ACSL by Example

Towards a Formally Verified Standard Library

Version 21.1.1
for
Frama-C 21.1 (Scandium)
September 2020

Jens Gerlach
Denis Efremov
Tim Sikatzki

Former Authors

Malte Brodmann, Jochen Burghardt,
Andreas Carben Robert Clausecker,
Liangliang Gu, Kerstin Hartig,
Timon Lapawczyk, Hans Werner Pohl,
Juan Soto, Kim Völlinger

Fraunhofer
FOKUS

This report was partially funded by the VESSEDIA project.[1]

The research leading to these results has received funding from the STANCE project within European Union's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 317753.[2]

This body of work was completed within the Device-Soft project, which was supported by the Programme Inter Carnot Fraunhofer from BMBF (Grant 01SF0804) and ANR.[3]

This document is hosted at

`https://github.com/fraunhoferfokus/acsl-by-example`

From there, you can also download the source code of all algorithms discussed here, their contracts, and the employed predicate definitions and lemmas. All examples are developed and proved with the Frama-C/WP [1] plugin.[4] We recommend using the GitHub issue tracker

`https://github.com/fraunhoferfokus/acsl-by-example/issues`

to report suggestions or errors. Alternatively, you can email them also to

`jens.gerlach@fokus.fraunhofer.de`

[1] The project VESSEDIA has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731453. Project duration: 2017–2019, see `https://vessedia.eu`

[2] Project duration: 2012–2016, see `http://www.stance-project.eu`

[3] Project duration: 2009–2012

[4] There is also full support for the Frama-C/AstraVer plugin which is developed at ISP RAS and can be installed with the instruction available on `https://forge.ispras.ru/projects/astraver/wiki`

1. Changes

For changes in previous versions we refer to Appendix B on Page 259.

1.1. New in Version 21.1.1 (Scandium, September 2020)

This release is intended for Frama-C [2, v21.1] issued in June 2020. We are also using for this release the Why3 platform [3, v1.3.3] and the provers listed in the following table.

Prover	Type	Version	Reference
Alt-Ergo	automatic	2.3.3	[4]
CVC4	automatic	1.7	[5]
CVC3	automatic	2.4.1	[6]
Z3	automatic	4.8.6	[7]
Coq	interactive	8.9.1	[8]

Table 1.1.: Information on automatic and interactive theorem provers

Note that all automatic provers use the Why3 interface. However, the interactive prover Coq still relies on the native interface provided by Frama-C/WP.

New examples

None.

Improvements

- general changes
  - disable CVC3 and switch back to Z3 `4.8.6`
  - refactor Coq proofs
  - reduce general timeout to 1s (Coq timeout 5s)
- improve loop invariants of `remove_copy3` and `remove` to reduce timeout
- add lemma `AdjacentDifference_InverseBounds` to reduce timeout of `adjacent_difference_inv` robust
- add lemmas `Count_Single`, `Count_Single_Bounds`, `Count_Single_Shift` and `Count_Cut`
- heap algorithms
  - rework `push_heap` and add more assertions to reduce timeout
  - rework `pop_heap` and finally verify property `reorder`

- add predicates ArrayUpdate, MultisetParity, MultisetUpdate and supporting lemmas
- remove predicate PushHeapAdjust and accompanying lemmas
- rename heap_child_max to heap_child and improve both contract and implementation
- add lemma HeapParent_Zero

- sorting and reordering
  - rework contract and annotations of merge
  - no need more for option -wp-split
  - add lemma WeaklyIncreasing_Shrink
  - add lemma WeaklyIncreasing_Unchanged
  - add more annotations to bubble_sort to reduce timeout
  - add lemma MultisetSwap_FrontMiddle

Open issues

- The contract of algorithm merge does not handle the reordering of the involved arrays.

Renaming of ACSL definitions

We sometimes rename predicates, logic functions or lemmas in order to make them more precise or make the naming more consistent.

- rename suffix _Read to _Unchanged in names of lemmas

Old name	New name
EqualRanges	Equal
Equal_Increasing	Increasing_Equal
Equal_Count	Count_Equal
MultisetUnchanged	MultisetReorder
MultisetUnchanged_Union	MultisetReorder_DisjointUnion
Reorder_Match	MultisetReorder_SomeEqual
Reorder_LowerBound	MultisetReorder_LowerBound
Reorder_LowerBounds	MultisetReorder_PartitionLowerBound
Reorder_UpperBound	MultisetReorder_UpperBound
SwappedInside	ArraySwap
SwappedInside_Reorder	MultisetSwap_Middle

Contents

List of Figures

List of Tables

Part I.

Basics

2. Introduction

This report provides various examples for the formal specification, implementation, and deductive verification of C programs using the ANSI/ISO-C Specification Language (ACSL [9]) and the Frama-C/WP plug-in [1] of Frama-C [2] (Framework for Modular Analysis of C programs).

We have chosen our examples from the C++ Standard Library whose initial version is still known as the *Standard Template Library* (STL). The C++ Standard Library contains a broad collection of *generic* algorithms that work not only on C arrays but also on more elaborate container data structures. For the purposes of this document we have selected representative algorithms, and converted their implementation from C++ function templates to C functions that work on arrays of type `int`.

We will continue to extend and refine this report by describing additional STL algorithms and data structures. Thus, step by step, this document will evolve from an ACSL tutorial to a report on a formally specified and deductively verified Standard Library for ANSI/ISO-C. Moreover, as ACSL is extended to a C++ specification language, our work may be extended to a deductively verified C++ Standard Library.

We encourage you to check vigilantly whether our formal specifications capture the essence of the informal description of the STL algorithms. We appreciate your feedback[5] and hope that this document helps foster the adoption of deductive verification techniques.

Acknowledgement

Many members from the Frama-C community provided valuable input and comments during the course of the development of this document. In particular, we wish to thank our project partners Patrick Baudin, Allan Blanchard, Loïc Correnson, Zaynah Dargaye, Florent Kirchner, Virgile Prevosto, and Armand Puccetti from CEA LIST[6] and Pascal Cuoq from TrustInSoft[7].

We also like to express our gratitude to Claude Marché (LRI/INRIA)[8] and Yannick Moy (AdaCore)[9] for their helpful comments and detailed suggestions for improvement. Finally, we would like to thank Aaron Rocha who sent us valuable improvement suggestions and error reports.

[5]We suggest GitHub's issue tracker: `https://github.com/fraunhoferfokus/acsl-by-example/issues`
[6]`http://www-list.cea.fr/en`
[7]`http://trust-in-soft.com`
[8]`https://www.lri.fr/index_en.php?lang=EN`
[9]`http://www.adacore.com`

2.1. Frama-C

The Framework for Modular Analyses of C, Frama-C [2], is a suite of software tools dedicated to the analysis of C source code. Its development efforts are conducted and coordinated at two French public institutions: CEA LIST [10], a laboratory of applied research on software-intensive technologies, and INRIA Saclay [11], the French National Institute for Research in Computer Science and Control in collaboration with LRI [12], the Laboratory for Computer Science at Université Paris-Sud.

ACSL (ANSI/ISO-C Specification Language) [9] is a formal language to express behavioral properties of C programs. This language can specify a wide range of functional properties by adding annotations to the code. It allows to create function contracts containing preconditions and postconditions. It is possible to define type and global invariants as well as logic specifications, such as predicates, lemmas, axioms or logic functions. Furthermore, ACSL allows statement annotations such as assertions or loop annotations.

Within Frama-C, the Frama-C/WP plug-in [1] enables deductive verification of C programs that have been annotated with ACSL. The Frama-C/WP plug-in uses Hoare-style weakest precondition computations to formally prove ACSL properties of C code. Verification conditions are generated and submitted to external automatic theorem provers or interactive proof assistants.

The Verification Group at Fraunhofer FOKUS [13] see the great potential for deductive verification using ACSL. However, we recognize that for a novice there are challenges to overcome in order to effectively use the Frama-C/WP plug-in for deductive verification. In order to help users gain confidence, we have written this tutorial that demonstrates how to write annotations for existing C programs. This document provides several examples featuring a variety of annotated functions using ACSL. For an in-depth understanding of ACSL, we strongly recommend users to read the official Frama-C introductory tutorial [14] first. The principles presented in this paper are also documented in the ACSL reference document [15].

2.2. Structure of this document

The functions presented in this document were selected from the C++ Standard Library. The original C++ implementation was stripped from its generic implementation and mapped to C arrays of type `value_type`.

Chapter 3 provides a short introduction into the Hoare Calculus. For a better understanding of Frama-C/WP and the theory behind it, we also recommend Allan Blanchard's ACSL tutorial [16].

We have grouped various standard algorithms in chapters as follows:

- non-mutating algorithms (Chapter 4)
- maximum/minimum algorithms (Chapter 5)
- binary search algorithms (Chapter 6)
- mutating algorithms (Chapter 7)
- numeric algorithms (Chapter 8)
- heap algorithms (Chapter 9)
- sorting algorithms and well-known classical implementations of sorting algorithms (Chapter 10)

The order of these chapters reflects their increasing complexity.

Using the example of a stack, we tackle in Chapter 11 the problem of how a data type and its associated C functions can be specified with ACSL and automatically verified with Frama-C.

Finally, Appendix A lists for each example the results of verification with Frama-C.

2.3. Types, arrays, ranges and valid indices

In order to keep algorithms and specifications as general as possible, we use abstract type names on almost all occasions. We currently defined the following types:

```
typedef int value_type;

typedef unsigned int size_type;

typedef int bool;
```

Programmers who know the types associated with C++ Standard Library containers will not be surprised that `value_type` refers to the type of values in an array whereas `size_type` will be used for the indices of an array.

This approach allows one to modify, say, an algorithm working on an **`int`** array to work on a **`char`** array by changing only one line of code, viz. the **`typedef`** of `value_type`. Moreover, we believe in better readability as it becomes clear whether a variable is used as an index or as a memory for a copy of an array element, just by looking at its type.

The latter reason also applies to the use of **`bool`**. To denote values of that type, we defined the identifiers **`false`** and **`true`** to be `0` and `1`, respectively. While any non-zero value is accepted to denote **`true`** in ACSL like in C the algorithms shown in this tutorial will always produce `1` for **`true`**. Due to the above definitions, the ACSL truth-value constant `\false` and `\true` can be used interchangeably with our **`false`** and **`true`**, respectively, in ACSL clauses, but not in C code.

2.3.1. Array and ranges

The C Standard describes an array as a "contiguously allocated nonempty set of objects" [17, §6.2.5.20]. If `n` is a constant integer expression with a value greater than zero, then

```
int a[n];
```

describes an array of type **`int`**. In particular, for each `i` that is greater than or equal to 0 and less than `n`, we can dereference the pointer `a+i`.

Let the following prototype represent a function, whose first argument is the address to a range and whose second argument is the length of this range.

```
void example(value_type* a, size_type n);
```

To be very precise, we have to use the term *range* instead of *array*. This is due to the fact, that functions may be called with empty ranges, i.e., with `n == 0`. Empty arrays, however, are not permitted according to the definition stated above. Nevertheless, we often use the term *array* and *range* interchangeably.

2.3.2. Specification of valid ranges in ACSL

The following ACSL fragment expresses the precondition that the function `example` expects that for each `i`, such that `0 <= i < n`, the pointer `a+i` may be safely dereferenced.

```
/*@
    requires 0 <= n;
    requires \valid(a + (0.. n-1));
*/
void example(value_type* a, size_type n);
```

In this case we refer to each index `i` with `0 <= i < n` as a *valid index* of `a`.

ACSL's built-in predicates `\valid(a + (0.. n))` and `\valid_read(a + (0.. n))` refer to all addresses `a+i` where `0 <= i <= n`. However, the array notation **`int`** `a[n]` of the C programming language refers only to the elements `a+i` where `i` satisfies `0 <= i < n`. Users of ACSL must therefore use the range notation `a+(0.. n-1)` in order to express a valid array of length `n`.

3. The Hoare calculus

In 1969, C.A.R. Hoare introduced a calculus for formal reasoning about properties of imperative programs [18], which became known as “Hoare Calculus”.

The basic notion is

```
//@ assert P;
Q;
//@ assert R;
```

where `P` and `R` denote logical expressions and `Q` denotes a source-code fragment. Informally, this means

If `P` holds before the execution of `Q`, then `R` will hold after the execution.

Usually, `P` and `R` are called *precondition* and *postcondition* of `Q`, respectively. The syntax for logical expressions is described in [15, §2.2] in full detail. For the purposes of this tutorial, the notions shown in Table 3.1 are sufficient. Note that they closely resemble the logical and relational operators in C.

ACSL syntax	Name	Reading
`!P`	negation	`P` is not true
`P && Q`	conjunction	`P` is true and `Q` is true
`P || Q`	disjunction	`P` is true or `Q` is true
`P ==> Q`	implication	if `P` is true, then `Q` is true
`P <==> Q`	equivalence	if, and only if, `P` is true, then `Q` is true
`x < y == z`	relation chain	`x` is less than `y` and `y` is equal to `z`
`\forall int x; P(x)`	universal quantifier	`P(x)` is true for every `int` value of `x`
`\exists int x; P(x)`	existential quantifier	`P(x)` is true for some `int` value of `x`

Table 3.1.: Some ACSL formula syntax

Here we show three example source-code fragments and annotations.

`//@ assert x % 2 == 1;`
`++x;`
`//@ assert x % 2 == 0;`
	If `x` has an odd value before execution of the code `++x` then `x` has an even value thereafter.

`//@ assert 0 <= x <= y;`
`++x;`
`//@ assert 0 <= x <= y + 1;`
	If the value of `x` is in the range $\{0, \ldots, y\}$ before execution of the same code, then `x`’s value is in the range $\{0, \ldots, y + 1\}$ after execution.

`//@ assert true;`
`while (--x != 0)`
`    sum += a[x];`
`//@ assert x == 0;`
	Under any circumstances, the value of `x` is zero after execution of the loop code.

Any C programmer will confirm that these properties are valid.[10] The examples were chosen to demonstrate also the following issues:

- For a given code fragment, there does not exist one fixed pre- or postcondition. Rather, the choice of formulas depends on the actual property to be verified, which comes from the application context. The first two examples share the same code fragment, but have different pre- and postconditions.
- The postcondition need not be the most restricting possible formula that can be derived. In the second example, it is not an error that we stated only that `0 <= x` although we know that even `1 <= x`.
- In particular, pre- and postconditions need not contain all variables appearing in the code fragment. Neither `sum` nor `a[]` is referenced in the formulas of the loop example.
- We can use the predicate **`true`** to denote the absence of a properly restricting precondition, as we did before the **`while`** loop.
- It is not possible to express by pre- and postconditions that a given piece of code will always terminate. The loop example only states that *if* the loop terminates, then `x == 0` will hold. In fact, if `x` has a negative value on entry, the loop will run forever. However, if the loop terminates, `x == 0` will hold, and that is what the loop example claims.

  Usually, termination issues are dealt with separately from correctness issues. Termination proofs may, however, refer to properties stated (and verified) using the Hoare Calculus.

Hoare provided the rules shown in Listing 3.2 to 3.12 in order to reason about programs. We will comment on them in the following sections.

[10] We leave the important issues of overflow aside for a moment.

3.1. The assignment rule

We start with the rule that is probably the least intuitive of all Hoare-Calculus rules, viz. the assignment rule. It is depicted in Listing 3.2, where

$$\mathtt{P}\,\{\mathtt{x} \mapsto \mathtt{e}\}$$

denotes the result of substituting each occurrence of the variable `x` in the predicate `P` by the expression `e`.

```
//@ assert P {x |--> e};
x = e;
//@ assert P;
```

Listing 3.2: The assignment rule

For example, if `P` is the predicate

```
  x > 0 && a[2*x] == 0
```

then $\mathtt{P}\,\{\mathtt{x} \mapsto \mathtt{y + 1}\}$ is the predicate

```
y+1 > 0 && a[2*(y+1)] == 0
```

Hence, we get Listing 3.3 as an example instance of the assignment rule. Note that parentheses are required in the index expression to get the correct `2*(y+1)` rather than the faulty `2*y+1`.

```
//@ assert y+1 > 0 && a[2*(y+1)] == 0;
x = y+1;
//@ assert x > 0 && a[2*x] == 0;
```

Listing 3.3: An assignment rule example instance

Note that after a substitution several different predicates `P` may result in the same predicate $\mathtt{P}\,\{\mathtt{x} \mapsto \mathtt{e}\}$. For example, after applying the substitution $\mathtt{P}\,\{\mathtt{x} \mapsto \mathtt{y + 1}\}$ each of the following four predicates

```
  x > 0 && a[2*x]      == 0
  x > 0 && a[2*(y+1)] == 0
y+1 > 0 && a[2*x]      == 0
y+1 > 0 && a[2*(y+1)] == 0
```

turns into

```
y+1 > 0 && a[2*(y+1)] == 0
```

For this reason, the same precondition and statement may result in several different postconditions (All four above expressions are valid postconditions in Listing 3.3, for example). However, given a postcondition and a statement, there is only one precondition that corresponds.

When first confronted with Hoare's assignment rule, most people are tempted to think of a simpler and more intuitive alternative, shown in Listing 3.4.

```
//@ assert P;
x = e;
//@ assert P && x == e;
```

Listing 3.4: Simpler, but *faulty* assignment rule

Listings 3.5–3.7 show some example instances of this faulty rule.

```
//@ assert y > 0;
x = y+1;
//@ assert y > 0 && x == y+1;
```

Listing 3.5: An example instance of the faulty rule from Listing 3.4

While Listing 3.5 happens to be ok, Listing 3.6 and 3.7 lead to postconditions that are obviously nonsensical formulas.

```
//@ assert true;
x = x+1;
//@ assert x == x+1;
```

Listing 3.6: An example instance of the faulty rule from Listing 3.4

The reason is that in the assignment in Listing 3.6 the left-hand side variable `x` also appears in the right-hand side expression `e`, while the assignment in Listing 3.7 just destroys the property from its precondition.

```
//@ assert x < 0;
x = 5;
//@ assert x < 0 && x == 5;
```

Listing 3.7: An example instance of the faulty rule from Listing 3.4

Note that the correct example Listing 3.5 can as well be obtained as an instance of the correct rule from Listing 3.2, since replacing `x` by `y+1` in its postcondition yields `y > 0 && y+1 == y+1` as precondition, which is logically equivalent to just `y > 0`.

3.2. The sequence rule

The sequence rule, shown in Listing 3.8, combines two code fragments `Q` and `S` into a single one `Q ; S`. Note that the postcondition for `Q` must be identical to the precondition of `S`. This just reflects the sequential execution ("first do `Q`, then do `S`") on a formal level. Thanks to this rule, we may "annotate" a program with interspersed formulas, as it is done in Frama-C.

```
//@ assert P;
Q;
//@ assert R;
```

and

```
//@ assert R;
S;
//@ assert T;
```

⇝

```
//@ assert P;
Q ; S;
//@ assert T;
```

Listing 3.8: The sequence rule

3.3. The implication rule

The implication rule, shown in Listing 3.9, allows us at any time to sharpen a precondition `P` and to weaken a postcondition `R`. More precisely, if we know that `P' ==> P` and `R ==> R'` then the we can replace the left contract in of Listing 3.9 by the right one.

```
//@ assert P;
Q;
//@ assert R;
```

⇝

```
//@ assert P';
Q;
//@ assert R';
```

Listing 3.9: The implication rule

3.4. The choice rule

The choice rule, depicted in Listing 3.10, is needed to verify conditional statements of the form

```
if (C) X;
else   Y;
```

Both the then and else branch must establish the same postcondition, viz. `S`. The implication rule can be used to weaken differing postconditions `S1` of a then-branch and `S2` of an else-branch into a unified postcondition `S1 || S2`, if necessary. In each branch, we may use what we know about the condition `C`. For example, in the else-branch, we may use that `C` is false. If the else-branch is missing, it can be considered as consisting of an empty sequence, having the postcondition `P && !C`.

```
//@ assert P && C;
X;
//@ assert S;
```

and

```
//@ assert P && !C;
Y;
//@ assert S;
```

$\rightsquigarrow$

```
//@ assert P;
if (C) X;
else   Y;
//@ assert S;
```

Listing 3.10: The choice rule

Listing 3.11 shows an example application of the choice rule.

```
//@ assert 0 <= i < n;           // given precondition
if (i < n-1) {
  //@ assert 0 <= i < n - 1;     // using that i < n-1 holds in this branch
  //@ assert 1 <= i+1 < n;       // by the implication rule
  i = i+1;
  //@ assert 1 <= i < n;         // by the assignment rule
  //@ assert 0 <= i < n;         // weakened by the implication rule
} else {
  //@ assert 0 <= i == n-1 < n; // using that !(i < n-1) holds in else part
  //@ assert 0 == 0 && 0 < n;    // weakened by the implication rule
  i = 0;
  //@ assert i == 0 && 0 < n;    // by the assignment rule
  //@ assert 0 <= i < n;         // weakened by the implication rule
}
//@ assert 0 <= i < n;           // by the choice rule from both branches
```

Listing 3.11: An example application of the choice rule

The variable `i` may be used as an index into a ring buffer `int a[n]`. The shown code fragment just advances the index `i` appropriately. We verified that `i` remains a valid index into `a[]` provided it was valid before. Note the use of the implication rule to establish preconditions for the assignment rule as needed, and to unify the postconditions of the then and else branches, as required by the choice rule.

3.5. The loop rule

The loop rule, shown in Listing 3.12, is used to verify a `while` loop. This requires to find an appropriate formula, `P`, which is preserved by each execution of the loop body. `P` is also called a loop invariant.

```
//@ assert P && B;
S;
//@ assert P;
```

$\leadsto$

```
//@ assert P;
while (B) {
  S;
}
//@ assert !B && P;
```

Listing 3.12: The loop rule

To find it requires some intuition in many cases; for this reason, automatic theorem provers usually have problems with this task.

As said above, the loop rule does not guarantee that the loop will always eventually terminate. It merely assures us that, if the loop has terminated, the postcondition holds. To emphasize this, the properties verifiable with the Hoare Calculus are usually called "partial correctness" properties, while properties that include program termination are called "total correctness" properties.

As an example application, let us consider an abstract ring-buffer. Listing 3.13 shows a verification proof for the index `i` lying always within the valid range `[0..n-1]` during, and after, the loop. It uses the proof from Listing 3.11 as a sub-part.

```
//@ assert  0 < n;                  // given precondition

int i = 0;
//@ assert  0 <= i < n;             // by the assignment rule

while (!done) {
  //@ assert 0 <= i < n && !done;   // may be assumed by the loop rule

  a[i] = getchar();
  //@ assert 0 <= i < n && !done;   // required property of getchar
  //@ assert 0 <= i < n;            // weakened by the implication rule

  i = (i < n-1) ? i+1 : 0;
  //@ assert 0 <= i < n;            // follows by the choice rule

  process(a, i, &done);
  //@ assert 0 <= i < n;            // required property of process
}
//@ assert 0 <= i < n;              // by the loop rule
```

Listing 3.13: An abstract ring buffer loop

To reuse the proof from Listing 3.11, we had to drop the conjunct `!done`, since we didn't consider it in Listing 3.11. In general, we may *not* infer

```
//@ assert P && S;
Q;
//@ assert R && S;
```

from

```
//@ assert P;
Q;
//@ assert R;
```

since the code fragment `Q` may just destroy the property `S`.

This is obvious for `Q` being the fragment from Listing 3.11, and `S` being e.g. `i != 0`.

Suppose for a moment that `process` had been implemented in a way such that it refuses to set `done` to **`true`** unless it is **`false`** at entry. In this case, we would really need that `!done` still holds after execution of Listing 3.11. We would have to do the proof again, looping-through an additional conjunct `!done`.

We have similar problems to carry the property `0 <= i < n && !done` and `0 <= i < n` over the statement `a[i] = getchar()` and `process(a, i, &done)`, respectively. We need to specify that neither `getchar` nor `process` is allowed to alter the value of `i` or `n`. In ACSL, there is a particular language construct `assigns` for that purpose, which is introduced in §7.3 on Page 105.

In our example, the loop invariant can be established between any two statements of the loop body. However, this need not be the case in general. The loop rule only requires the invariant holds before the loop and at the end of the loop body. For example, `process` could well change the value of `i`[11] and even `n` intermediately, as long as it re-establishes the property `0 <= i < n` immediately prior to returning.

The loop invariant, `0 <= i < n`, is established by the proof in Listing 3.11 also after termination of the loop. Thus, e.g., a final `a[i] = '\0'` after the loop would be guaranteed not to lead to a bounds violation.

Even if we would need the property `0 <= i < n` to hold only immediately before the assignment `a[i] = getchar()`, for example since `process`'s body didn't use `a` or `i`, we would still have to establish `0 <= i < n` as a loop invariant by the loop rule, since there is no other way to obtain any property inside a loop body. Apart from this formal reason it is obvious that `0 <= i < n` wouldn't hold during the second loop iteration unless we re-established it at the end of the first one, and that is just what the while rule requires.

[11] We would have to change the call to `process(a, &i, &done)` and the implementation of `process` appropriately. In this case we couldn't rely on the above-mentioned `assigns` clause for `process`.

3.6. Derived rules

The above rules do not cover all kinds of statements allowed in C. However, missing C-statements can be rewritten into a form that is semantically equivalent and covered by the Hoare rules.

For example, if the expression `E` doesn't have side-effects, then

```
switch (E) {
    case E1: Q1; break; ...
    case En: Qn; break;
    default: Q0; break;
}
```

is semantically equivalent to

```
if (E == E1) {
    Q1;
} else ... if (E == En) {
    Qn;
} else {
    Q0;
}
```

While the **if-else** form is usually slower in terms of execution speed on a real computer, this doesn't matter for verification purposes, which are separate from execution issues.

Similarly, a loop statement of the form

```
for (P; Q; R) {
  S;
}
```

can be re-expressed as

```
P;
while (Q) {
  S;
  R;
}
```

and so on.

It is then possible to derive a Hoare rule for each kind of statement not previously discussed, by applying the classical rules to the corresponding re-expressed code fragment. However, we do not present these derived rules here.

Although procedures cannot be re-expressed in the above way if they are (directly or mutually) recursive, it is still possible to derive Hoare rules for them. This requires the finding of appropriate "procedure invariants" similar to loop invariants. Non-recursive procedures can, of course, just be inlined to make the classical Hoare rules applicable.

Note that **goto** cannot be rewritten in the above way; in fact, programs containing **goto** statements cannot be verified with the Hoare Calculus. See [19] for a similar calculus that can deal with arbitrary flowcharts, and hence arbitrary jumps. In fact, Hoare's work was based on that calculus. Later calculi inspired from Hoare's work have been designed to re-integrate support for arbitrary jumps. However, in this tutorial, we will not discuss example programs containing a **goto**.

Part II.

Nonmutating and simple search algorithms

4. Non-mutating algorithms

In this chapter, we consider *non-mutating* algorithms of the C++ Standard Library [20, §28.5]. These algorithms neither change their arguments nor any objects outside their scope. This requirement can be formally expressed with the following *assigns clause*:

```
assigns \nothing;
```

Each algorithm in this chapter therefore uses this assigns clause in its specification.

The specifications of these algorithms are not very complex. Nevertheless, we have tried to arrange them so that the earlier examples are simpler than the later ones. Each algorithm works on one-dimensional arrays.

- `find` in §4.1 provides *sequential* or *linear search* and returns the smallest index at which a given value occurs in a given range. In §4.2, a user-defined ACSL predicate is introduced in order to simplify the reuse of various specification elements. We refer to the simplified version as `find2`. We provide in §4.3 a third specification of `find` (called `find3`) that relies on a user-defined ACSL function that expresses the ideas of linear search on the logic level.
- `find_if_not` in §4.4 is a small variation of `find` that searches the first occurrence where a given value does *not* occur.
- `find_first_of` in §4.5 provides similar to `find` a *sequential search*. However, unlike `find` it does not search for a particular value, but for an arbitrary member of a set.
- `adjacent_find` in §4.6 can be used to find equal neighbors in an array.
- `equal` and `mismatch` in §4.7 are useful for comparing two ranges element-by-element and identifying where they differ.
- `search` and `search_n` in §4.8 and §4.9 find a subsequence that is identical to a given sequence when compared element-by-element and returns the position of the first occurrence.
- `count` in §4.11 returns the number of occurrences of a given value in a range. Here we will explicitly define a logic function for elements counting and show that the implementation comply with it.
- `count2` in §4.12 contains different specification for the `count` function. In this case an inductive predicate defined for elements counting. The section allows one to compare different approaches of writing specifications and demonstrates the ACSL inductive predicates.

4.1. The `find` algorithm

The `find` algorithm in the C++ Standard Library [20, §28.5.5] implements *sequential search* for general sequences. We have modified the generic implementation, which relies heavily on C++ templates, to that of a range of type `value_type`. The signature now reads:

```
size_type find(const value_type* a, size_type n, value_type v);
```

The function `find` returns the least *valid* index `i` of `a` where the condition `a[i] == v` holds. If no such index exists then `find` returns the length `n` of the array.

