==> dst[i] == old(src[i])
  invariant forall i :: 0 <= i < src.Length ==> src[i] == old(src[i])
 {
  dst[n] := src[n];
  n := n + 1;
 }
}
```

Practice writing array modifications using loops in the following exercises:

Exercise 14.5.

Add `src != dst` as a precondition. Then, simplify the postcondition and invariants as much as possible (without changing the meaning of the postcondition).

Exercise 14.6.

Change **old**`(src[i])` to `src[i]` in the postcondition of `CopyArray`, as hypothetically considered above. Then, change the implementation of `CopyArray` so that it makes all elements of src the same if `src == dst`.

Exercise 14.7.

Specify and implement a method that copies the elements of one matrix to another of the same dimensions. Allow the source and destination matrices to be the same and make sure your specification says the right thing in that case. (Write a little test harness if you're uncertain.)

EBSCOhost - printed on 6/22/2023 5:12 AM via UNIVERSITTSBIBLIOTHEK MARBURG. All use subject to https://www.ebsco.com/terms-of-use

Exercise 14.8.

Specify and implement a method

```
method DoubleArray(src: array<int>, dst: array<int>)
```

that sets `dst[i]` to `2 * src[i]` for each `i`. Assume the given arrays have the same lengths, and allow the possibility that they reference the same array.

Exercise 14.9.

Specify and implement a method that reverses the elements of a given array.

Exercise 14.10.

Specify and implement a method that takes a square matrix and transposes its elements.

Exercise 14.11.

Specify and implement a method that in a given array rotates the elements “left”. That is, what used to be stored in `a[(i + 1) % a.Length]` gets stored in `a[i]`.

Exercise 14.12.

Specify and implement a method that in a given array rotates the elements “right”. That is, what used to be stored in `a[(i - 1) % a.Length]` gets stored in `a[i]`.

14.1.4.Loop-less array operations

Loops like the ones I showed in this section illustrate how you work with quantified loop invariants, and they provide useful practice for doing more complicated loops. Beyond that, loops like these are quite tedious—they are conceptually simple and yet require loop invariants that come across as far more complicated than the loop bodies that they are describing.

If you reflect on it, you realize that loops are the wrong programming construct for doing these simple, uniform operations. Some programming languages don't have anything better to offer, but others do. I'll show you two such constructs that are found in Dafny.

Array constructor

When allocating an array, you can give a function that specifies the initial elements, like

```
a := new int[25](F);
```

where `F` is a function from the indices of the new array (from `0` to `25` in this example) to the desired element values of the new array (of type **`int`** in this example).

Most often, the function `F` is given as an *inline function*, often known as a *lambda expression*. It has the form `x => E` where `x` is the formal parameter and expression `E` is the body of the function. For example,

```
a := new int[25](i => i * i);
```

EBSCOhost - printed on 6/22/2023 5:12 AM via UNIVERSITTSBIBLIOTHEK MARBURG. All use subject to https://www.ebsco.com/terms-of-use

allocates an array of length `25` whose elements are initialized to the first `25` squares. Another example is the commonly occurring

```
a := new int[n](_ => 0);
```

which allocates an array of length `n` where each element is initialized to `0`. As the example shows, if the formal parameter of the inline function is not used in the body of the inline function, you can omit a specific name and instead write _.

For multi-dimensional arrays, you use a function with more than one parameter. Using an inline function for that purpose, here is an example that initializes all elements of a new `50` by `50` matrix with `d` along the diagonal and `0` everywhere else:

```
m := new int[50, 50]((i, j) => if i == j then d else 0);
```

Note that this inline function puts the list of parameters in parentheses—in the special case of one parameter, the parentheses can be omitted, as I showed in the examples above.

Inline functions with specifications To allocate an array as a copy of another, you would write something like:

