Rotation of Sequences: Algorithms and Proofs

Carlo A. Furia
ETH Zurich, Switzerland
http://bugcounting.net
20 June 2014

Abstract

Sequence rotation consists of a circular shift of the sequence’s elements by a given number of positions. We present the four classic algorithms to rotate a sequence; the loop invariants underlying their correctness; detailed correctness proofs; and fully annotated versions for the Boogie verifier. The presentation illustrates in detail both how the algorithms work and what it takes to carry out mechanized proofs of their correctness.
# Contents

1 Introduction 3

2 Modular arithmetic and sequences 3

3 Rotation: the problem 5

4 Rotation: the algorithms 5

4.1 Rotation by copy 6

4.1.1 Rotation by copy: computational complexity 6

4.1.2 Rotation by copy: correctness 8

4.2 Rotation by reversal 8

4.2.1 Rotation by reversal: computational complexity 9

4.2.2 Rotation by reversal: correctness 10

4.3 Rotation by swapping 11

4.3.1 Rotation by swapping: computational complexity 13

4.3.2 Rotation by swapping: correctness 15

4.4 Rotation by modular visit 18

4.4.1 Rotation by modular visit: complexity 21

4.4.2 Rotation by modular visit: correctness 21

5 Rotation: mechanized proofs in Boogie 24

5.1 Sequences and rotated sequences 25

5.2 Rotation by copy: mechanized proofs as on paper 26

5.3 Rotation by reversal: mechanized lemmas 27

5.3.1 Axioms and lemmas about reversal 27

5.3.2 Mechanized proofs of reversal and rotation by reversal 29

5.4 Rotation by swapping: organizing code for proofs 31

5.4.1 Mechanized proof of swapping sections 31

5.4.2 Lemmas about swapping 31

5.4.3 Mechanized proof of rotation by swapping: recursive version 34

5.4.4 Mechanized proof of rotation by swapping: iterative version 37

5.5 Rotation by modular visit: ghost code and framing 41

5.5.1 Axioms about cycles 41

5.5.2 Outer loop 42

5.5.3 Inner loop 44

5.5.4 Proof conclusion 47

6 Discussion 49
1 Introduction

Rotating a sequence, typically represented by an array, is the problem of shifting all its elements in one direction by a fixed number of positions while cyclically wrapping over the sequence’s bounds. Rotations have various practical applications in programming; a classic example is in editors, to arrange the lines of text stored as sequences of characters.

There is an interesting choice of algorithms to rotate a sequence, exploring different trade-offs between performance, code complexity, and resource usage. The algorithms themselves have been part of computer science folklore knowledge for a long time [2, Sec. 2.3]; they are an excellent subject to discuss algorithm design and analysis [3].

Proving correctness of these algorithms is a challenging task too, even more so if we aim for (automated) mechanized proofs. The only document formally discussing how to establish the correctness of the rotation algorithms is a technical report by Gries and Mills [7]. They sketch the loop invariants underlying the correctness arguments; the level of detail of their presentation is, however, clearly insufficient to perform complete mechanized proofs, which require to exhaustively deal with issues such as framing and intermediate properties and assertions.

This paper presents each rotation algorithm for rotation first informally, then with detailed loop invariants and lemmas, and finally with all the gory low-level details necessary to carry out automated proofs using the Boogie prover [8]. This presentation can serve as a useful guide to carry out similar correctness proofs of the same algorithms using other automated tools, as well as a tutorial introduction to some idioms (lemma procedures, framing, ghost code, and so on) frequently used in automated verification of full functional correctness.

The code presented in the paper is available online (under directory rotation):

https://bitbucket.org/caf/verified/

2 Modular arithmetic and sequences

A precise definition of the notion of rotation relies on some mathematical concepts that we introduce in this section.

Modular arithmetic. Modular arithmetic makes extensive usage of the ‘mod’ binary operation, which is normally defined [6, Sec. 3.4] in terms of integer division and floor as

\[ x \mod y = x - y \lfloor x/y \rfloor, \text{ for } y \neq 0. \]  

(1)

This definition is not, however, always the best choice to specify and reason about programs. First, programming languages may implement definitions of ‘mod’ that differ in the sign of the result; for example, \( x \% y \) has the same sign as \( x \) in Java but as \( y \) in Python. Second, [1] relies on two other operations, whereas it would be more convenient to have a direct definition only in terms of basic operators that are universally available.

These considerations justify the introduction of the binary operation ‘wrap’ with recursive definition

\[ x \text{ wrap } y = \begin{cases} x & 0 \leq x < y, \\ (x - y) \text{ wrap } y & 0 < y \leq x. \end{cases} \]  

(2)
We only need to define ‘wrap’ for nonnegative arguments, although we could easily generalize it to handle negative arguments too.

When it is defined, it is easy to see that \( x \text{ wrap} \ y \) is the same as \( x \mod y \). Specifically, we will make use of the property that

\[
0 \leq (x \text{ wrap} \ y) < y, \quad \text{for } x \geq 0 \text{ and } y > 0,
\]

which can be proved by induction on \( x \) (base case: \( x < y \)).

Sequences. Sequences are finite ordered collections of elements, which we denote by sans-serif letters such as \( S \). The length of a sequence \( S \) is denoted by \(|S|\). An element of \( S \) is denoted by \( S[k] \) (also: \( S[k] \)), with \( 0 \leq k < |S| \) denoting the position (or index) of the element in the sequence starting from 0:

\[
S = S_0 S_1 \cdots S_{|S|-1}.
\]

For sequences \( S \) and \( T \) of equal length, \( S = T \) denotes that the two sequences consist of the same elements in the same order. The empty sequence \( \epsilon \) is such that \(|\epsilon| = 0\).

The concatenation \( S \circ T \) of sequences \( S \) and \( T \) is the sequence of length \(|S| + |T|\) obtained by juxtaposing \( S \) and \( T \). We define \( S \circ T \) element-wisely as

\[
(S \circ T)_k = \begin{cases} S_k & 0 \leq k < |S|, \\ T_{k-|S|} & |S| \leq k < |S| + |T|, \end{cases}
\]

for \( 0 \leq k < |S| + |T| \). For example, the concatenation of \( \text{A B} \) and \( \text{C D E F} \) is \( \text{A B C D E F} \).

The reverse \( S^{-1} \) of a sequence \( S \) is the sequence defined by \(|S^{-1}| = |S| = N\) and

\[
S_k^{-1} = S_{N-1-k}, \quad \text{for } 0 \leq k < N.
\]

For example, the reverse of sequence \( \text{A B C D E F} \) is sequence \( \text{F E D C B A} \).

Permutations and cycles. Rotations—introduced in the next section—are a special kind of permutations, that is bijections of a set onto itself; henceforth, \( \langle 0, 1, \ldots, N-1 \rangle \) is the set in question, which we denote \( \langle N \rangle \).

A cycle \( (a_0 a_1 \cdots a_m) \), for \( a_0, a_1, \ldots, a_m \) distinct values in \( \langle N \rangle \), is the permutation \( \lambda : \langle N \rangle \to \langle N \rangle \) such that:

\[
\lambda(a_k) = \begin{cases} a_{k+1} & 0 \leq k < m, \\ a_0 & k = m, \\ a_k & \text{otherwise}. \end{cases}
\]

In other words, a cycle \( (a_0 a_1 \cdots a_m) \) is a permutation that sends \( a_0 \) into \( a_1 \), \( a_1 \) into \( a_2 \), and so on until \( a_m \), which it sends back to \( a_0 \), while leaving all other elements in \( \langle N \rangle \setminus \{ a_0, \ldots, a_m \} \) unchanged. Two cycles \( (a_0 \cdots a_m) \) and \( (b_0 \cdots b_n) \) are disjoint if the intersection of the sets \( \{ a_0, \ldots, a_m \} \) and \( \{ b_0, \ldots, b_n \} \) is empty. A fundamental result of the theory of permutations [10, Th. 1.6] is that every permutation is uniquely expressible as the composition of disjoint cycles (the order of composition does not matter).
3 Rotation: the problem

The rotation $\rho^r S$ of sequence $S$ by $r$ (also called \"$r$-rotation\", \"$r$-circular shift\", or \"$r$-cyclic shift\") is the sequence obtained by shifting all elements in $S$ by $r$ positions while wrapping over the sequence’s bounds. We assume that positive values of $r$ denote shifts to the left; hence the definition

$$
\rho^r S = \begin{cases} 
S_r \cdots S_{|S|} & 0 \leq r < |S| , \\
S_{|S|+r} \cdots S_{|S|+r+1} & -|S| < r \leq 0 .
\end{cases} \quad (6)
$$

For simplicity, we ignore the case of rotations by more than $|S|$ in absolute value (although it is clear they correspond to applications of definition (6)). Notice that $\rho^0$ is the identify mapping. Figure 1 demonstrates the definition on the running example: rotating the 6-element sequence A B C D E F by 2 yields sequence C D E F A B.

![Figure 1](image)

Figure 1: The rotation of sequence A B C D E F of length 6 by 2 (or, equivalently, by $-(6-2) = -4$) yields sequence C D E F A B. Different colors highlight elements of the two subsequences that are swapped.

We can derive from (6) an equivalent element-wise representation of $\rho^r S$ as the sequence such that $|\rho^r S| = |S| = N$ and, for $0 \leq k < N$,

$$
(\rho^r S)_k = \begin{cases} 
S_{(k+r) \text{ mod } N} & 0 \leq r < N , \\
S_{(k+N+r) \text{ mod } N} & -N < r \leq 0 .
\end{cases} \quad (7)
$$

The duality between left rotation and $r$ on one side, and right rotation and $N + r$ on the other side suggests the inverse mapping of (7)

$$
S_k = \begin{cases} 
(\rho^r S)_{(k+N-r) \text{ mod } N} & 0 \leq r < N , \\
(\rho^r S)_{(k-r) \text{ mod } N} & -N < r \leq 0 .
\end{cases} \quad (8)
$$

4 Rotation: the algorithms

There are four main algorithms that compute the rotation of a sequence. We present them in increasing level of complexity, where “complexity” simultaneously refers to complexity of implementation and to complexity of understanding, reasoning about, and proving the correctness of the algorithms—but not computational complexity.

We present the algorithms in pseudo-code. Sequences are represented by arrays of a generic type $G$, indexed from 0. That is, we identify an array $a$ with the sequence $a[0] \ldots a[a.\text{count} - 1]$ of its elements, where $a.\text{count}$ denotes the length of $a$. An array slice $a[\text{low}..\text{high}]$ denotes the sequence $a[\text{low}] \ldots a[\text{high} - 1]$ if
\( \theta \leq \text{low} \leq \text{high} \leq a.\text{count} \), and the empty sequence in all other cases. Correspondingly, equality between slices of equal length corresponds to element-wise equality of their sequences of elements; this notational convention makes it possible to elide explicit quantification without ambiguity:

\[
x(a..b) = y(a..b) \iff \forall i: a \leq i < b \implies x[i] = y[i].
\]

Each algorithm operates on an array \( a \) of length \( N \) (an alias of \( a.\text{count} \)), and on an integer \( r \); it modifies \( a \) in place so that it is rotated by \( r \) when the algorithm terminates. For simplicity, we assume \( 0 < r < N \) as precondition. This is without loss of generality as the right rotation of a sequence \( S \) by some \( r \) such that \( |S| > r > 0 \) coincides with its left rotation by \( |S| - r \), and rotation by 0 is the identity. We also assume that \( a \)'s size does not change, so that \( N \) denotes \( a.\text{count} \) at any point during the computation.

Using this notation, Figure 2 shows the input/output specification of the rotation algorithms, where \( \text{old} \ a \) denotes the content of \( a \) upon calling \( \text{rotate} \).

1. \( \text{rotate} \ (a: \text{ARRAY}[G]; r: \text{INTEGER}) \)
2. \( \text{require} \ 0 < r < N \)
3. \( \text{ensure} \ a = \rho^r(\text{old} \ a) \)

Figure 2: Rotate array \( a \) by \( r \) to the left: specification.

### 4.1 Rotation by copy

A straightforward application of the definition of rotation, the first algorithm (shown in Figure 3) uses a second array \( b \) as scratch space. With two pointers \( s \) (source) and \( d \) (destination), it copies each element \( a[s] \) from \( a \) into \( b \) at its position in \( \rho^r a \). Matching the mapping in (8), initially \( s \) is \( 0 \) and \( d \) is \( N - r \); each loop iteration increments both, and resets \( d \) to 0 when it reaches \( N \) (the first non-valid position in \( a \)). When the loop terminates, \( b \) contains \( \rho^r a \), and its content is copied back into \( a \).

Figure 4 demonstrates some steps of the algorithm \( \text{rotate\_copy} \) on an example. The top left figure represents the state upon first entering the loop, followed by the state after one (top right) and two (bottom right) iterations; the bottom right figure is the state upon exiting the loop.

#### 4.1.1 Rotation by copy: computational complexity

Algorithm \( \text{rotate\_copy} \) takes \( \Theta(N) \) time and \( \Theta(N) \) space\(^2\). For large values of \( N \), the space complexity may be prohibitive. We can save some space by noticing that we only need scratch space for \( d = \min(r, N - r) \) elements, while we can swap the other \( N - d \) elements in place. For example, assuming \( r \leq N - r \) as in the running example, rotation of \( a \) by \( r \) reduces to:

1. \( b[\emptyset..r) := a[\emptyset..r) \quad // \text{copy} \ a[\emptyset..r) \text{ into} \ b \)
2. \( a[\emptyset..N - r) := a[r, N) \quad // \text{copy} \ a[r..N) \text{ to the left by} \ r \text{ in place} \)
3. \( a[N - r..N) := b[\emptyset..r) \quad // \text{copy} \ b \text{ back into} \ a[N - r..N) \)

\(^1\)Alternatively, it could initialize \( s \) to \( r \) and \( d \) to \( \emptyset \) and decrement indices following (7).

\(^2\)In the complexity analyses, \( N \) and \( r \) denote generic values of input length \( N \) and rotation \( r \).
rotate_copy (a: ARRAY[G]; r: INTEGER)

require 0 < r < N
ensure a = ρ(old a)

local b: ARRAY[G]; s, d: INTEGER

do
    b := [N] // initialize to size N
    s := 0; d := N − r
    while s < N do
        invariant 0 ≤ s ≤ N
        d = (s + N − r) wrap N
        ∀ i: 0 ≤ i < s ⇒ a[i] = b[(i + N − r) wrap N]
        b[d] := a[s]
        s, d := s + 1, d + 1
        // wrap over a’s bounds
        if d = N then d := 0 end
    end
    // copy b’s content back into a
    a.copy (b)
end

Figure 3: Rotate array a by r to the left through copy, using b as scratch space.

This takes time Θ(N + d) and space Θ(d), with d ≤ N/2. If we use native memory copy methods (such as Java’s System.arraycopy) this is quite fast in practice but only if enough memory is available. This is the case of the benchmarks reported in Table 1, which ran on a server with a lot of physical RAM: the implementation of reverse_copy using native memory copy methods is consistently the fastest (or very close to the fastest).

The algorithms presented in the following sections improve over the space requirements of the “rotation by copy” algorithm by trading time for space.

Figure 4: Rotating sequence A B C D E F by 2 through copy.
Algorithm Time Space Java Implementation On:

| Algorithm    | Time     | Space   | 3k  | 10k  | 100k | 1000k |
|--------------|----------|---------|-----|------|------|-------|
| copy         | $\Theta(N)$ | $\Theta(N)$ | 78 ms | 540 ms | 47 s | 4324 s |
| copy (native)| $\Theta(N + d)$ | $\Theta(d)$ | 43 ms | 188 ms | 12 s | 1159 s |
| reverse      | $\Theta(N)$ | $\Theta(1)$ | 44 ms | 323 ms | 17 s | 2354 s |
| swap (iterative) | $\Theta(N)$ | $\Theta(1)$ | 42 ms | 221 ms | 12 s | 1138 s |
| modulo       | $\Theta(N)$ | $\Theta(1)$ | 48 ms | 333 ms | 37 s | 5307 s |

Table 1: Time and space complexity of the various algorithms for rotating an array of size $N$ by $r$ (with $d = \min(r, N - r) \leq N/2$). The righthand columns show the times spent by Java 1.7 implementations of the algorithms over arrays of sizes from 3 thousand to 1000 thousand elements (for each $N$, an algorithm runs once for every $0 \leq r < N$). The experiments ran on an Intel Xeon 2.13 GHz server with 10 GB of physical RAM.

4.1.2 Rotation by copy: correctness

A correctness proof for $\text{rotate} \_\text{copy}$ relies on a suitable “essential” loop invariant [5] that characterizes the state of $b$ as reflecting definition (8). For the essential invariant to be well-defined, we first need a bounding invariant that constrains the variability of index $s$ to be within $a$’s bounds:

$$0 \leq s \leq N.$$  \hspace{1cm} (9)

Since $d - s = N - r$ initially, and both $s$ and $d$ are incremented in every iteration (while wrapping over $N$), a corresponding bounding loop invariant about $d$ is

$$d = (s + N - r) \text{\text{wrap}} N,$$  \hspace{1cm} (10)

whose inductiveness directly follows from the definition (2) of ‘\text{wrap}’ by case discussion.