As an example, we consider in Figure 4.1 an array. The arrows indicate which indices will be returned by `find` for a given value. Note that the index 9 points *one past end* of the array. Values that are not contained in the array are colored in gray.

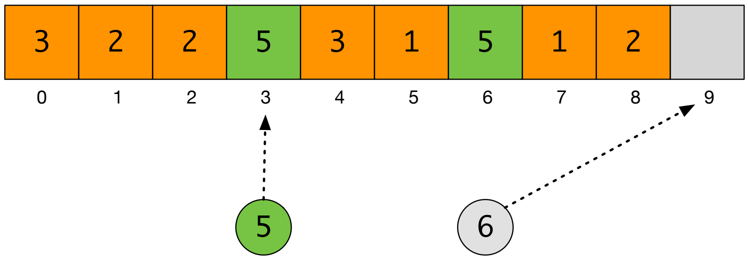

Figure 4.1.: Some simple examples for `find`

4.1.1. Formal specification of `find`

The following listing shows our first attempt specify `find` [4.2].

```
/*@
  requires    \valid_read(a + (0..n-1));
  assigns     \nothing;
  ensures     0 <= \result <= n;

  behavior some:
    assumes   \exists integer i; 0 <= i < n && a[i] == v;
    assigns   \nothing;
    ensures   0 <= \result < n;
    ensures   a[\result] == v;
    ensures   \forall integer i; 0 <= i < \result ==> a[i] != v;

  behavior none:
    assumes   \forall integer i; 0 <= i < n ==> a[i] != v;
    assigns   \nothing;
    ensures   \result == n;

  complete behaviors;
  disjoint behaviors;
*/
size_type
find(const value_type* a, size_type n, value_type v);
```

Listing 4.2: Formal specification of `find`

The requires-clause indicates that n is non-negative and that the pointer a points to *n* contiguously allocated objects of type value_type (see §2.3). The assigns-clause indicates that find (as a non-mutating algorithm), does not modify any memory location outside its scope (see Page 31).

Generally, we only know that find returns a non-negative index that is less or equal the length of the array. However, once we assume more specific situations, we can also make more precise statements about the returned valued. This is the reason why we have subdivided the specification of find into two behaviors (named some and none).

- The behavior some applies if the sought-after value is contained in the array. We express this condition by using the assumes-clause. The next line expresses that if the assumptions of the behavior are satisfied then find will return a valid index. The algorithm also ensures that the returned (valid) index i, a[i] == v holds. Therefore we define this property in the second postcondition of behavior some. Finally, it is important to express that find returns the smallest index i for which a[i] == v holds (see last postcondition of behavior some).
- The behavior none covers the case that the sought-after value is *not* contained in the array (see assumes-clause of behavior none in in the contract offind [4.2]. In this case, find must return the length n of the range a.

Note that the formula in the assumes-clause of the behavior some is the negation of the assumes-clause of the behavior none. Therefore, we can express that these two behaviors are *complete* and *disjoint*.

4.1.2. Implementation of find

The noteworthy elements of our implementation of find [4.3] are the *loop annotations*. The first loop *invariant* is needed to prove that accesses to a only occur with valid indices. The second loop *invariant* is needed for the proof of the postconditions of the behavior some in the contract of find [4.2]. It expresses that for each iteration the sought-after value is not yet found up to that iteration step. Finally, the loop *variant* n-i is needed to generate correct verification conditions for the termination of the loop.

```
size_type
find(const value_type* a, size_type n, value_type v)
{
  /*@
    loop invariant 0 <= i <= n;
    loop invariant \forall integer k; 0 <= k < i ==> a[k] != v;
    loop assigns i;
    loop variant n-i;
   */
  for (size_type i = 0u; i < n; i++) {
    if (a[i] == v) {
      return i;
    }
  }

  return n;
}
```

Listing 4.3: Implementation of find

4.2. The `find2` algorithm—reuse of specification elements

In this section we specify `find` in a slightly different way. Our approach is motivated by a considerable number of closely related ACSL formulas in the contract `find` [4.2] and the implementation `find` [4.3].

```
\exists integer i; 0 <= i < n         &&   a[i] == v;

\forall integer i; 0 <= i < \result  ==>  a[i] != v;

\forall integer i; 0 <= i < n         ==>  a[i] != v;

\forall integer k; 0 <= k < i         ==>  a[k] != v;
```

Note that the first formula is the negation of the third one.

4.2.1. The predicates `SomeEqual` and `NoneEqual`

In order to be more explicit about the commonalities of these formulas we define a predicate, called `SomeEqual` [4.4], which describes the situation that there is a valid index `i` where `a[i]` equals `v`.

```
/*@
  axiomatic SomeNone
  {
    predicate
    SomeEqual{A}(value_type* a, integer m, integer n, value_type v) =
      \exists integer i; m <= i < n && a[i] == v;

    predicate
    SomeEqual{A}(value_type* a, integer n, value_type v) =
      SomeEqual(a, 0, n, v);

    predicate
    NoneEqual(value_type* a, integer m, integer n, value_type v) =
      \forall integer i; m <= i < n  ==>  a[i] != v;

    predicate
    NoneEqual(value_type* a, integer n, value_type v) =
      NoneEqual(a, 0, n, v);

    lemma NotSomeEqual_NoneEqual:
      \forall value_type *a, v, integer m, n;
        !SomeEqual(a, m, n, v)  ==>  NoneEqual(a, m, n, v);

    lemma NoneEqual_NotSomeEqual:
      \forall value_type *a, v, integer m, n;
       NoneEqual(a, m, n, v)   ==>  !SomeEqual(a, m, n, v);
  }
*/
```

Listing 4.4: The logic definition(s) `SomeNone`

We first remark that the `SomeEqual`, its negation `NoneEqual` and the lemmas `NotSomeEqual_NoneEqual` and `NoneEqual_NotSomeEqual` are encapsulated in the *axiomatic block* `SomeNone` [4.4]. This is a *feeble* attempt to establish some modularization for the various predicates, logic functions and lemmas. We say *feeble* because axiomatic blocks are, in contrast to ACSL `modules`, *not* name spaces. ACSL modules, however, are not yet implemented by Frama-C.

We also remark that both predicates come in overloaded versions. The first of theses versions is a definition for array sections while the second definition is for the case of complete arrays.

Note that we have provided a label, viz. `A`, to the predicate `SomeEqual`. Its purposes to express that the evaluation of the predicate depends on a memory state, viz. the contents of `a[0..n-1]`. In general, we have to write

```
\exists integer i; 0 <= i < n && \at(a[i],A) == v;
```

in order to express that we refer to the value `a[i]` in the program state `A`. However, ACSL allows to abbreviate `\at(a[i],A)` by `a[i]` if, as in `SomeEqual` or `NoneEqual`, the label `A` is the only available label. In particular, we have omitted the label in the overloaded versions for complete arrays.

4.2.2. Formal specification of `find2`

With the predicates `SomeEqual` [4.4] and `NoneEqual` [4.4] we are able to encapsulate all uses of the universal and existential quantifiers in both the specification and implementation of `find2`.

As a result, the revised contract `find2` [4.5] is more concise than that of `find` [4.2]. In particular, it can be seen immediately that the conditions in the assumes clauses of the two behaviors `some` and `none` are mutually exclusive since one is the literal negation of the other. Moreover, the requirement that `find` returns the smallest index can also be expressed using the `NoneEqual` [4.4] predicate, as depicted with the last postcondition of behavior `some`.

```
/*@
  requires valid:    \valid_read(a + (0..n-1));
  assigns            \nothing;
  ensures result:    0 <= \result <= n;

  behavior some:
    assumes          SomeEqual(a, n, v);
    assigns          \nothing;
    ensures   bound:  0 <= \result < n;
    ensures   result: a[\result] == v;
    ensures   first:  NoneEqual(a, \result, v);

  behavior none:
    assumes          NoneEqual(a, n, v);
    assigns          \nothing;
    ensures   result: \result == n;

  complete behaviors;
  disjoint behaviors;
*/
size_type
find2(const value_type* a, size_type n, value_type v);
```

Listing 4.5: Formal specification of `find2`

We also enriched the specification of `find` by user-defined names (sometimes called *labels*, too, the distinction to program state identifiers being obvious) to refer to the `requires` and `ensures` clauses. We highly recommend this practice in particular for more complex annotations. For example, Frama-C can be instructed to verify only clauses with a given name.

4.2.3. Implementation of `find2`

The predicate `NoneEqual` is also used in the loop annotation inside the implementation of `find2` [4.6]. Note that, as in the case of the specification, we use labels to name individual annotations.

```
size_type
find2(const value_type* a, size_type n, value_type v)
{
  /*@
    loop invariant bound:     0 <= i <= n;
    loop invariant not_found: NoneEqual(a, i, v);
    loop assigns i;
    loop variant n-i;
   */
  for (size_type i = 0u; i < n; i++) {
    if (a[i] == v) {
      return i;
    }
  }

  return n;
}
```

Listing 4.6: Implementation of `find2`

4.3. The `find3` algorithm—using a logic function

In this section we specify linear search yet another way. This requires more preparing work but results in a more concise function contract.

4.3.1. The logic function `Find`

We start with a *recursive* definition of the ACSL function `Find`. Due to the considerable number of associated lemmas of the function `Find` we split its definition into several listings. Note that `Find` comes as two *overloaded* functions. While the first version is defined for *array sections* the latter is intend for *complete arrays*.

The listings start with lemmas which express elementary properties directly related to an incremental increase of the array `a[0..n-1]`. The latter lemmas are somewhat more higher-level and will be useful for the verification of `find3`. It will be there that we also reuse the predicates `SomeEqual` [4.4]and `NoneEqual` [4.4]. At the end of this section we will also discuss in what sense the contracts of `find2` and `find3` are equivalent.

```
/*@
  axiomatic Find
  {
    logic integer
    Find(value_type* a, integer m, integer n, value_type v) =
      (n <= m) ?
       0 : ((0 <= Find(a, m, n-1, v) < n-m-1) ?
         Find(a, m, n-1, v) : ((a[n-1] == v) ? n-m-1 : n-m));

    logic integer
    Find(value_type* a, integer n, value_type v) = Find(a, 0, n, v);

    lemma Find_Empty:
      \forall value_type *a, v, integer m, n;
        n <= m  ==>  Find(a, m, n, v) == 0;

    lemma Find_Hit:
      \forall value_type *a, v, integer m, n;
        m <= n                      ==>
        Find(a, m, n, v) < n-m  ==>
        Find(a, m, n+1, v) == Find(a, m, n, v);

    lemma Find_MissHit:
      \forall value_type *a, v, integer m, n;
        m <= n                         ==>
        a[n] == v                      ==>
        Find(a, m, n, v)   == n-m  ==>
        Find(a, m, n+1, v) == n-m;

    lemma Find_MissMiss:
      \forall value_type *a, v, integer m, n;
        m <= n                         ==>
        a[n] != v                      ==>
        Find(a, m, n, v)   == n-m  ==>
        Find(a, m, n+1, v) == (n+1)-m;

    lemma Find_Lower:
      \forall value_type *a, v, integer m, n;
        0 <= Find(a, m, n, v);

    lemma Find_Upper:
      \forall value_type *a, v, integer m, n;
        m <= n  ==>  Find(a, m, n, v) <= n-m;

    lemma Find_WeaklyIncreasing:
      \forall value_type *a, v, integer m, n;
        m <= n  ==>  Find(a, m, n, v) <= Find(a, m, n+1, v);

    lemma Find_Increasing:
      \forall value_type *a, v, integer k, m, n;
        m <= k <= n  ==>
        Find(a, m, k, v) <= Find(a, m, n, v);

    lemma Find_Extend:
      \forall value_type *a, v, integer k, m, n;
        m <= k < n                   ==>
        a[k] == v                    ==>
        Find(a, m, k, v) == k-m    ==>
        Find(a, m, n, v) == k-m;
```

Listing 4.7: The logic function `Find` (1)

```
    lemma Find_Limit:
      \forall value_type *a, v, integer k, m, n;
        m <= k < n  ==>
        a[k] == v   ==>
        Find(a, m, n, v) <= k-m;

    lemma Find_NoneEqual:
      \forall value_type *a, v, integer m, n;
        m <= n                 ==>
        NoneEqual(a, m, n, v)  ==>
        Find(a, m, n, v) == n-m;

    lemma Find_SomeEqual:
      \forall value_type *a, v, integer k, m, n;
        m <= k < n             ==>
        a[k] == v              ==>
        NoneEqual(a, m, k, v)  ==>
        Find(a, m, n, v) == k-m;

    lemma Find_ResultNoneEqual:
      \forall value_type *a, v, integer m, n;
        m <= n  ==>  NoneEqual(a, m, m + Find(a, m, n, v), v);

    lemma Find_ResultEqual:
      \forall value_type *a, v, integer m, n;
        0 <= Find(a, m, n, v) < n-m  ==>
        a[m + Find(a, m, n, v)] == v;
  }
*/
```

Listing 4.8: The logic function `Find` (2)

4.3.2. Formal specification of `find3`

Using the logic function `Find` we can now give a third specification of linear search. The contract of `find3` [4.9] is considerably shorter than that of `find2` [4.5]. Of course, we had to put much more effort into the definition of the ACSL function `Find` [4.7].

```
/*@
  requires valid:    \valid_read(a + (0..n-1));
  assigns            \nothing;
  ensures result:    0 <= \result <= n;
  ensures result:    \result == Find(a, n, v);
*/
size_type
find3(const value_type* a, size_type n, value_type v);
```

Listing 4.9: Formal specification of `find3`

4.3.3. Implementation of find3

The following listing shows the implementation of `find3` [4.10]. In order to achieve a complete verification we had to add the assertion `found`.

```
size_type
find3(const value_type* a, size_type n, value_type v)
{
  /*@
    loop invariant bound:      0 <= i <= n;
    loop invariant not_found: Find(a, i, v) == i;
    loop assigns i;
    loop variant n-i;
   */
  for (size_type i = 0u; i < n; i++) {
    if (a[i] == v) {
      //@ assert found: Find(a, n, v) == i;
      return i;
    }
  }

  return n;
}
```

Listing 4.10: Implementation of `find3`

A question that remains is in what sense the contract of `find2` [4.5] is equivalent to the one of `find3` [4.9]. We will answer this question in the following section.

4.3.4. The equivalence of find2 and find3

We consider the contracts of `find2` [4.5] and `find3` [4.9] as *equivalent* if each one is sufficient to verify the other. To this end we introduce yet another two examples `find4` and `find5`.

The implementation of `find4` [4.11] consists just of a call to `find3`.

```
size_type
find4(const value_type* a, size_type n, value_type v)
{
  return find3(a, n, v);
}
```

Listing 4.11: Implementation of `find4`

The contract of `find4` [4.12], however, is the same as the one of `find2` [4.5].

```
/*@
  requires valid:    \valid_read(a + (0..n-1));
  assigns            \nothing;
  ensures result:    0 <= \result <= n;

  behavior some:
    assumes          SomeEqual(a, n, v);
    assigns          \nothing;
    ensures  bound:  0 <= \result < n;
    ensures  result: a[\result] == v;
    ensures  first:  NoneEqual(a, \result, v);

  behavior none:
    assumes          NoneEqual(a, n, v);
    assigns          \nothing;
    ensures  result: \result == n;

  complete behaviors;
  disjoint behaviors;
*/
size_type
find4(const value_type* a, size_type n, value_type v);
```

Listing 4.12: Formal specification of `find4`

Analogously, the implementation of `find5` [4.13] is simply a call to `find2`.

```
size_type
find5(const value_type* a, size_type n, value_type v)
{
  return find2(a, n, v);
}
```

Listing 4.13: Implementation of `find5`

On the other hand, the contract of `find5` [4.14] is the same as the one of `find3` [4.9]. The verification of the functions `find4` and `find5` (cf. Table A.2) then shows the equivalence of the respective contracts of `find2` [4.5] and `find3` [4.9].

```
/*@
  requires valid:    \valid_read(a + (0..n-1));
  assigns            \nothing;
  ensures result:    0 <= \result <= n;
  ensures result:    \result == Find(a, n, v);
*/
size_type
find5(const value_type* a, size_type n, value_type v);
```

Listing 4.14: Formal specification of `find5`

4.4. The `find_if_not` algorithm

Many algorithms in the C++ standard library can be parameterized not only by the type of sequence but also using so-called *function objects*. One example is the `find_if_not` algorithm that accepts a *predicate function object P*. The algorithm then returns the first position *i* in the input sequence where *P*(*i*) does *not* hold.

While function objects could be emulated in C with *pointers to functions*, we will not follow this road here. The main reason is that function pointers are, so far, only supported momentarily by Frama-C. Moreover, there is as of now no support for parameterized ACSL predicates. For these reasons our implementation of `find_if_not` only returns the first position in an array where a given value does *not* occur. The signature, thus, reads

```
size_type find_if_not(const value_type* a, size_type n, value_type v);
```

On the one hand, this is not a very exciting addition to our collections of verified algorithms. It gives us, however, an opportunity to introduce the predicates `AllEqual` [4.15] and `SomeNotEqual` [4.15] and more importantly the logic function `FindNotEqual` [4.16] that will later play an essential role in the specification of the algorithm `remove_copy`, or more precisely, its variant `remove_copy3` [7.48].

```
/*@
  axiomatic AllSomeNot
  {
    predicate
    AllEqual(value_type* a, integer m, integer n, value_type v) =
      \forall integer i; m <= i < n  ==>  a[i] == v;

    predicate
    AllEqual(value_type* a, integer m, integer n) =
      AllEqual(a, m, n, a[m]);

    predicate
    AllEqual(value_type* a, integer n, value_type v) =
      AllEqual(a, 0, n, v);

    predicate
    SomeNotEqual{A}(value_type* a, integer m, integer n, value_type v) =
      \exists integer i; m <= i < n && a[i] != v;

    predicate
    SomeNotEqual{A}(value_type* a, integer n, value_type v) =
      SomeNotEqual(a, 0, n, v);

    lemma NotAllEqual_SomeNotEqual:
      \forall value_type *a, v, integer m, n;
        !AllEqual(a, m, n, v)   ==>  SomeNotEqual(a, m, n, v);

    lemma SomeNotEqual_NotAllEqual:
      \forall value_type *a, v, integer m, n;
       SomeNotEqual(a, m, n, v)    ==>  !AllEqual(a, m, n, v);
  }
*/
```

Listing 4.15: The logic definition(s) `AllSomeNot`

The predicate `AllEqual` expresses that each member of the array section

`a[m..n-1]` equals `v`. We also introduce the predicate `SomeNotEqual` which is the negation of `AllEqual` . Both predicates complement the predicates `SomeEqual` [4.4] and `NoneEqual` [4.4].

There are two additional overloaded versions of `AllEqual`. The first version uses the value `a[m]` as `v`. The second version is just a shortcut when the first index `m` equals 0.

4.4.1. The logic function `FindNotEqual`

The definition of the overloaded logic function `FindNotEqual` is shown in Listings 4.16 and 4.17. This function is very similar to `Find` [4.7] except that it finds the first element in a sequence that *differs* from a given value. Note that in lemma `FindNotEqual_Unchanged` we are using the predicate `Unchanged` [7.1] that will be defined in a later chapter.

```
/*@
  axiomatic FindNotEqual
  {
    logic integer
    FindNotEqual(value_type* a, integer m, integer n, value_type v) =
       (n <= m) ?
        0 : ((0 <= FindNotEqual(a, m, n-1, v) < n-m-1) ?
          FindNotEqual(a, m, n-1, v) : ((a[n-1] != v) ? n-m-1 : n-m));

    logic integer
    FindNotEqual(value_type* a, integer n, value_type v) =
      FindNotEqual(a, 0, n, v);

    lemma FindNotEqual_Empty:
      \forall value_type *a, v, integer m, n;
        n <= m  ==>  FindNotEqual(a, m, n, v) == 0;

    lemma FindNotEqual_Hit:
      \forall value_type *a, v, integer m, n;
        m <= n                          ==>
        FindNotEqual(a, m, n, v) < n-m  ==>
        FindNotEqual(a, m, n+1, v) == FindNotEqual(a, m, n, v);

    lemma FindNotEqual_MissHit:
      \forall value_type *a, v, integer m, n;
        m <= n                               ==>
        a[n] != v                            ==>
        FindNotEqual(a, m, n, v)    == n-m   ==>
        FindNotEqual(a, m, n+1, v) == n-m;

    lemma FindNotEqual_MissMiss:
      \forall value_type *a, v, integer m, n;
        m <= n                               ==>
        a[n] == v                            ==>
        FindNotEqual(a, m, n, v)    == n-m   ==>
        FindNotEqual(a, m, n+1, v) == (n+1)-m;

    lemma FindNotEqual_Lower:
      \forall value_type *a, v, integer m, n;
        0 <= FindNotEqual(a, m, n, v);
```

Listing 4.16: The logic function `FindNotEqual` (1)

```
    lemma FindNotEqual_Upper:
      \forall value_type *a, v, integer m, n;
        m <= n  ==>  FindNotEqual(a, m, n, v) <= n-m;

    lemma FindNotEqual_Unchanged{K,L}:
      \forall value_type *a, v, integer m, n;
        Unchanged{K,L}(a, m, n)  ==>
        FindNotEqual{K}(a, m, n, v) == FindNotEqual{L}(a, m, n, v);

    lemma FindNotEqual_WeaklyIncreasing:
      \forall value_type *a, v, integer m, n;
        m <= n  ==>  FindNotEqual(a, m, n, v) <= FindNotEqual(a, m, n+1, v);

    lemma FindNotEqual_Extend:
      \forall value_type *a, v, integer k, m, n;
        m <= k < n                          ==>
        a[k] != v                           ==>
        FindNotEqual(a, m, k, v) == k-m    ==>
        FindNotEqual(a, m, n, v) == k-m;

    lemma FindNotEqual_Increasing:
      \forall value_type *a, v, integer k, m, n;
        m <= k <= n  ==>  FindNotEqual(a, m, k, v) <= FindNotEqual(a, m, n, v);

    lemma FindNotEqual_Limit:
      \forall value_type *a, v, integer k, m, n;
        m <= k < n  ==>
        a[k] != v   ==>
        FindNotEqual(a, m, n, v) <= k-m;

    lemma FindNotEqual_AllEqual:
      \forall value_type *a, v, integer m, n;
        m <= n                ==>
        AllEqual(a, m, n, v)  ==>
        FindNotEqual(a, m, n, v) == n-m;

    lemma FindNotEqual_SomeNotEqual:
      \forall value_type *a, v, integer k, m, n;
        m <= k < n            ==>
        a[k] != v             ==>
        AllEqual(a, m, k, v)  ==>
        FindNotEqual(a, m, n, v) == k-m;

    lemma FindNotEqual_ResultAllEqual:
      \forall value_type *a, v, integer m, n;
        m <= n  ==>  AllEqual(a, m, m + FindNotEqual(a, m, n, v), v);

    lemma FindNotEqual_ResultNotEqual:
      \forall value_type *a, v, integer m, n;
        0 <= FindNotEqual(a, m, n, v) < n-m  ==>
        a[m + FindNotEqual(a, m, n, v)] != v;
  }
*/
```

Listing 4.17: The logic function `FindNotEqual` (2)

4.4.2. Formal specification of `find_if_not`

The contract of `find_if_not` [4.18] is, unsurprisingly, very similar to that of `find3` [4.9]. The only difference is that we replaced `Find` [4.7] by `FindNotEqual` [4.16].

```
/*@
  requires valid:     \valid_read(a + (0..n-1));
  assigns             \nothing;
  ensures result:     0 <= \result <= n;
  ensures result:     \result == FindNotEqual(a, n, v);
*/
size_type
find_if_not(const value_type* a, size_type n, value_type v);
```

Listing 4.18: Formal specification of `find_if_not`

4.4.3. Implementation of `find_if_not`

The implementation of `find_if_not` [4.19] also has a lot of similarities with of `find3` [4.10]. Here again we have replaced `Find` by `FindNotEqual` and, of course, we check in the loop body that the value `a[i]` *differs* from the given value `v`.

```
size_type
find_if_not(const value_type* a, size_type n, value_type v)
{
  /*@
    loop invariant bound:     0 <= i <= n;
    loop invariant not_found: FindNotEqual(a, i, v) == i;
    loop assigns i;
    loop variant n-i;
   */
  for (size_type i = 0u; i < n; i++) {
    if (a[i] != v) {
      //@ assert found: FindNotEqual(a, n, v) == i;
      return i;
    }
  }

  return n;
}
```

Listing 4.19: Implementation of `find_if_not`

4.5. The `find_first_of` algorithm

The `find_first_of` algorithm [20, §28.5.7] is closely related to `find` (see §4.1 and §4.2).

```
size_type
find_first_of(const value_type* a, size_type m,
              const value_type* b, size_type n);
```

Like `find`, it performs a sequential search. However, while `find` searches for a particular value, the function `find_first_of` returns the least index `i` such that `a[i]` is equal to one of the values `b[0..n-1]`.

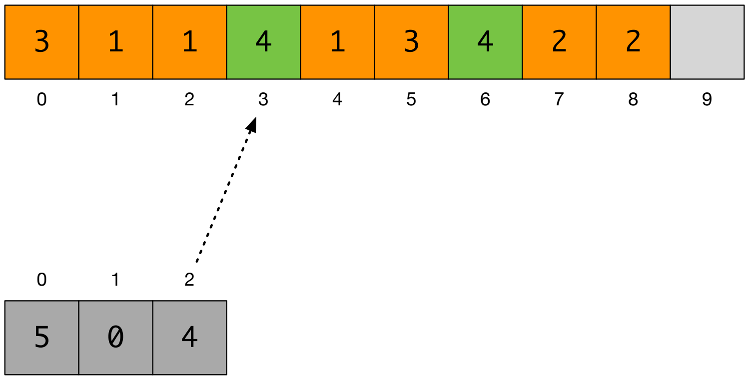

Figure 4.20.: A simple example for `find_first_of`

As an example, we consider in Figure 4.20 two arrays. The arrow indicates the smallest index where one of the elements of the three-element array occurs.

4.5.1. The predicate `HasValueOf`

Similar to our approach in §4.2, we define a predicate `HasValueOf` [4.21] that formalizes the fact that there are valid indices `i` and `j` of the respective arrays `a` and `b` such that `a[i] == b[j]` holds. We have chosen to reuse the predicate `SomeEqual` [4.4] to define `HasValueOf`.

```
/*@
  axiomatic HasValueOf
  {
    predicate
    HasValueOf{A}(value_type* a, integer m, value_type* b, integer n) =
      \exists integer i; 0 <= i < m && SomeEqual{A}(b, n, a[i]);
  }
*/
```

Listing 4.21: The logic definition(s) `HasValueOf`

4.5.2. Formal specification of `find_first_of`

The following listing shows the formal specification of `find_first_of`. The function contract uses the predicates `HasValueOf` [4.21] and `SomeEqual` [4.4] thereby making it very similar the specification of `find2` [4.5].

```
/*@
  requires valid:   \valid_read(a + (0..m-1));
  requires valid:   \valid_read(b + (0..n-1));
  assigns           \nothing;
  ensures result:   0 <= \result <= m;

  behavior found:
    assumes         HasValueOf(a, m, b, n);
    assigns         \nothing;
    ensures bound:  0 <= \result < m;
    ensures result: SomeEqual(b, n, a[\result]);
    ensures first:  !HasValueOf(a, \result, b, n);

  behavior not_found:
    assumes         !HasValueOf(a, m, b, n);
    assigns         \nothing;
    ensures result: \result == m;

  complete behaviors;
  disjoint behaviors;
*/
size_type
find_first_of(const value_type* a, size_type m,
              const value_type* b, size_type n);
```

Listing 4.22: Formal specification of `find_first_of`

4.5.3. Implementation of `find_first_of`

Our implementation of `find_first_of` [4.23] calls `find2` [4.5], thereby emphasizing reuse. Besides, leading to a more concise implementation, we also have to write fewer loop annotations.

```
size_type
find_first_of (const value_type* a, size_type m,
               const value_type* b, size_type n)
{
  /*@
    loop invariant bound:      0 <= i <= m;
    loop invariant not_found: !HasValueOf(a, i, b, n);
    loop assigns i;
    loop variant m-i;
  */
  for (size_type i = 0u; i < m; i++) {
    if (find2(b, n, a[i]) < n) {
      return i;
    }
  }

  return m;
}
```

Listing 4.23: Implementation of `find_first_of`

4.6. The `adjacent_find` algorithm

The `adjacent_find` algorithm of the C++ Standard Library [20, §28.5.8]

```
size_type adjacent_find(const value_type* a, size_type n);
```

returns the smallest valid index i, such that i+1 is also a valid index and such that

```
a[i] == a[i+1]
```

holds. The `adjacent_find` algorithm returns `n` if no such index exists.