```
b := new T[a.Length](i => a[i]); // error in accessing a
```

However, the verifier complains about this, because it checks the well-formedness of the inline function without regard to where the function is being used. In particular, for `i => a[i]` to be a well-formed function, we need to both make sure `i` is in range and that `a` is allowed by the frame of the function. If we declared a named function, this would be clear to us (we've seen function preconditions throughout the book and I introduced read frames in Section 14.0.4). Inline functions can also have **`requires`** and **`reads`** clauses, and we need to use both to do the copying example. Here is the correct form, which the verifier accepts:

```
b := new T[a.Length](i requires 0 <= i < a.Length reads a =>
                  a[i]);
```

This is long, but it still saves us the hassle of writing a loop and inventing a loop invariant.

Sequence constructors There is a similar construct for constructing a sequence. The expression

```
seq(500, i => i % 2 == 0)
```

denotes a sequence of `500` booleans, where the elements behind even indices are **`true`** and the elements behind odd indices are **`false`**.

Remember that sequences are values whereas arrays are references to mutable elements. Accordingly, the sequence constructor is an expression, not an allocation statement.

Aggregate statements

Dafny has an *aggregate statement* that can be used to update array elements. You can think of the aggregate statement as a generalization of simultaneous assignment. So, whereas

```
a[5], a[7] := a[5] + 25, a[7] + 49;
```

updates two array elements, the aggregate statement

EBSCOhost - printed on 6/22/2023 5:12 AM via UNIVERSITTSBIBLIOTHEK MARBURG. All use subject to https://www.ebsco.com/terms-of-use

```
forall i | 0 <= i < a.Length {
 a[i] := a[i] + i * i;
}
```

updates all array elements in a similar way.

Despite the syntactic similarity with the **`for`** and **`foreach`** statements in languages like Java or C#, the crucial point about the **`forall`** statement is that *it is not a loop*. This has two consequences.

One consequence is that the body of the **`forall`** statement is performed simultaneously for all values of the bound variable. Certain computations are easy to formulate in this way. For example,

```
forall i | 0 <= i < a.Length && i % 2 == 0 {
 a[i] := a[i] + 1;
}
```

increments even-indexed elements by `1`, the `IncrementArray` method in Section 14.1.2 can be implemented as just

```
forall i | 0 <= i < a.Length {
 a[i] := a[i] + 1;
}
```

and

```
forall i | 0 <= i < a.Length {
 a[i] := a[(i + 1) % a.Length];
}
```

rotates all the elements of a to the left (which Exercise 14.11 asked you to do with a loop). However, it is not possible to do something like sum the elements of an array, since this cannot be done as one simultaneous operation over all indices.

The other consequence of the **`forall`** statement not being a loop is that reasoning about it does not require a loop invariant! For example, you can say goodbye to the long and boring loop invariants of the nested loops in Section 14.1.1 and instead simply write