The relation between $d$ and $s$ also suggests the essential invariant that relates the content of $b$ to that of $a$:

$$\forall i : 0 \leq i < s \implies a[i] = b[(i + N - r) \text{\text{wrap}} N].$$  \hspace{1cm} (11)

Its inductiveness is a consequence of the other invariant (9) and of the assignment $b[d] := a[s]$ performed in the loop.

Upon exiting the loop, $s$ equals $N$; hence, (11) asserts that

$$\forall i : 0 \leq i < N \implies a[i] = b[(i + N - r) \text{\text{wrap}} N];$$

that is, $b$ is $\rho^r a$ according to (8), which establishes the postcondition after copying $b$’s content into $a$.

4.2 Rotation by reversal

The rotation by reversal algorithm conjugates simplicity and efficiency in a way that makes it a very effective solution in practice. In fact, it has been used in numerous text editors to reshuffle lines of text; Bentley reports usage as early as 1971—according to Ken Thompson, it was folklore even then [2, Sec. 2.3].
To rotate \(a[0..N)\) to the left by \(r\), the algorithm performs three in-place reversals. The first two reversals are partial, in that they reverse the slices \(a[0..r)\) and \(a[r..N)\). The last reversal targets the whole \(a[0..N)\). Figure 6 shows the resulting straightforward implementation, which calls a routine \texttt{reverse} to reverse \(a\) in place.

The algorithm works thanks to a fundamental property of reversal with respect to concatenation (which we prove in Section 4.2.2 below): the reversal \((X \circ Y)^{-1}\) of the concatenation of two sequences \(X\) and \(Y\) is the concatenation \(Y^{-1} \circ X^{-1}\) of \(Y\)’s reversal and \(X\)’s reversal. Then, consider a sequence \(S\) of length \(N\) as the concatenation \(X \circ Y\), where \(|X| = r\) and \(|Y| = N - r\). As demonstrated in Figure 5 on the running example, where \(N = 6\) and \(r = 2\), the rotation by reversal algorithm applies the following transformations to \(S\):

\[
S = X \circ Y \xrightarrow{\text{reverse } X} X^{-1} \circ Y \xrightarrow{\text{reverse } Y} X^{-1} \circ Y^{-1} \xrightarrow{\text{reverse all}} Y \circ X = \rho^r S,
\]

where the fundamental property justifies the last reversal of the whole \(X^{-1} \circ Y^{-1}\).

Completing the picture, Figure 7 provides an implementation of \texttt{reverse} that works by switching elements at opposite ends of \(a[\text{low}..\text{high})\) while working its way inward: each iteration of the main loop swaps \(a[p]\) and \(a[q]\) on line 48 and then increments \(p\) and decrements \(q\) (initialized to \(\text{low}\) and \(\text{high} - 1\)) on line 49.

```
rotate_reverse (a: ARRAY[G]; r: INTEGER)
require \(0 < r < N\)
ensure \(a = \rho^r(\text{old} \ a)\)
begin
   reverse (a, 0, r)
   reverse (a, r, N)
   reverse (a, 0, N)
end
```

Figure 6: Rotate array \(a\) by \(r\) to the left through three reversals.

### 4.2.1 Rotation by reversal: computational complexity

Since \texttt{rotate_reverse} just calls \texttt{reverse} three times, the asymptotic complexities of the two algorithms are the same. The implementation of \texttt{reverse} shown in Figure 7 has space complexity \(\Theta(1)\) (since it only needs one variable to swap) and time complexity \(\Theta(\text{high} - \text{low})\). Hence, \texttt{rotate_reverse} has time complexity \(\Theta(N)\) and space
reverse \( (a: \text{ARRAY}[G]; \text{low, high: INTEGER}) \)

**require** \( 0 \leq \text{low} \leq \text{high} \leq N \)

**ensure** \( a[\text{low}..\text{high}) = (\text{old } a)^{-1}[\text{low}..\text{high}) \)

**local** \( p, q: \text{INTEGER} \)

\[
\begin{align*}
\text{do} & \quad p, q := \text{low}, \text{high} - 1 \\
\text{while} & \quad p < q + 1 \\
\text{invariant} & \quad \text{low} \leq p \leq q + 2 \leq \text{high} + 1 \\
& \quad q = \text{high} + \text{low} - 1 - p \\
& \quad \forall i: \ \text{low} \leq i < p \implies (\text{old } a)[i] = a[\text{high} + \text{low} - 1 - i] \\
& \quad \forall i: \ q < i < \text{high} \implies (\text{old } a)[i] = a[\text{high} + \text{low} - 1 - i] \\
\text{do} & \quad \text{/ swap } a[p] \text{ and } a[q] \\
& \quad a[p], a[q] := a[q], a[p] \\
& \quad p, q := p + 1, q - 1 \\
\text{end} & \quad \text{end}
\end{align*}
\]

Figure 7: In-place reversal of \( a[\text{low}..\text{high}) \) by swapping elements at opposite ends.

Another appealing feature of rotation by reversal is its flexibility with respect to the data structure it operates on. As long as we can implement in-place reversal in constant space and linear time on the structure, rotate_reverse will still work with the same complexity. In particular, we can have linear-time in-place reversal on linked lists; rotation by reversal works there as well as it works on arrays.

### 4.2.2 Rotation by reversal: correctness

We sketched a correctness argument for rotate_reverse in Section 4.2; now we provide a complete proof, beginning with loop invariants sufficient to verify reverse.

**Reversal: correctness.** A basic bounding invariant requires the indexes \( p \) and \( q \) to be valid positions within \( a[\text{low}..\text{high}) \):

\[
\text{low} \leq p \leq q + 2 \leq \text{high} + 1.
\]

The loop exits when \( p = q + 1 \) if \( \text{high} - \text{low} \) is an even number; and when \( p = q + 2 \) if \( \text{high} - \text{low} \) is an odd number. This reveals that there is a bit of redundancy in reverse: when \( \text{high} - \text{low} \) is odd, \( p = q = \lfloor (\text{high} - \text{low})/2 \rfloor + 1 \) at the beginning of the last loop iteration, which consequently swaps \( a[\text{low}..\text{high}) \)’s central element with itself. To avoid this unnecessary swap, relax the loop staying condition (line 40) to \( p < q \). Here, however, we prefer the slightly redundant formulation because it makes for simpler loop invariants and correctness arguments, as we do not have to separately discuss what happens to the central element.
Another bounding invariant relates \( q \) to \( p \):

\[
q = \text{high} + \text{low} - 1 - p,
\]

which implies, together with (12), that \( q \) is also within bounds in the loop.

The essential loop invariant is two-fold, as it has to relate elements in the upper half \( a(q..\text{high}) \) to the corresponding elements in the lower half of \( \text{old} \ a \) that have been swapped; and vice versa for the lower half \( a(\text{low}..p) \) with respect to the upper half of \( \text{old} \ a \):

\[
\forall i: \text{low} \leq i < p \implies (\text{old} \ a)[i] = a[\text{high} + \text{low} - 1 - i],
\]

\[
\forall i: q < i < \text{high} \implies (\text{old} \ a)[i] = a[\text{high} + \text{low} - 1 - i].
\]

Initiation and consecution are trivial to prove for the bounding invariants, based on how \( p \) and \( q \) are initialized (line 39) and modified by every loop iteration (line 49). The bounding invariants are also the basis to prove inductiveness of the essential invariant: each iteration swaps the elements at positions \( p \) and \( q \), thus preserving (13) thanks to (14). Finally, one can check that (14) implies reverse’s postcondition when the loop exits with \( p \geq q + 1 \).

**Rotation by reversal: correctness.** We prove the lemma relating reversal and concatenation that underpins the correctness of rotate\_reverse.

**Lemma 1** (Reverse of concatenation). \((S \circ T)^{-1} = (T^{-1}) \circ (S^{-1})\), for any two sequences \( S \) and \( T \).

**Proof.** Let \( 0 \leq k < |S| + |T| \) be a generic position in \((S \circ T)^{-1}\). By (5), \((S \circ T)^{-1}_k\) equals \((S \circ T)_{k'}\), where \( k' = |S| + |T| - 1 - k \). We discuss two cases. First case: (a) \( 0 \leq k < |T| \), and hence \( |S| \leq k' < |S| + |T| \). Thus \((S \circ T)_{k'}\) equals \( T_{k'-|S|} \) by (4) and \( k' = |S| + |T| - 1 - k \); hence \( T_{k'-|S|} \) is the element at position \( k \) in \( T^{-1} \) according to (5). Otherwise, second case: (b) \( |T| \leq k < |S| + |T| \), and hence \( 0 \leq k' < |S| \). Thus \((S \circ T)_{k'}\) equals \( S_{k'} \) by (4); hence \( S_{k'} \) is the element at position \( k - |T| \) in \( S^{-1} \) according to (5). (a) and (b) show that \((S \circ T)^{-1}\) follows the definition of \((T^{-1}) \circ (S^{-1})\). □

Let \( 0 \leq x < r \) be an index in the lower half of \( a(\emptyset..N) \). The first reversal of rotate\_reverse maps \( x \) to \( r - 1 - x \) according to (5); the second reversal leaves it unchanged; the third reversal maps it to \( N + 1 - (r - 1 - x) = x + N - r \) still according to (5) and to Lemma 1. Following (6), the latter is the position in \( \text{old} \ a \)’s rotation of the element originally at \( x \) (note that \( 0 \leq x < r \) implies \( x + N - r < N \)). The dual argument, for \( r \leq y < N \), establishes that the element originally at \( y \) ends up in the position in \( \text{old} \ a \)’s rotation. Since such generic \( x \) and \( y \) span the whole interval \( \emptyset..N \), rotate\_reverse’s postcondition holds.

4.3 Rotation by swapping

The rotation by swapping algorithm applies a divide-and-conquer strategy to improve over the space requirements of the rotation by copy algorithm (Section 4.1).

The algorithm builds upon two key observations. First, as it is apparent from definition (6) of rotation, rotating \( a(\emptyset..N) \) to the left by \( r \) can be seen as swapping the adjacent array slices \( a(\emptyset..r) \) and \( a(r..N) \)—which have different length in general. Second, swapping two non-overlapping array slices of *equal* length can be done in
place in linear time, as shown in Figure 8: we simply maintain two index variables \( x \) and \( z \) pointing to the corresponding elements in each section, and swap the corresponding element elements in each iteration.

\[
\begin{align*}
\text{// swap } a[\text{low}..\text{low} + d) \text{ and } a[\text{high} - d..\text{high})
\end{align*}
\]

52  // swap a[low..low + d) and a[high − d..high)
53  swap_sections (a: ARRAY[G]; low, high: INTEGER; d: INTEGER)
54  // non overlapping slices
55  require 0 ≤ low ≤ low + d ≤ high − d ≤ high ≤ N
56  ensure a[low..low + d) = old a[high − d..high)
57        a[low + d..high − d) = old a[low + d..high − d)
58        a[high − d..high) = old a[low..low + d)
59  // pointers to left (x) and right (z) slices
60  local x, z: INTEGER
61  do
62     x, z := low, high − d
63  until x = low + d
64  invariant
65     low ≤ x ≤ low + d ∧ high − d ≤ z ≤ high
66     x − low = z − (high − d)
67     a[low..x) = (old a)[high − d..z)
68     a[x..high − d) = (old a)[x..high − d)
69     a[high − d..z) = (old a)[low..x)
70     a[z..high) = (old a)[z..high)
71  do
72     // swap a[x] and a[z]
73     a[x], a[z] := a[z], a[x]
74     x, z := x + 1, z + 1
75  end
76 end

Figure 8: In-place swap of \( a[\text{low}..\text{low} + d) \) and \( a[\text{low}..\text{low} + d) \).

The divide-and-conquer strategy implemented by the rotation by swapping algorithm calls swap_sections to compute part of the rotation, and then repeats on the smaller unrotated section until completion. To illustrate, consider the running example in Figure 9 where the goal is to swap the subsequence denoted by \( X \) with the rest. Since the size \( r = 2 \) of \( X \) is less than \( N − r = 4 \), we can select another subsequence of size \( r \) (denoted by \( Z \) in Figure 9), at the other end of the whole sequence, such that it does not overlap \( X \). After swapping \( X \) and \( Z \) by calling swap_sections, \( X \) acquires its final position in the rotation of the whole sequence. Then, we recursively apply the algorithm to the subsequence \( Z \circ Y \), which we rotate also by \( r \). In this case, the two subsequences \( Z \) and \( Y \) have equal length; hence swapping them concludes the overall rotation.

It is natural to generalize this approach using a recursive formulation. As Figure 10 shows, we rely on a helper procedure rotate_swap_helper that swaps the slices \( a[\text{low}..\text{p}) \) and \( a[\text{p}..\text{high}) \). If the two slices have equal length (case on line 91), then calling swap_sections with \( d = \text{p} − \text{low} = \text{high} − \text{p} \) does the job. Otherwise, suppose the first slice is smaller (case on line 95, such as in the running example of Figure 9 where \( p = r = 2 \)); that is, \( p − \text{low} < \text{high} − p \). Then, swap \( a[\text{low}..\text{p}) \) with \( a[\text{high} − (p − \text{low})..\text{high}) \); as a result, the latter slice is in place, and we
repeat on \(a[\text{low}..\text{high}] - (p - \text{low})\). Conversely, if \(p - \text{low} > \text{high} - p\) (case on line 100 such as in the other example of Figure 11, where \(p = r = 4\)), swap \(a[\text{low}..\text{low} + (\text{high} - p)]\) with \(a[\text{high} - p..\text{high}]\); as a result, the former slice is in place, and we repeat on \(a[\text{low} + (\text{high} - p)..'\text{high}]\).

In the remainder, we refer to the two recursive cases as the “left is smaller” case, for \(p - \text{low} < \text{high} - p\), and the “right is smaller” case, for \(p - \text{low} > \text{high} - p\). As we justify rigorously in Section 4.3.2, the correctness of the algorithm relies on two dual properties of rotation with respect to concatenation, one for each recursive case. In both cases, we represent \(a[\text{low}..\text{high}]\) as the concatenation \(X \circ Y \circ Z\) of three sequences. In the left is smaller case, \(|X| = |Z| = p - \text{low}\), and the property that \(\rho^{|X|}(X \circ Y \circ Z) = \rho^{|Z|}(Z \circ Y \circ X)\) justifies the recursive call. In the right is smaller case, \(|X| = |Z| = \text{high} - p\) (hence \(|X| + |Y| = p - \text{low}\)), and the property that \(\rho^{|X|+|Y|}(X \circ Y \circ Z) = Z \circ \rho^{|Y|}(Y \circ X)\) justifies the recursive call.

### 4.3.1 Rotation by swapping: computational complexity

Here is a back-of-the-envelope complexity analysis of rotate_swap via its helper function. Overall, rotate_swap_helper makes some \(n\) recursive calls to swap_sections; let \(d_k\) denote the value of argument \(d\) in the \(k\)th call, for \(1 \leq k \leq n\) (for example, \(d_1 = \text{min}(r, N - r)\)). Every such call to swap_sections takes time \(\Theta(d)\) and reduces the problem size by \(d\). Since recursion terminates when the yet-to-be-rotated array slice becomes empty, it must be \(d_1 + \cdots + d_n = N\). The overall time complexity is then \(\Theta(d_1) + \cdots + \Theta(d_n) = \Theta(N)\).

Gries and Mills [7, Sec. 5] provide a more rigorous analysis of the complexity of rotate_swap in terms of number of swaps between array elements. First, note the elegant property that rotate_swap_helper reduces to Euclid's algorithm for greatest common divisor by successive subtractions [5, Sec. 1.3] if we omit the calls to swap_sections: it computes \(\gcd(r, N - r)\). Hence, the last call to swap_sections takes place when \(p - \text{low} = \text{high} - p = \gcd(r, N - r)\); it places the remaining \(2 \cdot \gcd(r, N - r)\) elements in their final rotated position through exactly \(\gcd(r, N - r)\) swaps. The previous calls to swap_sections perform another \(N - 2 \cdot \gcd(r, N - r)\) swaps: each swap places one element in its final rotated position. Overall rotate_swap performs \(N - \gcd(r, N - r)\) swaps.