The arrow in Figure 4.24 indicates the smallest index where two adjacent elements are equal.

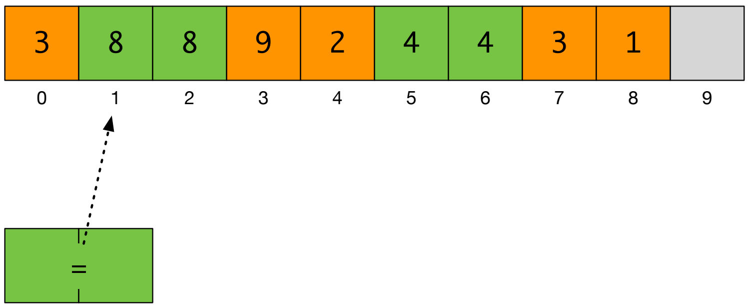

Figure 4.24.: A simple example for `adjacent_find`

4.6.1. The predicate `HasEqualNeighbors`

As in the case of other search algorithms, we first define a predicate `HasEqualNeighbors` [4.25] that captures the essence of finding two adjacent indices at which the array holds equal values.

```
/*@
  axiomatic HasEqualNeighbors
  {
    predicate
    HasEqualNeighbors{L}(value_type* a, integer n) =
      \exists integer i; 0 <= i < n-1 && a[i] == a[i+1];
  }
*/
```

Listing 4.25: The predicate `HasEqualNeighbors`

4.6.2. Formal specification of `adjacent_find`

We use the predicate `HasEqualNeighbors` [4.25] to define the formal specification of `adjacent_find` [4.26].

```
/*@
  requires valid:         \valid_read(a + (0..n-1));
  assigns                 \nothing;
  ensures result:         0 <= \result <= n;

  behavior some:
    assumes               HasEqualNeighbors(a, n);
    assigns               \nothing;
    ensures   result:     0 <= \result < n-1;
    ensures   adjacent:   a[\result] == a[\result+1];
    ensures   first:      !HasEqualNeighbors(a, \result);

  behavior none:
    assumes               !HasEqualNeighbors(a, n);
    assigns               \nothing;
    ensures   result:     \result == n;

  complete behaviors;
  disjoint behaviors;
*/
size_type
adjacent_find(const value_type* a, size_type n);
```

Listing 4.26: Formal specification of `adjacent_find`

4.6.3. Implementation of `adjacent_find`

In the implementation of `adjacent_find` [4.27] we check whether the array contains at least two elements. Otherwise, there is no point in looking for adjacent neighbors. Note the use of the predicate `HasEqualNeighbors` [4.25] in the loop invariant to match the similar postcondition of behavior `some`.

```
size_type
adjacent_find(const value_type* a, size_type n)
{
  if (1u < n) {
    /*@
      loop invariant bound:  0 <= i < n;
      loop invariant none:   !HasEqualNeighbors(a, i+1);
      loop assigns i;
      loop variant n-i;
    */
    for (size_type i = 0u; i + 1u < n; ++i) {
      if (a[i] == a[i + 1u]) {
        return  i;
      }
    }
  }

  return n;
}
```

Listing 4.27: Implementation of `adjacent_find`

4.7. The `equal` and `mismatch` algorithms

The algorithms `equal` [20, §28.5.11] and `mismatch` [20, §28.5.10] of the C++ Standard Library compare two generic sequences. For our purposes we have modified the generic implementation to that of an array of type `value_type`. The signatures read

```
bool        equal(const value_type* a, size_type n, const value_type* b);

size_type   mismatch(const value_type* a, size_type n, const value_type* b);
```

The function `equal` returns **`true`** if and only if `a[i] == b[i]` holds for each `0 <= i < n`. Otherwise, `equal` returns **`false`**.

The `mismatch` algorithm is slightly more general than the negation of `equal`: it returns the smallest index where the two ranges `a` and `b` differ. If no such index exists, that is, if both ranges are equal, then `mismatch` returns the (common) length `n` of the two ranges.

4.7.1. The `Equal` predicate

The fact that two arrays `a[0]..a[n-1]` and `b[0]..b[n-1]` are equal when compared element by element, is a property we might need again in other specifications, as it describes a very basic property.

The motto *don't repeat yourself* is not just good programming practice.[12] It is also true for concise and easy to understand specifications. We will therefore introduce specification elements that we can apply to the `equal` algorithm as well as to other specifications and implementations with the described property.

We start with introducing several *overloaded* versions of the predicate `Equal` [4.28].

```
/*@
  axiomatic Equal
  {
    predicate
    Equal{K,L}(value_type* a, integer m, integer n, value_type* b) =
      \forall integer i; m <= i < n  ==>  \at(a[i],K) == \at(b[i],L);

    predicate
    Equal{K,L}(value_type* a, integer n, value_type* b) =
      Equal{K,L}(a, 0, n, b);

    predicate
    Equal{K,L}(value_type* a, integer m, integer n,
               value_type* b, integer p) = Equal{K,L}(a+m, n-m, b+p);

    predicate
    Equal{K,L}(value_type* a, integer m, integer n, integer p) =
      Equal{K,L}(a, m, n, a, p);
  }
*/
```

Listing 4.28: The logic definition(s) `Equal`

The letters `K` and `L` in the definition of `Equal` are so-called *labels*[13] that refer to program states in which the ranges `a[..]` and `b[..]` are evaluated. Frama-C defines several standard labels, e.g. `Old` and `Post`,

[12] Compare `http://en.wikipedia.org/wiki/Don't_repeat_yourself`
[13] Labels are used in C to name the target of the *goto* jump statement.

a programmer can use to refer to the pre-state or post-state, respectively, of a function. For more details on labels we refer to the ACSL specification [15, §2.6.9].

4.7.2. Formal specification of `equal` and `mismatch`

Using predicate `Equal` [4.28] we can formulate the specification of `equal` [4.29] using the predefined label `Here`. When used in an `ensures` clause, the label `Here` refers to the post-state of a function. Note that the equivalence is needed in the ensures clause. Putting an equality instead is not legal in ACSL, because `Equal` is a predicate, not a function.

```
/*@
  requires valid:   \valid_read(a + (0..n-1));
  requires valid:   \valid_read(b + (0..n-1));
  assigns           \nothing;
  ensures result:   \result <==> Equal{Here,Here}(a, n, b);
*/
bool
equal(const value_type* a, size_type n, const value_type* b);
```

Listing 4.29: Formal specification of `equal`

The formal specification of `mismatch` [4.30] is more complex than that of `equal` [4.29] because the return value of mismatch provides more information than just reporting whether the two arrays are equal.

```
/*@
  requires valid:    \valid_read(a + (0..n-1));
  requires valid:    \valid_read(b + (0..n-1));
  assigns            \nothing;
  ensures  result:   0 <= \result <= n;

  behavior all_equal:
    assumes          Equal{Here,Here}(a, n, b);
    assigns          \nothing;
    ensures result:  \result == n;

  behavior some_not_equal:
    assumes          !Equal{Here,Here}(a, n, b);
    assigns          \nothing;
    ensures bound:   0 <= \result < n;
    ensures result:  a[\result] != b[\result];
    ensures first:   Equal{Here,Here}(a, \result, b);

  complete behaviors;
  disjoint behaviors;
*/
size_type
mismatch(const value_type* a, size_type n, const value_type* b);
```

Listing 4.30: Formal specification of `mismatch`

On the other hand, the specification is conceptually quite similar to that of `find2` [4.5]. While `find2` returns the smallest index `i` where `a[i] == v` holds, `mismatch` finds the smallest index `a[i] != b[i]`. Note in particular the use of `Equal` in the specification of `mismatch`. As in the specification of `find2` the completeness and disjointness of `mismatch`'s behaviors is quite obvious, because the `assumes` clauses of `all_equal` and `some_not_equal` are negations of each other.

4.7.3. Implementation of equal and mismatch

The implementation of equal [4.31] consists of a simple call of mismatch.

```
bool
equal(const value_type* a, size_type n, const value_type* b)
{
  return mismatch(a, n, b) == n;
}
```

Listing 4.31: Implementation of equal

The implementation of mismatch [4.32] has been enriched with some loop annotations to support the deductive verification.

```
size_type
mismatch(const value_type* a, size_type n, const value_type* b)
{
  /*@
    loop invariant bound:  0 <= i <= n;
    loop invariant equal:  Equal{Here,Here}(a, i, b);
    loop assigns i;
    loop variant n-i;
  */
  for (size_type i = 0u; i < n; i++) {
    if (a[i] != b[i]) {
      return i;
    }
  }

  return n;
}
```

Listing 4.32: Implementation of mismatch

We use again the predicate Equal [4.28] in order to express that all indices k that are less than the current index i satisfy the condition a[k] == b[k]. This is necessary to prove that mismatch indeed returns the smallest index where the two ranges differ.

4.8. The `search` algorithm

The `search` algorithm in the C++ Standard Library [20, §28.5.13] finds a subsequence that is identical to a given sequence when compared element-by-element. For our purposes we have modified the generic implementation to that of an array of type `value_type`. The signature now reads:

```
size_type search(const value_type* a, size_type n,
                 const value_type* b, size_type p);
```

The function `search` returns the first index `s` of the array `a` where the condition `a[s+k] == b[k]` holds for each index `k` with `0 <= k < p` (see Figure 4.33). If no such index exists, then `search` returns the length `n` of the array `a`.

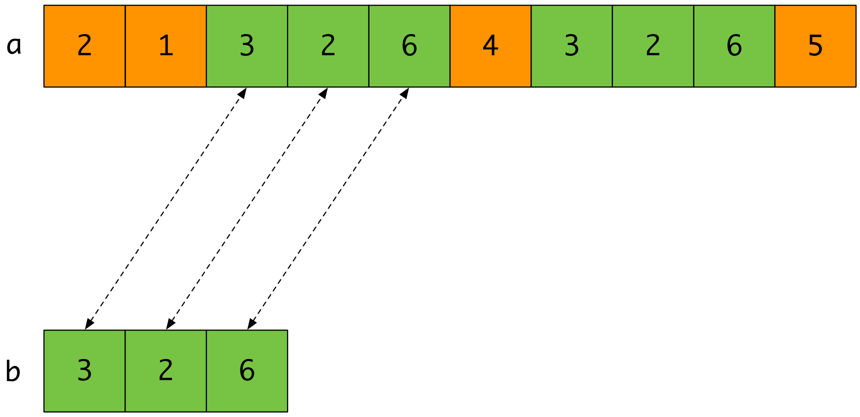

Figure 4.33.: Searching the first occurrence of `b[0..p-1]` in `a[0..n-1]`

4.8.1. The predicate `HasSubRange`

Our specification of `search` starts with introducing the predicate `HasSubRange` [4.34]. This predicate formalizes, using the predicate `Equal` [4.28], that the sequence `a` contains a subsequence which equal the sequence `b`. Of course, in order to contain a subsequence of length `p`, `a` must be at least that large; this is expressed by lemma `HasSubRange_Sizes`.

```
/*@
  axiomatic HasSubRange
  {
    predicate
    HasSubRange{L}(value_type* a, integer m, integer n, value_type* b, integer p) =
      \exists integer k; (m <= k <= n-p) && Equal{L,L}(a+k, p, b);

    predicate
    HasSubRange{L}(value_type* a, integer n, value_type* b, integer p) =
      HasSubRange{L}(a, 0, n, b, p);

    lemma HasSubRange_Sizes:
      \forall value_type *a, *b, integer m, n, p;
        HasSubRange(a, m, n, b, p)  ==>  p <= n-m;
  }
*/
```

Listing 4.34: The logic definition(s) `HasSubRange`

4.8.2. Formal specification of `search`

The following listing shows the specification of `search` [4.35].

```
/*@
  requires valid:   \valid_read(a + (0..n-1));
  requires valid:   \valid_read(b + (0..p-1));
  assigns           \nothing;
  ensures  result:  0 <= \result <= n;

  behavior has_match:
    assumes         HasSubRange(a, n, b, p);
    assigns         \nothing;
    ensures bound:  0 <= \result <= n-p;
    ensures result: Equal{Here,Here}(a+\result, p, b);
    ensures first:  !HasSubRange(a, \result+p-1, b, p);

  behavior no_match:
    assumes         !HasSubRange(a, n, b, p);
    assigns         \nothing;
    ensures result: \result == n;

  complete behaviors;
  disjoint behaviors;
*/
size_type
search(const value_type* a, size_type n,
       const value_type* b, size_type p);
```

Listing 4.35: Formal specification of `search`

Conceptually, the specification of `search` is very similar to that of `find` [4.2]. We therefore use again two behaviors to capture the essential aspects of `search`.

- The behavior `has_match` applies if the sequence `a` contains a subsequence identical to `b`. We express this condition with `assumes` using the predicate `HasSubRange` [4.34].

  The ensures clause `bound` of behavior `has_match` indicates that the returned index value must be in the range `[0..n-p]`. The clause `result` expresses that `search` returns an index where a copy of `b` can be found in `a`. Clause `first` indicates that the least index with that property is returned, i.e. that `b` can't be found in `a[0..\result+p-2]`.

- The behavior `no_match` covers the case that there is no subsequence `a` that equals `b`. In this case, `search` must return the length `n` of the range `a`. If the ranges `a` or `b` are empty then the return value will be `0`.

The formula in the assumes clause of the behavior `has_match` is the negation of the assumes clause of the behavior `no_match`. Therefore, we can express that these two behaviors are *complete* and *disjoint*.

4.8.3. Implementation of search

The implementation of `search` [4.36] is relatively easy to understand, but needs an order of magnitude of `n*p` operations. In contrast, the sophisticated algorithm from [21] needs only `n+p` operations.[14]

The loop invariant `not_found` is needed for the proof of the postconditions of the behavior `has_match` in the contract of `search` [4.35]. It expresses that the subsequence `b` has not been found up to the current iteration step. Neither `p == 0` nor `n == 0` need to be handled separately, not even for efficiency reasons: in the former case, `equal(a+i, p, b)` will succeed in the first iteration, while in the latter, `p > n` will apply.

```
size_type
search(const value_type* a, size_type n,
       const value_type* b, size_type p)
{
  if (p <= n) {
    /*@
      loop invariant bound:     i <= n-p+1;
      loop invariant not_found: !HasSubRange(a, p+i-1, b, p);
      loop assigns i;
      loop variant n-i;
    */
    for (size_type i = 0u; i <= n - p; ++i) {
      if (equal(a + i, p, b)) {
        //@ assert has_match: HasSubRange(a, n, b, p);
        return i;
      }
    }
  }

  //@ assert no_match: !HasSubRange(a, n, b, p);
  return n;
}
```

Listing 4.36: Implementation of `search`

[14] The efficiency question has been also discussed by the C++ standardization committee, see `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3905.html`

4.9. The `search_n` algorithm

The `search_n` algorithm in the C++ Standard Library [20, §28.5.13] finds the first place where a given value starts to occur a given number of times in a given sequence. For our purposes we have modified the generic implementation to that of an array of type `value_type`. The signature now reads:

```
size_type
search_n(const value_type* a, size_type n, size_type p, value_type v);
```

Note the similarity to the signature of `search` (§4.8). The only difference is that `v` now is a single value rather than an array.

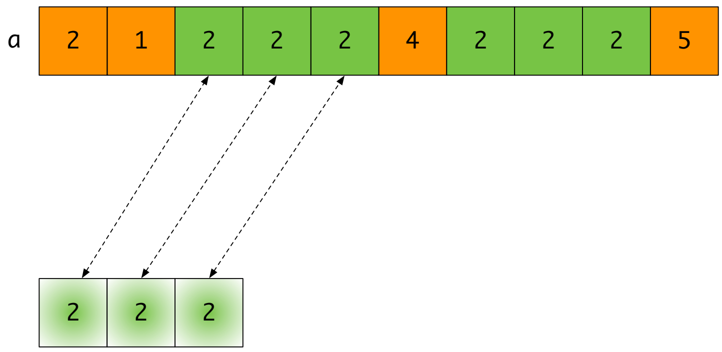

Figure 4.37.: Searching the first occurrence a given constant sequence in `a[0..n-1]`

The function `search_n` returns the first index `s` of the array `a` where the condition `a[s+k] == v` holds for each index `k` with `0 <= k < p` (see Figure 4.37). If no such index exists, then `search_n` returns the length `n` of the array `a`.

4.9.1. The predicate `HasConstantSubRange`

Our specification of `search_n` starts with introducing the predicate `HasConstantSubRange` [4.38].

```
/*@
  axiomatic HasConstantSubRange
  {
    predicate
    HasConstantSubRange{L}(value_type* a, integer m, integer n, value_type v,
        integer p) =
      \exists integer k; m <= k <= n-p && AllEqual(a, k, k+p, v);

    predicate
    HasConstantSubRange{L}(value_type* a, integer n, value_type v, integer p) =
      HasConstantSubRange(a, 0, n, v, p);

    lemma HasConstantSubRange_Sizes:
      \forall value_type *a, v, integer n, p;
        HasConstantSubRange(a, n, v, p)  ==>  p <= n;
  }
*/
```

Listing 4.38: The logic definition(s) `HasConstantSubRange`

This predicate formalizes that the sequence a of length n contains a subsequence of p times the value v. It thereby reuses the predicate AllEqual [4.15].

Similar to predicate HasSubRange [4.34], in order to contain p repetitions, the size of the array a[0..n-1] must be at least that large; this is what lemma HasConstantSubRange_Sizes [4.38] says.

4.9.2. Formal specification of search_n

Like for search [4.35], our specification of search_n [4.39] is very similar to that of find2 [4.5].

```
/*@
  requires valid:     \valid_read(a + (0..n-1));
  assigns             \nothing;
  ensures  result:    0 <= \result <= n;

  behavior has_match:
    assumes           HasConstantSubRange(a, n, v, p);
    assigns           \nothing;
    ensures result:   0 <= \result <= n-p;
    ensures match:    AllEqual(a, \result, \result+p, v);
    ensures first:    !HasConstantSubRange(a, \result+p-1, v, p);

  behavior no_match:
    assumes           !HasConstantSubRange(a, n, v, p);
    assigns           \nothing;
    ensures result:   \result == n;

  complete behaviors;
  disjoint behaviors;
*/
size_type
search_n(const value_type* a, size_type n, value_type v, size_type p);
```

Listing 4.39: Formal specification of search_n

We again use two behaviors to capture the essential aspects of search_n.

- The behavior has_match applies if the sequence a contains an n-fold repetition of b. We express this condition with assumes by using the predicate HasConstantSubRange [4.38]. The result ensures clause of behavior has_match indicates that the return value must be in the range [0..n-p]. The match ensures clause expresses that the return value of search_n actually points to an index where b can be found p or more times in a. The first ensures clause expresses that the minimal index with this property is returned.
- The behavior no_match covers the case that there is no matching subsequence in sequence a. In this case, search_n must return the length n of the range a.

```
size_type
search_n(const value_type* a, size_type n, value_type v, size_type p)
{
  if (0u < p) {
    if (p <= n) {
      size_type start = 0u;

      /*@
        loop invariant match:     AllEqual(a, start, i, v);
        loop invariant start:     0 < start ==> a[start-1] != v;
        loop invariant bound:     start <= i + 1 <= start + p;
        loop invariant not_found: !HasConstantSubRange(a, i, v, p);
        loop assigns i, start;
        loop variant n - i;
      */
      for (size_type i = 0u; i < n; ++i) {
        if (a[i] != v) {
          start = i + 1u;
          //@ assert not_found: !HasConstantSubRange(a, i+1, v, p);
        }
        else {
          //@ assert match: a[i] == v;
          //@ assert match: AllEqual(a, start, i+1, v);
          if (p == i + 1u - start) {
            //@ assert bound: start + p == i + 1;
            //@ assert match: AllEqual(a, start, start+p, v);
            //@ assert match: \exists integer k; 0 <= k <= n-p && AllEqual(a, k, k+p
                , v);
            //@ assert match: HasConstantSubRange(a, n, v, p);
            return start;
          }
          else {
            //@ assert bound: i + 1 < start + p;
            continue;
          }
        }

        //@ assert not_found: !HasConstantSubRange(a, i+1, v, p);
      }

      //@ assert not_found: !HasConstantSubRange(a, n, v, p);
      return n;
    }
    else {
      //@ assert not_found: n < p;
      //@ assert not_found: !HasConstantSubRange(a, n, v, p);
      return n;
    }
  }
  else {
    //@ assert bound: p == 0;
    //@ assert match: AllEqual(a, 0, 0, v);
    //@ assert match: HasConstantSubRange(a, n, v, 0);
    return 0u;
  }
}
```

Listing 4.40: Implementation of `search_n`

4.9.3. Implementation of `search_n`

Although the specification of `search_n` [4.39] strongly resembles that of `search` [4.35], their implementations differ significantly. The implementation of `search_n` [4.40] has a time complexity of $O(n)$, whereas the implementation of `search` [4.36] employs an easy, but a non-optimal algorithm needing $O(n \cdot p)$ time.

Our implementation maintains in the variable `start` the beginning of the most recent consecutive range of values `v`. The loop invariant `not_found` states that we didn't find an `p`-fold repetition of `b` up to now; if we find one, we terminate the loop, returning `start`. We handle the boundary cases `n < p` and `p == 0` in explicit else branches. We found this easier when trying to ensure a verification by automatic provers.

4.10. The `find_end` algorithm

The `find_end` algorithm in the C++ Standard Library [20, §28.5.6] searches for the last subsequence that is identical to a given sequence when compared element-by-element. For our purposes we have modified the generic implementation to that of an array of type `value_type`. The signature now reads:

```
size_type
find_end(const value_type* a, size_type n, const value_type* b, size_type p);
```

The function `find_end` returns the greatest index `s` of the array `a` where the condition `a[s+k] == b[k]` holds for each index `k` with `0 <= k < p` (see Figure 4.41). If no such index exists, then `find_end` returns the length `n` of the array `a`. One has to remark the special case `p == 0`. In this case the last position of the empty string is found (the length `n`) and returned.

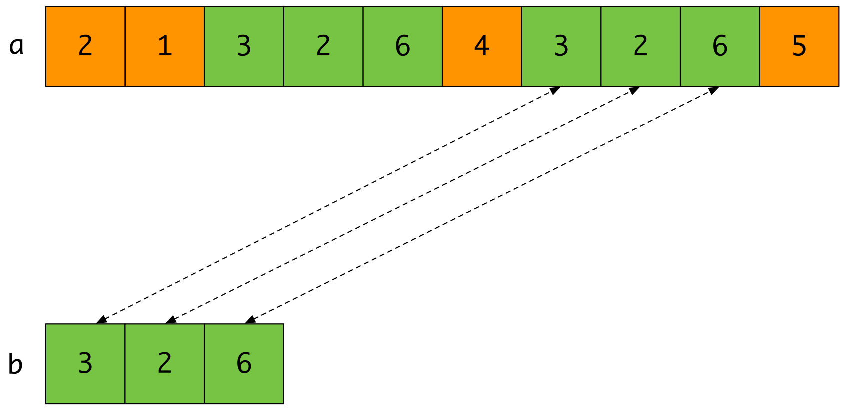

Figure 4.41.: Finding the last occurrence `b[0..p-1]` in `a[0..n-1]`

4.10.1. Formal specification of `find_end`

The following listing shows the specification of `find_end` [4.42]. Conceptually, the specification of the function `find_end` is very similar to that of `find2` [4.5]. We therefore use again behaviors to capture the essential aspects of `find_end`. It is quite clear that these behaviors are *complete* and *disjoint*.

The behavior `has_match` applies if the sequence `a` contains a subsequence identical to `b`. We express this condition with `assumes` using the predicate `HasSubRange` [4.34]. The `ensures` clause `bound` indicates that the return value must be in the range `0..n-p`. The clause `result` of behavior `has_match` expresses that `find_end` returns an index where `b` can be found in `a`. Finally, the clause `last` indicates that the sequence `a` does not contain `b` beginning at a position larger than `\result`.

The behavior `no_match` covers the case that there is no subsequence of `a` that equals `b`. In this case, `find_end` must return the length `n` of the range `a`.

```
/*@
  requires valid:   \valid_read(a + (0..n-1));
  requires valid:   \valid_read(b + (0..p-1));
  assigns           \nothing;
  ensures result:   0 <= \result <= n;

  behavior has_match:
    assumes         HasSubRange(a, n, b, p);
    assigns         \nothing;
    ensures bound:  0 <= \result <= n-p;
    ensures result: Equal{Here,Here}(a + \result, p, b);
    ensures last:   !HasSubRange(a, \result + 1, n, b, p);

  behavior no_match:
    assumes         !HasSubRange(a, n, b, p);
    assigns         \nothing;
    ensures result: \result == n;

  complete behaviors;
  disjoint behaviors;
*/
size_type
find_end(const value_type* a, size_type n,
         const value_type* b, size_type p);
```

Listing 4.42: Formal specification of `find_end`

4.10.2. Implementation of `find_end`

Our implementation of `find_end` [4.43] is similar to the one of `search` [4.36].

```
size_type
find_end(const value_type* a, size_type n,
         const value_type* b, size_type p)
{
  size_type r = n;

  if ((0u < p) && (p <= n)) {
    /*@
      loop invariant bound   :  r <= n - p || r == n;
      loop invariant not_found: r == n ==> !HasSubRange(a, p+i-1, b, p);
      loop invariant found:     r < n  ==> Equal{Here,Here}(a+r, p, b);
      loop invariant last:      r < n  ==> !HasSubRange(a, r+1, i+p-1, b, p);
      loop assigns i, r;
      loop variant n - i;
    */
    for (size_type i = 0u; i <= n - p; ++i) {
      if (equal(a + i, p, b)) {
        r = i;
      }
    }
  }

  return r;
}
```

Listing 4.43: Implementation of `find_end`

We maintain in the variable `r` the prospective value to be returned, according to the current knowledge. Initially, it is set to `n`, meaning “no occurrence of `b` found yet”. Whenever an occurrence is found, `r` is updated to its starting position.

The invariant `bound` states that `r` either still has the value `n` or has a value up to `n-p`. For the former case, invariant `not_found` indicates that no occurrence of `b` has been found. For the latter case, the loop invariant `found` indicates that an occurrence `b[0..p-1]` at `r` has indeed been found. The invariant `last`, on the other hand states that none was found *after* the index `r`.

4.11. The `count` algorithm

The `count` algorithm in the C++ Standard Library [20, §28.5.9] counts the frequency of occurrences for a particular element in a sequence. For our purposes we have modified the generic implementation to that of arrays of type `value_type`. The signature now reads:

```
size_type
count(const value_type* a, size_type n, value_type v);
```

Informally, the function returns the number of occurrences of `v` in the array `a`.

4.11.1. The logic function `Count`

When trying to specify `count` we are faced with the situation that ACSL does not provide a definition of counting a value in an array.[15] We therefore start with an axiomatic definition of *logic function* `Count` that captures the basic intuitive features of counting on an array section. The expression `Count(a,m,n,v)` returns the number of occurrences of `v` in `a[m],...,a[n-1]`.

The specification of `count` will then be fairly short because it employs our *logic function* `Count` whose (considerably) longer definition is given in the Listings 4.44 and 4.45.[16]

- The ACSL keyword `axiomatic` is used to structure the specification and gather the logic function `Count` and related lemmas. Note that the interval bounds `m` and `n` and the return value for `Count` are of type `integer`.
- The logic functions `Count` is recursively defined. It consist of two checks: whether the range is empty and for the value of the "current" element in the array. The recursion goes down on the range length. We also provide an overloaded version of `Count` that accepts only the length of an array, thus relieving the use the supply the argument $m = 0$ for the case of a complete array.
- Lemma `Count_Empty` [4.44] covers the cases of empty ranges.
- Lemmas `Count_Hit` [4.44] and `Count_Miss` [4.44] reduce counting of a range of length $n - m$ to a range of length $n - m - 1$.
- Lemmas `Count_One` [4.44] and `Count_Single` [4.44] built on on top of `Count_Hit` and `Count_Miss`. Using them simplifies several Coq proofs. They also slightly change the induction scheme from $n - 1 \to n$ to $n \to n + 1$.

[15] This statement is not quite true because the ACSL documentation lists `numof` as one of several *higher order logic constructions* [15, §2.6.7]. However, these *extended quantifiers* are mentioned only as experimental features.