```
forall i, j | 0 <= i < a.Length0 && 0 <= j < a.Length1 {
 a[i,j] := d;
}
```

To make compilation work out without too many complications, there are some restrictions on the **`forall`** statement. The most noticeable of these is that the body must contain just one assignment statement.

In the future, I will use a **`forall`** statement when possible, but we'll see plenty of programs where iteration is really needed. For those, you'll be glad to have mastered the technique of writing loop invariants on the simpler programs first.

EBSCOhost - printed on 6/22/2023 5:12 AM via UNIVERSITTSBIBLIOTHEK MARBURG. All use subject to https://www.ebsco.com/terms-of-use

Exercise 14.13.

Do Exercise 14.7 using a **`forall`** statement.

Exercise 14.14.

Do Exercise 14.8 using a **`forall`** statement.

Exercise 14.15.

Do Exercise 14.9 using a **`forall`** statement.

Exercise 14.16.

Do Exercise 14.10 using a **`forall`** statement.

14.2.Summary

In this chapter, I introduced *read and write frames*, which specify which parts of the heap a function may depend on and which parts of the heap a method may modify, respectively. Read frames are declared with **`reads`** clauses and write frames with **`modifies`** clauses, and both of them denote a set of references into the heap.

I also introduced some features that are useful in specifications when array elements or other object fields are modified in the heap. **`old`**`(E)` denotes the value of `E` in the prestate of the method, **`fresh`**`(E)` says that the reference (or set of references) `E` has been allocated since entry to the enclosing method.

It is important to understand how to specify and write loops that set array elements in various ways. I gave a number of examples that initialize arrays or perform simple updates on arrays. These highlight the importance of specifying what has changed and what hasn't changed in the heap.

Finally, I showed some language features that can eliminate the need to write loops in the first place.

In the next chapter, we'll consider some programs that manipulate arrays in more interesting ways.

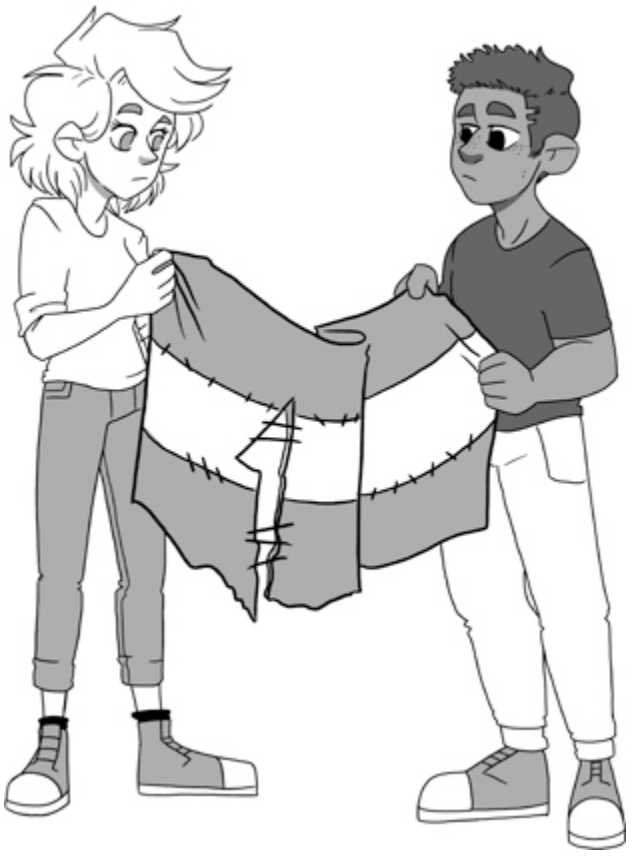
Notes

The design of the pioneering programming language Euclid [74] targeted the construction of verifiable programs. It featured declarations of pre- and postcondition, inline assertions, and module invariants. In order for a procedure to access a variable declared outside the procedure, the procedure had to explicitly *import* the variable. Such an import clause also indicated if the procedure might modify the variable or only read it.

EBSCOhost - printed on 6/22/2023 5:12 AM via UNIVERSITTSBIBLIOTHEK MARBURG. All use subject to https://www.ebsco.com/terms-of-use

Chapter 15

In-situ Sorting



In Chapter 8, I covered specifications and programs for sorting inductive lists. Since lists are immutable, those programs necessarily returned sorted versions of their inputs, without modifying the input. In this chapter, we'll consider programs that reorder the elements of a given array to make it sorted. That is, the sorting is done *in situ*, Latin for “in place”.

15.0.Dutch National Flag

I'll start us with a problem of sorting an array with three kinds of values. The values are the three colors red, white, and blue, and the order in which we'll sort them is what has given this problem its name, the Dutch National Flag (first red, then white, then blue).

We'll start by introducing the colors and a way to compare them.

```
datatype Color = Red | White | Blue



ghost predicate Below(c: Color, d: Color) {
```

EBSCOhost - printed on 6/22/2023 5:12 AM via UNIVERSITTSBIBLIOTHEK MARBURG. All use subject to https://www.ebsco.com/terms-of-use