The space complexity is \(\Theta(N)\) due to recursion: the worst case is \(r = 1\), when the maximum recursion depth is \(N\). However, it is straightforward to produce an equivalent iterative version of rotate_swap, as shown in Figure 12. The condition \(-1(< p < \text{high})\) that terminates recursion becomes the exit condition for a loop.
rotate_swap (a: ARRAY[G]; r: INTEGER)
require 0 < r < N
ensure a = ρ(old a)
do rotate_swap_helper (a, θ, r, N) end

// Rotate a[low..high) at p by swapping a[low..p) and a[p..high)
rotate_swap_helper (a: ARRAY[G]; low, p, high: INTEGER)
require 0 ≤ low ≤ p < high ≤ N
ensure a[low..high) = ρ(p−low)(old a)[low..high)
a[θ..low) = (old a)[θ..low) ∧ a[high..N) = (old a)[high..N)
do
if low < p < high then
  if p − low = high − p then
    // swap a[low..p) and a[p..high)
    swap_sections (a, low, high, p − low)
    // now the whole a[low..high) is in place
  elseif p − low < high − p then
    // swap a[low..p) and a[high − (p − low)..high)
    swap_sections (a, low, high, p − low)
    // now a[high − (p − low)..high) is in place
    rotate_swap_helper (a, low, p, high − (p − low))
  elseif p − low > high − p then
    // swap a[low..low + (high − p)) and a[p..high)
    swap_sections (a, low, high, high − p)
    // now a[low..low + (high − p)) is in place
    rotate_swap_helper (a, low + (high − p), p, high)
  end
end
end

Figure 10: Rotate array a by r to the left by swapping sections of equal length: recursive algorithm.

Figure 11: Rotating sequence LMNOPQ by 4 through swapping sections.

(line 14 in Figure 12) that calls swap_sections and moves low or high closer to each other accordingly. The iterative version clearly has space complexity Θ(1). Practical
implementations will use iteration even if enough memory is available, since limits on recursion stack size would become a bottleneck. As Table 1 shows, such iterative version is quite fast in practice, often nearly as fast as rotation by copy using native methods, but with only constant memory usage.

4.3.2 Rotation by swapping: correctness

We first quickly illustrate the invariants for a correctness proof of swap_sections. Then, we discuss the key steps of a correctness proof for rotate_swap in its recursive and iterative versions.

Swap sections: correctness. Variables \( x \) and \( z \) span the intervals \([\text{low}..\text{low} + d)\) and \([\text{high} - d.. \text{high})\); hence the bounding invariant

\[
\text{low} \leq x \leq \text{low} + d,
\]
\[
\text{high} - d \leq z \leq \text{high}.
\]

(15)

At the beginning of every loop iteration, they point to the pair of elements that are about to be swapped; hence the other bounding invariant

\[
x - \text{low} = z - (\text{high} - d).
\]

(16)

Based on the bounding invariants (15) and (16), we characterize the content of \( a \) during swap_sections’s execution as partitioned into six sections:

| \( a \) | \( 0 \) | \( \text{low} \) | \( x \) | \( \text{high} - d \) | \( z \) | \( \text{high} \) | \( N \)
|---|---|---|---|---|---|---|---|
| untouched | region X: swapped with (Z) | unchanged | region Z: swapped with (X) | unchanged | untouched |

Thus, we have the essential invariants:

\[
a[\text{low}..x) = (\text{old } a)[\text{high} - d..z),
\]
\[
a[x..\text{high} - d) = (\text{old } a)[x..\text{high} - d),
\]
\[
a[\text{high} - d..z) = (\text{old } a)[\text{low}..x),
\]
\[
a[z..\text{high}) = (\text{old } a)[z..\text{high}).
\]

(17)

It is not difficult to prove initiation and consecution of (17). In particular, swapping the elements \( a[x] \) and \( a[z] \) on line 73 in Figure 8 maintains invariance of the swapped slices \( a[\text{low}..x) \) and \( a[\text{high} - d..z) \).

Rotation by swapping: correctness of recursive version. As mentioned in the overview, the proof makes usage of a fundamental lemma, which we now prove.

Lemma 2 (Rotation and swap). For any three sequences \( X, Y, Z \), with \(|X| = |Z| = d \) and \(|X| + |Y| + |Z| = N\):

\[
\rho^d(X \circ Y \circ Z) = \rho^d(Z \circ Y) \circ X,
\]

(18a)

\[
\rho^{N-d}(X \circ Y \circ Z) = Z \circ \rho^{N-2d}(Y \circ X).
\]

(18b)
Proof. We prove \((18a)\); the proof of \((18b)\) can be constructed by similar means. Let \(0 \leq k < N\) be a generic position in \(X \circ Y \circ Z\); the goal is showing that the mapping \(k \mapsto k_1\) determined by the left-hand side transformation \(\rho^l(X \circ Y \circ Z)\) and the mapping \(k \mapsto k_2\) determined by the right-hand side transformation \(\rho^d(Z \circ Y) \circ X\) are such that \(k_1 = k_2\). We discuss two cases: (a) \(0 \leq k < d\) and (b) \(d \leq k < N\).

In case (a), \(k + N - d < N\); hence \(k_1\) is \(k + N - d\) according to \((5)\). Also in case (a), \(k\) denotes a position of \(X\); hence, mapping \(\mapsto 2\) shifts \(k\) by \(|Z| + |Y|\) to the right; that is, \(k_2 = k + (N - 2d) + d = k_1\), which concludes case (a).

In case (b), \(N \leq k + N - d < 2N\); hence \(k_1\) is \((k + N - d) - N = k - d\) according to \((6)\) (and the definition \((2)\) of ‘wrap’). To determine the value of \(k_2\), we describe \(\mapsto 2\) as the application of \(\mapsto 2_{1,1}\) followed by \(\mapsto 2_{2,2}\) accounts for the swapping of \(Z\) and \(X\) in \(X \circ Y \circ Z\), and \(\mapsto 2_{2,2}\) accounts for the rotation \(\rho^d(Z \circ Y)\). Accordingly, we further split case (b) into: (b.1) \(d \leq k < N - d\) and (b.2) \(N - d \leq k < N\). In case (b.1), \(k\) denotes a position within \(Y\) in sequence \(X \circ Y \circ Z\). Hence, mapping \(\mapsto 2_{1,1}\) leaves \(k\) unchanged (since \(|Z| = |X|\)), and then \(\mapsto 2_{2,2}\) maps it to \((k + |Z|)\) \(\rightarrow\) \(Y\) \(|\rightarrow\) \(Z\) \(\rightarrow\) \(Y\) \(|\rightarrow\) \(X\) \(|\rightarrow\) \(Y\). This concludes the proof that \(k_1 = k_2\) in all cases.

The proof of \(\text{rotate_swap_helper}\) now discusses the three main cases, for the three conditional branches on lines \(91\), \(95\) and \(100\) in Figure \(10\). In the first case, \(p - \text{low} = \text{high} - p\), note that the postcondition of \(\text{swap_sections}\) called on \(a(\text{low}..\text{high})\) with \(d = p - \text{low}\) satisfies definition \((4)\) of rotation for \(r = p - \text{low}\) and \(N = \text{high} - \text{low}\). In the \(\text{left is smaller}\) case, \(p - \text{low} < \text{high} - p\), after the call to \(\text{swap_sections}\) the content of \(a(\text{low}..\text{high})\) is the concatenation \((\text{old} \ a)\)\(\text{high} - (p - \text{low})\)\(\text{high} \circ (\text{old} \ a)\)\(\text{p.high} - (p - \text{low}) \circ (\text{old} \ a)\)\(\text{low.p} \)\).

Using the names assigned to each slice, the call to \(\text{swap_sections}\) turns sequence \(X \circ Y \circ Z\) into \(Z \circ Y \circ X\); the following recursive call rotates \(Z \circ Y\) by \(r = p - \text{low}\). According to \((18a)\), for \(X = X, Y = Y, Z = Z\), and \(d = r\), this achieves a rotation of the original sequence \((\text{old} a)\)\(\text{low.high} \) \(= X \circ Y \circ Z\) by the same \(r\), which establishes \(\text{rotate_swap_helper}\)’s postcondition. The \(\text{right is smaller}\) case is symmetric and crucially relies on \((18b)\) for \(d = \text{high} - p\) and \(N = \text{high} - \text{low}\).

**Rotation by swapping: correctness of iterative version.** As usual, we start by identifying the straightforward bounding invariants. Variables \(\text{low}\) and \(\text{high}\) mark a shrinking slice of \(a\) as they get closer to \(p\), hence the obvious invariant

\[
0 \leq \text{low} \leq p \leq \text{high} \leq N.
\] (19)

As we prove inductiveness of this invariant based on how \(\text{low}\) and \(\text{high}\) are updated in every iteration, we notice that when the loop exits both \(p = \text{low}\) and \(p = \text{high}\) hold. We record this fact with another bounding invariant

\[
\text{low} = p \iff p = \text{high}.
\] (20)

which lets us establish that the interval \(\text{low}..\text{high}\) is empty when the loop exits.

The bounding invariants suggest an essential invariant that predicates about three slices of \(a\):
rotate_swap_iterative(a: ARRAY[G]; r: INTEGER)

require 0 < r < N
ensure a = ρ¹(r (old a))

local low, high, p: INTEGER

do
  low, p, high := 0, r, N
  while low < p < high
    invariant
    0 ≤ low ≤ p ≤ high ≤ N
    low = p ⇐⇒ p = high
    // rotated on the left
    ∀i: 0 ≤ i < low =⇒ a[i] = ρ¹(r (old a))[i]
    // to be rotated
    ∀i: low ≤ i < high =⇒
        ρ⁻¹(low)(a[low..high])[i − low] = ρ¹(r (old a))[i]
    // rotated on the right
    ∀i: high ≤ i < N =⇒ a[i] = ρ¹(r (old a))[i]
    do
      if p − low = high − p then
        // swap a[low..p) and a[p..high)
        swap_sections (a, low, high, p − low)
        // now the whole a[low..high) is in place
        low, high := low + (p − low), high − (high − p)
      elseif p − low < high − p then
        // swap a[low..p) and a[high − (p − low)..high)
        swap_sections (a, low, high, p − low)
        // now a[high − (p − low)..high) is in place
        high := high − (p − low)
      elseif p − low > high − p then
        // swap a[low..low + (high − p)) and a[p..high)
        swap_sections (a, low, high, high − p)
        // now a[low..low + (high − p)) is in place
        low := low + (high − p)
      end
    end
  end
end

Figure 12: Rotate array a by r to the left by swapping sections of equal length: iterative algorithm.

The leftmost and rightmost slices are initially empty and invariably in place as the loop iterates:

\[ a[0..\text{low}] = \rho^r(\text{old } a)[0..\text{low}] , \]
\[ a[\text{high}..N] = \rho^r(\text{old } a)[\text{high}..N] . \]
The mid slice has to be rotated; precisely, slices \(a[\text{low}..p)\) and \(a[p..\text{high})\) have to be swapped relative to each other:

\[
\forall i: \text{low} \leq i \leq \text{high} \implies \rho^p - \text{low}(a[\text{low}..\text{high}])[i - \text{low}] = \rho^r(\text{old } a)[i]. \tag{22}
\]

Notice the index shift in the left-hand side of the equality in (22): the first element (index 0) of the rotation of sequence \(a[\text{low}..\text{high})\) by \(p - \text{low}\) corresponds to the element at position \(\text{low}\) in the rotation of the whole \(a\) by \(r = p\).

Proving inductiveness of the essential invariants crucially relies on Lemma 2, following the same overall argument as the proof of the recursive version \rotate_swap. Consider, for example, the right is smaller case: \(p - \text{low} > \text{high} - p\). The call to \swap_sections in the corresponding branch of \rotate_swap_iterative’s loop (line 138 in Figure 12) swaps \(X = (\text{old } a)[\text{low}..\text{low} + (\text{high} - p))\) with \(Z = (\text{old } a)[p..\text{high})\), while leaving \(Y = (\text{old } a)[\text{low}..(\text{high} - p)..p)\) untouched. Thus, \(a[\text{low}..\text{high})\) consists of \(Z \circ Y \circ Z\) after the swap. For \(N = \text{high} - \text{low}, d = \text{high} - p, X = X, Y = Y,\) and \(Z = Z, \tag{18b}\) shows that \(Z\) is in place, whereas \(a[\text{low}..(\text{high} - p)..\text{high})\) must be rotated by \(N - 2d = 2p - \text{high} - \text{low}\). After incrementing \(\text{low}\) by \(\text{high} - p\) (on line 140 in Figure 12), this corresponds to a rotation by \(p - \text{low}\) of \(a[\text{low}..\text{high})\), thus establishing that (22) is inductive in this case.

4.4 Rotation by modular visit

The rotation by modular visit algorithm has the property that it directly moves elements into their final position. To understand how it works, we see \(\rho^r\) as a permutation of the set \(\langle N \rangle\)—that is as the mapping \(k \mapsto (k + (N - r)) \text{ wrap } N\) defined in (8). (Remind that we only deal with left rotations: \(0 \leq r < N\).)

Cycle decomposition of rotations. As recalled in Section 2, \(\rho^r\) has a unique decomposition in disjoint cycles. The first cycle starts from the element at index 0, goes through the elements at indexes

\[
0 \rightarrow (0 + (N - r)) \text{ wrap } N \rightarrow (0 + 2(N - r)) \text{ wrap } N \rightarrow \cdots
\]

until it reaches index 0 again. Similarly, the second cycle goes through

\[
1 \rightarrow (1 + (N - r)) \text{ wrap } N \rightarrow (1 + 2(N - r)) \text{ wrap } N \rightarrow \cdots
\]

until it reaches 1 again. And a generic cycle that starts from \(s\) is

\[
s \rightarrow (s + (N - r)) \text{ wrap } N \rightarrow (s + 2(N - r)) \text{ wrap } N \rightarrow \cdots \tag{23}
\]

until \(s\).

The number of elements in each cycle is the smallest positive integer \(t\) such that \(s + (t(N - r)) \text{ wrap } N = s\), which we equivalently express as the modular equation

\[
t(N - r) \equiv 0 \pmod{N}. \tag{24}
\]

The Linear Congruence Theorem [11, Th. 1.6.14] says that (24) has solutions for \(t \in \{kN / \gcd(N, N - r) \mid k \in \mathbb{Z}\}\). The smallest positive integer in this set is obviously \(N / \gcd(N, N - r)\), which is then the length of each cycle.

\[\text{[11]} \text{Also: http://en.wikipedia.org/wiki/Linear_congruence_theorem}\]
rotate_modulo (a: ARRAY[G]; r: INTEGER)
require 0 < r < N
ensure a = ρ'(old a)
local start, moved, v: INTEGER; displaced: G
do
  start := 0
  moved := 0
  while moved ≠ N
    invariant
    0 ≤ moved ≤ N
    0 ≤ start ≤ gcd(N, N − r)
    moved = start · τ(N, N − r)
    ∀i, s, p: 0 ≤ i < τ(N, N − r) ∧ 0 ≤ s < start ∧ p = πN−r(s, i)
    ⇒ a[p] = ρ'(old a)[p]
    do
      displaced := a[start]
      v := start
      repeat
        v := v + N − r
        // wrap over a's bounds
        if v ≥ N then v := v − N end
        // swap a[v] and displaced
        a[v], displaced := displaced, a[v]
        moved := moved + 1
      invariant
      0 < moved − start · τ(N, N − r) ≤ τ(N, N − r)
      v = πN−r(start, moved − start · τ(N, N − r))
      displaced = (old a)[v]
      ∀i, s, p: 0 ≤ i < τ(N, N − r) ∧ 0 ≤ s < start ∧ p = πN−r(s, i)
      ⇒ a[p] = ρ'(old a)[p]
      ∀j, q: 0 < j ≤ moved − start · τ(N, N − r)
      ∧ q = πN−r(start, j) ⇒ a[q] = ρ'(old a)[q]
    until v = start end
  start := start + 1
end

Figure 13: Rotate array a by r to the left through modular visit.

Rotation by visiting cycles. We finally have all elements to describe the rotation by modular visit algorithm, presented in Figure 13 and demonstrated on the running example in Figure 14. The basic idea is to go through elements in the order given by the decomposition, one cycle at a time until all elements are moved. During the visit, the element originally at position 0 moves to position (0 + (N − r)) wrap N; the element originally at (0 + (N − r)) wrap N moves to (0 + 2(N − r)) wrap N, and so on for all elements in the cycle. Thanks to the unique decomposition property of permutations, this procedure eventually reaches all elements in the sequence; when they are all moved, the whole sequence has been rotated in place.

In the implementation of Figure 13 the outermost loop (line 151) performs a series
of cyclic visits starting with the element at index $\text{start}$—which is 0 initially (line 149). Variable $\text{moved}$ is a counter that records the number of number of elements that are in place; correspondingly, the outermost loop exits when $\text{moved} = N$ and the rotation is complete.

The inner loop (line 161) actually performs the visits of the cycles; precisely, each iteration of the outer loop executes the inner loop to completion for the current value of $\text{start}$, which visits all elements in the cycle beginning at $\text{start}$ as follows. With every iteration of the inner loop, a local variable $v$ takes on the values in the cycle beginning at $\text{start}$: $\text{start}$, $\text{start} + N - r$, and so on, where each new value of $v$ (line 162) is wrapped over when it overflows $N$ (line 164). After updating $v$, the inner loop exchanges $a[v]$ with the element at the previous position in the cycle (line 166), which is stored in variable $\text{displaced}$ (initially equal to $a[\text{start}]$ and successively updated after updating $v$). It then continues with the next iteration. In the running example, the first iteration of the inner loop begins with $\text{displaced} = a[0] = A$ (top left in Figure 14), which it writes to position $0 + N - r = 4$ (its position in the rotation by $r$) while saving (old a)[4] into displaced for the next iteration (mid right in Figure 14).