[16] This definition of `Count` is a generalization of the *logic function* `nb_occ` of the ACSL specification [15].

```
/*@
  axiomatic Count
  {
    logic integer
    Count(value_type* a, integer m, integer n, value_type v) =
      n <= m ? 0 : Count(a, m, n-1, v) + (a[n-1] == v ? 1 : 0);

    logic integer
    Count(value_type* a, integer n, value_type v) = Count(a, 0, n, v);

    lemma Count_Empty:
      \forall value_type *a, v, integer m, n;
        n <= m  ==>  Count(a, m, n, v) == 0;

    lemma Count_Hit:
      \forall value_type *a, v, integer n, m;
        m < n          ==>
        a[n-1] == v    ==>
        Count(a, m, n, v) == Count(a, m, n-1, v) + 1;

    lemma Count_Miss:
      \forall value_type *a, v, integer n, m;
        m < n          ==>
        a[n-1] != v    ==>
        Count(a, m, n, v) == Count(a, m, n-1, v);

    lemma Count_One:
      \forall value_type *a, v, integer m, n;
        m <= n  ==>  Count(a, m, n+1, v) == Count(a, m, n, v) + Count(a, n, n+1, v);

    lemma Count_Single{K,L}:
      \forall value_type *a, *b, v, integer m, n;
        \at(a[m],K) == \at(b[n],L)   ==>
        Count{K}(a, m, m+1, v) == Count{L}(b, n, n+1, v);

    lemma Count_Equal{K,L}:
      \forall value_type *a, v, integer m, n, p;
        0 <= m <= n              ==>
        Equal{K,L}(a, m, n, p)   ==>
        Count{K}(a, m, n, v) == Count{L}(a, p, p + (n-m), v);

    lemma Count_Unchanged{K,L}:
      \forall value_type *a, v, integer m, n;
        Unchanged{K,L}(a, m, n)   ==>  Count{K}(a, m, n, v) == Count{L}(a, m, n, v);
```

Listing 4.44: The logic function `Count` (1)

- The logic function `Count` depends only on the set `a[m..n-1]` of memory locations. Lemma `Count_Unchanged` [4.44] makes this claim explicit by ensuring that `Count` produces the same result if the values `a[0..n-1]` do not change between two program states indicated by the labels `K` and `L`. We use here predicate `Unchanged` [7.1] to express the premise.
- Lemma `Count_Equal` [4.44] is a generalization of lemma `Count_Unchanged` for the case of comparing `Count` on two arrays.
- Lemmas `Count_Union` [4.44] and `Count_Cut` [4.44] allow to deal with partitions of arrays.

```
    lemma Count_Union:
      \forall value_type *a, v, integer k, m, n;
        0 <= k <= m <= n  ==>
        Count(a, k, n, v) == Count(a, k, m, v) + Count(a, m, n, v);

    lemma Count_Cut:
      \forall value_type *a, v, integer k, m, n;
        0 <= k <= m < n  ==> Count(a, k, n, v) ==
          Count(a, k, m, v) + Count(a, m, m+1, v) + Count(a, m+1, n, v);

    lemma Count_Single_Bounds:
      \forall value_type *a, v, integer n;
        0 <= Count(a, n, n+1, v) <= 1;

    lemma Count_Bounds:
      \forall value_type *a, v, integer m, n;
        0 <= m <= n  ==>  0 <= Count(a, m, n, v) <= n-m;

    lemma Count_Increasing:
      \forall value_type *a, v, integer m, n, p;
        m <= n <= p  ==>  Count(a, m, n, v) <= Count(a, m, p, v);

    lemma Count_Single_Shift:
      \forall value_type *a, v, integer n;
        0 <= n  ==>  Count(a+n, 0, 1, v) == Count(a, n, n+1, v);

    lemma Count_Shift:
      \forall value_type *a, v, integer m, n;
        0 <= m  ==>  0 <= n  ==>  Count(a+m, 0, n, v) == Count(a, m, m+n, v);
  }
*/
```

Listing 4.45: The logic function Count (2)

- Lemmas Count_Single_Bounds [4.44] and Count_Bounds [4.44] express lower and upper bounds of Count. Lemma Count_Increasing [4.44] states that Count is a monotonically increasing.
- Finally, lemmas Count_Single_Shift [4.44] and Count_Shift [4.44] state that Count is invariant under array shifts.

We mention here also lemma Count_SomeEqual [4.46] which brings together properties of Count [4.44] and Find [4.7].

```
/*@
  axiomatic CountFind
  {
    lemma Count_SomeEqual:
      \forall value_type *a, v, integer m, n;
        0 <= m < n             ==>
        0 < Count(a, m, n, v)  ==>
        SomeEqual(a, m, n, v);
  }
*/
```

Listing 4.46: The logic definition(s) CountFind

4.11.2. Formal specification of `count`

In the contract of `count` [4.47] we use the logic function `Count` [4.44] Note that our specification also states that the result of `count` is non-negative and less than or equal the size of the array.

```
/*@
  requires valid: \valid_read(a + (0..n-1));
  assigns         \nothing;
  ensures bound:  0 <= \result <= n;
  ensures count:  \result == Count(a, n, v);
*/
size_type
count(const value_type* a, size_type n, value_type v);
```

Listing 4.47: Formal specification of `count`

4.11.3. Implementation of `count`

The following listing shows a possible implementation of `count` [4.48]. Note that we refer to the logic function `Count` in one of the loop invariants.

```
size_type
count(const value_type* a, size_type n, value_type v)
{
  size_type counted = 0u;

  /*@
    loop invariant bound: 0 <= i <= n;
    loop invariant bound: 0 <= counted <= i;
    loop invariant count: counted == Count(a, i, v);
    loop assigns i, counted;
    loop variant n-i;
  */
  for (size_type i = 0u; i < n; ++i) {
    if (a[i] == v) {
      counted++;
    }
  }

  return counted;
}
```

Listing 4.48: Implementation of `count`

4.12. The `count2` algorithm

In this section, we specify the `count` algorithm in a different way, namely using the *inductively* defined predicate `CountInd` [4.49] from the following listing.

```
/*@
  inductive CountInd{L}(value_type *a, integer n, value_type v, integer sum)
  {
    case Nil{L}:
      \forall value_type *a, v, integer n;
        n <= 0  ==>  CountInd{L}(a, n, v, 0);

    case Hit{L}:
      \forall value_type *a, v, integer n, sum;
        0 < n  &&  a[n-1] == v  &&  CountInd{L}(a, n-1, v, sum)  ==>
        CountInd{L}(a, n, v, sum + 1);

    case Miss{L}:
      \forall value_type *a, v, integer n, sum;
        0 < n  &&  a[n-1] != v  &&  CountInd{L}(a, n-1, v, sum)  ==>
        CountInd{L}(a, n, v, sum);
  }
*/
```

Listing 4.49: Inductive definition `CountInd`

The definition consists of three cases.

- The `Nil` case states for arrays of negative pf zero length, the predicate only holds is `sum` is zero.
- The `Hit` and `Miss` define `CountInd` for arrays `a[0..n-1]` of size `n` referring to the array `a [0..n-2]` and the value `a[n-1]`.

We remark that the cases are very similar to the lemmas `Count_Empty` [4.44], `Count_Hit` [4.44] and `Count_Miss` [4.44], except we have use the additional argument `sum` to refer to the number of counted elements since `CountInd` is a predicate.

We have intentionally used the scheme $n - 1 \Rightarrow n$ instead of $n \Rightarrow n + 1$. In this particular case, it allows theorem provers to match loop indices with premises without additional hints to prove loop invariants.

4.12.1. Additional lemmas for the inductive predicate

The lemmas of `CountIndImplicit` [4.50] complement the lemmas of `Count` [4.44]. They demonstrate how existing lemmas can be rewritten for an inductive predicate. These lemmas are not required to prove the `count` function, but we provide them to complete the illustrative example of how inductive predicates could be utilized in the specifications.

The inductive definition is the "complete" definition which means that the predicate does not hold for cases outside of its domain of definition. We state this property explicitly through lemma `CountInd_Inverse` [4.51] in the following listing. Frama-C does not automatically generate this kind of property. The reason for not adding such a corresponding axiom apparently is that it "could confuse first-order theorem provers".[17]

[17] `https://stackoverflow.com/a/32457870`

```
/*@
  axiomatic CountIndImplicit
  {
    lemma CountInd_Empty{L}:
      \forall value_type *a, v, integer n;
       n <= 0  ==>  CountInd(a, n, v, 0);

    lemma CountInd_Hit{L}:
      \forall value_type *a, v, integer n, sum;
        0 < n                     ==>
        a[n-1] == v               ==>
        CountInd(a, n-1, v, sum)  ==>
        CountInd(a,   n, v, sum+1);

    lemma CountInd_Miss{L}:
      \forall value_type *a, v, integer n, sum;
        0 < n                     ==>
        a[n-1] != v               ==>
        CountInd(a, n-1, v, sum)  ==>
        CountInd(a,   n, v, sum);

    lemma CountInd_Unchanged{K,L}:
      \forall value_type *a, v, integer n, sum;
        Unchanged{K,L}(a, n)  ==>
        (CountInd{K}(a, n, v, sum)  <==>  CountInd{L}(a, n, v, sum));
  }
*/
```

Listing 4.50: The logic definition(s) `CountIndImplicit`

There is also the lemma `CountInd_NonNegative` [4.51] which states that the lower bound for the number of the counted elements is zero. The relation between the inductive definition `CountInd` and the explicit definition of `Count` [4.44] is expressed by lemma `CountInd_Count` [4.51].

```
/*@
  axiomatic CountIndLemmas
  {
    lemma CountInd_Inverse:
      \forall value_type *a, v, integer n, sum;
        CountInd(a, n, v, sum)  ==>
          (n <= 0 && sum == 0) ||
          (0 < n && a[n-1] != v && CountInd(a, n-1, v, sum)) ||
          (0 < n && a[n-1] == v && CountInd(a, n-1, v, sum-1));

    lemma CountInd_NonNegative{L}:
      \forall value_type *a, v, integer n, sum;
        CountInd(a, n, v, sum)  ==>  0 <= sum;

    lemma CountInd_Count{L}:
      \forall value_type *a, v, integer n;
        CountInd(a, n, v, Count(a, n, v));
  }
*/
```

Listing 4.51: The logic definition(s) `CountIndLemmas`

4.12.2. Specification of count2

The following listing contains the contracts of count2 [4.52]. It shows the use of the inductive predicate CountInd [4.49].

```
/*@
  requires valid: \valid_read(a + (0..n-1));
  assigns         \nothing;
  ensures bound:  0 <= \result <= n;
  ensures count:  CountInd(a, n, v, \result);
*/
size_type
count2(const value_type* a, size_type n, value_type v);
```

Listing 4.52: Formal specification of count2

4.12.3. Implementation of count2

The only difference between the implementation of count2 [4.53] and the implementation of count [4.48] is that we have to supply the value counted as an argument of the predicate CountInd [4.49].

```
size_type
count2(const value_type* a, size_type n, value_type v)
{
  size_type counted = 0u;

  /*@
    loop invariant bound: 0 <= i <= n;
    loop invariant bound: 0 <= counted <= i;
    loop invariant count: CountInd(a, i, v, counted);
    loop assigns i, counted;
    loop variant n-i;
  */
  for (size_type i = 0u; i < n; ++i) {
    if (a[i] == v) {
      counted++;
      //@ assert count: CountInd(a, i+1, v, counted);
    }
  }

  return counted;
}
```

Listing 4.53: Implementation of count2

5. Maximum and minimum algorithms

In this chapter we discuss the formal specification of algorithms in the C++ Standard Library [20, §28.7.8] that compute the maximum or minimum values of their arguments. As the algorithms in Chapter 4, they also do not modify any memory locations outside their scope. The most important new feature of the algorithms in this chapter is that they compare values using binary operators such as <.

We consider in this chapter the following algorithms.

- We discuss some properties of relations operators in §5.1.
- We introduce in §5.2 various predicates that describe basic order properties for arrays whose elements are of `value_type`.
- `clamp`, which is discussed in §5.3, is a very simple algorithms that “clamps” (or “clips”) a value between a pair of boundary values.
- `max_element` returns an index to a maximum element in a range. Similar to `find` it also returns the smallest of all possible indices. This algorithm is discussed in §5.5. In §5.6, we introduce an alternative specification `max_element2` which relies on user-defined predicates.
- `max_seq` in §5.7 is very similar to `max_element` and will serve as an example of *modular verification*. It returns the maximum value itself rather than an index to it.
- `min_element` in §5.8 can be used to find the smallest element in an array.
- `minmax_element` in §5.9 is used to find simultaneously the smallest and largest element in a given range. This algorithms relies on the auxiliary function `make_pair` (§5.4).

First, however, we discuss in §5.1 general properties that must be satisfied by the relational operators.

5.1. A note on relational operators

Note that in order to compare values, algorithms in the C++ Standard Library [20, §28.7.8] usually rely solely on the *less than* operator < or special function objects. To be precise, the operator < must be a *partial order*,[18] which means that the following rules must hold.

$$\begin{array}{lll} \text{irreflexivity} & \forall x & : \neg(x < x) \\ \text{asymmetry} & \forall x, y & : x < y \implies \neg(y < x) \\ \text{transitivity} & \forall x, y, z & : x < y \wedge y < z \implies x < z \end{array}$$

If you wish to check that the operator < of our `value_type`[19] satisfies these properties one can formulate the lemmas of `Less` [5.1] and verify them with Frama-C.

[18] See `http://en.wikipedia.org/wiki/Partially_ordered_set`
[19] See §2.3

```
/*@
  axiomatic Less
  {
    lemma Less_Irreflexivity:
      \forall value_type a; !(a < a);

    lemma Less_Antisymmetry:
      \forall value_type a, b; (a < b)  ==>  !(b < a);

    lemma Less_Transitivity:
      \forall value_type a, b, c; (a < b) && (b < c)  ==>  (a < c);

    lemma Greater_Less:
      \forall value_type a, b; (a > b)  <==>  (b < a);

    lemma LessOrEqual_Less:
      \forall value_type a, b; (a <= b)  <==>  !(b < a);

    lemma GreaterOrEqual_Less:
    \forall value_type a, b; (a >= b)  <==> !(a < b);
  }
*/
```

Listing 5.1: The logic definition(s) Less

It is of course possible to specify and implement the algorithms of this chapter by only using operator <. For example, a <= b can be written as a < b || a == b, or, for our particular ordering on value_type, as !(b < a). Listing Less [5.1] therefor also contains lemmas on representing the operator >, <=, and >= through operator <.

5.2. Predicates for bounds and extrema of arrays

We define in the following listing the predicates MaxElement [5.2] and MinElement [5.2] that we will use for the specification of various algorithms. We will discuss these predicates in more detail in §5.6 and §5.8.

```
/*@
  axiomatic ArrayExtrema
  {
    predicate
    MaxElement{L}(value_type* a, integer n, integer max) =
      0 <= max < n && UpperBound(a, n, a[max]);

    predicate
    MinElement{L}(value_type* a, integer n, integer min) =
      0 <= min < n && LowerBound(a, n, a[min]);
  }
*/
```

Listing 5.2: The logic definition(s) ArrayExtrema

The aforementioned predicates rely on the predicates `LowerBound` [5.3] and `UpperBound` [5.3] which are shown in the following listing together with the related predicates `StrictUpperBound` [5.3] and `StrictLowerBound` [5.3].

```
/*@
  axiomatic ArrayBounds
  {
    predicate
    LowerBound{L}(value_type* a, integer m, integer n, value_type v) =
      \forall integer i; m <= i < n  ==>  v <= a[i];

    predicate
    LowerBound{L}(value_type* a, integer n, value_type v) =
      LowerBound{L}(a, 0, n, v);

    predicate
    StrictLowerBound{L}(value_type* a, integer m, integer n, value_type v) =
      \forall integer i; m <= i < n  ==>  v < a[i];

    predicate
    StrictLowerBound{L}(value_type* a, integer n, value_type v) =
      StrictLowerBound{L}(a, 0, n, v);

    predicate
    UpperBound{L}(value_type* a, integer m, integer n, value_type v) =
      \forall integer i; m <= i < n  ==>  a[i] <= v;

    predicate
    UpperBound{L}(value_type* a, integer n, value_type v) =
      UpperBound{L}(a, 0, n, v);

    predicate
    StrictUpperBound{L}(value_type* a, integer m, integer n, value_type v) =
      \forall integer i; m <= i < n  ==>  a[i] < v;

    predicate
    StrictUpperBound{L}(value_type* a, integer n, value_type v) =
      StrictUpperBound{L}(a, 0, n, v);
  }
*/
```

Listing 5.3: The logic definition(s) `ArrayBounds`

These predicates concisely express the comparison of the elements in an array (segment) with a given value. We will heavily rely on these predicates both in this chapter and in Chapter 6.

5.3. The `clamp` algorithm

The `clamp` algorithm in the C++ Standard Library [20, §28.7.9] "clamps" a value between a pair of boundary values. The signature of our version of `clamp` reads:

```
value_type clamp(value_type v, value_type lower, value_type upper);
```

The function `clamp` returns `v` if the value is greater than `lower` and smaller than `upper`. Otherwise, if `v` is smaller than `lower`, then `lower` is returned. Finally, if `v` is greater than `upper`, `upper` is the returned.

5.3.1. Formal specification of `clamp`

The following listing contains the specification of `clamp` [5.4]. Note that we require that `lower` must be less or equal than `upper`.

```
/*@
  requires bound:   lower < upper;
  assigns           \nothing;
  ensures bound:    lower <= \result <= upper;

  behavior lower_bound:
    assumes          v < lower;
    assigns          \nothing;
    ensures result: \result == lower;

  behavior between:
    assumes          lower <= v <= upper;
    assigns          \nothing;
    ensures result: \result == v;

  behavior upper_bound:
    assumes          upper < v;
    assigns          \nothing;
    ensures result: \result == upper;

  complete behaviors;
  disjoint behaviors;
*/
value_type
clamp(value_type v, value_type lower, value_type upper);
```

Listing 5.4: Formal specification of `clamp`

5.3.2. Implementation of `clamp`

The implementation of `clamp` [5.5] can be verified without any additional annotations.

```
value_type
clamp(value_type v, value_type lower, value_type upper)
{
  return (v < lower) ? lower : (upper < v) ? upper : v;
}
```

Listing 5.5: Implementation of `clamp`

5.4. The auxiliary function `make_pair`

In order to be able to specify functions that work on pairs of indices we introduce in the following listing the type `size_type_pair`.

```
struct size_type_pair {
  size_type first;
  size_type second;
};

typedef struct size_type_pair size_type_pair;
```

Listing 5.6: The type `size_type_pair`

We will also use the auxiliary function `make_pair` which turns two indices `first` and `second` into an object of `size_type_pair`. The specification and implementation of `make_pair` [5.7] is shown here.

```
/*@
    assigns          \nothing;
    ensures result: \result.first  == first;
    ensures result: \result.second == second;
*/
static inline
size_type_pair
make_pair(size_type first, size_type second)
{
  size_type_pair pair;

  pair.first  = first;
  pair.second = second;

  return pair;
}
```

Listing 5.7: Formal specification of `make_pair`

5.5. The `max_element` algorithm

The `max_element` algorithm in the C++ Standard Library [20, §28.7.8] searches the maximum of a general sequence. The signature of our version of `max_element` reads:

```
size_type max_element(const value_type* a, size_type n);
```

The function finds the largest element in the range `a[0..n-1]`. More precisely, it returns the unique valid index `i` such that:

1. for each index `k` with `0 <= k < n` the condition `a[k] <= a[i]` holds and
2. for each index `k` with `0 <= k < i` the condition `a[k] < a[i]` holds.

The return value of `max_element` is `n` if and only if there is no maximum, which can only occur if `n == 0`.

5.5.1. Formal specification of `max_element`

The following listings shows the formal specification of `max_element` [5.8]. Note that we have subdivided the specification of `max_element` into the two behaviors `empty` and `not_empty`. The behavior `empty` contains the specification for the case that the range contains no elements. The behavior `not_empty` applies if the range has a positive length.

The ensures clause `max` of behavior `not_empty` indicates that the returned valid index `k` refers to a maximum value of the array. The postcondition `first` expresses that `k` is indeed the *first* occurrence of a maximum value in the array.

```
/*@
  requires valid:   \valid_read(a + (0..n-1));
  assigns           \nothing;
  ensures  result:  0 <= \result <= n;

  behavior empty:
    assumes          n == 0;
    assigns          \nothing;
    ensures result:  \result == 0;

  behavior not_empty:
    assumes          0 < n;
    assigns          \nothing;
    ensures result:  0 <= \result < n;
    ensures upper:   \forall integer i; 0 <= i < n       ==> a[i] <= a[\result];
    ensures first:   \forall integer i; 0 <= i < \result ==> a[i] <  a[\result];

  complete behaviors;
  disjoint behaviors;
*/
size_type
max_element(const value_type* a, size_type n);
```

Listing 5.8: Formal specification of `max_element`

5.5.2. Implementation of `max_element`

In our description, we concentrate on the *loop annotations* of the implementation of `max_element` [5.9].

```
size_type
max_element(const value_type* a, size_type n)
{
  if (0u < n) {
    size_type max = 0u;

    /*@
      loop invariant bound:  0 <= i <= n;
      loop invariant max:    0 <= max <  n;
      loop invariant upper:  \forall integer k; 0 <= k < i   ==> a[k] <= a[max];
      loop invariant first:  \forall integer k; 0 <= k < max ==> a[k] <  a[max];
      loop assigns max, i;
      loop variant n-i;
    */
    for (size_type i = 1u; i < n; i++) {
      if (a[max] < a[i]) {
        max = i;
      }
    }

    return max;
  }

  return n;
}
```

Listing 5.9: Implementation of `max_element`

The loop invariant `max` is needed to prove the postcondition `result` of the behavior `not_empty` of `max_element` [5.8]. Using loop invariant `upper` we prove the postcondition `upper` of the behavior `not_empty` of `max_element` [5.8]. Finally, the postcondition `first` of this behavior can be verified with the loop invariant `first`.

5.6. The max_element algorithm with predicates

In this section we present another specification of the max_element algorithm. The main difference is that we employ the predicate UpperBound [5.3] which basically expresses that a given value is greater or equal than all elements of a given array. Closely related to the predicate UpperBound is the predicate StrictUpperBound [5.3].

We also employ the predicate MaxElement [5.2]. This predicate states that the element at a given index max is an *upper bound* of the sequence a[0..n-1], and, by construction, a member of that sequence.

5.6.1. Formal specification of max_element2

The formal specification of max_element2 [5.10] is shown in the following listing. Note that we also use the predicate StrictUpperBound [5.3] in order to express that max_element2 returns the *first* maximum position in a[0..n-1].

```
/*@
  requires valid:     \valid_read(a + (0..n-1));
  assigns             \nothing;
  ensures  result:    0 <= \result <= n;

  behavior empty:
    assumes           n == 0;
    assigns           \nothing;
    ensures result:   \result == 0;

  behavior not_empty:
    assumes           0 < n;
    assigns           \nothing;
    ensures result:   0 <= \result < n;
    ensures max:      MaxElement(a, n, \result);
    ensures first:    StrictUpperBound(a, \result, a[\result]);

  complete behaviors;
  disjoint behaviors;
*/
size_type
max_element2(const value_type* a, size_type n);
```

Listing 5.10: Formal specification of max_element2

5.6.2. Implementation of `max_element2`

The implementation of `max_element2` [5.11] is of course very similar to that of `max_element` [5.9]—except that the loop invariants now also use the above mentioned predicates.

```
size_type
max_element2(const value_type* a, size_type n)
{
  if (0u < n) {
    size_type max = 0u;

    /*@
      loop invariant bound:   0 <= i <= n;
      loop invariant max:     0 <= max < n;
      loop invariant upper:   UpperBound(a, i, a[max]);
      loop invariant first:   StrictUpperBound(a, max, a[max]);
      loop assigns max, i;
      loop variant n-i;
    */
    for (size_type i = 0u; i < n; i++) {
      if (a[max] < a[i]) {
        max = i;
      }
    }

    return max;
  }

  return n;
}
```

Listing 5.11: Implementation of `max_element2`

5.7. The max_seq algorithm

In this section we consider the function max_seq [14, Ch. 3]) which is very similar to the function max_element [5.8]. The main difference between max_seq and max_element is that max_seq returns the maximum value (not just the index of it). Therefore, it requires a *non-empty* range as an argument.

Of course, max_seq can easily be implemented using max_element2 [5.11]. Moreover, relying only on the formal specification of max_element2 [5.10], we are also able to deductively verify the correctness of this implementation. Thus, we have a simple example of *modular verification* in the following sense:

> Any implementation of max_element2 that is separately proven to implement the contract max_element2 [5.10] makes max_seq behave correctly. Once the contracts have been defined, the function max_element2 could be implemented in parallel, or just after max_seq, without affecting the verification of max_seq.

5.7.1. Formal specification of max_seq

The following listing shows the formal specification of max_seq [5.12].

```
/*@
  requires    0 < n;
  requires    \valid_read(p + (0..n-1));
  assigns     \nothing;
  ensures     \forall integer i; 0 <= i <= n-1 ==> \result >= p[i];
  ensures     \exists integer e; 0 <= e <= n-1 &&  \result == p[e];
*/
value_type
max_seq(const value_type* p, size_type n);
```

Listing 5.12: Formal specification of max_seq

Using the first requires-clause we express that max_seq needs a *non-empty* range as input. Our postconditions formalize that max_seq indeed returns the maximum value of the range.

5.7.2. Implementation of max_seq

The implementation of max_seq [5.13] consists of a simple call to max_element2 [5.11]. Since max_seq requires a non-empty range the call of max_element2 returns an index to a maximum value in the range. The fact that max_element2 returns the smallest index is of no importance in this context.

```
value_type
max_seq(const value_type* p, size_type n)
{
  return p[max_element2(p, n)];
}
```

Listing 5.13: Implementation of max_seq

5.8. The min_element algorithm

The min_element algorithm in the C++ Standard Library [20, §28.7.8] searches the minimum in a general sequence. The signature of our version of min_element reads:

```
size_type min_element(const value_type* a, size_type n);
```

The function min_element finds the smallest element in the range a[0..n-1]. More precisely, it returns the unique valid index i such that a[i] is minimal among the values a[0], ..., a[n-1], and i is the first position with that property. The return value of min_element is n if and only if n == 0.

We use the predicate LowerBound [5.3] that basically expresses that a given value is less or equal than all elements of a given array (section). Closely related to the predicate LowerBound is the predicate StrictLowerBound [5.3]. We also use the predicate MinElement [5.2] which states that the element at a given index min is a *lower bound* of the sequence a[0..n-1], and, by construction, a member of that sequence.

5.8.1. Formal specification of min_element

The following listing contains the specification of min_element [5.14]. Note that we also use the predicate StrictLowerBound [5.3] in order to express that min_element returns the *first* minimum position in a[0..n-1].

```
/*@
  requires valid:   \valid_read(a + (0..n-1));
  assigns           \nothing;
  ensures  result:  0 <= \result <= n;

  behavior empty:
    assumes         n == 0;
    assigns         \nothing;
    ensures result: \result == 0;

  behavior not_empty:
    assumes         0 < n;
    assigns         \nothing;
    ensures result: 0 <= \result < n;
    ensures min:    MinElement(a, n, \result);
    ensures first:  StrictLowerBound(a, \result, a[\result]);

  complete behaviors;
  disjoint behaviors;
*/
size_type
min_element(const value_type* a, size_type n);
```

Listing 5.14: Formal specification of min_element

5.8.2. Implementation of `min_element`

The implementation of `min_element` [5.15] uses the predicates `LowerBound` [5.3] and `StrictLowerBound` [5.3] in its loop annotations.

```
size_type
min_element(const value_type* a, size_type n)
{
  if (0u < n) {
    size_type min = 0u;

    /*@
      loop invariant bound:  0 <= i   <= n;
      loop invariant min:    0 <= min <  n;
      loop invariant lower:  LowerBound(a, i, a[min]);
      loop invariant first:  StrictLowerBound(a, min, a[min]);
      loop assigns min, i;
      loop variant n-i;
    */
    for (size_type i = 0u; i < n; i++) {
      if (a[i] < a[min]) {
        min = i;
      }
    }

    return min;
  }

  return n;
}
```

Listing 5.15: Implementation of `min_element`

5.9. The `minmax_element` algorithm

The `minmax_element` algorithm in the C++ Standard Library [20, §28.7.8] searches *both* the minimum *and* the maximum in a sequence. The signature of our version of `min_element` reads:

```
size_type_pair minmax_element(const value_type* a, size_type n);
```

Note that `minmax_element` returns a *pair* of indices (see §5.4). This pair contains the *first* position where the minimum occurs in the sequence `a[0..n-1]` and the *last* position where maximum occurs.

The properties of the index for the minimum value are the same as the properties of `min_element` [5.14]. However, the properties of the index that marks the maximum element, are slightly different from the properties of `max_element` [5.8]. The `max_element` algorithm returns the position of the *first* occurrence of the maximum element if it occurs multiple times in the sequence. The `minmax_element` algorithm returns the position of the last occurrence of the maximum element.

5.9.1. Formal specification of `minmax_element`

The following listing shows the acsl specification of `minmax_element` [5.16]. Note that we use the predicates `StrictLowerBound` [5.3] and `StrictUpperBound` [5.3] in order to express that the algorithm returns the positions of both the *first minimum* and the *last maximum*. We also use the predicates `MinElement` [5.2] and `MaxElement` [5.2]. Thus reflects of course the use of this predicates for the algorithms `min_element` [5.14] and `max_element` [5.8].

```
/*@
  requires valid:    \valid_read(a + (0..n-1));
  assigns            \nothing;
  ensures result:    0 <= \result.first  <= n;
  ensures result:    0 <= \result.second <= n;

  behavior empty:
    assumes          0 == n;
    assigns          \nothing;
    ensures result:  \result.first == 0;
    ensures result:  \result.second == 0;

  behavior not_empty:
    assumes          0 < n;
    assigns          \nothing;
    ensures result:  0 <= \result.first < n;
    ensures result:  0 <= \result.second < n;

    ensures min:     MinElement(a, n, \result.first);
    ensures first:   StrictLowerBound(a, \result.first, a[\result.first]);
    ensures max:     MaxElement(a, n, \result.second);
    ensures last:    StrictUpperBound(a, \result.second+1, n, a[\result.second]);
*/
size_type_pair
minmax_element(const value_type* a, size_type n);
```

Listing 5.16: Formal specification of `minmax_element`

The specification is similar to the specifications of `min_element` and `max_element`. The only difference lies in the postcondition `last`. Here the postcondition states that after the position of the maximum element there is no value greater or equal the maximum element. This differs from the specification of max_element, where the first occurrence of the maximum value has to be returned.

5.9.2. Implementation of minmax_element

The implementation of minmax_element [5.17] uses the auxiliary function make_pair [5.7] to construct a pair of indices. We will focus on the loop invariant last, because it is the only loop invariant that differs from the implementations of min_element [5.15] and max_element [5.9].

```
size_type_pair
minmax_element(const value_type* a, size_type n)
{
  if (0u < n) {
    size_type min = 0u;
    size_type max = 0u;

    /*@
      loop invariant bound: 0 <= i     <= n;
      loop invariant min:   0 <= min   <  n;
      loop invariant max:   0 <= max   <  n;
      loop invariant lower: LowerBound(a, i, a[min]);
      loop invariant upper: UpperBound(a, i, a[max]);
      loop invariant first: StrictLowerBound(a, min, a[min]);
      loop invariant last:  StrictUpperBound(a, max+1, i, a[max]);
      loop assigns min, max, i;

      loop variant n-i;
    */
    for (size_type i = 0u; i < n; i++) {
      if (a[i] >= a[max]) {
        max = i;
      }

      if (a[i] < a[min]) {
        min = i;
      }
    }

    return make_pair(min, max);
  }

  return make_pair(n, n);
}
```

Listing 5.17: Implementation of minmax_element

As already mentioned we had to alter the range for the predicate StrictUpperBound [5.3] to fit into the property of returning the last maximum position that occurred.

6. Binary search algorithms

In this chapter, we consider the four *binary search* algorithms of the C++ Standard Library [20, §28.7.3], namely

- `lower_bound` in §6.1
- `upper_bound` in §6.2
- two variants for the implementation of `equal_range` in §6.3
- two variants for the formal specification of `binary_search` in §6.4

As in the case of the of maximum/minimum algorithms from Chapter 5 the binary search algorithms primarily use the less-than operator < (and the derived operators <=, > and >=) to determine whether a particular value is contained in an increasing range. Thus, different to the `find` algorithm in §4.1, the equality operator == will play only a supporting part in the specification of binary search.

In order to make the specifications of the binary search algorithms more compact and (arguably) more readable we re-use the predicates `LowerBound` [5.3], `StrictLowerBound` [5.3], `UpperBound` [5.3], and `StrictUpperBound` [5.3].

All binary search algorithms require that their input array is arranged in increasing order. The following listing shows two versions of predicate `Increasing` [6.1]. The first one defines when a section of an array is in increasing order. The second version uses the first one to express that the whole array is in increasing order.

```
/*@
  axiomatic Increasing
  {
    predicate
    Increasing{L}(value_type* a, integer m, integer n) =
      \forall integer i, j; m <= i < j < n  ==>  a[i] <= a[j];

    predicate
    Increasing{L}(value_type* a, integer n) = Increasing{L}(a, 0, n);
  }
*/
```

Listing 6.1: The logic definition(s) `Increasing`

There is also the overloaded predicate `WeaklyIncreasing` [6.2] that we will user for the verification of other algorithms.

```
/*@
  axiomatic WeaklyIncreasing
  {
    predicate
    WeaklyIncreasing{L}(value_type* a, integer m, integer n) =
      \forall integer i; m <= i < n-1  ==>  a[i] <= a[i+1];

    predicate
    WeaklyIncreasing{L}(value_type* a, integer n) = WeaklyIncreasing{L}(a, 0, n);
  }
*/
```

Listing 6.2: The logic definition(s) `WeaklyIncreasing`

Users inexperienced in formal verification often have a blind spot at the difference between `Increasing` and `WeaklyIncreasing`. Both versions are logically equivalent, and proving that `Increasing` implies `WeaklyIncreasing` is even trivial. However, proving the converse direction is not, and requires an induction on the array size `n`, employing the transitivity of `<=` in the induction step. Humans are trained to perform such inductions unnoticed, but none of the automated provers supported by Frama-C is able to perform induction. The following Listing contains several lemmas on the relationship of `WeaklyIncreasing` and `Increasing`.

```
/*@
  axiomatic IncreasingLemmas
  {
    lemma Increasing_WeaklyIncreasing{L}:
      \forall value_type* a, integer m, n;
        0 <= m <= n                 ==>
        Increasing(a, m, n)         ==>
        WeaklyIncreasing(a, m, n);

    lemma WeaklyIncreasing_Increasing{L}:
      \forall value_type* a, integer m, n;
        0 <= m <= n                ==>
        WeaklyIncreasing(a, m, n)  ==>
        Increasing(a, m, n);

    lemma Increasing_Shift{L}:
      \forall value_type *a, integer l, r;
        0 <= l <= r             ==>
        Increasing{L}(a, l, r)  ==>
        Increasing{L}(a+l, r-l);

    lemma Increasing_Equal{K,L}:
      \forall value_type* a, integer m, n, p;
        Increasing{K}(a, m, n)     ==>
        Equal{K,L}(a, m, n, m+p)   ==>
        Increasing{L}(a, m+p, n+p);
  }
*/
```

Listing 6.3: The logic definition(s) `IncreasingLemmas`

We usually exploit the relationship of the predicates `Increasing` and `WeaklyIncreasing` in the following way:

- We use the predicate `Increasing` in the preconditions and postconditions of function contracts.
- The `WeaklyIncreasing` is employed for assertions and loop invariants whenever we have to verify that an algorithm (typically a sorting algorithm) produces an increasing array.
- Finally, to conclude that a *weakly increasing* array is in fact *increasing* we rely on lemma `WeaklyIncreasing_Increasing` [6.3] .

6.1. The `lower_bound` algorithm

The `lower_bound` algorithm is one of the four binary search algorithms of the C++ Standard Library [20, §28.7.3.1]. For our purposes we have modified the generic implementation to that of an array of type `value_type`. The signature now reads:

```
size_type
lower_bound(const value_type* a, size_type n, value_type v);
```

As with the other binary search algorithms `lower_bound` requires that its input array is in increasing order. The index `lb`, that `lower_bound` returns satisfies the inequality

$$0 \leq \mathtt{lb} \leq n \tag{6.1}$$

and has the following properties for a valid index `k` of the array under consideration

$$0 \leq k < \mathtt{lb} \quad \Longrightarrow \quad a[k] < \mathtt{v} \tag{6.2}$$
$$\mathtt{lb} \leq k < n \quad \Longrightarrow \quad \mathtt{v} \leq a[k] \tag{6.3}$$

Conditions (6.2) and (6.3) imply that `v` can only occur in the array section `a[lb..n-1]`. In this sense `lower_bound` returns a *lower bound* for the potential indices.

As an example, we consider in Figure 6.4 an increasingly ordered array. The arrows indicate which indices will be returned by `lower_bound` for a given value. Note that the index 9 points *one past end* of the array. Values that are not contained in the array are colored in gray.

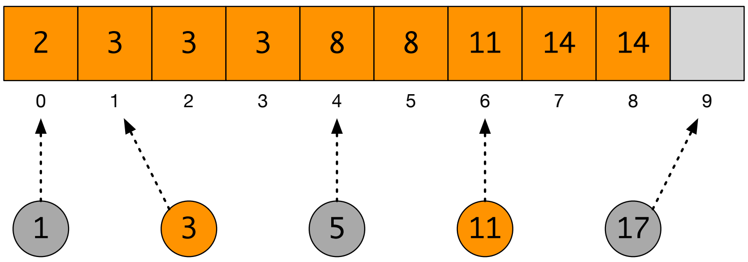

Figure 6.4.: Some examples for `lower_bound`

Figure 6.4 also clarifies that care must be taken when interpreting the return value of `lower_bound`. An important difference to the algorithms in Chapter 4 is that a return value of `lower_bound` that is less than *n* does not necessarily implies `a[lb] == v`. We can only be sure that `v <= a[lb]` holds.

6.1.1. Formal specification of `lower_bound`

The specification of `lower_bound` [6.5] is shown in the following listing. The preconditions `increasing` expresses that the array values need to be in increasing order. The postconditions reflect the conditions listed above and can be expressed using the predicates `LowerBound` [5.3] and `StrictUpperBound` [5.3].

- Condition (6.1) becomes postcondition `result`

- Condition (6.2) becomes postcondition `left`
- Condition (6.3) becomes postcondition `right`

```
/*@
  requires valid:      \valid_read(a + (0..n-1));
  requires increasing: Increasing(a, n);
  assigns              \nothing;
  ensures result:      0 <= \result <= n;
  ensures left:        StrictUpperBound(a, 0, \result, v);
  ensures right:       LowerBound(a, \result, n, v);
*/
size_type
lower_bound(const value_type* a, size_type n, value_type v);
```

Listing 6.5: Formal specification of `lower_bound`

6.1.2. Implementation of `lower_bound`

The following listing shows our implementation of `lower_bound` [6.6]. Each iteration step narrows down the range that contains the sought-after result. The loop invariants express that in each iteration step all indices less than the temporary left bound `left` contain values that are less than `v` and all indices not less than the temporary right bound `right` contain values that are greater or equal than `v`. The expression to compute `middle` is slightly more complex than the naïve `(left+right)/2`, but it avoids potential overflows.

```
size_type
lower_bound(const value_type* a, size_type n, value_type v)
{
  size_type left  = 0u;
  size_type right = n;

  /*@
    loop invariant bound:  0 <= left <= right <= n;
    loop invariant left:   StrictUpperBound(a, 0,  left, v);
    loop invariant right:  LowerBound(a, right, n, v);

    loop assigns left, right;
    loop variant right - left;
  */
  while (left < right) {
    const size_type middle = left + (right - left) / 2u;

    if (a[middle] < v) {
      left = middle + 1u;
    }
    else {
      right = middle;
    }
  }

  return left;
}
```

Listing 6.6: Implementation of `lower_bound`

6.2. The `upper_bound` algorithm

The `upper_bound` algorithm of the C++ Standard Library [20, §28.7.3.2] is a variant of binary search and closely related to `lower_bound` [6.5]. The signature reads:

```
size_type
upper_bound(const value_type* a, size_type n, value_type v)
```

As with the other binary search algorithms, `upper_bound` requires that its input array is in increasing order. The index `ub` returned by `upper_bound` satisfies the inequality

$$0 \leq \mathtt{ub} \leq n \tag{6.4}$$

and is involved in the following implications for a valid index `k` of the array under consideration

$$0 \leq k < \mathtt{ub} \quad \Longrightarrow \quad a[k] \leq \mathtt{v} \tag{6.5}$$

$$\mathtt{ub} \leq k < n \quad \Longrightarrow \quad \mathtt{v} < a[k] \tag{6.6}$$

Conditions (6.5) and (6.6) imply that `v` can only occur in the array section `a[0..ub-1]`. In this sense `upper_bound` returns a *upper bound* for the potential indices where `v` can occur. It also means that the searched-for value `v` can *never* be located at the index `ub`.

Figure 6.7 is a variant of Figure 6.4 for the case of `upper_bound` and the same example array. The arrows indicate which indices will be returned by `upper_bound` for a given value. Note how, compared to Figure 6.4, only the arrows from values that *are present* in the array change their target index.

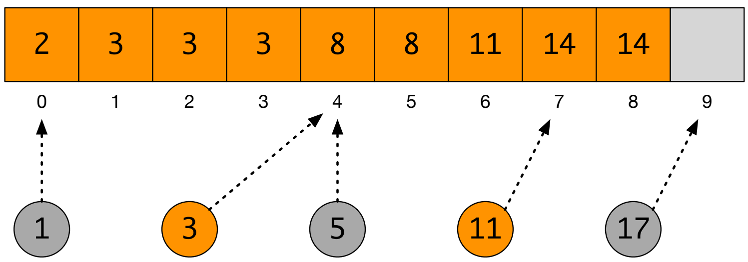

Figure 6.7.: Some examples for `upper_bound`

6.2.1. Formal specification of `upper_bound`

The following listing shows the specification of `upper_bound` [6.8] which is quite similar to the specification of `lower_bound` [6.5]. The precondition `increasing` expresses that the array values need to be in increasing order.

The postconditions reflect the conditions listed above and can be expressed using predicates `UpperBound` [5.3] and `StrictLowerBound` [5.3], namely,

- Condition (6.4) becomes postcondition `result`
- Condition (6.5) becomes postcondition `left`
- Condition (6.6) becomes postcondition `right`

```
/*@
  requires valid:      \valid_read(a + (0..n-1));
  requires increasing: Increasing(a, n);
  assigns              \nothing;
  ensures result:      0 <= \result <= n;
  ensures left:        UpperBound(a, 0, \result, v);
  ensures right:       StrictLowerBound(a, \result, n, v);
*/
size_type
upper_bound(const value_type* a, size_type n, value_type v);
```

Listing 6.8: Formal specification of `upper_bound`

6.2.2. Implementation of upper_bound

Our implementation of `upper_bound` [6.9] is shown in the following listing. The loop invariants express that for each iteration step all indices less than the temporary left bound `left` contain values not greater than `v` and all indices not less than the temporary right bound `right` contain values greater than `v`.

```
size_type
upper_bound(const value_type* a, size_type n, value_type v)
{
  size_type left  = 0u;
  size_type right = n;

  /*@
    loop invariant bound:  0 <= left <= right <= n;
    loop invariant left:   UpperBound(a, 0, left, v);
    loop invariant right:  StrictLowerBound(a, right, n, v);

    loop assigns left, right;
    loop variant right - left;
  */
  while (left < right) {
    const size_type middle = left + (right - left) / 2u;

    if (a[middle] <= v) {
      left = middle + 1u;
    }
    else {
      right = middle;
    }
  }

  return right;
}
```

Listing 6.9: Implementation of `upper_bound`

6.3. The `equal_range` algorithm

The `equal_range` algorithm is one of the four binary search algorithms of the C++ Standard Library [20, §28.7.3.3]. As with the other binary search algorithms `equal_range` requires that its input array is in increasing order. The specification of `equal_range` states that it *combines* the results of the algorithms `lower_bound` [6.5] and `upper_bound` [6.8].

For our purposes we have modified `equal_range` to take an array of type `value_type`. Moreover, instead of a pair of iterators, our version returns a pair of indices. To be more precise, the return type of `equal_range` is the struct `size_type_pair` from Listing 5.6. Thus, the signature of `equal_range` now reads:

```
size_type_pair
equal_range(const value_type* a, size_type n, value_type v);
```

Figure 6.10 combines Figure 6.4 with Figure 6.7 in order visualize the behavior of `equal_range` for select test cases. The two types of arrows → and ⇢ represent the respective fields `first` and `second` of the return value. For values that are not contained in the array, the two arrows point to the same index. More generally, if `equal_range` returns the pair $(\texttt{lb}, \texttt{ub})$, then the difference $\texttt{ub} - \texttt{lb}$ is equal to the number of occurrences of the argument `v` in the array.

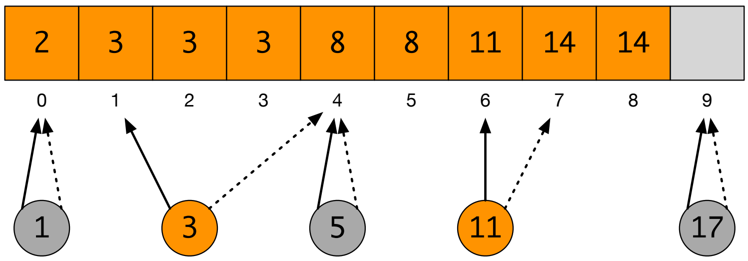

Figure 6.10.: Some examples for `equal_range`

We will provide two implementations of `equal_range` and verify both of them. The first implementation `equal_range` [6.12] just straightforwardly calls `lower_bound` [6.5] and `upper_bound` [6.8] and simply returns the pair of their respective results. The second implementation `equal_range2` [6.13], which is more elaborate, follows the original C++ code by attempting to minimize duplicate computations. Let $(\texttt{lb}, \texttt{ub})$ be the return value `equal_range`, then the conditions (6.1)–(6.6) can be merged into the inequality

$$0 \le \texttt{lb} \le \texttt{ub} \le n \tag{6.7}$$

and the following three implications for a valid index k of the array under consideration

$$0 \le k < \texttt{lb} \quad\Longrightarrow\quad a[k] < \texttt{v} \tag{6.8}$$
$$\texttt{lb} \le k < \texttt{ub} \quad\Longrightarrow\quad a[k] = \texttt{v} \tag{6.9}$$
$$\texttt{ub} \le k < n \quad\Longrightarrow\quad a[k] > \texttt{v} \tag{6.10}$$

Here are some justifications for these conditions.

- Conditions (6.8) and (6.10) are just the Conditions (6.2) and (6.6), respectively.
- The Inequality (6.7) follows from the Inequalities (6.1) and (6.4) and the following considerations: If ub were less than lb, then according to (6.8) we would have $a[\texttt{ub}] < \texttt{v}$. One the other hand, we know from (6.10) that opposite inequality $\texttt{v} < a[\texttt{ub}]$ holds. Therefore, we have $\texttt{lb} \le \texttt{ub}$.

- Condition (6.9) follows from the combination of (6.3) and (6.5) and the fact that ≤ is a total order on the integers.

6.3.1. Formal specification of equal_range

The following listing show the specification of equal_range [6.11] (and of equal_range2).

```
/*@
  requires valid:      \valid_read(a + (0..n-1));
  requires increasing: Increasing(a, n);
  assigns              \nothing;
  ensures result:      0 <= \result.first <= \result.second <= n;
  ensures left:        StrictUpperBound(a, 0, \result.first, v);
  ensures middle:      AllEqual(a, \result.first, \result.second, v);
  ensures right:       StrictLowerBound(a, \result.second, n, v);
 */
size_type_pair
equal_range(const value_type* a, size_type n, value_type v);
```

Listing 6.11: Formal specification of equal_range

The precondition increasing expresses that the array values need to be in increasing order.

The postconditions reflect the conditions listed above and can be expressed using the already introduced predicates AllEqual [4.15], StrictUpperBound [5.3] and StrictLowerBound [5.3].

- Condition (6.7) becomes postcondition result
- Condition (6.8) becomes postcondition left
- Condition (6.9) becomes postcondition middle
- Condition (6.10) becomes postcondition right

6.3.2. Implementation of equal_range

Our first implementation of equal_range [6.12] is shown in the following listing. We just call the two functions lower_bound [6.5] and upper_bound [6.8] and return their respective results as a pair.

```
size_type_pair
equal_range(const value_type* a, size_type n, value_type v)
{
  size_type first  = lower_bound(a, n, v);
  size_type second = upper_bound(a, n, v);
  //@ assert aux: second < n  ==>  v < a[second];
  return make_pair(first, second);
}
```

Listing 6.12: Implementation of equal_range

In a very early version of this document we had proven the similar assertion first <= second with the interactive theorem prover Coq. After reviewing this proof we formulated the new assertion aux that uses a fact from the postcondition of upper_bound [6.8]. The benefit of this reformulation is that both the assertion aux and the postcondition first <= second can now be verified automatically.

6.3.3. Implementation of `equal_range2`

The first implementation of `equal_range` [6.12] does more work than needed. In the following listing `equal_range2` [6.13] we show that it is possible to perform as much range reduction as possible before calling `upper_bound` [6.8] and `lower_bound` [6.5] on the reduced ranges.

```
size_type_pair
equal_range2(const value_type* a, size_type n, value_type v)
{
  size_type first  = 0u;
  size_type middle = 0u;
  size_type last   = n;

  /*@
    loop invariant bounds: 0 <= first <= last <= n;
    loop invariant left:   StrictUpperBound(a, 0, first, v);
    loop invariant right:  StrictLowerBound(a, last, n, v);
    loop assigns first, last, middle;
    loop variant last - first;
   */
  while (last > first) {
    middle = first + (last - first) / 2u;

    if (a[middle] < v) {
      first = middle + 1u;
    }
    else if (v < a[middle]) {
      last = middle;
    }
    else {
      break;
    }
  }

  if (first < last) {
    //@ assert increasing: Increasing(a, first, middle);
    size_type left = first + lower_bound(a + first, middle - first, v);
    //@ assert middle: LowerBound(a, left, middle, v);
    //@ assert left:   StrictUpperBound(a, first, left, v);
    ++middle;
    //@ assert increasing: Increasing(a, middle, last);
    size_type right = middle + upper_bound(a + middle, last - middle, v);
    //@ assert middle: UpperBound(a, middle, right, v);
    //@ assert right:  StrictLowerBound(a, right, last, v);
    //@ assert middle: AllEqual(a, left, right, v);
    return make_pair(left, right);
  }
  else {
    return make_pair(first, first);
  }
}
```

Listing 6.13: Implementation of `equal_range2`

Due to the higher code complexity of the second implementation, additional assertions had to be inserted in order to ensure that Frama-C/WP is able to verify the correctness of the code. All of these assertions are related to pointer arithmetic and shifting base pointers. They fall into three groups and are briefly discussed below. In order to enable the automatic verification of these properties we added the following collection of `ArrayBoundsShift` [6.14].

```
/*@
  axiomatic ArrayBoundsShift
  {
    lemma LowerBound_Shift{L}:
      \forall value_type *a, val, integer b, c, d;
        LowerBound{L}(a+b, c,   d,   val)  ==>
        LowerBound{L}(a,   c+b, d+b, val);

    lemma StrictLowerBound_Shift{L}:
      \forall value_type *a, val, integer b, c, d;
        StrictLowerBound{L}(a+b, c,   d,   val)  ==>
        StrictLowerBound{L}(a,   c+b, d+b, val);

    lemma UpperBound_Shift{L}:
      \forall value_type *a, val, integer b, c;
        UpperBound{L}(a+b,  0, c-b, val)  ==>
        UpperBound{L}(a,    b, c,   val);

    lemma StrictUpperBound_Shift{L}:
      \forall value_type *a, val, integer b, c;
        StrictUpperBound{L}(a+b, 0, c-b, val)  ==>
        StrictUpperBound{L}(a,   b, c,   val);
  }
*/
```

Listing 6.14: The logic definition(s) `ArrayBoundsShift`

The `increasing` properties

Both `lower_bound` [6.5] and `upper_bound` [6.8] expect that they operate on increasingly ordered arrays. This is of course also true for `equal_range` [6.11], however, inside our second implementation we need a more specific formulation, namely,

```
Increasing(a + middle, last - middle)
```

With the three-argument form of predicate `Increasing` [6.1] we can formulate out an intermediate step. This enables the provers to verify the preconditions of the call to `lower_bound` [6.5] automatically. A similar assertion is present before the call to `upper_bound` [6.8].

The `strict` and `constant` properties

Part of the post conditions of `equal_range` [6.11] is that `v` is both a strict upper and a strict lower bound. However, the calls to `lower_bound` and `upper_bound` only give us

```
StrictUpperBound(a + first, 0, left - first, v)

StrictLowerBound(a + middle, right - middle, last - middle, v)
```

which is not enough to reach the desired post conditions automatically. One intermediate step for each of the assertions was sufficient to guide the prover to the desired result.

Conceptually similar to the `strict` properties the `constant` properties guide the prover towards

```
LowerBound(a, left, n, v)
```

```
UpperBound(a, 0, right, v)
```

Combining these properties allow the postcondition `middle` to be derived automatically.

6.4. The `binary_search` algorithm

The `binary_search` algorithm is one of the four binary search algorithms of the C++ Standard Library [20, §28.7.3.4]. For our purposes we have modified the generic implementation to that of an array of type `value_type`. The signature now reads:

```
bool binary_search(const value_type* a, size_type n, value_type  v);
```

Again, `binary_search` requires that its input array is in increasing order. It will return **`true`** if there exists an index `i` in `a` such that `a[i] == v` holds.[20]

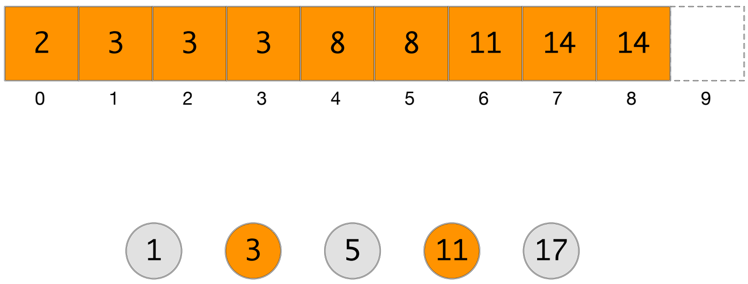

Figure 6.15.: Some examples for `binary_search`

In Figure 6.15 we do not need to use arrows to visualize the effects of `binary_search`. The colors orange and grey of the sought-after values indicate whether the algorithm returns true or false, respectively.

6.4.1. Formal specification of `binary_search` and `binary_search2`

The ACSL specification of `binary_search` [6.16] is shown in the following listing.

```
/*@
  requires valid:      \valid_read(a + (0..n-1));
  requires increasing: Increasing(a, n);
  assigns              \nothing;
  ensures  result:     \result <==> \exists integer i; 0 <= i < n && a[i] == v;
*/
bool
binary_search(const value_type* a, size_type n, value_type v);
```

Listing 6.16: Formal specification of `binary_search`

Note that instead of the somewhat lengthy existential quantification of `binary_search` [6.16] we can use our previously introduced predicate `SomeEqual` [4.4] in order to achieve the following more concise formal specification `binary_search2` [6.17].

[20] To be more precise: The C++ Standard Library requires that `(a[i] <= v) && (v <= a[i])` holds. For our definition of `value_type` (see §2.3) this means that `v` equals `a[i]`.

```
/*@
  requires valid:       \valid_read(a + (0..n-1));
  requires increasing: Increasing(a, n);
  assigns              \nothing;
  ensures  result:      \result <==> SomeEqual(a, n, v);
*/
bool
binary_search2(const value_type* a, size_type n, value_type v);
```

Listing 6.17: Formal specification of `binary_search2`

It is interesting to compare the specification of `binary_search` [6.16] with that of `find2` [4.5]. Both algorithms allow to determine whether a value is contained in an array. The fact that the C++ Standard Library requires that `find` has *linear* complexity whereas `binary_search` must have a *logarithmic* complexity can currently not be expressed with ACSL.

6.4.2. Implementation of `binary_search`

Our implementation `binary_search2` [6.18] first calls `lower_bound` [6.5]. Remember that if the latter returns an index `0 <= i < n`, then we can be sure that `v <= a[i]` holds.

```
bool
binary_search2(const value_type* a, size_type n, value_type v)
{
  const size_type i = lower_bound(a, n, v);
  return (i < n) && (a[i] <= v);
}
```

Listing 6.18: Implementation of `binary_search2`

Part III.

Mutating and numeric algorithms

7. Mutating algorithms

Let us now turn our attention to another class of algorithms, viz. *mutating* algorithms of the C++ Standard Library [20, §28.6], i.e., algorithms that change one or more ranges. In Frama-C, you can explicitly specify that, e.g., entries in an array `a` may be modified by a function `f`, by including the following *assigns clause* into the `f`'s specification:

```
assigns a[0..length-1];
```

The expression `length-1` refers to the value of `length` when `f` is entered, see [15, §2.3.2]. Below are the algorithms we will discuss in this chapter.