Earlier in this section, we established that each cycle has $N / \gcd(N, N - r)$ elements. Hence, the inner loop iterates this many times before reaching the exit condition $v = \text{start}$ (line 176). In the running example, $N = 6$, $r = 2$, and $\gcd(6, 4) = 2$, and in fact the inner loop has put $6/2 = 3$ elements in place when it reaches $\text{start}$ again (bottom right in Figure 14). The outer loop correspondingly performs exactly $N / (N / \gcd(N, N - r)) = \gcd(N, N - r)$ iterations, which is when the last cycle in the decomposition is visited (mid left in Figure 14 where the outer loop iterates twice).

![Figure 14: Rotating sequence A B C D E F by 2 through modular visit.](image)

According to Gries and Mills [7], the “jumping around” pattern of the cyclic visits
suggested the name “dolphin algorithm” by which it is sometimes referred to—like a dolphin that leaps out of water and plunges back into it someplace forth.

4.4.1 Rotation by modular visit: complexity

The illustration of the algorithm suggests the complexity of rotation by modular visit. Clearly, only a finite amount of scratch memory is needed; hence the space complexity is $\Theta(1)$. The outer loop iterates $\gcd(N - r, N)$ times, each of which sees the inner loop iterate $N / \gcd(N - r, N)$; hence the time complexity is $\Theta(N)$.

This corresponds to $N$ array writes (one per element put in place). Gries and Mills [7] present a variant of the algorithm that puts the elements in place in each cycle backwards, using $N + \gcd(N - r, N)$ array accesses: one for each element plus $\gcd(N - r, N)$ to temporarily save the array value put in place last and overwritten first (i.e., $a[\text{start}]$). If we count swapping a pair of array elements as three array accesses (using a temporary variable for the swap), this variant of the modular visit algorithm performs the fewest number of array writes among the rotation algorithms. Even in the form of Figure 13, rotation by modular visit has the property that it swaps elements directly into their final position (using displaced as pivot).

Nevertheless, Table 1 suggests that the algorithm does not scale well in practice. While we have not thoroughly investigated the reasons for this lackluster practical performance, it might have to do with (lack of) locality in access: the “jumping around” of modular visits accesses non-adjacent elements which may generate many cache misses when a large array cannot be stored in the fastest level of the memory hierarchy.

4.4.2 Rotation by modular visit: correctness

A proof that it works is remarkably difficult…
— Richard Bornat about rotation by modular visit [3]

We introduce the loop invariants necessary to prove correctness; we then discuss how to prove their inductiveness.

Loop invariants. The formal analysis starts with the outer loop: each iteration visits (and puts in place) all elements whose index is in the cycle that begins at start. The bounding invariants

\begin{align}
0 \leq \text{moved} &\leq N, \\
0 \leq \text{start} &\leq \gcd(N, N - r). \tag{25} \\
0 \leq \text{start} &\leq \gcd(N, N - r). \tag{26}
\end{align}

are then easy to justify (but not so easy to prove!). (25) follows from the number of moved elements being initially zero; and the outer loop exiting when all $N$ elements have been moved. (26) captures the fact that the outer loop executes exactly $\gcd(N, N - r)$ times—as each cycle visits $N / \gcd(N, N - r)$ elements. Since the quantity $N / \gcd(N, N - r)$ will feature often in the invariants and proof of the algorithm, we give it an abbreviation:

\begin{equation}
\tau(N, N - r) = \frac{N}{\gcd(N, N - r)}. \tag{27}
\end{equation}

---

It is somewhat surprising that method rotate of java.util.Collections in OpenJDK 6 uses rotation by modular visit (with an implementation very similar to the one used for the experiments reported in Table 1) not only for small collections but also whenever a collection supports constant-time random access—as in arrayed lists.
The first usage of this definition is to express an exact relation between moved and start. As we repeatedly discussed, every cycle consists of exactly \( \tau(N, N - r) \) elements, and start is incremented on line 177 after every cycle is completed; hence the invariant

\[
\text{moved} = \text{start} \cdot \tau(N, N - r).
\]  

The outer loop’s essential loop invariant precisely characterizes the elements put in place by each iteration of the loop. Given an initial value \( s \) of start, such an iteration visits all elements whose indexes are in the cycle of \( \tau(N, N - r) \) elements defined by

\[
(23)
\]

for \( s = \varpi \). Using the abbreviation

\[
\pi^M_N(s, k) = (s + kM) \text{wrap} N
\]  

(29)
to denote the \( k \)th index in the cycle of step \( M \) modulo \( N \) that starts at \( s \), the essential loop invariant is

\[
\forall i, s, p: \left( \begin{array}{c} 0 \leq i < \tau(N, N - r) \\ 0 \leq s < \text{start} \\ p = \pi^N_N(s, i) \end{array} \right) \implies a[p] = \rho^r(\text{old } a)[p].
\]  

(30)

That is, all elements of all cycles originating in values of start less than the current one have been put in place.

Moving on to the inner loop, variable \( v \) takes on all the values in the currently visited cycle (beginning at start on line 160). Since all cycles previously visited have the same length \( \tau(N, N - r) \), we can express the value of \( v \) as a function of start and moved:

\[
v = \pi^N_N(\text{start}, \text{moved} - \text{start} \cdot \tau(N, N - r)).
\]  

(31)

Expression \( \text{moved} - \text{start} \cdot \tau(N, N - r) \) is 1 initially (that is, after the first unconditionally executed loop iteration); it is \( \tau(N, N - r) \) when the inner loop exits with \( v = \text{start} \). Hence the bounding loop invariant

\[
0 < \text{moved} - \text{start} \cdot \tau(N, N - r) \leq \tau(N, N - r).
\]  

(32)

Given the current value of \( v \), displaced is simply the value in \( a \) initially at index \( v \):

\[
\text{displaced} = (\text{old } a)[v].
\]  

(33)

The outer loop’s essential invariant (30) is also maintained by the inner loop: [30] only involves elements whose indexes are in a cycle starting at some \( s < \text{start} \), but these cycles are disjoint from the currently visited cycle (which begins at start). To describe partial progress made by the inner loop in visiting the current cycle, we introduce another essential invariant:

\[
\forall j, q: \left( \begin{array}{c} 0 < j \leq \text{moved} - \text{start} \cdot \tau(N, N - r) \\ q = \pi^N_N(\text{start}, j) \end{array} \right) \implies a[q] = \rho^r(\text{old } a)[q].
\]  

(34)

Quantified variable \( j \) determines the position in the current cycle; correspondingly, \( j \)’s range of quantification excludes 0, since \( a[\pi^N_N(\text{start}, 0)] \) is set last, and includes \( \text{moved} - \text{start} \cdot \tau(N, N - r) \), corresponding to the element set in the latest loop iteration.

---

[3] Notice it is a repeat...until loop, and hence initiation for its invariants means that they have to hold after one iteration.
Proving initiation. Initiation is trivial for the outer loop invariants, so let’s focus on initiation for the inner loop invariants. The outer loop’s \(28\) still holds at the beginning of the inner loop body, since neither \(\text{start}\) nor \(\text{moved}\) has changed. Thus, incrementing \(\text{moved}\) at the end of the inner loop body makes \(\text{moved} - \text{start} \cdot \tau(N, N - r) = 1\), which proves initiation of \(32\). Since \(\pi_{N-r}^{N-r}(\text{start}, 1)\) is \((\text{start} + N - r) \mod N, 31\) also holds initially. For similar reasons, \(\text{displaced}\) stores the value originally at index \(v\) that has just been assigned to; hence \(33\) initially.

We already discussed that the outer loop essential invariant’s validity is not affected by the inner loop’s work; hence \(30\) satisfies initiation and consecution with respect to the inner loop. Finally, initiation for \(34\) amounts to proving that the value assigned to \(a[\pi_{N-r}^{N-r}(\text{start}, 1)]\) in the first execution of line \(166\) is the one of \(\rho^r(\text{old } a)\); this can be done by matching the definitions of \(\pi_{N-r}\) and of rotation \(8\).

Proving consecution. The outer loop’s bounding invariant \(26\) is unaffected by the inner loop, which does not modify \(\text{start}\). Its inductiveness follows from the bound \(25\) on \(\text{moved}\) and on the connection \(28\) between the latter and \(\text{start}\).

For the consecution proofs of the remaining outer loop invariant, we rely on the inner loop invariants. When the inner loop terminates, \(v = \text{start}\); through \(31\), it follows that \(\text{moved} - \text{start} \cdot \tau(N, N - r) = \tau(N, N - r)\); hence \(28\) is restored after incrementing \(\text{start}\) by one on line \(177\).

We now move to proving inductiveness of the inner loop invariants. Since the inner loop exits when \(v = \text{start}\), \(31\) implies that the increment of \(\text{moved}\) does not overflow \(\tau(N, N - r)\) relative to the initial value at the current outer loop iteration; hence \(32\) is maintained. Conversely, \(31\) is maintained because the inner loop body implements the definition of \(\pi_{N-r}^{N-r}\) with respect to the current \(\text{moved}\) that is incremented by one. Along the same lines one can prove that \(33\) is inductive.

Disjointness of the cycles visited by the inner loop ensures that \(30\) is also maintained. Proving consecution of \(34\) is more involved. Thanks to the inductive hypothesis, we only have to establish progress for \(q = v\). The inner loop, on line \(166\) assigns to \(a[v]\) the value that was assigned to \(\text{displaced}\) in the previous loop iteration; from \(33\), noting that \(\text{moved}\) has just been incremented, this is the element in \(\text{old } a\) whose index is given by the previous value in the cycle, that is the previous value of \(v\). This is in fact \(\rho^r(\text{old } a)[v]\) because of how \(\pi_{N-r}\) is defined.

Final correctness. The final step in the correctness proof of \textit{rotate modulo} is establishing the postcondition from the outer loop invariants. When the outer loop terminates, \(\text{moved} = N\); \(28\) implies that \(\text{start} = \gcd(N, N - r)\). Therefore, proving the postcondition boils down to the following lemma.

**Lemma 3.** Given \(r\) and \(N\) satisfying the precondition of \textit{rotate modulo}: for every \(0 \leq k < N\), there exist \(0 \leq i < \tau(N, N - r)\) and \(0 \leq s < \gcd(N, N - r)\) such that \(\pi_{N-r}^{N-r}(s, i) = k\).

**Proof.** The lemma ultimately follows from the property of decomposition in cycles of permutations (Section 2).

Since \(k\) is listed in some cycle \(23\), the three variables \(k, s, i\) satisfy \(s + i(N - r) \equiv k \pmod{N}\), which indicates that \(s = k \mod (N - r)\). It follows, from standard properties of modular arithmetic \(11\) Sec. 1.6.2], that \(N - r\) divides \(k - s\). Hence, we
can rewrite the expression that relates \( k, s, \) and \( i \)—where now only \( i \) is unknown—as

\[
i \cdot \frac{N - r}{g} \equiv \frac{k - s}{g} \left( \mod \frac{N}{g} \right),
\]

for \( g = \gcd(N, N - r) \). \((35)\) has exactly one solution: use the extended Euclidean algorithm to find \( x \) and \( y \) such that \( x(N - r)/g + yN/g = \gcd((N - r)/g, N/g) = 1 \) satisfies Bézout’s identity. Then, \( i = x(k - s)/g \) is the unique solution.

\[\square\]

5 Rotation: mechanized proofs in Boogie

Notwithstanding our efforts to be as rigorous as possible in the correctness arguments of Section 4, there still is substantial ground to cover before we can have mechanized proofs. Part of the remaining gap is due to the unforgiving level of precision that is required by mechanical proof tools; another part is more specific to the nature of a specific tool we may choose, such as its level of automation and limitations. In this section, we provide a detailed account of what is necessary to turn the proof ideas of Section 4 into successful verification using Boogie [8].

Boogie is an auto-active tool, providing a level of automation intermediate between completely automatic (such as in static analyzers) and interactive (such as in proof assistants). In practice, users interact with the tool offline by providing annotations (such as assertions and lemmas) that guide proof attempts.

Boogie is mainly used as an intermediate language for verification; hence we will have to provide annotations at a relatively low level of detail. This will turn out to be instructive and will showcase several fundamental categories of annotations and annotation styles that are present, in one form or another, in practically every auto-active tool—and possibly in other kinds of tools as well.

The four rotation algorithms make for a gradual introduction to these features of automated verification, as each of them requires new specific annotation techniques:

**Rotation by copy** is simple enough that it only requires basic definitions; we can replicate the proof essentially as done on paper.

**Rotation by reversal** requires intermediate assertions to guide the prover and explicit lemmas proved separately and applied where appropriate in the main correctness proof.

**Rotation by swapping** requires lemmas with non-trivial proofs and modularization tailored to the proof outline; it also requires a little usage of triggers to curb instantiation patterns of the underlying automatic theorem prover.

**Rotation by modular visit** requires clever axiomatization, as well as non-trivial ghost code added to the implementation specifically to represent additional information about program state required to justify the correctness proof; a framing specification is of the essence.

In each case, the features used (such as ghost code or framing annotations) are not necessarily the only way to carry out a proof of that algorithm using Boogie. However, they support a natural approach, and one that is often idiomatic to using auto-active tools of the same family.

---

\( ^6 \)The presentation assumes basic familiarity with the Boogie language and tool.
5.1 Sequences and rotated sequences

Before delving into the details of the algorithms, Figure 15 introduces some basic definitions that we will use in all the Boogie annotations and proofs. As arrays, we use Boogie maps from integers to integers (type \([\text{int}]\text{int}\)). While we could use a generic type as codomain, sticking to plain integers generally works better as it has better support with the underlying SMT solver (in other words, it requires fewer explicit axioms).

First, it is convenient to have an explicit representation of array slices as sequences, so that the lower index of the sequence corresponding to \(a[low..high)\) is zero. Thus, function \(\text{seq}\) takes a map \(a\), a lower index \(low\), and an upper index \(high\) and returns another map whose content over indexes \(\{0..N\}\) coincides with \(a[low..high)\).

The two axioms defining \(\text{wrap}\) replicate definition (2) verbatim. And the definition of rotated sequence follows (6): precisely, function \(\text{rot}\) takes a slice \(a[low..high)\) and a rotation coefficient \(r\) and returns a sequence (that is, a zero-based map) representing the left rotation of \(a[low..high)\) by \(r\). Since we only consider nonnegative values of \(r\), the axiomatic definition of \(\text{rot}\) consists of two axioms, in the same order as definition (6): the first one describes the head of \((high - low) - r\) elements; and the second one describes the tail of \(r\) elements of the rotated sequence.

Using functions \(\text{seq}\) and \(\text{rot}\), Figure 15 shows the generic signature and input/output specification of a Boogie procedure that performs rotation. Since input arguments are read only in Boogie, \(\text{rotate}\) returns another map \(b\) whose content represents the input slice \(a[0..N)\) after processing. The rest is as in Figure 2, but we have to make explicit, in the postcondition, the quantification over range that was implicit in the notation \(a = \rho'(old\ a)\).
procedure rotate(a: [int]int, N: int, r: int) returns(b: [int]int)
  requires 0 < r ∧ r < N;
  ensures (∀ i: int • 0 ≤ i ∧ i < N ⇒ seq(b, θ, N)[i] = rot(a, θ, N, r)[i]);

Figure 16: Rotate array a by r to the left: signature and specification in Boogie.

 wiring a[0..N) to the left by r by copying

procedure rotate_copy(a: [int]int, N: int, r: int)
  returns(b: [int]int)
  requires 0 < r ∧ r < N;
  ensures (∀ i: int • 0 ≤ i ∧ i < N ⇒ seq(b, θ, N)[i] = rot(a, θ, N, r)[i]);
  {
  var s, d: int;
  s, d := 0, N - r;
  while (s < N)
  invariant 0 ≤ s ∧ s ≤ N;
  invariant d = wrap(s + N - r, N);
  invariant (∀ i: int • 0 ≤ i ∧ i < s ⇒ seq(a, θ, N)[i] = seq(b, θ, N)[wrap(i + N - r, N)]);
  {
  b[d] := a[s];
  s := s + 1;
  d := d + 1;
  if (d = N) { d := 0; }
  }
  assert (∀ i: int • 0 ≤ i ∧ i < N ⇒
  rot(a, θ, N, r)[i] = seq(a, θ, N)[wrap(i + r, N)];
  }

Figure 17: Verified Boogie annotated implementation of the rotation by copy algorithm of Figure 3.

5.2 Rotation by copy: mechanized proofs as on paper

Rotation by copy retains most of its simplicity in Boogie. As Figure [7] shows, the same implementation and loop invariants of Figure [6 work in Boogie.