- In order to allow for a finer control of which parts of an array, we introduce in §7.1 the auxiliary predicate `Unchanged`.
- `fill` in §7.2 initializes each element of an array by a given fixed value.
- `swap` in §7.3 exchanges two values.
- `swap_ranges` in §7.4 exchanges the contents of the arrays of equal length, element by element. We use this example to present "modular verification", as `swap_ranges` reuses the verified properties of `swap`.
- `copy` in §7.5 copies a source array to a destination array.
- `copy_backward` in §7.6 also copies a source array to a destination array. This version, however, uses another separation condition than `copy`.
- `reverse_copy` and `reverse` in §7.7 and §7.8, respectively, reverse an array. Whereas `reverse_copy` copies the result to a separate destination array, the `reverse` algorithm works in place.
- `rotate_copy` in §7.9 rotates a source array by `m` positions and copies the results to a destination array.
- `rotate` in §7.10 rotates *in place* a source array by `m` positions.
- `replace_copy` and `replace` in §7.11 and §7.12, respectively, substitute each occurrence of a value by a given new value. Whereas `replace_copy` copies the result to a separate array, the `replace` algorithm works in place.
- `remove_copy` and `remove` in §7.13–§7.16 *filter* all occurrences of a given value from an array. Whereas `remove_copy` copies the result to a separate array, the `remove` algorithm works in place. Note that we provide altogether three versions of how to specify `remove_copy`. This shall help the reader to understand that finding appropriate contracts is an iterative process and that it is usually a good idea to *not* strive for a "complete" contract right from the beginning.
- `shuffle` in §7.17 randomly reorders the elements of an array thereby relying on the simple random number generator `random_number` in §7.18.

7.1. The predicate `Unchanged`

Many of the algorithms in this section iterate sequentially over one or several sequences. For the verification of such algorithms it is often important to express that a section of an array, or the complete array, have remained *unchanged*. As this cannot always be expressed by an `assigns` clause, we introduce in the following listing the overloaded predicate `Unchanged` [7.1]. The expression `Unchanged{K,L}(a,m,n)` is true if the range `a[m..n-1]` in state `K` is element-wise equal to that range in state `L`.

```
/*@
  axiomatic Unchanged
  {
    predicate
    Unchanged{K,L}(value_type* a, integer m, integer n) =
      \forall integer i; m <= i < n ==>  \at(a[i],K) == \at(a[i],L);

    predicate
    Unchanged{K,L}(value_type* a, integer n) = Unchanged{K,L}(a, 0, n);
  }
*/
```

Listing 7.1: The logic definition(s) `Unchanged`

In some situations we use the predicate `ArrayUpdate`, which relies on the predicate `Unchanged` and the the logic function `At` [7.49], to concisely describe which parts of an array have changed or remained unchanged when updating an individual array element.

```
/*@
  axiomatic ArrayUpdate
  {
    predicate
      ArrayUpdate{K,L}(value_type* a, integer n, integer i, value_type v) =
        0 <= i < n                  &&
        Unchanged{K,L}(a, 0, i)     &&
        Unchanged{K,L}(a, i+1, n)   &&
        At{K}(a, i) != v            &&
        At{L}(a, i) == v;

    lemma ArrayUpdate_Shrink{K,L}:
      \forall value_type *a, v, integer n, i;
        0 <= i < n-1                    ==>
        ArrayUpdate{K,L}(a, n, i, v)    ==>
        ArrayUpdate{K,L}(a, n-1, i, v);

    lemma ArrayUpdate_UpperBound{K,L}:
      \forall value_type *a, v, w, integer n, i;
        ArrayUpdate{K,L}(a, n, i, v)    ==>
        v <= w                          ==>
        UpperBound{K}(a, n, w)          ==>
        UpperBound{L}(a, n, w);
  }
*/
```

Listing 7.2: The logic definition(s) `ArrayUpdate`

In the following listing we show a few lemmas for `Unchanged` [7.1] that we need for the verification of various algorithms.

```
/*@
  axiomatic UnchangedLemmas
  {
    lemma Unchanged_Shrink{K,L}:
      \forall value_type *a, integer m, n, p, q;
         m <= p <= q <= n          ==>
         Unchanged{K,L}(a, m, n)   ==>
         Unchanged{K,L}(a, p, q);

    lemma Unchanged_Extend{K,L}:
      \forall value_type *a, integer n;
        Unchanged{K,L}(a, n)          ==>
        \at(a[n],K) == \at(a[n],L)   ==>
        Unchanged{K,L}(a, n+1);

    lemma Unchanged_Shift{K,L}:
      \forall value_type *a, integer p, q, r;
        Unchanged{K,L}(a+p, q, r)   ==>  Unchanged{K,L}(a, p+q, p+r);

    lemma Unchanged_Symmetric{K,L}:
      \forall value_type *a, integer m, n;
        Unchanged{K,L}(a, m, n)   ==>
        Unchanged{L,K}(a, m, n);

    lemma Unchanged_Transitive{K,L,M}:
      \forall value_type *a, integer m, n;
        Unchanged{K,L}(a, m, n)   ==>
        Unchanged{L,M}(a, m, n)   ==>
        Unchanged{K,M}(a, m, n);
  }
*/
```

Listing 7.3: The logic definition(s) `UnchangedLemmas`

- Lemma `Unchanged_Shrink` [7.3] states that if the range `a[m..n-1]` does not change when going from state `K` to state `L`, then `a[p..q-1]` does not change either, provided the latter is a subrange of the former, i.e. provided $0 \le m \le p \le q \le n$ holds.
- Lemma `Unchanged_Extend` [7.3] expresses the simple fact that "unchangedness" is an inductive property.
- Lemma `Unchanged_Shift` [7.3] states how `Unchanged` behaves under pointer additions.
- Lemmas `Unchanged_Symmetric` [7.3] and `Unchanged_Transitive` [7.3] express respectively the symmetry and transitivity of `Unchanged` with respect to program states.

7.2. The `fill` algorithm

The `fill` algorithm in the C++ Standard Library [20, §28.6.6] initializes general sequences with a particular value. The signature of our modified variant reads:

```
void fill(value_type* a, size_type n, value_type v);
```

7.2.1. Formal specification of `fill`

The following listing shows the formal specification of `fill` [7.4]. We can express the postcondition of `fill` simply by using the overloaded predicate `AllEqual` [4.15].

```
/*@
  requires valid:   \valid(a + (0..n-1));
  assigns           a[0..n-1];
  ensures constant: AllEqual(a, n, v);
*/
void
fill(value_type* a, size_type n, value_type v);
```

Listing 7.4: Formal specification of `fill`

The `assigns`-clauses formalize that `fill` modifies only the entries of the range `a[0..n-1]`. In general, when more than one *assigns clause* appears in a function's specification, it is permitted to modify any of the referenced memory locations. However, if no *assigns clause* appears at all, the function is free to modify any memory location, see [15, §2.3.2]. To forbid a function to do any modifications outside its scope, a clause `assigns \nothing;` must be used, as we practised in the example specifications in Chapter 4.

7.2.2. Implementation of `fill`

The implementation of `fill` [7.5] comes with the loop invariant `constant` expresses that for each iteration the array is *filled* with the value of `v` up to the index `i` of the iteration. Note that we use here again the predicate `AllEqual` [4.15].

```
void
fill(value_type* a, size_type n,  value_type v)
{
  /*@
    loop invariant bound:    0 <= i <= n;
    loop invariant constant: AllEqual(a, i, v);
    loop assigns i, a[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0u; i < n; ++i) {
    a[i] = v;
  }
}
```

Listing 7.5: Implementation of `fill`

7.3. The swap algorithm

The `swap` algorithm [20, §28.6.3] in the C++ Standard Library exchanges the contents of two variables. Similarly, the `iter_swap` algorithm [20, §28.6.3] exchanges the contents referenced by two pointers. Since C and hence ACSL, does not support an `&` type constructor ("declarator"), we will present an algorithm that processes pointers and refer to it as `swap`.

7.3.1. Formal specification of swap

The contract of `swap` [7.6] is shown in the following listing. The preconditions state that both pointer arguments of `swap` must be dereferenceable.

```
/*@
  requires valid:    \valid(p);
  requires valid:    \valid(q);
  assigns            *p;
  assigns            *q;
  ensures  exchange: *p == \old(*q);
  ensures  exchange: *q == \old(*p);
*/
void
swap(value_type* p, value_type* q);
```

Listing 7.6: Formal specification of `swap`

Upon termination of `swap` the entries must be mutually exchanged. The expression `\old(*p)` refers to the value of `*p` before `swap` has be called. By default, a postcondition refers the values after the functions has been terminated.

7.3.2. Implementation of swap

The following listing shows the straight-forward implementation of `swap` [7.7]. No interspersed ACSL annotations are needed achieve a verification by Frama-C/WP.

```
void
swap(value_type* p, value_type* q)
{
  value_type save = *p;
  *p = *q;
  *q = save;
}
```

Listing 7.7: Implementation of `swap`

7.4. The swap_ranges algorithm

The swap_ranges algorithm in the C++ Standard Library [20, §28.6.3] exchanges the contents of two expressed ranges element-wise. After translating C++ reference types and iterators to C, our version of the original signature reads:

```
void swap_ranges(value_type* a, size_type n, value_type* b);
```

We do not return a value since it would equal n, anyway.

7.4.1. Formal specification of swap_ranges

The following listing shows a specification for the swap_ranges [7.8] algorithm.

```
/*@
  requires valid:   \valid(a + (0..n-1));
  requires valid:   \valid(b + (0..n-1));
  requires sep:     \separated(a+(0..n-1), b+(0..n-1));
  assigns           a[0..n-1];
  assigns           b[0..n-1];
  ensures equal:    Equal{Old,Here}(a, n, b);
  ensures equal:    Equal{Old,Here}(b, n, a);
*/
void
swap_ranges(value_type* a, size_type n, value_type* b);
```

Listing 7.8: Formal specification of swap_ranges

The swap_ranges algorithm works correctly only if a and b do not overlap. Because of that fact we use the clause sep to tell Frama-C that a and b must not overlap.

With the assigns-clause we postulate that the swap_ranges algorithm alters the elements contained in two distinct ranges, modifying the corresponding elements and nothing else.

The postconditions of swap_ranges specify that the content of each element in its post-state must equal the pre-state of its counterpart. We can use the predicate Equal [4.28] together with the label Old and Here to express the postcondition of swap_ranges. In our specification, for example, we specify that the array a in the memory state that corresponds to the label Here is equal to the array b at the label Old. Since we are specifying a postcondition Here refers to the post-state of swap_ranges whereas Old refers to the pre-state.

7.4.2. Implementation of swap_ranges

The implementation of swap_ranges [7.9] together with the necessary loop annotations is shown in the following listing. Unsurprisingly, we are repeatedly calling swap [7.6].

```
void
swap_ranges(value_type* a, size_type n, value_type* b)
{
  /*@
    loop invariant bound:  0 <= i <= n;
    loop invariant equal:  Equal{Pre,Here}(a, i, b);
    loop invariant equal:  Equal{Pre,Here}(b, i, a);

    loop invariant unchanged:  Unchanged{Pre,Here}(a, i, n);
    loop invariant unchanged:  Unchanged{Pre,Here}(b, i, n);

    loop assigns i, a[0..n-1], b[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0u; i < n; ++i) {
    swap(a + i, b + i);
  }
}
```

Listing 7.9: Implementation of swap_ranges

For the postcondition swap_ranges [7.8] to hold, our loop invariants must ensure that at each iteration all of the corresponding elements that have already been visited are swapped.

Note that there are two additional loop invariants which claim that all the elements that have not visited yet equal their original values. This annotation allows us to prove the postconditions of swap_ranges despite the fact that the loop assigns is coarser than it should be. The predicate Unchanged [7.1] is used to express this property.

7.5. The copy algorithm

The `copy` algorithm in the C++ Standard Library [20, §28.6.1] implements a duplication algorithm for general sequences. For our purposes we have modified the generic implementation to that of a range of type `value_type`. The signature now reads:

```
void copy(const value_type* a, size_type n, value_type* b);
```

Informally, the function copies every element from the source range `a[0..n-1]` to the destination range `b[0..n-1]`, as shown in Figure 7.10.

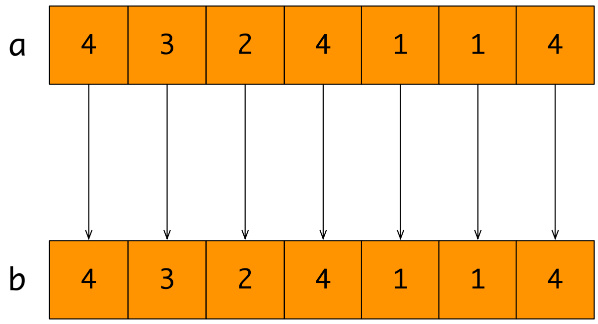

Figure 7.10.: Effects of `copy`

7.5.1. Formal specification of copy

Figure 7.10 might suggest that the ranges `a[0..n-1]` and `b[0..n-1]` must not overlap. However, since the informal specification requires that elements are copied in the order of increasing indices only a weaker condition is necessary. To be more specific, it is required that the pointer `b` does not refer to elements of `a[0..n-1]` as shown in the example in Figure 7.11.

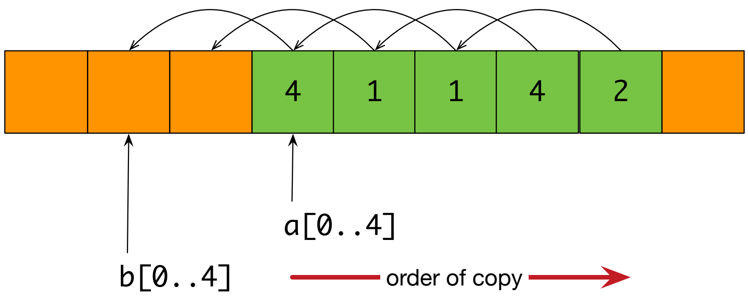

Figure 7.11.: Possible overlap of `copy` ranges

The specification of `copy` is shown in the following listing. The `copy` algorithm expects that the ranges `a` and `b` are valid for reading and writing, respectively. Note the precondition `sep` that expresses the previously discussed non-overlapping property.

```
/*@
  requires valid:  \valid_read(a + (0..n-1));
  requires valid:  \valid(b + (0..n-1));
  requires sep:    \separated(a + (0..n-1), b);
  assigns          b[0..n-1];
  ensures equal:   Equal{Old,Here}(a, n, b);
*/
void
copy(const value_type* a, const size_type n, value_type* b);
```

Listing 7.12: Formal specification of `copy`

Again, we can use the `Equal` [4.28] predicate to express that the array `a` equals `b` after `copy` has been called. Nothing else must be altered. To state this we use the `assigns`-clause.

7.5.2. Implementation of `copy`

The following listing shows an implementation of the `copy` function.

```
void
copy(const value_type* a, size_type n, value_type* b)
{
  /*@
    loop invariant bound:     0 <= i <= n;
    loop invariant equal:     Equal{Pre,Here}(a, i, b);
    loop invariant unchanged: Unchanged{Pre,Here}(a, i, n);
    loop assigns   i, b[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0u; i < n; ++i) {
    b[i] = a[i];
  }
}
```

Listing 7.13: Implementation of `copy`

For the postcondition `equal` to be true, we must ensure that for every index `i`, the value `a[i]` must not yet have been changed before it is copied to `b[i]`. We express this by using the `Unchanged` predicate.[21]

The `assigns` clause ensures that nothing but the range `b[0..n-1]` and the loop variable `i` is modified. Keep in mind, however, that parts of the source range `a[0..n-1]` might change due to its potential overlap with the destination range.

[21] Alternatively, this could also be expressed by changing the `loop assigns` clause to `i, b[0..i-1]`; however, Frama-C doesn't yet support `loop assigns` clauses containing the loop variable.

7.6. The copy_backward algorithm

The copy_backward algorithm in the C++ Standard Library [20, §28.6.1] implements another duplication algorithm for general sequences. For our purposes we have modified the generic implementation to that of a range of type value_type. The signature now reads:

```
void copy_backward(const value_type* a, size_type n, value_type* b);
```

The main reason for the existence of copy_backward is to allow copying when the start of the destination range a[0..n-1] is contained in the source range b[0..n-1]. In this case, copy can't be employed since its precondition sep is violated, as can be seen in the contract of copy [7.12].

The informal specification of copy_backward states that copying starts at the end of the source range. For this to work, however, the pointer b+n must not be contained in the source range. Note that the order of operation (or procedure) calls cannot be specified in ACSL.[22] A similar remark about order of operations tacitly applied to earlier functions as well, e.g. to copy, where the C++ order was prescribed by confining the signature to a ForwardIterator.

Figure 7.14 gives an example where copy_backward, but *not* copy, can be applied.

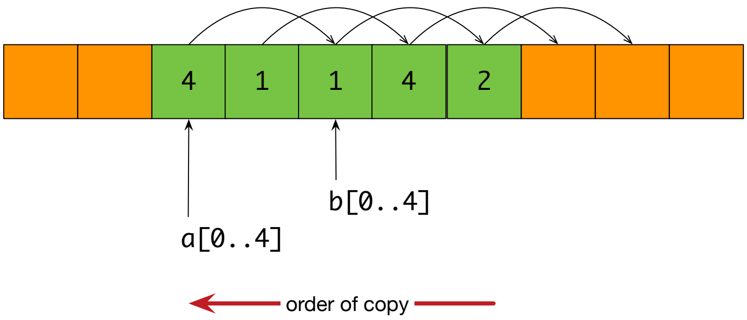

Figure 7.14.: Possible overlap of copy_backward ranges

Note that in the original signature the argument b refers to one past the end of the destination range. Here, however, it refers to its start. The reason for this change is that in C++ copy_backward is defined for *bidirectional iterators* which do not provide random access operations such as adding or subtracting an index. Since our C version works on pointers we do not consider it as necessary to use the one past the end pointer.

7.6.1. Formal specification of copy_backward

The specification of copy_backward is shown in the following listing. The copy_backward algorithm expects that the ranges a[0..n-1] and b[0..n-1] are valid for reading and writing, respectively. Precondition sep formalizes the constraints on the overlap of the source and destination ranges as discussed at the beginning of this section.

[22] The Aoraï specification language and the corresponding Frama-C plugin are provided to specify and verify temporal properties of code; however, they are beyond the scope of this tutorial.

```
/*@
  requires valid:  \valid_read(a + (0..n-1));
  requires valid:  \valid(b + (0..n-1));
  requires sep:    \separated(a + (0..n-1), b + n);
  assigns          b[0..n-1];
  ensures equal:   Equal{Old,Here}(a, n, b);
*/
void
copy_backward(const value_type* a, size_type n, value_type* b);
```

Listing 7.15: Formal specification of `copy_backward`

The function `copy_backward` assigns the elements from the source range `a` to the destination range `b`, modifying the memory of the elements pointed to by `b`. Again, we can use the `Equal` [4.28] predicate to express that the array `a` equals `b` after `copy_backward` has been called.

7.6.2. Implementation of `copy_backward`

The following listing shows an implementation of the `copy_backward` function.

```
void
copy_backward(const value_type* a, size_type n, value_type* b)
{
  /*@
    loop invariant bound:     0 <= i <= n;
    loop invariant equal:     Equal{Pre,Here}(a, i, n, b);
    loop invariant unchanged: Unchanged{Pre,Here}(a, i);
    loop assigns i, b[0..n-1];
    loop variant i;
   */
  for (size_type i = n; i > 0u; --i) {
    b[i - 1u] = a[i - 1u];
  }
}
```

Listing 7.16: Implementation of `copy_backward`

We have loop invariants similar to `copy`, stating the loop variable's range (`bound`) and the area that has already been copied in each cycle (`equal`).

7.7. The `reverse_copy` algorithm

The `reverse_copy` algorithm of the C++ Standard Library [20, §28.6.10] inverts the order of elements in a sequence. `reverse_copy` does not change the input sequence, and copies its result to the output sequence. For our purposes we have modified the generic implementation to that of a range of type `value_type`. The signature now reads:

```
void reverse_copy(const value_type* a, size_type n, value_type* b);
```

Informally, `reverse_copy` copies the elements from the array `a` into array `b` such that the copy is a reverse of the original array. In order to concisely formalize these conditions we define in the following listing the predicate `Reverse` [7.17] (see also Figure 7.18).

```
/*@
  axiomatic Reverse
  {
    predicate
    Reverse{K,L}(value_type* a, integer n, value_type* b) =
      \forall integer i; 0 <= i < n  ==>  \at(a[i],K) == \at(b[n-1-i], L);

    predicate
    Reverse{K,L}(value_type* a, integer m, integer n,
                 value_type* b, integer p) = Reverse{K,L}(a+m, n-m, b+p);

    predicate
    Reverse{K,L}(value_type* a, integer m, integer n, value_type* b) =
      Reverse{K,L}(a, m, n, b, m);

    predicate
    Reverse{K,L}(value_type* a, integer m, integer n, integer p) =
      Reverse{K,L}(a, m, n, a, p);

    predicate
    Reverse{K,L}(value_type* a, integer m, integer n) =
      Reverse{K,L}(a, m, n, m);

    predicate
    Reverse{K,L}(value_type* a, integer n) = Reverse{K,L}(a, 0, n);
  }
*/
```

Listing 7.17: The logic definition(s) `Reverse`

We also define several overloaded variants of `Reverse` that provide default values for some of the parameters. These overloaded versions enable us to write later more concise ACSL annotations.

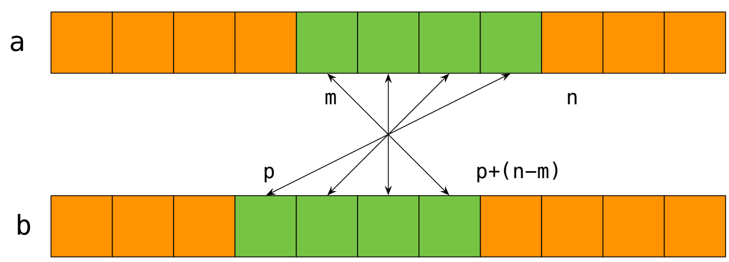

Figure 7.18.: Sketch of predicate `Reverse`

7.7.1. Formal specification of reverse_copy

The specification of reverse_copy [7.19] is shown in the following listing We use the second version of predicate Reverse [7.17] in order to formulate the postcondition of reverse_copy.

```
/*@
  requires valid:      \valid_read(a + (0..n-1));
  requires valid:      \valid(b + (0..n-1));
  requires sep:        \separated(a + (0..n-1), b + (0..n-1));
  assigns              b[0..(n-1)];
  ensures reverse:     Reverse{Old,Here}(a, n, b);
  ensures unchanged:   Unchanged{Old,Here}(a, n);
*/
void
reverse_copy(const value_type* a, size_type n, value_type* b);
```

Listing 7.19: Formal specification of reverse_copy

7.7.2. Implementation of reverse_copy

The implementation of reverse_copy [7.20] is shown in the following listing. For the postcondition to be true, we must ensure that for every element i, the comparison b[i] == a[n-1-i] holds. This is formalized by the loop invariant reverse where we employ the first version of Reverse [7.17].

```
void
reverse_copy(const value_type* a, size_type n, value_type* b)
{
  /*@
    loop invariant bound:   0 <= i <= n;
    loop invariant reverse: Reverse{Here,Pre}(b, 0, i, a, n-i);
    loop assigns i, b[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0u; i < n; ++i) {
    b[i] = a[n - 1u - i];
  }
}
```

Listing 7.20: Implementation of reverse_copy

7.8. The `reverse` algorithm

The `reverse` algorithm of the C++ Standard Library [20, §28.6.10] inverts the order of elements *within* a sequence. The signature of our version of `reverse` reads.

```
void reverse(value_type* a, size_type n);
```

7.8.1. Formal specification of `reverse`

The specification for the `reverse` [7.21] function is shown in the following listing.

```
/*@
  requires valid:  \valid(a + (0..n-1));
  assigns          a[0..n-1];
  ensures reverse: Reverse{Old,Here}(a, n);
*/
void
reverse(value_type* a, size_type n);
```

Listing 7.21: Formal specification of `reverse`

7.8.2. Implementation of `reverse`

Since the implementation of `reverse` [7.22] operates *in place* we use `swap` [7.6] in order to exchange the elements of the first half of the array with the corresponding elements of the second half. We reuse the predicates `Reverse` [7.17] and `Unchanged` [7.1] in order to write concise loop invariants.

```
void
reverse(value_type* a, size_type n)
{
  const size_type half = n / 2u;

  //@ assert half: half <= n - half;
  //@ assert half: 2*half <= n <= 2*half + 1;
  /*@
    loop invariant bound:    0 <= i <= half <= n-i;
    loop invariant left:     Reverse{Pre,Here}(a, 0, i, n-i);
    loop invariant middle: Unchanged{Pre,Here}(a, i, n-i);
    loop invariant right:    Reverse{Pre,Here}(a, n-i, n, 0);
    loop assigns i, a[0..n-1];
    loop variant half - i;
  */
  for (size_type i = 0u; i < half; ++i) {
    swap(&a[i], &a[n - 1u - i]);
  }
}
```

Listing 7.22: Implementation of `reverse`

7.9. The `rotate_copy` algorithm

The `rotate_copy` algorithm of the C++ Standard Library [20, §28.6.11] copies, in a particular way, the elements of one sequence of length n into a separate sequence. More precisely,

- the first m elements of the first sequence become the last m elements of the second sequence, and
- the last $n - m$ elements of the first sequence become the first $n - m$ elements of the second sequence.

Figure 7.23 illustrates the effects of `rotate_copy` by highlighting how the initial and final segments of the array `a[0..n-1]` are mapped to corresponding subranges of the array `b[0..n-1]`.

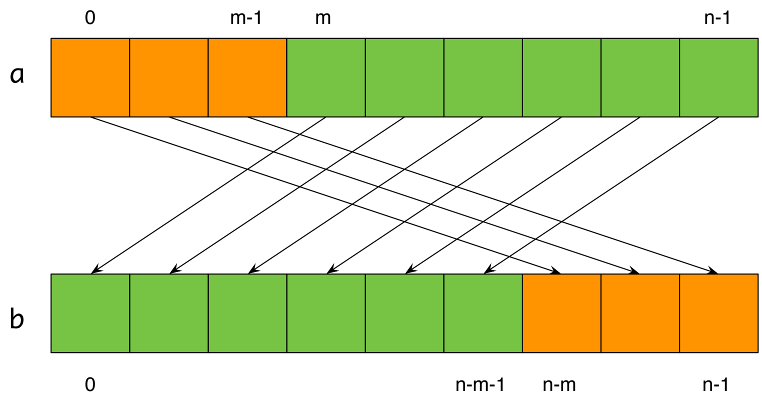

Figure 7.23.: Effects of `rotate_copy`

For our purposes we have modified the generic implementation to that of a range of type `value_type`. The signature now reads:

```
void rotate_copy(const value_type* a, size_type m, size_type n, value_type* b);
```

7.9.1. Formal specification of `rotate_copy`

The specification of `rotate_copy` is shown in the following listing. Note that we require explicitly that both ranges do not overlap and that we are only able to *read* from the range `a[0..n-1]`.

```
/*@
  requires bound:      0 <= m <= n;
  requires valid:      \valid_read(a + (0..n-1));
  requires valid:      \valid(b + (0..n-1));
  requires sep:        \separated(a + (0..n-1), b + (0..n-1));
  assigns              b[0..(n-1)];
  ensures left:        Equal{Old,Here}(a, 0, m,   b, n-m);
  ensures right:       Equal{Old,Here}(a, m, n-m, b, 0);
  ensures unchanged:   Unchanged{Old,Here}(a, n);
*/
void
rotate_copy(const value_type* a, size_type m, size_type n, value_type* b);
```

Listing 7.24: Formal specification of `rotate_copy`

7.9.2. Implementation of `rotate_copy`

The following listing shows an implementation of the `rotate_copy` function. The implementation simply calls the function `copy` twice.

```
void
rotate_copy(const value_type* a, size_type m, size_type n, value_type* b)
{
  copy(a,   m, b + (n - m));
  copy(a + m, n - m, b);
}
```

Listing 7.25: Implementation of `rotate_copy`

7.10. The `rotate` algorithm

The algorithm `rotate` is an *in-place* variant of the algorithm `rotate_copy` [7.24]. We have modified the generic specification of `rotate` [20, §28.6.11] such that it refers to a range of objects of `value_type`. The signature now reads:

```
size_type rotate(const value_type* a, size_type m, size_type n);
```

7.10.1. Formal specification of `rotate`

Figure 7.26 shows informally the behavior of `rotate`. The figure is of course very similar to the one for `rotate_copy` (see Figure 7.23). The notable difference is that `rotate` operates *in place* of the array `a[0..n-1]`.

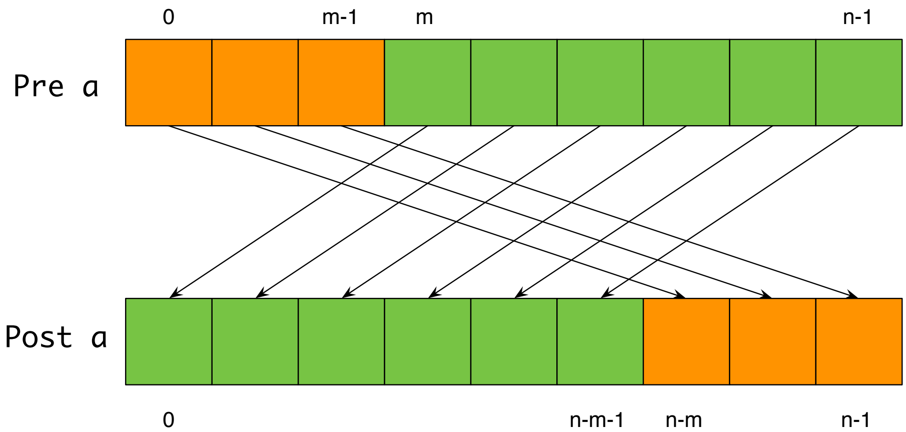

Figure 7.26.: Effects of `rotate`

The specification of `rotate` is shown in the following listing.

```
/*@
  requires valid: \valid(a + (0..n-1));
  requires bound: m <= n;
  assigns         a[0..n-1];
  ensures result: \result == n-m;
  ensures left:   Equal{Old,Here}(a, 0, m, n-m);
  ensures right:  Equal{Old,Here}(a, m, n, 0);
*/
size_type
rotate(value_type* a, size_type m, size_type n);
```

Listing 7.27: Formal specification of `rotate`

7.10.2. Implementation of `rotate`

The following listing shows an implementation of the `rotate` function together with several ACSL annotations. Actually, there are several ways to implement `rotate`. We have chosen a particularly simple one that is derived from an implementation of `std::rotate` for *bidirectional iterators* [20, §27.2.6] and which essentially consists of several calls to the algorithm `reverse` [7.21].

Note the statement contract of the final call of `reverse` [7.21]. Here we use both the labels `Pre` and `Old` which refer to the pre-states of `reverse` and the function `rotate` itself, respectively.

```
size_type
rotate(value_type* a, size_type m, size_type n)
{
  // if one subrange is empty, then nothings needs to be done
  if ((0u < m) && (m < n)) {
    reverse(a,  m);
    reverse(a + m, n - m);
    /*@
      requires left:   Reverse{Pre,Here}(a, 0, m, 0);
      requires right:  Reverse{Pre,Here}(a, m, n, m);

      assigns          a[0..n-1];

      ensures left:    Reverse{Old,Here}(a, 0, m, n-m);
      ensures right:   Reverse{Old,Here}(a, m, n, 0);
    */
    reverse(a, n);
    //@ assert left:   Equal{Pre,Here}(a, 0, m, n-m);
    //@ assert right:  Equal{Pre,Here}(a, m, n, 0);
  }

  return n - m;
}
```

Listing 7.28: Implementation of `rotate`

7.11. The `replace_copy` algorithm

The `replace_copy` algorithm of the C++ Standard Library [20, §28.6.5] substitutes specific elements from general sequences. Here, the general implementation has been altered to process `value_type` ranges. The new signature reads:

```
size_type replace_copy(const value_type* a, size_type n, value_type* b,
                       value_type v, value_type w);
```

The `replace_copy` algorithm copies the elements from the range `a[0..n]` to range `b[0..n]`, substituting every occurrence of `v` by `w`. The return value is the length of the range. As the length of the range is already a parameter of the function this return value does not contain new information.

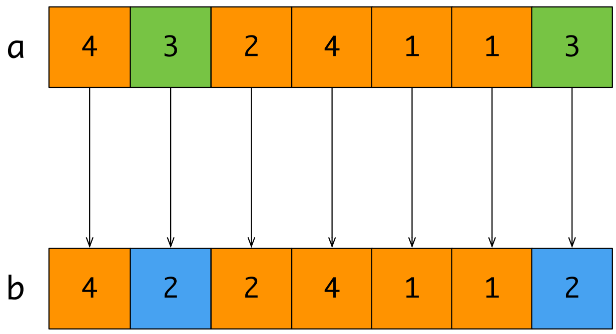

Figure 7.29.: Effects of `replace`

Figure 7.29 shows the behavior of `replace_copy` at hand of an example where all occurrences of the value 3 in `a[0..n-1]` are replaced with the value 2 in `b[0..n-1]`.

7.11.1. The predicate `Replace`

We start with defining in the following listing the predicate `Replace` [7.30] that describes the intended relationship between the input array `a[0..n-1]` and the output array `b[0..n-1]`. Note the introduction of *local bindings* `\let ai = ...` and `\let bi = ...` in the definition of `Replace` (see [15, §2.2]).

```
/*@
  axiomatic Replace
  {
    predicate
    Replace{K,L}(value_type* a, integer n, value_type* b,
                 value_type v, value_type w) =
      \forall integer i; 0 <= i < n  ==>
        \let ai = \at(a[i],K);
        \let bi = \at(b[i],L);
        (ai == v  ==>  bi == w) && (ai != v  ==>  bi == ai) ;

    predicate
    Replace{K,L}(value_type* a, integer n, value_type v, value_type w) =
      Replace{K,L}(a, n, a, v, w);
  }
*/
```

Listing 7.30: The logic definition(s) `Replace`

This listing also contains a second, overloaded version of `Replace` which we will use for the specification of the related in-place algorithm `replace` [7.33].

7.11.2. Formal specification of `replace_copy`

Using predicate `Replace` the specification of `replace_copy` [7.31] is as simple as shown in the following listing. Note that we also require that the input range `a[0..n-1]` and output range `b[0..n-1]` do not overlap.

```
/*@
  requires valid:     \valid_read(a + (0..n-1));
  requires valid:     \valid(b + (0..n-1));
  requires sep:       \separated(a + (0..n-1), b + (0..n-1));
  assigns             b[0..n-1];
  ensures result:     \result == n;
  ensures replace:    Replace{Old,Here}(a, n, b, v, w);
  ensures unchanged: Unchanged{Old,Here}(a, n);
*/
size_type
replace_copy(const value_type* a, size_type n, value_type* b,
             value_type v, value_type w);
```

Listing 7.31: Formal specification of `replace_copy`

7.11.3. Implementation of `replace_copy`

The implementation (including loop annotations) of `replace_copy` [7.32] is shown in the following listing. Note how the structure of the loop annotations resembles the specification of `replace_copy` [7.31].

```
size_type
replace_copy(const value_type* a, size_type n, value_type* b, value_type v,
             value_type w)
{
  /*@
    loop invariant bounds:     0 <= i <= n;
    loop invariant replace:    Replace{Pre,Here}(a, i, b, v, w);
    loop assigns i, b[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0u; i < n; ++i) {
    b[i] = (a[i] == v ? w : a[i]);
  }

  return n;
}
```

Listing 7.32: Implementation of `replace_copy`

7.12. The `replace` algorithm

The `replace` algorithm of the C++ Standard Library [20, §28.6.5] substitutes specific values in a general sequence. Here, the general implementation has been altered to process `value_type` ranges. The new signature reads

```
void replace(value_type* a, size_type n, value_type v, value_type w);
```

The `replace` algorithm substitutes all elements from the range `a[0..n-1]` that equal `v` by `w`.

7.12.1. Formal specification of `replace`

Using the second predicate `Replace` [7.30] the specification of `replace` [7.33] can be expressed as in the following listing.

```
/*@
  requires valid:   \valid(a + (0..n-1));
  assigns           a[0..n-1];
  ensures replace:  Replace{Old,Here}(a, n, v, w);
*/
void
replace(value_type* a, size_type n, value_type v, value_type w);
```

Listing 7.33: Formal specification of `replace`

7.12.2. Implementation of `replace`

The implementation of `replace` [7.34] is shown in the following listing. The loop invariant `unchanged` expresses that when entering iteration `i` the elements `a[i..n-1]` have not yet changed.

```
void
replace(value_type* a, size_type n, value_type v, value_type w)
{
  /*@
    loop invariant bounds:    0 <= i <= n;
    loop invariant replace:   Replace{Pre,Here}(a, i, v, w);
    loop invariant unchanged: Unchanged{Pre,Here}(a, i, n);
    loop assigns i, a[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0u; i < n; ++i) {
    if (a[i] == v) {
      a[i] = w;
    }
  }
}
```

Listing 7.34: Implementation of `replace`

7.13. The `remove_copy` algorithm (basic contract)

The `remove_copy` algorithm of the C++ Standard Library [20, §28.6.8] copies all elements of a sequence other than a given value. Here, the general implementation has been altered to process `value_type` ranges. The new signature reads:

```
size_type
remove_copy(const value_type* a, size_type n, value_type* b, value_type v);
```

The requirements of `remove_copy` are:

Requirements	Description
Remove Copy Size	The output range has to fit in all the elements of the input range, except the ones that equal the value `v` by `remove_copy`.
Remove Copy Separated	The input range and the output range do not overlap
Remove Copy Elements	The `remove_copy` algorithm copies elements that are not equal to `v` from range `a[0..n-1]` to the range `b[0..\result-1]`.
Remove Copy Stability	The algorithm is stable, that is, the relative order of the elements in `b` is the same as in `a`.
Remove Copy Return	The return value is the length of the resulting range.
Remove Copy Complexity	The algorithm takes *n* comparisons in every case.

Table 7.35.: Properties of `remove_copy`

Figure 7.36 shows an example of how `remove_copy` is supposed to copy elements that differ from `4` from the input range to the output range.

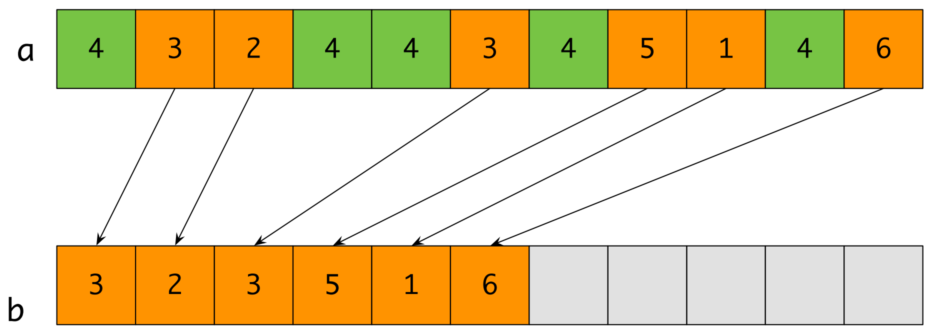

Figure 7.36.: Effects of `remove_copy`

7.13.1. Formal specification of `remove_copy`

The following listing shows our first attempt to specify `remove_copy`. In postcondition `discard` we use of the predicate `NoneEqual` [4.4] to show that the value `v` does not occur in the range `b[0..\result]`.

```
/*@
  requires valid:    \valid_read(a + (0..n-1));
  requires valid:    \valid(b + (0..n-1));
  requires sep:      \separated(a + (0..n-1), b + (0..n-1));
  assigns            b[0..n-1];
  ensures bound:     0 <= \result <= n;
  ensures discard:   NoneEqual(b, \result, v);
  ensures unchanged: Unchanged{Old,Here}(a, n);
  ensures unchanged: Unchanged{Old,Here}(b, \result, n);
*/
size_type
remove_copy(const value_type *a, size_type n, value_type *b, value_type v);
```

Listing 7.37: Formal specification of `remove_copy`

One shortcoming of this specification is that the postcondition `bound` only makes very general and not very precise statements about the number of copied elements. We will address this problem in the contract of `remove_copy2` [7.41]. A more serious shortcoming is, however, that we haven't specified what the relationship between the elements of the input range `a[0..n-1]` and the output range `b[0..\result -1]` looks like. This problem will be tackled in the contract of `remove_copy3` [7.48].

7.13.2. Implementation of `remove_copy`

An implementation of `remove_copy` is shown in the following listing.

```
size_type
remove_copy(const value_type *a, size_type n, value_type *b, value_type v)
{
  size_type k = 0u;

  /*@
    loop invariant bound:     0 <= k <= i <= n;
    loop invariant discard:   NoneEqual(b, k, v);
    loop invariant unchanged: Unchanged{Pre,Here}(b, k, n);
    loop assigns   k, i, b[0..n-1];
    loop variant   n-i;
  */
  for (size_type i = 0u; i < n; ++i) {
    if (a[i] != v) {
      b[k++] = a[i];
    }
  }

  return k;
}
```

Listing 7.38: Implementation of `remove_copy`

Here we also need to add another loop invariant `discard` which basically checks if `v` occurs in `b[0..k]` for each iteration of the loop.

7.14. The `remove_copy2` algorithm (number of copied elements)

In this section we improve the contract of `remove_copy` [7.37] by formally specifying the number `\result` of elements copied by `remove_copy`.

The number of copied elements equals of course the number of elements in the input range `a[0..n-1]` that are different from `v`. One can formally describe this number by relying on the logic function `Count` [4.44].

```
logic integer
CountNotEqual(value_type* a, integer n, value_type v) =  n - Count(a, n, v);
```

In fact, we have used this kind of definition in earlier version of this document. We have found it, however, worthwhile to provide a separate definition of `CountNotEqual` and express the relationship with `Count` as a lemma. This definition is shown in the Listings 7.39 and 7.40.

```
/*@
  axiomatic CountNotEqual
  {
    logic integer
    CountNotEqual(value_type* a, integer m, integer n, value_type v) =
      n <= m ? 0 : CountNotEqual(a, m, n-1, v) + (a[n-1] == v ? 0 : 1);

    logic integer
    CountNotEqual(value_type* a, integer n, value_type v) =
      CountNotEqual(a, 0, n, v);

    lemma CountNotEqual_Empty:
      \forall value_type *a, v, integer m, n;
        n <= m  ==>  CountNotEqual(a, m, n, v) == 0;

    lemma CountNotEqual_Hit:
      \forall value_type *a, v, integer m, n;
        m <= n      ==>
        a[n] != v   ==>
        CountNotEqual(a, m, n+1, v) == CountNotEqual(a, m, n, v) + 1;

    lemma CountNotEqual_Miss:
      \forall value_type *a, v, integer m, n;
        m <= n      ==>
        a[n] == v   ==>
        CountNotEqual(a, m, n+1, v) == CountNotEqual(a, m, n, v);

    lemma CountNotEqual_Lower:
      \forall value_type *a, v, integer m, n;
        m <= n  ==>  0 <= CountNotEqual(a, m, n, v);

    lemma CountNotEqual_Upper:
      \forall value_type *a, v, integer m, n;
        m <= n  ==>  CountNotEqual(a, m, n, v) <= n-m;
```

Listing 7.39: The logic function `CountNotEqual` (1)

The above mentioned relationship with `Count` [4.44] is expressed as lemma `CountNotEqual_Count` [7.39] in the following listing.

```
    lemma CountNotEqual_WeaklyIncreasing:
      \forall value_type *a, v, integer m, n;
        m <= n  ==>  CountNotEqual(a, m, n, v) <= CountNotEqual(a, m, n+1, v);

    lemma CountNotEqual_Increasing:
      \forall value_type *a, v, integer k, m, n;
        m <= k <= n  ==>  CountNotEqual(a, m, k, v) <= CountNotEqual(a, m, n, v);

    lemma CountNotEqual_Unchanged{K,L}:
      \forall value_type *a, v, integer m, n;
        Unchanged{K,L}(a, m, n)  ==>
        CountNotEqual{K}(a, m, n, v) == CountNotEqual{L}(a, m, n, v);

    lemma CountNotEqual_Count:
      \forall value_type *a, v, integer m, n;
        m <= n  ==>  CountNotEqual(a, m, n, v) == n - m - Count(a, m, n, v);

    lemma CountNotEqual_Union:
      \forall value_type *a, v, integer k, m, n;
        0 <= k <= m <= n  ==>
        CountNotEqual(a, k, n, v) ==
        CountNotEqual(a, k, m, v) + CountNotEqual(a, m, n, v);
  }
*/
```

Listing 7.40: The logic function `CountNotEqual` (2)

7.14.1. Formal specification of `remove_copy2`

We extend our formal specification by using `CountNotEqual` [7.39] and add the new postcondition `size`, which states that the returning value of `remove_copy2` equals `CountNotEqual`. The following listing shows the formal specification of `remove_copy2` [7.41].

```
/*@
  requires valid:     \valid_read(a + (0..n-1));
  requires valid:     \valid(b + (0..n-1));
  requires sep:       \separated(a + (0..n-1), b + (0..n-1));
  assigns             b[0..n-1];
  ensures size:       \result == CountNotEqual(a, n, v);
  ensures bound:      0 <= \result <= n;
  ensures discard:    NoneEqual(b, \result, v);
  ensures unchanged:  Unchanged{Old,Here}(a, n);
  ensures unchanged:  Unchanged{Old,Here}(b, \result, n);
*/
size_type
remove_copy2(const value_type* a, size_type n, value_type* b, value_type v);
```

Listing 7.41: Formal specification of `remove_copy2`

7.14.2. Implementation of `remove_copy2`

The following listing shows the implementation of our extended of `remove_copy2`. Here we added the loop invariant `size` which corresponds to the postcondition in `remove_copy2` [7.41]. In order to ensure that the loop invariant `size` can be verified we have added the assertions `size` and `unchanged`.

```
size_type
remove_copy2(const value_type* a, size_type n, value_type* b, value_type v)
{
  size_type k = 0u;

  /*@
    loop invariant size:      k == CountNotEqual(a, i, v);
    loop invariant bound:     0 <= k <= i <= n;
    loop invariant discard:   NoneEqual(b, k, v);
    loop invariant unchanged: Unchanged{Pre,Here}(b, k, n);
    loop assigns   k, i, b[0..n-1];
    loop variant   n-i;
  */
  for (size_type i = 0u; i < n; ++i) {
    if (a[i] != v) {
      b[k++] = a[i];
      //@ assert unchanged: Unchanged{LoopCurrent,Here}(a, n);
      //@ assert size:      k == CountNotEqual(a, 0, i+1, v);
    }
  }

  return k;
}
```

Listing 7.42: Implementation of `remove_copy2`

While we now can precisely speak of the number of copied elements, it is still not possible to say something about the exact relationship between the elements of range `a[0..n-1]` and range `b[0..n-1]`. We will address this question the contract of `remove_copy3` [7.48].

7.15. The `remove_copy3` algorithm (final contract)

In this section we extend the contracts of `remove_copy` [7.37] and `remove_copy2` [7.41] by introducing a logic function, which describes the relationship between the elements of input range `a[0..n-1]` and the output range `b[0..\result-1]`. Note that we have shown in the previous section that `\result` equals `CountNotEqual(a, n, v)`.

7.15.1. A closer look on the properties of `remove_copy`

Figure 7.43 shows a modified version of the Figure 7.36. We left out the indices of values that were not copied into the target array. Furthermore we have added a dashed arrow which points to the index that corresponds to the *one past the end* location of the input and output range.

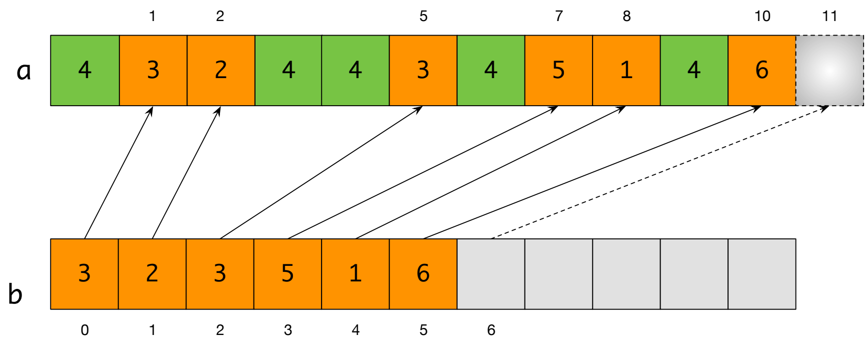

Figure 7.43.: Partitioning the input of `remove_copy`

These arrows between the indices of the array `b` and array `a` define the following sequence p of seven indices. The index of the *one past the end* is underlined. $p = (1, 2, 5, 7, 8, 10, \underline{11})$

More generally, we refer to the sequence p as *partitioning sequence* of `remove_copy` for the array `a[0.. n-1]`. For the **length of a partitioning sequence** m we get the following **strictly monotone increasing** sequence:

$$0 \leq p_0 < ... < p_m = n \tag{7.1}$$

and the open index intervals

$$(p_i, p_{i+1}) \qquad \forall i : 0 \leq i < m$$

mark **consecutive ranges** of the value `v` in the source array, that is,

$$a[k] = v \qquad \forall k : p_i < k < p_{i+1} \tag{7.2}$$

Additionally, the half open interval

$$[0, p_0)$$

also marks another **consecutive range** of the value `v` in the source array:

$$a[k] = v \qquad \forall k : 0 \leq k < p_0 \tag{7.3}$$

Another observation is that

$$a[p_i] \neq v \qquad \forall i : 0 \leq i < m \tag{7.4}$$

holds. Finally, we have

$$a[p_i] = b[i] \qquad \forall i : 0 \leq i < m \tag{7.5}$$

which, together with the inequality (7.4) states, that the target does not contain the value `v`

$$b[i] \neq v \qquad \forall i : 0 \leq i < m$$

7.15.2. More lemmas on `CountNotEqual`

Our formalization the properties of §7.15.1 relies on the logic function `CountNotEqual` [7.39]. We also rely on the logic function `FindNotEqual` [4.16] and the lemmas of `CountFindNotEqual` [7.44] in the following listing that provide more facts about `CountNotEqual` and `FindNotEqual`.

```
/*@
  axiomatic CountFindNotEqual
  {
    lemma CountNotEqual_AllEqual:
      \forall value_type *a, v, integer m, n;
        0 <= m <= n            ==>
        AllEqual(a, m, n, v)    ==>
        CountNotEqual(a, m, n, v) == 0;

    lemma CountNotEqual_SomeNotEqual:
      \forall value_type *a, v, integer m, n;
        0 <= m < n                    ==>
        0 < CountNotEqual(a, m, n, v)   ==>
        SomeNotEqual(a, m, n, v);

    lemma CountNotEqual_FindNotEqual:
      \forall value_type *a, v, integer m, n;
        0 <= m < n                    ==>
        0 < CountNotEqual(a, m, n, v)   ==>
        FindNotEqual(a, m, n, v) < n-m;

    lemma CountNotEqual_Zero:
      \forall value_type *a, v, integer m, n;
        0 <= m < n    ==>
        CountNotEqual(a, m, m + FindNotEqual(a, m, n, v), v) == 0;

    lemma CountNotEqual_Decrement:
      \forall value_type *a, v, integer m, n;
        0 <= m < n   ==>
        CountNotEqual(a, m + FindNotEqual(a, m, n, v), n, v) ==
        CountNotEqual(a, 0, n, v) - CountNotEqual(a, 0, m, v);
  }
*/
```

Listing 7.44: The logic definition(s) `CountFindNotEqual`

7.15.3. Formalizing the properties of the partitions

The function `RemovePartition`, whose axiomatic definition is given in Listings 7.45 and 7.46 defines the partitioning sequence p from §7.15.1.

```
/*@
  axiomatic RemovePartition
  {
    logic integer
    RemovePartition(value_type* a, integer n, value_type v, integer p) =
      \let c = CountNotEqual(a, n, v);
      \let x = RemovePartition(a, n, v, p-1) + 1;
        p < 0 ? -1 : // 0 <= p
          (n <= 0 ? 0 : // 0 < n
            p < c ? x + FindNotEqual(a, x, n, v) : n
          );

    lemma RemovePartition_Empty:
      \forall value_type *a, v, integer n, p;
        n <= 0 <= p  ==>
        RemovePartition(a, n, v, p) == 0;

    lemma RemovePartition_Left:
      \forall value_type *a, v, integer n, p;
        p < 0  ==>  RemovePartition(a, n, v, p) == -1;

    lemma RemovePartition_Right:
      \forall value_type *a, v, integer n, p;
        0 <= n                     ==>
        CountNotEqual(a, n, v) <= p  ==>  RemovePartition(a, n, v, p) == n;

    lemma RemovePartition_Next:
      \forall value_type *a, v, integer n, p;
        \let x = RemovePartition(a, n, v, p-1) + 1;
        0 <= n                         ==>
        0 <= p < CountNotEqual(a, n, v)  ==>
        RemovePartition(a, n, v, p) == x + FindNotEqual(a, x, n, v);

    lemma RemovePartition_Lower:
      \forall value_type *a, v, integer i, n, p;
        0 < n                          ==>
        0 <= p < CountNotEqual(a, n, v)  ==>
        0 <= RemovePartition(a, n, v, p);

    lemma RemovePartition_Core:
      \forall value_type *a, v, integer i, n, p;
        \let R = RemovePartition(a, n, v, p);
        0 < n                          ==>
        0 <= p < CountNotEqual(a, n, v)  ==>
        (R < n      &&
         a[R] != v  &&
         CountNotEqual(a, R, n, v) == CountNotEqual(a, 0, n, v) - p);

    lemma RemovePartition_Upper:
      \forall value_type *a, v, integer i, n, p;
        0 < n                          ==>
        0 <= p < CountNotEqual(a, n, v)  ==>
        RemovePartition(a, n, v, p) < n;
```

Listing 7.45: The logic function `RemovePartition` (1)

Before we begin to relate the various lemmas to the formulas from §7.15.1 we want to remind the reader that logic functions (and predicates) must be total that is they must be defined for all possible argument values.

```
    lemma RemovePartition_NotEqual:
      \forall value_type *a, v, integer n, p;
        0 < n  ==>
        0 <= p < CountNotEqual(a, n, v)   ==>
        a[RemovePartition(a, n, v, p)] != v;

    lemma RemovePartition_Count:
      \forall value_type *a, v, integer n, p;
        0 < n                             ==>
        0 <= p < CountNotEqual(a, n, v)   ==>
        CountNotEqual(a, RemovePartition(a, n, v, p), n, v) ==
        CountNotEqual(a, 0, n, v) - p;

    lemma RemovePartition_StrictlyWeakIncreasing:
      \forall value_type *a, v, integer n, p;
        0 < n                            ==>
        0 < p < CountNotEqual(a, n, v)   ==>
        RemovePartition(a, n, v, p-1) < RemovePartition(a, n, v, p);

    lemma RemovePartition_Segment:
      \forall value_type *a, v, integer i, n, p;
        0 < n                             ==>
        0 <= p                            ==>
        p + 1 < CountNotEqual(a, n, v)    ==>
          AllEqual(a, RemovePartition(a, n, v, p) + 1,
                      RemovePartition(a, n, v, p+1), v);

    lemma RemovePartition_Extend:
      \forall value_type *a, v, integer n, p;
        0 < n                             ==>
        0 <= p < CountNotEqual(a, n, v)   ==>
        RemovePartition(a, n, v, p) == RemovePartition(a, n+1, v, p);

    lemma RemovePartition_Unchanged{K,L}:
      \forall value_type *a, v, integer n, p;
        Unchanged{K,L}(a, n)   ==>
        RemovePartition{K}(a, n, v, p) == RemovePartition{L}(a, n, v, p);
  }
*/
```

Listing 7.46: The logic function RemovePartition (2)

The lemmas for RemovePartition are related to the properties of §7.15.1 in the following way.

- Property (7.1) is expressed by the lemmas RemovePartition_Empty, RemovePartition_Left RemovePartition_Right, and RemovePartition_StrictlyWeakIncreasing
- Properties (7.2) and (7.3) are described by lemmas RemovePartition_Segment.
- Property (7.4) is expressed by lemma RemovePartition_NotEqual.
- Property (7.5) is formulated using the predicate Remove [7.47].

We would like to point out lemma RemovePartition_Core which subsumes the statements of the subsequent lemmas RemovePartition_Upper, RemovePartition_NotEqual, and RemovePartition_Count. While these three lemmas add nothing new we have kept them because they correspond directly to individual properties of §7.15.1. The question may arise why there is the lemma RemovePartition_Core in the first place. The answer is that we found the individual properties so intertwined that we were not able to verify them separately but only their joint embodiment.