The proof outline presented in Section 4.1.2 mentioned that the essential loop invariant implies the postcondition thanks to the equivalent definition of rotation (8). In a similar way, Boogie has to prove that the representation of rot in terms of wrap, used in the loop invariant, and the axiomatic definition of rot in Figure [15], used in the postcondition, are equivalent. To this end, we introduce an assert expressing (7), on line 224 in Figure [17].

Since (7) is (8)’s inverse, it is the former that translates from a representation based on the latter into one conforming to rot’s axioms. Namely, after proving the assert from rot’s definition, Boogie’s reasoning follows this chain of equalities, for a generic index 0 ≤ k < N:

rot(a, θ, N, r)[k]
= seq(a, θ, N)[k + r] (assert on line 224 in Figure 17)
5.3 Rotation by reversal: mechanized lemmas

Rotation by reversal requires expressing Lemma 1 in Boogie, which is crucial to prove that the three reversals achieve a rotation of the original sequence. In order to be able to do that, we first axiomatize reversal along the same lines as done for rotation.

5.3.1 Axioms and lemmas about reversal

Figure 18 shows an axiomatization based on (5): rev(a, low, high) is the sequence obtained by reversal of a[low..high). We also introduce a function rp(i, low, high) that represents the mapping used in the essential loop invariant of reverse. This is merely a convenience, since we could equivalently use the expanded expression high + low − 1 − i wherever rp(i, low, high) occurs. However, this choice may have an impact in practice because Boogie introduces different triggers for integer expressions than for uninterpreted function applications. For lack of space, we won’t discuss every single alternative in detail; trying out some of them is a useful exercise, also to realize the sensitivity of Boogie’s encoding to changes in annotation style.

Lemmas as procedures. In Boogie, lemmas are encoded as procedures without returned values: preconditions express the lemma’s hypotheses; preconditions express the lemma’s statement; and the procedure body outlines steps in the lemma’s proof. For example, this is a lemma procedure expressing the conclusion of a classic syllogism:

```boogie
procedure syllogism(p: P) requires greek(p) ensures mortal(p)
{ assert human(p); /* p is human, and hence mortal */ }
```

Since lemmas are procedures in Boogie, asserting a lemma is done by calling the corresponding procedure. For example, we can ask Boogie to derive the fact that Socrates is mortal by the instruction `call syllogism(socrates)`, which checks that greek(socrates) and derives that mortal(socrates). To instantiate a lemma for a generic value of some of its arguments, there is the `call forall` instruction: `call forall syllogism(*)` makes the fact that every Greek is mortal available at the call site.

Two lemmas about rotation and reversal in Boogie. The first lemma we present is the Boogie version of (7) as an alternative definition of rot. We already used this fact in the proof of rotate_copy, where we introduced it as an ad hoc `assert`; now, we proceed systematically and formalize it as procedure lemma_rot.

The second lemma is the fundamental Lemma 1. Procedure lemma_rev_cat formalizes it in a form that is readily usable with maps: it asserts that, given slices s[sl..sh), t[tl..th), and c(tl..th) such that c’s slice equals the concatenation of t[tl..th)’s reversal and s[sl..sh)’s reversal, reversing the whole c’s slice gives the concatenation of s[sl..sh) and t[tl..th). Precisely, only the last two ensures, lines 266 and 268 in Figure 19 express the lemma’s conclusion. By
242 axiom \( \forall a: [\text{int}] \text{ int}, \text{ low: int}, \text{ high: int}, \text{ i: int} \bullet \)
243 \(0 \leq i \land i < \text{high} - \text{low} \implies\)
244 \(\text{rev}(a, \text{low}, \text{high})[i] = \text{seq}(a, \text{low}, \text{high})[\text{high} - \text{low} - 1 - i]};\)
245

246 function \(\text{rev}(a: [\text{int}] \text{ int}, \text{ low: int}, \text{ high: int}) \text{ returns}( [\text{int}] \text{ int});\)
247 axiom \(\forall a: [\text{int}] \text{ int}, \text{ low: int}, \text{ high: int}, \text{ i: int} \bullet \)
248 \(0 \leq i \land i < \text{high} - \text{low} \implies\)
249 \(\text{rev}(a, \text{low}, \text{high})[i] = \text{seq}(a, \text{low}, \text{high})[\text{high} - \text{low} - 1 - i]};\)
250

251 // The position \(i\) maps to in a reversal of \([\text{low}..\text{high}]\)
252 function \(\text{rp}(i: \text{ int}, \text{ low: int}, \text{ high: int}) \text{ returns}(\text{int});\)
253 axiom \(\forall i, \text{ low}, \text{ high: int} \bullet \text{rp}(i, \text{low}, \text{high}) = \text{high} + \text{low} - 1 - i};\)
254

Figure 18: Boogie declarations and axiomatic definitions of reversed sequence \text{rev} and inverse index mapping \text{rp} in a reversal.

257 // Representation \(\square\) is equivalent to rot’s definition
258 procedure \text{lemma_rot}(a: [\text{int}] \text{ int}, \text{ low: int}, \text{ high: int}, r: \text{ int}, p: \text{ int})
259 \text{requires} \text{low} \leq \text{high};
260 \text{requires} \text{low} \leq r \land r < \text{high} - \text{low};
261 \text{requires} \text{low} \leq p \land p < \text{high} - \text{low};
262 \text{ensures} \text{rot}(a, \text{low}, \text{high}, r)[p] = \text{seq}(a, \text{low}, \text{high})[\text{wrap}(p + r, \text{high} - \text{low})];
263 { }
264

267 // Lemma \(\square\) (reverse of concatenation)
268 procedure \text{lemma_rev_cat}(t: [\text{int}] \text{ int}, tl: \text{ int}, th: \text{ int},
269 s: [\text{int}] \text{ int}, sl: \text{ int}, sh: \text{ int},
270 c: [\text{int}] \text{ int}, p: \text{ int})
271 \text{requires} \text{tl} \leq \text{th} \land \text{sl} \leq \text{sh};
272 \text{requires} \text{low} \leq p \land p < \text{sh} - \text{sl} + \text{th} - \text{tl};
273 \text{requires} \forall i: \text{ int} \bullet \text{low} \leq i \land i < \text{th} - \text{tl} \implies\)
274 \text{seq}(c, \text{th} - \text{tl})[i] = \text{rev}(t, \text{tl}, \text{th})[i]);
275 \text{requires} \forall i: \text{ int} \bullet \text{low} \leq i \land i < \text{th} - \text{tl} \implies\)
276 \text{seq}(c, \text{th} - \text{tl}, \text{sh} - \text{sl} + \text{th} - \text{tl})[p] = \text{seq}(c, \text{sl}, \text{sh})[i]);
277 \text{ensures} \text{low} \leq p \land p < \text{sh} - \text{sl} \implies\)
278 \text{rev}(c, \text{th} - \text{tl}, \text{sh} - \text{sl} + \text{th} - \text{tl})[p] = \text{seq}(c, \text{sl}, \text{sh})[p];
279 \text{ensures} \text{low} \leq p \land p < \text{sh} - \text{sl} \implies\)
280 \text{rev}(c, \text{th} - \text{tl})[p] = \text{seq}(t, \text{tl}, \text{th})[p];
281 \text{ensures} \text{low} \leq p \land p < \text{sh} - \text{sl} \implies\)
282 \text{rev}(c, \text{th} - \text{tl} + \text{sh} - \text{sl})[p] = \text{seq}(s, \text{sl}, \text{sh})[p];
283 \text{ensures} \text{low} \leq p \land p < \text{sh} - \text{sl} \implies\)
284 \text{rev}(c, \text{th} - \text{tl} + \text{sh} - \text{sl})[p + \text{sh} - \text{sl}] = \text{seq}(t, \text{tl}, \text{th})[p];
285 { }
286

Figure 19: Two Boogie lemma procedures about rotation and reversal. Procedure \text{lemma_rot} establishes that \(\square\) is equivalent to rot’s axiomatic definition. Procedure \text{lemma_rev_cat} represents, for a generic index \(0 \leq p < |S| + |T|, \text{Lemma} \(\square\) with \(S = s[sl..sh], T = t[tl..th], \text{and} c[0..|S| + |T|] = T^{-1} \circ S^{-1}.\)
contrast, using the more readily understandable slice notation, the first two ensures express that
\[
c[a_{\text{ah}} - a_{\text{al}} + b]^{-1} = b[b_{\text{bl}}..b_{\text{bh}}],
c[0..a_{\text{ah}}]^{-1} = a[a_{\text{al}}..a_{\text{ah}}],
\]
which is part of the information used in proving the lemma. If we wanted to directly reflect the lemma’s structure on paper, we would move the formulas on lines 262 and 264 in Figure 19 as asserts inside lemma_rev_cat’s body. This is another alternative that we do not explore in full. It turns out, however, that having those formulas as ensures rather than asserts makes for an overall faster verification—probably because the extra ensures are useful facts where the lemma is employed: not having to derive them again from other available facts at the call site is advantageous.

Short of this, both lemma procedures have empty bodies: Boogie can round up the facts required to prove them without additional guidance.

5.3.2 Mechanized proofs of reversal and rotation by reversal

We now have all the ingredients to present the implementation and proof of the main algorithms.
Proof of in-place reversal. Figure 20 shows Boogie procedure reverse\textsuperscript{7}, which renders the pseudo-code implementation of Figure 7 using the same convention on input and output used for rotate in Figure 16.

Boogie can convert between the representation of b’s content with respect to a’s given by the loop invariants and the one used in the definition of rev, and hence in reverse’s postcondition. When the loop terminates, the essential invariant characterizes the program state in a way that can be expressed as follows, for \(0 \leq k < \text{high} - \text{low} \):

\[
\text{seq}(a, \text{low}, \text{high})[k] = \text{seq}(b, \text{low}, \text{high})[\text{high} - \text{low} - 1 - k]. \quad (36)
\]

Hence, Boogie verifies the following chain of equalities:

\[
\begin{align*}
\text{rev}(a, \text{low}, \text{high})[k] &= \text{seq}(a, \text{low}, \text{high})[\text{high} - \text{low} - 1 - k] \quad \text{(definition of rev: axiom on line 236)} \\
&= \text{seq}(b, \text{low}, \text{high})[\text{high} - \text{low} - 1 - (\text{high} - \text{low} - 1 - k)] \quad \text{(equation 36)} \\
&= \text{seq}(b, \text{low}, \text{high})[k] \quad \text{(arithmetic)} \\
&\quad \text{(postcondition ensures).}
\end{align*}
\]

The code in Figures 20 and 7 is structurally very similar. The only major—yet unsurprising—difference is that Boogie procedure reverse includes information (in the postcondition and, correspondingly, in the loop invariants) about what is not changed by the body: b is the same as a for indexes smaller than low and greater than or equal to higher. This is a simple form of framing necessary because Boogie’s reasoning is modular: the effects of calls to reverse within any of its callers are limited to what is explicit in reverse’s specification irrespective of its implementation; anything that is not explicitly defined in reverse’s postcondition may have changed.

Another, minor, difference between the Boogie code in Figures 20 and the pseudo-code in Figure 7 is that the former’s reverse has no precondition, and simply returns the input a when \([\text{low}..\text{high})\) is an empty range of indexes. In fact, the conditional return statement on line 281 is actually not needed, since the following loop exits immediately when \(\text{low} \geq \text{high}\) (but note that the invariant \(s \leq d + 2\) may fail initially if \(s \geq d + 1\)). Boogie needs a little nudge to understand how to handles this case separately: providing a conditional return is one way to do it with code; ways to do it with annotations are inserting a precondition requires \(\text{low} \leq \text{high}\), or making the failing invariant conditional, so that it holds vacuously when \(\text{low} \geq \text{high}\).

Proof of rotation by reversal. As we can see in Figure 21, the Boogie annotated implementation of the rotation by reversal algorithm closely follows its presentation in Section 4.2.2. After reversing in-place \(b(\emptyset..r)\) and then \(b(r..N)\), \texttt{lemma\_rev\_cat} ensures that reversing \(b(\emptyset..N)\) again yields a rotation of \(a(\emptyset..N)\) by \(r\). The last call to reverse performs this final reversal; and \texttt{lemma\_rot} helps convert between the index representation in \texttt{lemma\_rev\_cat}’s postcondition and the one used in the definition of rot. Even if wrap is directly used in neither, it is applicable to “invert” the former for \(\text{sh} - \text{sl} = N - r\). This reasoning is similar to the argument at the end of Section 5.2 that should have become familiar by now.

\footnote{Boogie’s performance in this example is affected by the names given to local variables \(s\) and \(d\), as well as the temporary local \(t\). For instance, using \(p\) and \(q\) as in the original pseudo-code listing triggers a timeout. We could not investigate this bizarre (and somewhat distressing) issue in depth, but it probably has to do with how the SMT solver’s instantiation rewrite order depends on Boogie’s translation of variable names in the encoding of verification conditions.}
// Rotate a[0..N) to the left by r by reversal

procedure rotate_reverse(a: [int]int, N: int, r: int)
    returns(b: [int]int)

    requires 0 < r ∧ r < N;
    ensures (∀ i: int • 0 ≤ i ∧ i < N ⇒ seq(b, 0, N)[i] = rot(a, 0, N, r)[i]);
{
    b := a;
    call b := reverse(b, 0, r);
    call b := reverse(b, r, N);
    call forall lemma_rev_cat(a, 0, r, a, r, N, b, *);
    call b := reverse(b, 0, N);
    call forall lemma_rot(a, 0, N, r, *);
}

Figure 21: Verified Boogie annotated implementation of the rotation by reversal algorithm of Figure 6.

5.4 Rotation by swapping: organizing code for proofs

Mechanizing the rotation by swapping algorithm requires more complex usage of lemma procedures; and a careful organization of the imperative code to help guide the proof search so that it terminates in reasonable time. The first step is, however, straightforward: verifying the auxiliary routine swap_sections—which we discuss next.

5.4.1 Mechanized proof of swapping sections

Procedure swap_sections in Figure 22 directly translates the pseudo-code in Figure 8. The are only few, unsurprising differences:

- The Boogie procedure uses an output map since input arguments are read only; maps have infinite domains, and hence there is no need to require that 0 ≤ low and high ≤ N.
- The postcondition (and correspondingly the loop invariants) has two additional clauses about framing on lines 329 and 330: the output b is the same as the input a for indexes outside the range [low..high].
- A while loop in Boogie renders the semantics of the until...do loop in pseudocode.

The correctness proof goes through without additional annotations. In fact, unlike the case of reverse, Boogie’s translation seems much more robust with respect to inessential changes such as variable names or equivalent orderings of declarations.

5.4.2 Lemmas about swapping

The most elaborate component for a Boogie proof of rotation by swapping is a translation of Lemma 2, which is used to justify the recursive calls in the main algorithm. We provide two distinct lemma procedures with symmetric structures, one for each of the left is smaller case (18a) and right is smaller case (18b).

31
procedure swap_sections(a: [int]int, low: int, high: int, d: int)
returns(b: [int]int)
requires low ≤ low + d ∧ low + d ≤ high − d ∧ high − d ≤ high;
ensures (∀ i: int • 0 ≤ i ∧ i < d ⇒ seq(b, low, high)[i] = seq(a, low, high)[i + (high − low − d)]);
ensures (∀ i: int • d ≤ i ∧ i < high − low − d ⇒ seq(b, low, high)[i] = seq(a, low, high)[i]);
ensures (∀ i: int • high − low − d ≤ i ∧ i < high − low ⇒ seq(b, low, high)[i] = seq(a, low, high)[i − (high − low − d)]);
ensures (∀ i: int • i < low ⇒ b[i] = a[i]);
ensures (∀ i: int • high ≤ i ⇒ b[i] = a[i]);
{
var x, z: int;
var tmp: int;  // Temporary variable for swap
b := a;
x, z := low, high − d;
while (x < low + d)
  invariant low ≤ x ∧ x ≤ low + d;
  invariant high − d ≤ z ∧ z ≤ high;
  invariant x − low = z − (high − d);
  invariant (∀ i: int • 0 ≤ i ∧ i < x − low ⇒ seq(b, low, high)[i] = seq(a, low, high)[i + (high − low − d)]);
  invariant (∀ i: int • x − low ≤ i ∧ i < high − low − d ⇒ seq(b, low, high)[i] = seq(a, low, high)[i]);
  invariant (∀ i: int • high − low − d ≤ i ∧ i < z − low ⇒ seq(b, low, high)[i] = seq(a, low, high)[i − (high − low − d)]);
  invariant (∀ i: int • i < low ⇒ b[i] = a[i]);
  invariant (∀ i: int • z ≤ i ⇒ b[i] = a[i]);
  { // swap b[x] and b[z]
    tmp := b[z]; b[z] := b[x]; b[x] := tmp;
    x, z := x + 1, z + 1;
  }
}

Figure 22: Verified Boogie annotated implementation of the in-place slice swapping algorithm of Figure 8

Lemma for left is smaller case. Figure 23 shows the Boogie translation of Lemma 2 in case (18a). Recall how the lemma justifies the main algorithm, demonstrated in Figure 9: to compute \( \rho^d(X \circ Y \circ Z) \), first swap equal-length sequences X and Z, and then recur on \( Z \circ Y \).

Procedure lemma_left_smaller traces these two macro steps through an additional input map \( c \). Then, \( a(al..ah) \) represents the input consisting of \( X \circ Y \circ Z \), with \( d \) the length of both leftmost \( X \) and rightmost \( Z \) segments, as in the top-left picture of Figure 9. Slice \( c(cl..ch \!- \! d) \) represents \( Z \circ Y \): the initial part of the processed array after swapping \( X \) and \( Z \), as in the top-right picture of Figure 9. Slice \( b(bl..bh) \) represents the final output after recursively rotating \( c(cl..ch \!- \! d) \) by \( d \), as in the bottom-right picture of Figure 9. The preconditions of lemma_left_smaller encode these assumptions: line 359 describes the right-most slice \( b(bh \!- \! d..bh) \) as \( X \), which is in place in the rotation of \( a(al..ah) \); line 361 describes the other slice \( b(bl..bh \!- \! d) \)
procedure lemma_left_smaller(a: [int]int, al, ah: int, c: [int]int, cl, ch: int, b: [int]int, bl, bh: int, d: int)
requires al < ah;
requires ah - al = bh - bl ∧ ah - al = ch - cl;
requires 0 < d ∧ d < ah - al - d;
requires (∀ i: int • ah - (d + al) ≤ i ∧ i < ah - al ⟹
  rot(a, al, ah, d)[i] = seq(b, bl, bh)[i]);
requires (∀ i: int • 0 ≤ i ∧ i < ah - (d + al) ⟹
  rot(c, cl, ch - d, d)[i] = seq(b, bl, bh - d)[i]);
requires (∀ i: int • 0 ≤ i ∧ i < d ⟹
  seq(c, cl, ch)[i] = seq(a, al, ah)[i + (ah - d - al)]);
requires (∀ i: int • d ≤ i ∧ i < ah - (d + al) ⟹
  seq(c, cl, ch)[i] = seq(a, al, ah)[i]);
ensures (∀ i: int • 0 ≤ i ∧ i < ah - al ⟹
  rot(a, al, ah, d)[i] = seq(b, bl, bh)[i]);
{
assert (∀ i: int • 0 ≤ i ∧ i < ah - (d + al) ⟹
  rot(c, cl, ch - d, d)[i] = seq(b, bl, bh)[i]);
assert (∀ i: int • 0 ≤ i ∧ i < ch - cl - d - d ⟹
  rot(c, cl, ch - d, d)[i] = seq(c, cl, ch)[i + d]);
assert (∀ i: int • 0 ≤ i ∧ i < ch - cl - d - d ⟹
  rot(c, cl, ch - d, d)[i] = seq(c, cl, ch)[i + d]);
assert (∀ i: int • ch - cl - d - d ≤ i ∧ i < ch - cl - d ⟹
  rot(c, cl, ch - d, d)[i] =
  seq(c, cl, ch)[i - (ch - cl - d - d)];
assert (∀ i: int • ch - cl - d - d ≤ i ∧ i < ch - cl - d ⟹
  rot(c, cl, ch - d, d)[i] =
  seq(c, cl, ch)[i - (ch - cl - d - d)];
assert (∀ i: int • 0 ≤ i ∧ i < ah - (d + al) ⟹
  rot(a, al, ah, d)[i] = seq(b, bl, bh)[i]);
assert (∀ i: int • ah - (d + al) ≤ i ∧ i < ah - al ⟹
  rot(a, al, ah, d)[i] = seq(b, bl, bh)[i]);
}

Figure 23: Lemma \ref{lem:invariant_left} for case \textit{left is smaller} \((18a)\) as a Boogie lemma procedure, with
\(d = d, N = ah - al = bh - bl = ch - cl, a(al..ah) = X \circ Y \circ Z, c(cl..ch - d) = Z \circ Y,\) and
\(b(bl..bh) = \rho^d(c(cl..ch - d)) \circ X.\)

as \(\rho^d(c(cl..ch - d));\) lines \ref{ln:assert363} and \ref{ln:assert363} respectively describe \(c(cl..cl + d)\)
as \(Z\) and \(c(cl + d..ch - d)\) as \(Y.\) The postcondition on line \ref{ln:assert367} concludes that the
\(b(bl..bh)\) described in the precondition is indeed the rotation of \(a(al..ah)\) by \(d.\)

The procedure body consists of a series of seven \textbf{assert} that guide Boogie through
the proof of the postcondition from the preconditions. As usual, there is room for
variations, but this particular sequence of \textbf{assert} is fairly natural and produces a fast
proof; to illustrate, this is an informal explanation of what each \textbf{assert} establishes:

\textbf{Line} \ref{ln:assert370} \textbf{relaxes} the right bound of seq\((b)\) in the precondition on line \ref{ln:assert361}
from \(bh - d \) to \(bh,\) since indexes beyond \(bh - d\) are out of the quantification range.

\textbf{Line} \ref{ln:assert372} \textbf{recalls} the definition of \textbf{rot} for \(c(cl + d..ch - d - d)\) or \(Y.\)
Line 374 relaxes the right bound of seq(c) in the previous assert from ch − d to ch, since indexes beyond ch − d are out of the quantification range.

Line 376 recalls the definition of rot for c[cl..cl + d) or Z.

Line 379 relaxes the right bound of seq(c) in the previous assert from ch − d to ch, since indexes beyond ch − d are out of the quantification range.

Line 382 concludes that b[bl..bh − d) is ρd(a[al..ah)) between [al..ah − d), using the facts about c proved so far (specifically, lines 374 and 379), and the relations between b and c and between b and a in the preconditions.

Line 384 recalls that b[bh − d..bh) coincides with ρd(a[al..ah)][ah − d..ah); even if this assert is the very same as the precondition on line 359, it is necessary to recall it explicitly in the body so that Boogie uses it to close the proof.

Once we have understood the rationale behind lemma_left_smaller, it is not difficult to derive the dual lemma_right_smaller shown in Figure 24 and corresponding to case (18b) of Lemma 2. Following the example of Figure 11 going from a (top-left picture) through c (top-right picture) to b (bottom-right picture) helps understand the lemma procedure. Compared to lemma_left_smaller, there now is one more assert due to an additional index rescaling (the second assert refers to i − d, which becomes i in the third assert).

The real twist is, however, the need for a trigger in the last assert on line 422:

\{ seq(b, bl, bh)[i] \}.

Even if the assert is just a repetition of the precondition on line 393, Boogie needs help to pick the relevant facts among the many instantiated terms that are available. The trigger directs the SMT solver to only instantiate the universal quantifier in the assert for those i’s such that seq(b, bl, bh)[i] is a term in the current proof context. In this particular case, using the trigger makes a dramatic difference in terms of performance when proving the whole lemma procedure.

5.4.3 Mechanized proof of rotation by swapping: recursive version

Presenting annotated versions of rotate_swap and rotate_swap_helper, Figure 25 is the Boogie counterpart to Figure 10.

With respect to the pseudo-code version of Figure 10, rotate_swap_helper in Boogie has some structural differences that are worth discussing. The most pronounced one is a different conditional structures. The pseudo-code algorithm clearly distinguishes between three cases (equal length of slices to be swapped, left is smaller, right is smaller), and each case has a call to swap_sections followed, in the last two cases, by a recursive call to the helper; the trivial base case low = p is handled by an enclosing if. By contrast, the Boogie procedure handles the trivial base case initially introducing abrupt termination (i.e., a return). Then, the call to swap_sections on line 448 applies to two cases: “equal length” and “left is smaller”. This structure helps reduce repetitions in reasoning along different conditional branches, and in fact it makes for quicker verification. Having one fewer call to swap_sections with respect to the pseudo-code version avoids checking swap_sections’s precondition twice with the same arguments.

---

8The description of the SMT solver Simplify discusses how triggers work; see also [9, 1] and [8, Sec. 11.2].
procedure lemma_right_smaller(a: [int]int, al, ah: int,
c: [int]int, cl, ch: int,
b: [int]int, bl, bh: int, d: int)
requires al < ah;
requires ah − al = bh − bl ∧ ah − al = ch − cl;
requires 0 < d ∧ d < ah − al − d;
requires (∀ i: int • 0 ≤ i ∧ i < d →
rot(a, al, ah, ah − al − d)[i] = seq(b, bl, bh)[i]);
requires (∀ i: int • 0 ≤ i ∧ i < ah − (d + al) →
rot(c, cl + d, ch, ah − al − d − d)[i] = seq(b, bl + d, bh)[i]);
requires (∀ i: int • ah − (d + al) ≤ i ∧ i < ah − al →
seq(c, cl, ch)[i] = seq(a, al, ah)[i(i − (ah − d − al))]);
requires (∀ i: int • d ≤ i ∧ i < ah − (d + al) →
seq(c, cl, ch)[i] = seq(a, al, ah)[i]);
ensures (∀ i: int • 0 ≤ i ∧ i < ah − al →
rot(a, al, ah, ah − al − d)[i] = seq(b, bl, bh)[i]);
{
assert (∀ i: int • 0 ≤ i ∧ i < ah − (d + al) →
rot(c, cl + d, ch, ah − al − d − d)[i] = seq(b, bl, bh)[i + d]);
assert (∀ i: int • 0 ≤ i ∧ i < ch − cl − d →
rot(c, cl + d, ch, ah − al − d − d)[i] = seq(c, cl, ch)[i − d]);
assert (∀ i: int • 0 ≤ i ∧ i < ch − cl − d − d →
rot(c, cl + d, ch, ah − al − d − d)[i + d] =
seq(c, cl + d, ch)[i]);
assert (∀ i: int • d ≤ i ∧ i < ch − cl − d →
rot(c, cl + d, ch, ah − al − d − d)[i] =
seq(c, cl + d, ch)[i + (ch − cl − d − d)]);
assert (∀ i: int • 0 ≤ i ∧ i < d →
rot(c, cl + d, ch, ah − al − d − d)[i] =
seq(c, cl, ch)[i + (ch − cl − d)]);
assert (∀ i: int • d ≤ i ∧ i < ah − al →
rot(a, al, ah, ah − al − d)[i] = seq(b, bl, bh)[i]);
assert (∀ i: int • { seq(b, bl, bh)[i] } // trigger
0 ≤ i ∧ i < d →
rot(a, al, ah, ah − al − d)[i] = seq(b, bl, bh)[i]);
}

Figure 24: Lemm[2] for case right is smaller (18b) as a Boogie lemma procedure, with
d = d, N = ah − al = bh − bl = ch − cl, a(al..ah) = X ◦ Y ◦ Z, c(cl + d..ch) = Y ◦ X, and b(bl..bh) = Z ◦ ρ^{N−2d}(c(cl + d..ch)).

in different contexts; and the return in the “equal length” case drives a direct proof
of the helper’s postcondition from swap_sections’s postcondition and the few other
facts available at that location, instead of having to consider many other inapplicable
facts in a conditional reasoning at the unique exit point of the structured pseudo-code
version. Of course, other solutions are possible in Boogie with some trial and error,
but it should be clear that the two versions are semantically equivalent. To help unravel
procedure rotate_swap(a: [int]int, N: int, r: int)
    returns(b: [int]int)
    requires 0 < r ∧ r < N;
    ensures (∀ i: int • 0 ≤ i ∧ i < N ⇒ seq(b, 0, N)[i] = rot(a, 0, N, r)[i]);
    { call b := rotate_swap_helper(a, r, 0, N); }

procedure rotate_swap_helper(a: [int]int, p: int, low, high: int)
    returns(b: [int]int)
    requires low ≤ p ∧ p < high;
    ensures (∀ i: int • 0 ≤ i ∧ i < high - low ⇒ rot(a, low, high, p - low)[i] = seq(b, low, high)[i]);
    ensures (∀ i: int • i < low ⇒ b[i] = a[i]);
    ensures (∀ i: int • high ≤ i ⇒ b[i] = a[i]);
    { var c: [int]int; // ghost: value of b before recursive call
        if (p = low) { b := a; return; }
        if (p - low ≤ high - p) {
            // swap a[low..p] and a[high - (p - low)..high)
            call b := swap_sections(a, low, high, p - low);
            if (p - low = high - p) {
                // now the whole b[low..high) is in place
                return;
            } else {
                // now b[high - (p - low)..high) is in place
                c := b; // ghost
                call b := rotate_swap_helper(b, p, low, high - (p - low));
                call lemma_left_smaller(a, low, high, c, low, high, b, low, high, p - low);
            }
        } else {
            assert p - low > high - p;
            assert 0 ≤ high - p ∧ high - p ≤ high - low;
            // swap a[low..low + (high - p)] and a[p..high)
            call b := swap_sections(a, low, high, high - p);
            // now b[low..low + (high - p)) is in place
            c := b; // ghost
            call b := rotate_swap_helper(b, p, low + (high - p), high);
            call lemma_right_smaller(a, low, high, c, low, high, b, low, high, high - p);
        }
    }

Figure 25: Verified Boogie annotated implementation of the rotation by swapping recursive algorithm of Figure 10
the branching structure with more clarity, we have two assert in the “right is smaller” branch; they also are crucial for performance.

The usage of a ghost variable c is another novelty of Figure 25 compared to the previous Boogie examples. It is no coincidence that the name c is also used for one argument of the lemma procedures presented in Section 5.4.2. In rotate_swap_helper, c keeps track of the value of b after the first macro-step (call to swap_sections) and before the second one (recursive call to the helper). Thanks to c, we conclude the proof of each recursive case by calling the corresponding lemma procedure, which relates the input a to the final output b through c to establish rotate_swap_helper’s postcondition. As discussed in the upcoming Section 5.5, the rotation by modular visit algorithm contemplates a much richer usage of ghost code, but the idea is already clear here: ghost code keeps track of program state beyond what is explicit in the non-ghost program variables (that is, variables used in the actual computation), capturing information that is readily useful for proofs.

A final aspect of modularization leveraged in the proof of rotation by swapping is not apparent in the presentation on paper. We split the proof of the procedures in separate files. Each file contains only one procedure with implementation (for example, rotate_swap_helper) together with only the signature and specification of other procedures called in the single implementation (for example, lemma_left_smaller, lemma_right_smaller, and swap_sections). We invoke Boogie separately on each file. Even if Boogie works modularly, there clearly is interference between different proofs originating in the same file; having only one procedure to prove per invocation significantly reduces the possible problems—ultimately causing slower proofs or timeouts due to unfruitful proof search heuristics being applied. The bottom line is that how code and annotations are structured can make a significant different when mechanizing verification of algorithms.

5.4.4 Mechanized proof of rotation by swapping: iterative version

A careful organization of code and annotations is also central to the proof of the iterative version of rotation by swapping. A formalization of Lemma 2 is still at the core of the correctness argument; but we now proceed using a different approach than in the recursive version: since lemmas and imperative code are both encoded as procedures in Boogie, we combine them in the same procedure.

To this end, we introduce three variants of swap_sections, one for each of the by-now familiar cases: “equal length” sections, “left is smaller”, and “right is smaller”. We name the three variants accordingly: swap_equal, swap_left, and swap_right. The operational part of the variants is identical, and simply consists of a suitable call to swap_sections of Figure 22. What is different is their specification: besides describing output in terms of input, it also relates the output to the original reversal problem as per Lemma 2. Take for example swap_left in Figure 26 which swaps c[l..p] and c[h - (p - l)..h] under the assumption p - l < h - p. Its input arguments also include the original input a[low..high] to be rotated. Its precondition assumes that c[low..l] and c[h..high] correspond to already rotated slices of a[low..high]. Its postcondition guarantees that output b[low..l] and b[h - (p - l)..high] will consist of rotated slices of a[low..high], thus ensuring progress; and that rotating b[l..h - (p - l)] by p - l will complete the rotation of a[low..high]. Of course, the names a, b, and c correspond to the three macro-step also underlying the recursive version and the running example in Figure 25. Similar comments apply to the augmented specification of swap_equal and swap_right shown in Figure 27.
procedure swap_left(a: [int]int, c: [int]int, low, high: int,
l, h: int, p: int) returns (b: [int]int)
  requires low ≤ l ∧ l < p ∧ p < h ∧ h ≤ high;
  requires p − l < h − p;  // left is smaller
  requires (∀ i: int • 0 ≤ i ∧ i < l − low ⇒
    rot(a, low, high, p − low)[i] = seq(c, low, high)[i]);
  requires (∀ i: int • l − low ≤ i ∧ i < h − low ⇒
    rot(a, low, high, p − low)[i] =
    rot(c, l, h, p − l)[i − (l − low)]);
  requires (∀ i: int • h − low ≤ i ∧ i < high − low ⇒
    rot(a, low, high, p − low)[i] = seq(c, low, high)[i]);
  requires (∀ i: int • i < low ⇒
    c[i] = a[i]);
  requires (∀ i: int • high ≤ i ⇒
    c[i] = a[i]);
  ensures (∀ i: int • 0 ≤ i ∧ i < l − low ⇒
    rot(a, low, high, p − low)[i] = seq(b, low, high)[i]);
  ensures (∀ i: int • l − low ≤ i ∧ i < h − low − (p − l) ⇒
    rot(a, low, high, p − low)[i] =
    rot(b, l, h − (p − l), p − l)[i − (l − low)]);
  ensures (∀ i: int • h − low − (p − l) ≤ i ∧ i < high − low ⇒
    rot(a, low, high, p − low)[i] = seq(b, low, high)[i]);
  ensures (∀ i: int • i < low ⇒
    b[i] = a[i]);
  ensures (∀ i: int • high ≤ i ⇒
    b[i] = a[i]);
  {
    call b := swap_sections(c, l, h, p − l);
    // Asserts to prove post from pre and swap_sections's post
    // ...
  }
Figure 26: Verified Boogie annotated implementation of the in-place slice swapping algorithm Figure 8, postcondition augmented with Lemma 2 for case left is smaller (18a).
procedure swap.equal(a: [int]int, c: [int]int, low, high: int,
    l, h: int, p: int) returns(b: [int]int);
requires low ≤ l ∧ l < p ∧ p < h ∧ h ≤ high;
requires p = l = h − p; // left same size as right
requires (∀ i: int • 0 ≤ i ∧ i < l − low)⇒
    rot(a, low, high, p − low)[i] = seq(c, low, high)[i]);
requires (∀ i: int • l − low ≤ i ∧ i < h − low)⇒
    rot(a, low, high, p − low)[i] =
    rot(c, l, h, p − l)[i − (l − low)));
requires (∀ i: int • h − low ≤ i ∧ i < high − low)⇒
    rot(a, low, high, p − low)[i] = seq(c, low, high)[i]);
requires (∀ i: int • i < low)⇒ c[i] = a[i]);
requires (∀ i: int • high ≤ i)⇒ c[i] = a[i]);
ensures (∀ i: int • 0 ≤ i ∧ i < p − low)⇒
    rot(a, low, high, p − low)[i] = seq(b, low, high)[i]);
ensures (∀ i: int • p − low ≤ i ∧ i < high − low)⇒
    rot(a, low, high, p − low)[i] = seq(b, low, high)[i]);
ensures (∀ i: int • i < low)⇒ b[i] = a[i]);
ensures (∀ i: int • high ≤ i)⇒ b[i] = a[i]);

procedure swap.right(a: [int]int, c: [int]int, low, high: int,
    l, h: int, p: int) returns(b: [int]int);
requires low ≤ l ∧ l < p ∧ p < h ∧ h ≤ high;
requires p − l > h − p; // right is smaller
requires (∀ i: int • 0 ≤ i ∧ i < l − low)⇒
    rot(a, low, high, p − low)[i] = seq(c, low, high)[i]);
requires (∀ i: int • h − low ≤ i ∧ i < high − low)⇒
    rot(a, low, high, p − low)[i] = seq(c, low, high)[i]);
requires (∀ i: int • l − low ≤ i ∧ i < h − low)⇒
    rot(a, low, high, p − low)[i] =
    rot(c, l, h, p − l)[i − (l − low)));
requires (∀ i: int • i < low)⇒ c[i] = a[i]);
requires (∀ i: int • high ≤ i)⇒ c[i] = a[i]);
ensures (∀ i: int • 0 ≤ i ∧ i < l − low + (h − p))⇒
    rot(a, low, high, p − low)[i] = seq(b, low, high)[i]);
ensures (∀ i: int • l − low + (h − p) ≤ i ∧ i < h − low)⇒
    rot(a, low, high, p − low)[i] =
    rot(b, l + (h − p), h, p − (l + (h − p)))[i − (l − low + h − p )]);
ensures (∀ i: int • h − low ≤ i ∧ i < high − low)⇒
    rot(a, low, high, p − low)[i] = seq(b, low, high)[i]);
ensures (∀ i: int • i < low)⇒ b[i] = a[i]);
ensures (∀ i: int • high ≤ i)⇒ b[i] = a[i]);

Figure 27: Boogie specifications of the in-place slice swapping algorithm. The postcondition of swap.equal is augmented with the property that, when p − l = h − p, swapping c(l..p) and c(p..h) in place is tantamount to rotating c(l..h) by p − l. The postcondition of swap.right is augmented with Lemma 2 for case right is smaller.
procedure rotate_swap_iterative(a: [int]int, N: int, r: int)
    returns(b: [int]int)
    requires 0 < r ∧ r < N;
    ensures (∀ i: int • 0 ≤ i ∧ i < N ⇒ seq(b, 0, N)[i] = rot(a, 0, N, r)[i]);
{
    var low, p, high: int;
    low, p, high := 0, r, N;
    b := a;
    while (low < p ∧ p < high)
        invariant θ ≤ low ∧ low ≤ p ∧ p ≤ high ∧ high ≤ N;
        invariant low = p ⇐⇒ p = high;
        invariant (∀ i: int • 0 ≤ i ∧ i < low ⇒
            rot(a, 0, N, p)[i] = seq(b, 0, N)[i]);
        invariant (∀ i: int • low ≤ i ∧ i < high ⇒
            rot(a, 0, N, p)[i] =
            rot(b, low, high, p - low)[i - low]);
        invariant (∀ i: int • high ≤ i ∧ i < N ⇒
            rot(a, 0, N, p)[i] = seq(b, 0, N)[i]);
        invariant (∀ i: int • i < 0 ⇒ b[i] = a[i]);
        invariant (∀ i: int • N ≤ i ⇒ b[i] = a[i]);
        {
            goto equal_length, left_smaller, right_smaller;
            equal_length:
                assume p - low = high - p;
                call b := swap_equal(a, b, 0, N, low, high, p);
                low, high := low + (p - low), high - (high - p);
                goto continue;
            left_smaller:
                assume p - low < high - p;
                call b := swap_left(a, b, 0, N, low, high, p);
                high := high - (p - low);
                goto continue;
            right_smaller:
                assume p - low > high - p;
                call b := swap_right(a, b, 0, N, low, high, p);
                low := low + (high - p);
                goto continue;
            continue:
        }
}
The advantage of this approach is that we can reason about special properties of swapping separately in each case. The call to \texttt{swap\_sections} in the bodies of \texttt{swap\_left}, \texttt{swap\_right}, and \texttt{swap\_equal} is followed by a sequence of \texttt{assert} that proves the special properties of the swapping declared in the augmented postconditions. For brevity, we omit the proofs; suffice it to say that \texttt{swap\_equal} is sufficiently simple that it requires no explicit proof, whereas \texttt{swap\_left} and \texttt{swap\_right}'s proofs are quite involved and require elaborate assertions and careful usage of triggers.

With this organization, Boogie can prove \texttt{rotate\_swap\_iterative} in Figure 28 with the same invariants as the pseudo code in Figure 12 without additional annotations (the only exception being the straightforward framing invariants to keep track of the unchanged parts of the map domain before 0 and after N). While we could have used nested \texttt{if}s to replicate the three-case structure in the loop body of Figure 12 we demonstrate another construct, nondeterministic \texttt{goto}, which emphasizes the three-way case split. Embedding the proof of Lemma 2 in separate procedures makes for a simple and efficient high-level proof that reflects the argument on paper.

5.5 Rotation by modular visit: ghost code and framing

Underlying the proof of the rotation by modular visit algorithm discussed informally in Section 4.4.2 were properties of modular arithmetic and cyclic decompositions of permutations. Mechanizing the proofs of those properties all the way down to fundamental arithmetic would be exceedingly complicated and out of the scope of the present discussion; instead, we capture the fundamental mathematical properties as axioms whose correctness is intuitively clear, and build the main correctness proof atop them.

This approach has the additional advantage that it lets us focus on other aspects central to mechanizing the proof of rotation by modular visit, and in particular on keeping track of implicit information in the program state by means of ghost code. We already encountered ghost code among the annotations of rotation by swapping (Section 5.4.3), but proving rotation by modular visit will require more complex usage, especially to detail framing of the result array \( b \).

5.5.1 Axioms about cycles

Figure 29 shows declarations and axiomatic definitions of three fundamental quantities featuring in the proof of rotation by modular visit: \( \pi_{M}^{N}(s, k) \), \( \text{gcd}(N, M) \), and \( \tau(N, M) \) corresponding to \( \text{mp}(N, M, s, k) \), \( \text{gcd}(N, M) \), and \( \tau(N, M) \) in Boogie.

The first two axioms characterize \( \pi_{N}^{M}(s, k) \)—which gives the \( k \)-th index in a cycle starting at \( s \) with step \( M \) wrapping over \( N \)—inductively as

\[
\pi_{N}^{M}(s, k) = \begin{cases} 
  s & k = 0, \\
  (M + \pi_{N}^{M}(s, k - 1)) \text{ wrap } N & k > 0.
\end{cases}
\]

This definition and the one in (29) are equivalent (a fact which could be proved from a suitable axiomatization of modular arithmetic), but the inductive definition has the advantage of directly matching the program's logic: each iteration of the inner loop moves \( v \) to the “next” value in the modular visit. In contrast, (29) is inductive only indirectly through definition (2) of ‘\text{wrap}’.

The following two axioms, lines 590 and 594 in Figure 29, define how gcd and \( \tau \) are bounded by their arguments.
function mp(N: int, M: int, s: int, p: int) returns(int);

axiom (∀ N: int, M: int, s: int •

\[ 0 < M \land M < N \land 0 \leq s \land s < N \implies mp(N, M, s, 0) = s; \]

axiom (∀ N: int, M: int, s: int, k: int •

\[ 0 < M \land M < N \land 0 \leq s \land s < N \land 0 < k \implies mp(N, M, s, k) = \text{wrap}(mp(N, M, s, k - 1) + M, N); \]

function gcd(N: int, M: int) returns(int);

axiom (∀ N, M: int • 0 < N \land 0 < M \implies

\[ 0 < \gcd(N, M) \land \gcd(N, M) \leq N \land \gcd(N, M) \leq M; \]

function \( \tau \)(N: int, M: int) returns(int);

axiom (∀ N: int, M: int • 0 < M \land M < N \implies

\[ \gcd(N, M) \neq \tau(N, M) = N; \]

axiom (∀ N: int, M: int, s: int • mp(N, M, s, \tau(N, M)) = s);

axiom (∀ N: int, M: int, s: int, p: int •

\[ 0 < p \land p < \tau(N, M) \implies mp(N, M, s, p) \neq s; \]

axiom (∀ N: int, M: int, s: int, p, q: int •

\[ 0 \leq p \land p < \tau(N, M) \land 0 \leq q \land q < \tau(N, M) \land p \neq q \implies mp(N, M, s, p) \neq mp(N, M, s, q)); \]

axiom (∀ N: int, M: int, s, t: int, p, q: int •

\[ 0 \leq p \land p < \tau(N, M) \land 0 \leq q \land q < \tau(N, M) \land \]

\[ 0 \leq s \land s < t \land t < s + \gcd(N, M) \land t < N \implies \]

\[ mp(N, M, s, p) \neq mp(N, M, t, q)); \]

Figure 29: Boogie declarations and axiomatic definitions of \( mp \), \( gcd \), and \( \tau \).

The remaining axioms in Figure 29 complete the characterization of \( \pi \), \( gcd \), and \( \tau \) in terms of mutual properties. The axiom on line 597 is equivalent to (27); in the proof, it is necessary to conclude that the inner and outer loops combined visit all the \( N \) elements of the input. The three axioms on lines 599–604 define the \( \tau(N, M) \) elements in the same cycle starting at a generic \( s \); in the proof, these characterize the elements visited by the inner loop (executed to completion for a given value of \( s \)). By contrast, the last axiom in Figure 29 declares disjointness between elements of the cycles a rotation can be decomposed into; in the proof, it is necessary to combine the effect of each iteration of the outer loop (in fact, the axiom covers exactly \( \tau(N, M) \) different consecutive values of \( s \)).

5.5.2 Outer loop

The Boogie version of the algorithm is a mouthful; we begin looking at the annotated outer while loop, whose Boogie version is shown in Figure 30 (with references to the parts presented later).

Framing using a ghost map. A fundamental difficulty we encounter trying to translate the annotated algorithm of Figure 13 into Boogie is the lack of readily available
procedure rotate_modulo(a: [int]int, N: int, r: int)
    returns (b: [int]int)
requires 0 < r ∧ r < N;
ensures (∀ i: int • 0 ≤ i ∧ i < N ⇒ seq(b, 0, N)[i] = rot(a, 0, N, r)[i]);
{
    var start, v, displaced: int;
    // ghost:
    var k: int; // index mp(N, N − r, start, k) currently visited
    var set: [int]bool; // set[i] iff b[i] has been assigned to
    b := a;
    assume (∀ i: int • ¬set[i]); // ghost: initialize b
    start := 0;
    while (start < gcd(N, N − r))
        invariant (0 ≤ start ∧ start ≤ gcd(N, N − r));
        invariant (∀ i: int • 0 ≤ i ∧ i < N ∧ ¬set[i] ⇒ b[i] = a[i]);
        invariant (∀ i: int, s: int • 0 ≤ i ∧ i < τ(N, N − r) ∧ start ≤ s ∧ s < gcd(N, N − r)
            ⇒ ¬set[mp(N, N − r, s, i)]);
        invariant (∀ i: int, s: int • 0 ≤ i ∧ i < τ(N, N − r) ∧ 0 ≤ s ∧ s < start
            ⇒ set[mp(N, N − r, s, i)]);
        invariant (∀ i: int • 0 ≤ i ∧ i < N ∧ set[i] ⇒
            seq(b, 0, N)[i] = rot(a, 0, N, r)[i]);
    {
        v, displaced := start, b[start];
        k := 0; // ghost
        // one unconditional iteration of the inner loop
        k := k + 1; // ghost
        v := v + N − r;
        if (v ≥ N) { v := v − N; }
        b[v], displaced := displaced, b[v];
        set[v] := true; // ghost
        // Inner loop here: see Figure 31
        assert k = τ(N, N − r);
        start := start + 1;
    }
    // Concluding assertions here: see Figure 33
}

Figure 30: Verified Boogie annotated implementation of the rotation by modular visit algorithm of Figure 13.
framing annotations. To prove that the essential outer loop invariant (30) is induc-
tive, we have to establish that each new iteration works on new elements of b or,
equivalently, that it does not touch the elements set by previous iterations. This non-
interference property ultimately boils down to the fact that the cycle visited by each
outer loop iteration is disjoint from the other cycles. To put this fact to use in the
mechanized proof, we introduce ghost state that keeps track precisely of the visited
locations. The axioms in Figure 29 can then be used to prove that the ghost state changes
following invariants that reflect progress as in the original loop invariant (30).

Concretely, we introduce a Boolean map \( \text{set} \) as ghost state: \( \text{set}[k] \) is \text{true} iff the
imperative code has changed the value of \( b[k] \) from its initial input value \( a[k] \) to its
correct value in the rotation underway. This convention makes it possible to decouple
framing (“what elements the algorithm modifies”) from functional properties (“how
the algorithm modifies the elements”), which simplifies the life of the theorem prover
by bringing the annotations closer in form to the axioms used to verify them, and hence
also simplifies the task of checking each of them individually.

The invariant on line 634 in Figure 30 restates the essential outer loop invariant
(30) in terms if set: if \( \text{set}[i] \) is \text{true} then \( b[i] \) represents the elements at position \( i \)
in a rotation of \( a \) by \( r \). This is equivalent to (30) if combined with the other invariant
on line 631 \( \text{set}[p] \) is \text{true} for the same values of \( i, s, p \) as in the antecedent of (30).

The two other outer loop invariants about \( \text{set} \) (lines 626 and 628) provide the com-
plementary information about what elements have not been modified: \( \text{set}[i] \) is \text{false}
for all \( i \)’s corresponding to values in cycles not visited yet (beginning at indexes larger
than or equal to the current value of \( \text{start} \)); and \( b[i] \) is unchanged for these \( i \)’s.

We have to appropriately update ghost variable \( \text{set} \) during the computation. The
inner loop, which performs the actual visits, also sets \( \text{set}[i] \) to \text{true} whenever it
assigns to \( b[i] \). The rest of the program is only responsible for initializing \( \text{set} \) to all
\text{false} values, which we do with an \text{assume} (line 621) rather than with imperative code
that would needlessly increase the complexity of verification.

Simplifying program state. The remaining bounding outer loop invariants, (25),
(26), and (28), constrain the values of moved and \( \text{start} \). They are redundant since the
value of \( \text{moved} \) between iterations of the outer loop is uniquely determined by the
value \( \text{start} \) through (28). We simplify the program state by omitting \( \text{moved} \) and using
(28) to rewrite properties of \( \text{moved} \) in terms of \( \text{start} \). Then, (26) remains the only
bounding invariant of the outer loop, whose staying condition changes from \( \text{moved} \neq N \)
to \( \text{start} < \gcd(N, N - r) \).

In practice, we realized that this simplification was very useful, if not necessary,
only late while arranging the mechanized proof. Boogie became very sensitive to
adding more annotations and invariants, and it struggled to connect to the postcondition
the final state characterized by the outer loop invariant. Removing the dependence on
\( \text{moved} \) greatly helped, since it simplified the logic of the whole program down to the
inner loop (which incremented \( \text{moved} \)). Since the imperative parts of the program are
modified only minimally (just the assignments that initialize and update \( \text{moved} \)), and we
still prove the same postcondition, we can still consider this a full-fledged mechanized
correctness proof of the original algorithm in Figure 13.

5.5.3 Inner loop

The inner loop in Figure 13 is a \text{repeat...until}, whose body is executed at least
once. Boogie only has one kind of loop (the \text{while} loop), and hence the inner loop body

44
call lemma_rotmp(start, a, 0, N, r, k);

while (v ≠ start)
    invariant 0 ≤ v ∧ v < N;
    invariant 0 < k ∧ k ≤ τ(N, N – r);
    invariant v = mp(N, N – r, start, k);
    invariant displaced = a[mp(N, N – r, start, k)];
    invariant (∀ i: int • 0 ≤ i ∧ i < N ∧ ¬set[i] ⇒ b[i] = a[i]);
    invariant (∀ i: int • k < i ∧ i ≤ τ(N, N – r) ⇒ ¬set[mp(N, N – r, start, i)]);
    invariant (∀ i: int • 0 < i ∧ i ≤ k ⇒ set[mp(N, N – r, start, i)]);
    invariant (∀ i: int, s: int • 0 ≤ i ∧ i < τ(N, N – r) ∧ start < s ∧ s < gcd(N, N – r) ⇒
      ¬set[mp(N, N – r, s, i)]);
    invariant (∀ i: int, s: int • 0 ≤ i ∧ i < τ(N, N – r) ∧ 0 ≤ s ∧ s < start ⇒
      set[mp(N, N – r, s, i)]);
    invariant (∀ i: int • 0 ≤ i ∧ i < N ∧ set[i] ⇒
      seq(b, 0, N)[i] = rot(a, 0, N, r)[i]);
    {
        k := k + 1; // ghost
        v := v + N – r;
        if (v ≥ N) { v := v – N; }
        b[v], displaced := displaced, b[v];
        set[v] := true; // ghost
    }
    call forall lemma_mp(N, N – r, start, *);
    call lemma_rotmp(start, a, 0, N, r, k);
}

Figure 31: Inner loop of the verified Boogie annotated implementation of the rotation by modular visit algorithm of Figure 13.

appears twice: once executed unconditionally right before the inner loop, in Figure 30, and once as body of the inner while loop, in Figure 31. The following discussion applies to both but focuses on the latter.

Progress in the current cycle using a ghost variable. The major novelty in the inner loop is the introduction of a new ghost variable k. The need for k comes quite naturally from observing that the expression \( moved – start \cdot \tau(N, N – r) \) appears twice in the inner loop invariants. The value of this expression enumerates the indexes of the current cycle, each visited by an iteration of the inner loop: a value of 0 corresponds to the first index, a value of 1 to the second index, and so on. Thus, we introduce a ghost variable k that keeps track of this value; this is also consistent with our choice to drop moved and represent its information by means of other variables. k is initialized to 0 in the outer loop before every execution of the inner loop; and is incremented by one in the inner loop body. The inner loop’s bounding invariants 31 and 32 become the invariants on lines 658 and 659 in Figure 31 after substituting k for \( moved – start \cdot \tau(N, N – r) \).
The invariant (33) defines the value of displaced as \((\text{old } a)[v]\), corresponding to just \(a[v]\) in Boogie. However, this formulation does not work well with Boogie, which reasons more directly if the definition of \(v\) is replicated, giving the invariant on line 660.

We express the essential inner loop invariants—in particular (34), specific to the inner loop—in terms of \(set\) as we expressed the essential outer loop invariant. (34) determines two new invariants on lines 662 and 664. Both predicate about indexes in the currently visited cycle. The former invariant targets those not visited yet, for positions larger than \(k\); the latter targets those visited, for positions up to \(k\). The remaining framing invariants are as in the outer loop.

\begin{verbatim}
684 procedure lemma_mp(N: int, M: int, s: int, p: int)
685 requires 0 < M ∧ M < N;
686 requires 0 ≤ s ∧ s < N;
687 requires p ≥ 0;
688 ensures 0 ≤ mp(N, M, s, p) < N;
689 { // proof by induction
690 if (p = 0) { } else { call lemma_mp(N, m, s, p - 1); }
691 }
692
693 procedure lemma_rotmp(s: int, a: [int]int, low: int, high: int, r: int, k: int)
694 requires 0 < r ∧ r < high - low ∧ 0 ≤ s ∧ s < high - low;
695 requires k > 0;
696 ensures rot(a, low, high, r)[mp(high - low, high - low - r, s, k)]
697 = seq(a, low, high)[mp(high - low, high - low - r, s, k - 1)];
698 { // proof by induction
699 if (k = 1) { } else {
700 // lemma_mp makes it possible to apply the definition of rot
701 call lemma_mp(high - low, high - low - r, s, k - 1);
702 }
703 }
\end{verbatim}

Figure 32: Lemmas used to prove the inner loop of rotation by modular visit.

**Lemmas to prove inductiveness.** To prove the inductiveness of the inner loop invariants, Boogie needs a little help in the form of two lemmas about properties of function \(mp\), whose statements and proofs are shown in Figure 32. \texttt{lemma_mp} simply bounds \(mp(N, M, s, p)\) to be nonnegative and less than \(N\). This is a consequence of the definition of \(mp\) in terms of \(\text{wrap } N\), but we need to nudge Boogie to use this property among the many others that could be proved. The Boogie proof is by induction, corresponding to a conditional \texttt{if} in the lemma procedure: the inductive step calls the lemma for the previous value of \(p - 1\) assumed by inductive hypothesis; since the definition of \(mp(N, M, s, p)\) is in terms of \(mp(N, M, s, p - 1)\), this is enough to close the proof.

\texttt{lemma_rotmp} asserts that two elements at consecutive indexes in a cycle (that is, two evaluations of function \(mp\) for successive values of its last argument), relate el-
ments in a rotation. This is an important property that explicitly connects the indexes in the cycles to the definition of rotation. Boogie can prove it by induction: the inductive step calls lemma_mp whose bounds justify the application of the definition of rot; based on this, the SMT solver combines the axiomatic definitions of rot and mp to prove the lemma.

We close the body of the inner loop by calling lemma_mp followed by lemma_rotmp; the order matters since the former asserts a more fundamental property on which the latter builds. Note that we also need to recall lemma_rotmp before entering the inner loop, to prove initiation after one unconditional execution of the loop body.

There remains one simple element of specification needed to guide Boogie’s proof to success. Even if this is, once again, a consequence of the definition of mp, we have to express it as a new bounding loop invariant on \( v \):

\[
0 \leq v < N
\]

This guarantees that the accesses to \( b[v] \) are within the bounds the other invariants predicate about. In fact, recalling lemma_mp in the loop body helps prove this invariant, which is then used in the rest of the proof.

**Variants and performance.** The Boogie proof is sensitive to the order in which some invariants appear and the ghost state is updated. To achieve a bit more robustness, we could add ghost state to make for a more step-wise proof of inductiveness. For example, we could add a ghost \( c \) map that represents the value of \( b \) in the previous iteration, so that the inductiveness proof uses facts about \( c \) as inductive hypotheses and only has to prove the inductive step about the latest update in \( b \). We do not discuss this variant in more detail and prefer the terser proof presented above.

```boogie
assert (\( \forall i: \text{int}, s: \text{int} \bullet \)
\[
0 \leq i \land i < \tau(N, N - r) \land 0 \leq s \land s < \gcd(N, N - r)
\] 
\implies
\] 
set[mp(N, N - r, s, i)]);
assert (\( \forall i: \text{int} \bullet 0 \leq i \land i < N \land set[i] \implies
\] 
seq(b, 0, N)[i] = rot(a, 0, N, r)[i]);
assert \( 0 < N - r \land N - r < N \);
call forall lemma_wrap_bounds(*, gcd(N, N - r));
assert (\( \forall i: \text{int} \bullet 0 \leq i \land i < N \implies
\] 
set[mp(N, N - r, wrap(i, gcd(N, N - r))),
\] 
yp(N, N - r, wrap(i, gcd(N, N - r)), i)]);
call forall lemma_yp_mp(N, N - r, *, set);
call lemma_extensional(N, N - r, set);
assert (\( \forall i: \text{int} \bullet 0 \leq i \land i < N \implies
\] 
set[i]);
```

Figure 33: Concluding assertions in the verified Boogie annotated implementation of the rotation by modular visit algorithm of Figure 13

5.5.4 Proof conclusion

At high level, the Boogie proof of the postcondition from the outer loop invariants and exit condition follows the same steps as the one illustrated in Section 4.4.2. The assertions in Figure 33 correspond to such final steps: the first two assertions recall the essential outer loop invariants upon exiting the loop; then two simple arithmetic facts about \( N - r \) and wrap are recalled (the second fact in the form of a lemma procedure wrap_bounds corresponding to a formal statement of (3)); then an assertion
\[ yp(N, m, s, i) = \text{p iff } mp(N, m, s, p) = i \]

```plaintext
function yp(N: int, m: int, s: int, i: int) returns (int);

axiom (\forall N: int, m: int, i: int •
\[ 0 < m \land m < N \land 0 \leq i \land i < N \implies \]
\[ 0 \leq yp(N, m, wrap(i, gcd(N, m)), i) < \tau(N, m) \]);

axiom (\forall N: int, m: int, i: int •
\[ 0 < m \land m < N \land 0 \leq i \land i < N \implies \]
\[ mp(N, m, wrap(i, gcd(N, m)), yp(N, m, wrap(i, gcd(N, m)), i)) = i \]);

procedure lemma_yp_mp(N: int, m: int, i: int, set: [int]bool);
requires 0 \leq i \land i < N;
requires 0 < m \land m < N;
requires 0 \leq wrap(i, gcd(N, m)) < gcd(N, m);
requires 0 \leq yp(N, m, wrap(i, gcd(N, m)), i) < \tau(N, m);
requires mp(N, m, wrap(i, gcd(N, m)),
\[ yp(N, m, wrap(i, gcd(N, m)), i)) = i ; \]
requires set[mp(N, m, wrap(i, gcd(N, m)),
\[ yp(N, m, wrap(i, gcd(N, m)), i))] ; \]
ensures set[i];
{
}
```

```plaintext
procedure lemma_extensional(N: int, m: int, set: [int]bool);
requires 0 < m \land m < N;
requires (\forall i: int • 0 \leq i \land i < N \implies \]
\[ 0 \leq wrap(i, gcd(N, m)) < gcd(N, m) ; \]
requires (\forall i: int • 0 \leq i \land i < N \implies \]
\[ 0 \leq yp(N, m, wrap(i, gcd(N, m)), i) < \tau(N, m) ; \]
requires (\forall i: int • 0 \leq i \land i < N \implies \]
\[ mp(N, m, wrap(i, gcd(N, m)), yp(N, m, wrap(i, gcd(N, m)), i)) = i ; \]
requires (\forall i: int • 0 \leq i \land i < N \implies \]
\[ set[mp(N, m, wrap(i, gcd(N, m)),
\[ yp(N, m, wrap(i, gcd(N, m)), i))] ; \]
free ensures (\forall i: int • 0 \leq i \land i < N \implies set[i]) ;
```

Figure 34: Lemmas and additional definitions used in the conclusion of the proof of rotation by modular visit.

and a lemma yp. mp capture and use the statement of Lemma 3 the concluding call to lemma_extensional and assertion on the last line in Figure 33 are technicalities that we discuss last.

Let us focus on the interesting part of expressing Lemma 3 in Boogie. Informally, Lemma 3 shows how to “invert” mp so that the indexes it enumerates can be shown to span the whole domain of the input array. The proof of Lemma 3 uses fundamental properties of modular arithmetic that we avoided axiomatizing in detail in Boogie. Instead, we extend the axiomatization at the same level of abstraction used so far in the mechanized proofs by introducing the definitions in Figure 34. Function yp is like an inverse of mp, as declared by the axioms in Figure 34—the second one in particular which declares that \[ mp(\ldots, yp(\ldots, i)) = i \]. We postulate its existence, instead of proving it from simpler principles as Lemma 3 does. Then, lemma procedure lemma_yp_mp connects mp to its inverse in the context in which they are used in
the proof: if \( \text{set}(\mp(..., \yp(..., i))) \) for any \( i \) between \( 0 \) and \( N \), then \( \text{set}(i) \) as well. The concluding proof in Figure 33 recalls \( \text{lemma}_\text{yp}_\text{mp} \). For speed, it asserts one of the procedure’s preconditions before calling to focus the proof context.

At this point it would seem that all facts are available to prove the postcondition. Procedure \( \text{lemma}_\text{yp}_\text{mp} \) concludes that \( \text{set}(i) \) for all \( 0 \leq i < N \) and the essential outer loop invariant (repeated by an \text{assert}) upon exiting says that \( b[0...N) \) is \( a[0...N) \)'s rotation for all \( i \) such that \( \text{set}(i) \). Nonetheless, we have to shoehorn the final conclusion into lemma procedure \text{extensional}, followed by an \text{assert} that reaffirms its postcondition. Specifically, the prover refuses to match the identical quantifications over \( [0..N) \) in the \text{call forall} of \( \text{lemma}_\text{yp}_\text{mp} \)'s postcondition and in:

\[
\text{assert} \ (\forall i: \text{int} \ \bullet \ 0 \leq i \land i < N \implies \text{set}(i)).
\]

While Boogie can prove the same assertion if put in the body of \text{lemma extensional}, it still cannot match it to the lemma's identical postcondition. Using triggers does not seem to help. Since this is clearly due to inessential details of quantifier instantiation in the SMT solver, we simply declare the property as a \text{free ensures} and use it without guilt. Finally, note that using a \text{free ensures} in a lemma procedure is generally preferable to using an axiom with the same statement, because its application is on demand—where it is really needed—rather than being a fact always available—cluttering the proof anywhere else.

6 Discussion

We have seen that the four main algorithms require increasingly more complex correctness proofs—both on paper and, even more so, when mechanizing them. This verification complexity does have relevant practical implications: the more complex the correctness argument, the more complex it is to get an implementation correct or to modify an existing implementation (for example to work on a different kind of data structure) without introducing subtle errors. In my own experience of implementing the algorithms in Java, getting rotation by reversal right is straightforward, rotation by swapping requires more attention mostly to corner cases, rotation by modular visit can be tricky to implement correctly at the first attempt. The likelihood of introducing errors is yet another feature to be traded off against others such as performance when choosing which algorithm to implement.

Auto-active verifiers such as Boogie generate verification conditions in multiple steps, often involving heuristics that may be hard to express and opaque to users. We have seen several cases in the mechanized proofs, especially of the more complex algorithms, were changing seemingly irrelevant details of the input (such as the order or names of declarations) transformed a successful proof into a very slow or even non-terminating one. The best defense against such brittleness is to put great care into modularizing proofs into files and procedures that are as separate as possible: provided each individual input to the verifier is sufficiently small (or large but structurally simple) the sensitivity on low-level details is kept at bay and does not interfere with the high-level goals of the prover.

\[10\]The name is because the form of the property reminds one of extensionality.
References

[1] Nada Amin, K. Rustan M. Leino, and Tiark Rompf. Computing with an SMT solver. In Proceedings of the 8th International Conference on Tests and Proofs (TAP), Lecture Notes in Computer Science. Springer, 2014.

[2] Jon Bentley. Programming Pearls. Addison-Wesley Professional, 2nd edition, 1999.

[3] Richard Bornat. Lecture slides on algorithms (circular shift). http://www.eis.mdx.ac.uk/staffpages/r_bornat/oldteaching/I2A/ 1998.

[4] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. Journal of the ACM, 52(3):365–473, 2005.

[5] Carlo A. Furia, Bertrand Meyer, and Sergey Velder. Loop invariants: Analysis, classification, and examples. ACM Computing Surveys, 46(3):Article 34, January 2014.

[6] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. Concrete Mathematics: A foundation for computer science. Addison-Wesley, 2nd edition, 1994.

[7] David Gries and Harlan Mills. Swapping sections. Technical Report TR 81-452, Cornell University, January 1981. http://hdl.handle.net/1813/6292

[8] K. Rustan M. Leino. This is Boogie 2, 2008. http://goo.gl/QsH6g

[9] K. Rustan M. Leino and Rosemary Monahan. Reasoning about comprehensions with first-order SMT solvers. pages 615–622. ACM, 2009.

[10] Joseph J. Rotman. Advanced Modern Algebra. American Mathematical Society, 2nd edition, 2010.

[11] Song Y. Yan. Number Theory for Computing. Springer, 2nd edition, 2002.