7.15.4. The predicate Remove

The predicate Remove [7.47] primarily serves in order to improve the readability of our specification remove_copy3 [7.48]. As mentioned before this predicate encapsulates the Property (7.5) from §7.15.1. Note that Remove [7.47] also contains an overloaded version of Remove which will be used for the specification of the *in-place* variant remove [7.52] of remove_copy.

```
/*@
  axiomatic Remove
  {
    predicate
    Remove{K,L}(value_type* a, integer n, integer i, value_type* b, value_type v) =
      \forall integer k; 0 <= k < CountNotEqual{K}(a, i, v)  ==>
        \let j = RemovePartition{K}(a, n, v, k);
        \at(b[k],L) == \at(a[j],K);

    predicate
    Remove{K,L}(value_type* a, integer n, value_type* b, value_type v) =
      Remove{K,L}(a, n, n, b, v);

    predicate
    Remove{K,L}(value_type* a, integer n, integer i, value_type v) =
      \forall integer k; 0 <= k < CountNotEqual{K}(a, i, v)  ==>
        \let j = RemovePartition{K}(a, n, v, k);
        \at(a[k],L) == \at(a[j],K);

    predicate
    Remove{K,L}(value_type* a, integer n, value_type v) =
      Remove{K,L}(a, n, n, v);
  }
*/
```

Listing 7.47: The logic definition(s) Remove

7.15.5. Formal specification of `remove_copy3`

The following listing shows the formal specification of `remove_copy` [7.37]. The additional postcondition `remove` makes use of the predicate `Remove` [7.47] which we have just described. Furthermore, we have again the postcondition `unchanged` which states that the source array `a[0..n-1]` does not change.

```
/*@
  requires valid:    \valid_read(a + (0..n-1));
  requires valid:    \valid(b + (0..n-1));
  requires sep:      \separated(a + (0..n-1), b + (0..n-1));
  assigns            b[0..n-1];
  ensures size:      \result == CountNotEqual{Old}(a, n, v);
  ensures bound:     0 <= \result <= n;
  ensures remove:    Remove{Old,Here}(a, n, b, v);
  ensures discard:   NoneEqual(b, \result, v);
  ensures unchanged: Unchanged{Old,Here}(a, n);
  ensures unchanged: Unchanged{Old,Here}(b, \result, n);
*/
size_type
remove_copy3(const value_type* a, size_type n, value_type* b, value_type v);
```

Listing 7.48: Formal specification of `remove_copy3`

7.15.6. Implementation of `remove_copy3`

We discuss now some aspects of the implementation of `remove_copy3` [7.50]. We introduce the loop invariant `mapping`. This invariant states that the variable `i` will always be smaller or equal to the result of `RemovePartition(a, n, v, k)`. We also add the assertion `mapping` to our implementation as stepping stone for the provers to verify the correctness of this loop invariant.

Somewhat surprisingly, in order to reduce excessive verification times we had to add an else-branch to our implementation that besides the assertion `unchanged` is empty.

Regarding the assertion `update`, one might wonder why we do not simply write `\at(a[i], Pre)`. However, this expression would be wrong because the index `i` would then be interpreted as `\at(i,Pre)` which doesn't makes sense for a local variable. Frama-C/WP consequently rejects this expression with the following error message.

```
Warning: unbound logic variable i. Ignoring code annotation
```

We could explicitly refer to the current value of `i` by using the subexpression `\at(i,Here)` inside the assertion `update`. We felt, however, tow introduce the predicate `At` [7.49] to simplify the comparison of array elements in programme states where the particular index variable isn't visible.

```
/*@
  axiomatic At
  {
    logic value_type At{L}(value_type* x, integer i) = \at(x[i],L);
  }
*/
```

Listing 7.49: The logic definition(s) `At`

The second argument `At` is interpreted at the programme point here it appears, that is, `Here`. Using this auxiliary logic function the assertion `update` is arguably more readable.

```
size_type
remove_copy3(const value_type* a, size_type n, value_type* b, value_type v)
{
  size_type k = 0u;

  /*@
    loop invariant size:       k == CountNotEqual{Pre}(a,i,v);
    loop invariant bound:      0 <= k <= i <= n;
    loop invariant remove:     Remove{Pre,Here}(a, n, i, b, v);
    loop invariant discard:    NoneEqual(b, k, v);
    loop invariant interval:   RemovePartition{Pre}(a, n, v, k-1) <= i;
    loop invariant interval:   i <= RemovePartition{Pre}(a, n, v, k);
    loop invariant unchanged:  Unchanged{Pre,Here}(a, n);
    loop invariant unchanged:  Unchanged{Pre,Here}(b, k, n);
    loop assigns   k, i, b[0..n-1];
    loop variant   n-i;
  */
  for (size_type i = 0u; i < n; ++i) {
    if (a[i] != v) {
      b[k++] = a[i];
      //@ assert size:      k == CountNotEqual{Pre}(a, i+1, v);
      //@ assert update:    b[k-1] == At{Pre}(a, i);
      //@ assert interval:  i == RemovePartition{Pre}(a, n, v, k-1);
      //@ assert remove:    Remove{Pre,Here}(a, n, i, b, v);
      //@ assert remove:    Remove{Pre,Here}(a, n, i+1, b, v);
      //@ assert unchanged: Unchanged{Pre,Here}(a, n);
      //@ assert unchanged: Unchanged{Pre,Here}(b, k, n);
    }
    else {
      //@ assert unchanged: Unchanged{Pre,Here}(a, n);
    }

    //@ assert unchanged: Unchanged{Pre,Here}(a, n);
  }

  return k;
}
```

Listing 7.50: Implementation of `remove_copy3`

7.16. The `remove` algorithm

The C++ Standard Library also contains a function `remove` [20, 28.6.8] performing the same operation as `remove_copy` as an in-place algorithm. Its signature is very similar to that of `remove_copy`, except that there is no need for an output array.

```
size_type remove(value_type* a, size_type n, value_type v);
```

Figure 7.51 shows how `remove` is supposed to remove all occurrences of the given value 4 from a range.

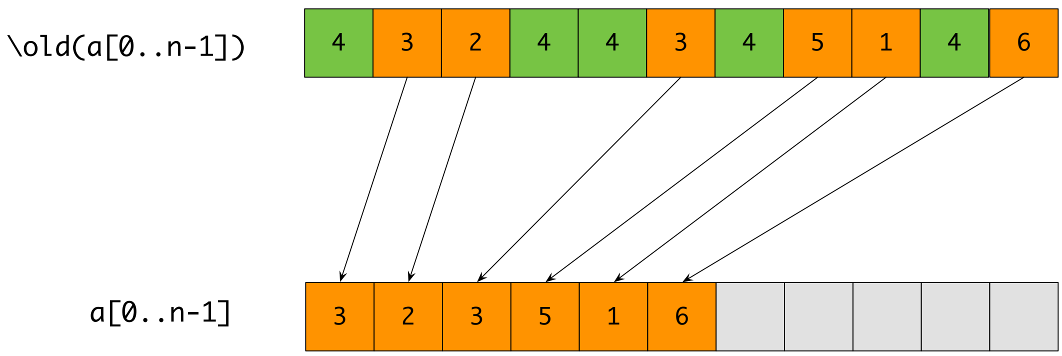

Figure 7.51.: Effects of `remove`

7.16.1. Formal specification of `remove`

The following listing shows a formal specification of the function `remove` [7.52]. Our specification is very similar to the one of `remove_copy3` [7.48] except that we using a version of `Remove` [7.47] that takes only one pointer argument.

```
/*@
  requires valid:     \valid(a + (0..n-1));
  assigns             a[0..n-1];
  ensures size:       \result == CountNotEqual{Old}(a, n, v);
  ensures bound:      0 <= \result <= n;
  ensures remove:     Remove{Old,Here}(a, n, v);
  ensures discard:    NoneEqual(a, \result, v);
  ensures unchanged:  Unchanged{Old,Here}(a, \result, n);
*/
size_type
remove(value_type* a, size_type n, value_type v);
```

Listing 7.52: Formal specification of `remove`

7.16.2. Implementation of `remove`

In the following listing we show our implementation of `remove` [7.53] together with the additional loop annotations. Again, the annotations are very similar to those of the implementation of `remove_copy3` [7.50].

```
size_type
remove(value_type* a, size_type n, value_type v)
{
  size_type k = 0u;

  /*@
    loop invariant size:       k == CountNotEqual{Pre}(a,i,v);
    loop invariant bound:      0 <= k <= i <= n;
    loop invariant remove:     Remove{Pre,Here}(a, n, i, v);
    loop invariant discard:    NoneEqual(a, k, v);
    loop invariant interval:   RemovePartition{Pre}(a, n, v, k-1) <= i;
    loop invariant interval:   i <= RemovePartition{Pre}(a, n, v, k);
    loop invariant unchanged:  Unchanged{Pre,Here}(a, k, n);
    loop invariant unchanged:  a[k] == At{Pre}(a, k);
    loop assigns   k, i, a[0..n-1];
    loop variant   n-i;
  */
  for (size_type i = 0u; i < n; ++i ) {
    if (a[i] != v) {
      a[k++] = a[i];
      //@ assert size:      k == CountNotEqual{Pre}(a, 0, i+1, v);
      //@ assert update:    a[k-1] == At{Pre}(a, i);
      //@ assert interval:  i == RemovePartition{Pre}(a, n, v, k-1);
      //@ assert remove:    Remove{Pre,Here}(a, n, i,   v);
      //@ assert remove:    Remove{Pre,Here}(a, n, i+1, v);
    }
  }

  return k;
}
```

Listing 7.53: Implementation of `remove`

Also note the use of the predicate `At` [7.49] in the loop invariant `unchanged` and the assertion `update`.

7.17. The `shuffle` algorithm

The `shuffle` algorithm in the C++ Standard Library [20, §28.6.13] randomly rearranges the elements of a given range, that is, it randomly picks one of its possible orderings. For our purposes we have modified the generic implementation to that of a range of type `value_type`. The signature now reads:

```
void shuffle(value_type* a, size_type n, unsigned short* rand);
```

The argument `rand` holds the state of a simple random number generator that is used in the implementation of `shuffle`.

Figure 7.54 illustrates an example run of `shuffle`. In this figure, the values 1, 2, 3, and 4 occur twice, once, once, and three times, respectively, both before and after the `shuffle` run. This expresses that the range has been reordered.

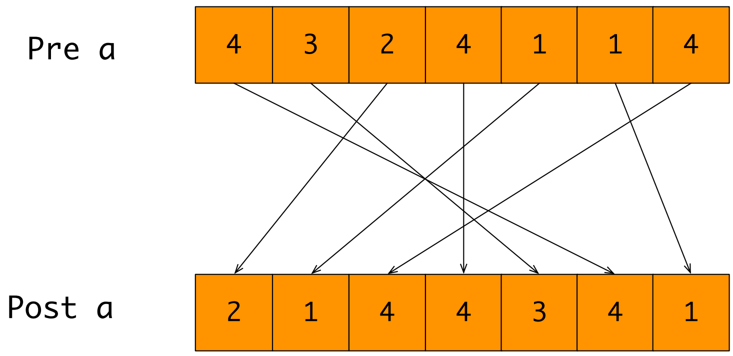

Figure 7.54.: Effects of `shuffle`

7.17.1. The predicate `MultisetReorder`

The `shuffle` algorithm is the first example in this document where we have to specify a *rearrangement* or *reordering* of the elements of a given range. We say that an array has been reordered between two states if the number of each element in the array remains unchanged. In other words, reordering leaves the *multiset*[23] of elements in the range unchanged.

We use the predicate `MultisetReorder` [7.55] to formally describe this property. This predicate, which is given in two overloaded versions, relies on the logic function `Count` [4.44]. We list here several lemma with basic properties of `MultisetReorder`. We will use these lemmas during the verification of various algorithms.

```
/*@
  axiomatic MultisetReorder
  {
    predicate
    MultisetReorder{K,L}(value_type* a, integer m, integer n) =
      \forall value_type v;
        Count{K}(a, m, n, v) == Count{L}(a, m, n, v);

    predicate
    MultisetReorder{K,L}(value_type* a, integer n) =
      MultisetReorder{K,L}(a, 0, n);

    lemma Unchanged_MultisetReorder{K,L}:
      \forall value_type *a, integer k, n;
        Unchanged{K,L}(a, k, n)  ==>  MultisetReorder{K,L}(a, k, n);

    lemma MultisetReorder_DisjointUnion{K,L}:
      \forall value_type *a, integer i, k, n;
        0 <= i <= k <= n                ==>
        MultisetReorder{K,L}(a, i, k)   ==>
        MultisetReorder{K,L}(a, k, n)   ==>
        MultisetReorder{K,L}(a, i, n);

    lemma MultisetReorder_Symmetric{K,L}:
      \forall value_type *a, integer m, n;
        MultisetReorder{K,L}(a, m, n)   ==>  MultisetReorder{L,K}(a, m, n);

    lemma MultisetReorder_Transitive{K,L,M}:
      \forall value_type *a, integer m, n;
        MultisetReorder{K,L}(a, m, n)   ==>
        MultisetReorder{L,M}(a, m, n)   ==>
        MultisetReorder{K,M}(a, m, n);
  }
*/
```

Listing 7.55: The logic definition(s) `MultisetReorder`

[23] See `http://en.wikipedia.org/wiki/Multiset`

7.17.2. Formal specification of shuffle

In the specification of the shuffle [7.56] algorithm we demand that the range a[0..n-1] is valid for reading and writing. We use the predicate MultisetReorder [7.55] to express that the contents of a [0..n-1] is just permuted, i.e., the number of occurrences of each of its members remains unchanged. The array rand contains a seed for the random number generator used to randomize the shuffle. By specifying that the function assigns to rand we capture that the function may return a different permutation every time.

Note that our specification only states that the resulting range is a reordering of the input range; nothing more and nothing less. Ideally, we would also specify that sequence of reorderings obtained by repeated calls of shuffle is required to be random. The informal specification [20, §28.6.13] of shuffle states that *that each possible permutation of those elements has equal probability of appearance*. However, ACSL does currently not support the specification of temporal properties related to repeated call results.

```
/*@
  requires valid:    \valid(a + (0..n-1));
  requires valid:    \valid(seed + (0..2));
  requires sep:      \separated(a + (0..n-1), seed + (0..2));
  assigns            a[0..n-1];
  assigns            seed[0..2];
  ensures  reorder:  MultisetReorder{Old,Here}(a,n);
*/
void
shuffle(value_type* a, size_type n, unsigned short* seed);
```

Listing 7.56: Formal specification of shuffle

More generally speaking, it is not trivial to capture the notion of randomness in a mathematically precise way. As a typical example, we refer to a paper [22, p.6–8], which just gives four statistical tests indicating the randomness of the permutations computed with their algorithm. From a theoretical point of view, a sequence of permutations can be called “random” if its Kolmogorov complexity exceeds a certain measure, however, this property is undecidable [23].

7.17.3. Implementation of `shuffle`

The following listing shows our implementation of the function `shuffle` [7.57]. It repeatedly calls the function `swap` [7.6] to *transpose* (randomly) selected elements. For details of out source of randomness we refer to the function `random_number` [7.60].

```
void
shuffle(value_type* a, size_type n, unsigned short* seed)
{
  if (0u < n) {
    /*@
      loop invariant bounds:    1 <= i <= n;
      loop invariant reorder:   MultisetReorder{Pre,Here}(a, 0, i);
      loop invariant unchanged: Unchanged{Pre,Here}(a, i, n);
      loop assigns   i, a[0..n-1], seed[0..2];
      loop variant   n - i;
    */
    for (size_type i = 1u; i < n; ++i) {
      size_type k = random_number(seed, i) + 1u;

      //@ assert less: 0 <= k <= i;
      if (k < i) {
        swap(&a[k], &a[i]);
        //@ assert swapped: ArraySwap{LoopCurrent,Here}(a, k, i, n);
        //@ assert reorder: MultisetReorder{LoopCurrent,Here}(a, i+1);
        //@ assert reorder: MultisetReorder{Pre,Here}(a, i+1);
      }
      else {
        //@ assert unchanged: Unchanged{LoopCurrent,Here}(a, i+1);
        //@ assert reorder:   MultisetReorder{Pre,Here}(a, i+1);
      }

      //@ assert reorder: MultisetReorder{Pre,Here}(a, i+1);
    }
  }
}
```

Listing 7.57: Implementation of `shuffle`

The loop invariants `reorder` and `unchanged` of `shuffle` are necessary for the verification of the postcondition `reorder`: in the `i`th loop cycle, the subrange `a[0..i-1]` has been reordered, while the remaining subrange `a[i..n-1]` is yet unchanged. We also formulate several auxiliary assertions `reorder` which use the the predefined label `LoopCurrent`, to guide the automatic verification the loop invariant `reorder`. Please not the empty **`else`**-branch hat only contains an assertion `reorder`. We introduced this assertion to support the verification of the `reorder` property.

In addition, we rely on the predicate `ArraySwap` [7.58] rather than the literal postcondition of `swap` [7.6], since this leads to to more concise annotations and better a performance of the automatic provers.

```
/*@
  axiomatic ArraySwap
  {
    predicate
    ArraySwap{K,L}(value_type* a, integer i, integer k, integer n) =
      0 <= i < k < n                &&
      At{K}(a, i) == At{L}(a, k)    &&
      At{K}(a, k) == At{L}(a, i)    &&
      Unchanged{K,L}(a, 0,   i)     &&
      Unchanged{K,L}(a, i+1, k)     &&
      Unchanged{K,L}(a, k+1, n);
  }
*/
```

Listing 7.58: The logic definition(s) `ArraySwap`

The lemma `MultisetSwap_Middle` [7.59] states that swapping the elements `a[i]` and `a[k]` is a particular kind of reordering on the range `a[i..k]`.

```
/*@
  axiomatic MultisetSwap
  {
    lemma MultisetSwap_Middle{K,L}:
      \forall value_type* a, integer i, k, n;
        ArraySwap{K,L}(a, i, k, n)  ==>  MultisetReorder{K,L}(a, i, k+1);

    lemma MultisetSwap_FrontMiddle{K,L}:
      \forall value_type* a, integer i, k, n;
        ArraySwap{K,L}(a, i, k, n)  ==>  MultisetReorder{K,L}(a, 0, k+1);
  }
*/
```

Listing 7.59: The logic definition(s) `MultisetSwap`

7.18. Verifying a random number generator

We describe in this section `random_number` [7.60] which implements a simple random-number generator. As in the case of `shuffle` [7.56] itself, we do not formulate precise properties of randomness and only require its result to be in the specified range `[0..n-1]`. Again, the `assigns` clause to the array `state` models the dependency on an additional state.

Note that in the following listing, we also provide the rather simple specification of the function `random_init` that is called to initialize the state of the random generator.

```
/*@
  requires  pos:    0 < n;
  requires  valid:  \valid(state + (0..2));
  assigns           state[0..2];
  ensures   result: 0 <= \result < n;
*/
size_type
random_number(unsigned short* state, size_type n);


/*@
  requires  \valid(state + (0..2));

  assigns   state[0..2];
*/
void
random_init(unsigned short* state);
```

Listing 7.60: Formal specification of `random_number`

The implementations of `random_number` and `random_init` are shown in the following listing. Internally, we rely on a custom implementation of the POSIX.1 random number generator `lrand48()` [24] This random number generator is a linear congruence generator with a 48 bit state and the iteration procedure

$$x_{n+1} = ax_n + c \bmod 2^{48} \tag{7.6}$$

where $a = 25214903917$ and $c = 11$ are relatively prime integers.

As a part of the iteration procedure in Equation (7.6) an unsigned overflow may occur. This does not affect the result as we are only interested in its lowest 48 bits. However, as one of the options we use, `-warn-unsigned-overflow`, causes Frama-C/WP assert the absence of unsigned overflow this algorithm does not verify under the same options used for the other algorithms. As an exception, we have therefore decided to disable `-warn-unsigned-overflow` for this function as the unsigned overflow is both benign and well-defined (cf. [17, §6.2.5, 9]).

[24] See `http://pubs.opengroup.org/onlinepubs/9699919799/functions/lrand48.html`

```
// see IEEE 1003.1-2008, 2016 Edition for specification
/*@
  requires valid: \valid(seed + (0..2));
  assigns seed[0..2];
  ensures lower:  0 <= \result;
  ensures upper:  \result <= 0x7fffffff;
*/
static long
my_lrand48(unsigned short* seed)
{
  unsigned long long state = (unsigned long long)seed[0] << 32
                           | (unsigned long long)seed[1] << 16
                           | (unsigned long long)seed[2];
  state = (0x5deece66dull * state + 0xbull) % (1ull << 48);
  //@ assert lower: state < (1ull << 48);
  long result = state / (1ull << 17);
  //@ assert lower: 0 <= result;
  seed[0u] = state >> 32 & 0xffff;
  seed[1u] = state >> 16 & 0xffff;
  seed[2u] = state >>  8 & 0xffff;
  return result;
}

size_type
random_number(unsigned short* state, size_type n)
{
  return my_lrand48(state) % n;
}

void
random_init(unsigned short* state)
{
  state[0] = 0x243f;
  state[1] = 0x6a88;
  state[2] = 0x85a3;
}
```

Listing 7.61: Implementation of `random_number`

Note that we use the custom acsl lemma `RandomNumberModulo` [7.62] from the following listing to support the verification of some assertions.

```
/*@
  axiomatic C_Bit
  {
    lemma RandomNumberModulo:
      \forall unsigned long long a;
        (a % (1ull << 48)) < (1ull << 48);
  }
*/
```

Listing 7.62: The logic definition(s) `C_Bit`

8. Numeric algorithms

The algorithms that we considered so far only *compared*, *read* or *copied* values in sequences. In this chapter, we consider so-called *numeric* algorithms of the C++ Standard Library [20, §29.8] that use arithmetic operations on `value_type` to combine the elements of sequences.

```
#define VALUE_TYPE_MAX  INT_MAX
#define VALUE_TYPE_MIN  INT_MIN
```

Listing 8.1: Limits of `value_type`

In order to refer to potential arithmetic overflows we introduce the two constants shown in Listing 8.1 which refer to the numeric limits of `value_type` (see also §2.3).

We consider the following algorithms.

- `iota` writes sequentially increasing values into a range (§8.1)
- `accumulate` computes the sum of the elements in a range (§8.2)
- `inner_product` computes the inner product of two ranges (§8.3)
- `partial_sum` computes the sequence of partial sums of a range (§8.4)
- `adjacent_difference` computes the differences of adjacent elements in a range (§8.5)
- Finally, in §8.6 we show that under appropriate preconditions the algorithms `partial_sum` and `adjacent_difference` are inverse to each other.

The formal specifications of these algorithms raise new questions. In particular, we now have to deal with arithmetic overflows in `value_type`.

8.1. The `iota` algorithm

The `iota` algorithm in the C++ Standard Library [20, §29.8.12] assigns sequentially increasing values to a range, where the initial value is user-defined. Our version of the original signature reads:

```
void iota(value_type* a, size_type n, value_type v);
```

Starting at `v`, the function assigns consecutive integers to the elements of the range `a`. When specifying `iota` we must be careful to deal with possible overflows of the argument `v`.

8.1.1. Formal specification of `iota`

The specification of `iota` relies on the logic function `IotaGenerate` [8.2] that is defined in the following listing.

```
/*@
  axiomatic IotaGenerate
  {
    predicate
    IotaGenerate(value_type* a, integer n, value_type v) =
      \forall integer i; 0 <= i < n  ==>  a[i] == v+i;
  }
*/
```

Listing 8.2: The logic definition(s) `IotaGenerate`

The specification of `iota` is shown in the following listing. It uses the logic function `IotaGenerate` [8.2] in order to express the postcondition `increment`.

```
/*@
  requires valid:    \valid(a + (0..n-1));
  requires limit:    v + n <= VALUE_TYPE_MAX;
  assigns            a[0..n-1];
  ensures increment: IotaGenerate(a, n, v);
*/
void
iota(value_type* a, size_type n, value_type v);
```

Listing 8.3: Formal specification of `iota`

The specification of `iota` refers to `VALUE_TYPE_MAX` which is the maximum value of the underlying integer type (see Listing 8.1). In order to avoid integer overflows the sum `v+n` must not be greater than the constant `VALUE_TYPE_MAX`.

8.1.2. Implementation of `iota`

The following listing shows an implementation of the `iota` function.

```
void
iota(value_type* a, size_type n, value_type v)
{
  /*@
    loop invariant bound:     0 <= i <= n;
    loop invariant limit:     v == \at(v, Pre) + i;
    loop invariant increment: IotaGenerate(a, i, \at(v, Pre));

    loop assigns i, v, a[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0u; i < n; ++i) {
    a[i] = v++;
  }
}
```

Listing 8.4: Implementation of `iota`

The loop invariant `increment` describes that in each iteration of the loop the current value `v` is equal to the sum of the value `v` in state of function entry and the loop index `i`. We have to refer here to `\at(v,Pre)` which is the value on entering `iota`.

8.2. The `accumulate` algorithm

The `accumulate` algorithm in the C++ Standard Library [20, §29.8.2] computes the sum of an given initial value and the elements in a range. Our version of the original signature reads:

```
value_type
accumulate(const value_type* a, size_type n, value_type init);
```

The result of `accumulate` shall equal the value $\texttt{init} + \sum_{i=0}^{\mathtt{n}-1} \mathtt{a}[i]$. This implies that `accumulate` will return `init` for an empty range.

8.2.1. The logic function `Accumulate`

As in the case of `count` [4.47] we specify `accumulate` by first defining the *logic function* `Accumulate` [8.5] that formally defines the summation of elements in an array.

```
/*@
  axiomatic Accumulate
  {
    logic integer
    Accumulate{L}(value_type* a, integer n, integer init) =
      n <= 0 ? init : Accumulate(a, n-1, init) + a[n-1];

    predicate
    AccumulateBounds{L}(value_type* a, integer n, value_type init) =
      \forall integer i; 0 <= i <= n  ==>
        VALUE_TYPE_MIN <= Accumulate(a, i, init) <= VALUE_TYPE_MAX;

    lemma Accumulate_Init:
      \forall value_type *a, init, integer n;
         n <= 0  ==>  Accumulate(a, n, init) == init;

    lemma Accumulate_Unchanged{K,L}:
      \forall value_type *a, init, integer n;
        Unchanged{K,L}(a, n)  ==>
        Accumulate{K}(a, n, init) == Accumulate{L}(a, n, init);

    lemma Accumulate_Unchanged_Shrink{K,L}:
      \forall value_type *a, init, integer m, n;
        0 <= m <= n           ==>
        Unchanged{K, L}(a, n)  ==>
        Accumulate{K}(a, m, init) == Accumulate{L}(a, m, init);

    lemma AccumulateBounds_Unchanged{K,L}:
      \forall value_type *a, init, integer n;
        Unchanged{K, L}(a, n)           ==>
        AccumulateBounds{K}(a, n, init) ==>
        AccumulateBounds{L}(a, n, init);
  }
*/
```

Listing 8.5: The logic definition(s) `Accumulate`

With this definition the following equation holds for $n \geq 0$

$$\texttt{Accumulate}(\texttt{a}, \texttt{n}, \texttt{init}) = \texttt{init} + \sum_{i=0}^{\texttt{n}-1} \texttt{a}[i] \tag{8.1}$$

The predicate `AccumulateBounds` [8.5] that we will subsequently use in order to compactly express requirements that exclude numeric overflows while accumulating value. This predicate states that for $0 \leq i < n$ the *partial sums*

$$\texttt{init} + \sum_{k=0}^{\texttt{i}} \texttt{a}[k] \tag{8.2}$$

do not overflow. If one of them did, one couldn't guarantee that the result of C implementation of `accumulate` equals the mathematical description of `Accumulate`.

8.2.2. `AccumulateDefault`—a variant of `Accumulate`

The following listing shows another version of `Accumulate` [8.5], called `AccumulateDefault` [8.6].

```
/*@
  axiomatic AccumulateDefault
  {
    logic integer
    AccumulateDefault{L}(value_type* a, integer n) =
      Accumulate(a, n, (value_type)(0));

    predicate
    AccumulateDefaultBounds{L}(value_type* a, integer n) =
      AccumulateBounds(a, n, (value_type)(0));

    lemma AccumulateDefault_Unchanged{K,L}:
      \forall value_type *a, integer n;
        0 <= n                ==>
        Unchanged{K,L}(a, n)   ==>
        AccumulateDefault{K}(a, n) == AccumulateDefault{L}(a, n);

    lemma AccumulateDefault_Zero{L}:
      \forall value_type* a; AccumulateDefault(a, 0) == 0;

    lemma AccumulateDefault_One{L}:
      \forall value_type* a; AccumulateDefault(a, 1) == a[0];

    lemma AccumulateDefault_Next{L}:
      \forall value_type* a, integer n;
        0 <= n  ==>
        AccumulateDefault(a, n+1) == AccumulateDefault(a, n) + a[n];

    lemma AccumulateDefaultBounds_Shrink{L}:
      \forall value_type* a, integer m, n;
        0 <= m <= n                    ==>
        AccumulateDefaultBounds(a, n)   ==>  AccumulateDefaultBounds(a, m);
  }
*/
```

Listing 8.6: The logic definition(s) `AccumulateDefault`

The function `AccumulateDefault` uses `a[0]` as default value of `init`. Thus, for `AccumulateDefault` we have

$$\texttt{AccumulateDefault}(\texttt{a}, \texttt{n}) = \sum_{i=0}^{\texttt{n}-1} \texttt{a}[i] \tag{8.3}$$

We will use this version for the specification of the algorithm `partial_sum` [8.13].

This listing also includes additional properties of observable `AccumulateDefault` behavior, here given as a lemmas. It also contains the predicate `AccumulateDefaultBounds` [8.6] with corresponding numeric limits for the predicate `AccumulateDefault`.

8.2.3. Formal specification of `accumulate`

Using the logic function `Accumulate` and the predicate `AccumulateBounds`, the specification of `accumulate` is then as simple as shown in the following listing.

```
/*@
  requires valid:  \valid_read(a + (0..n-1));
  requires bounds: AccumulateBounds(a, n, init);
  assigns          \nothing;
  ensures  result: \result == Accumulate(a, n, init);
*/
value_type
accumulate(const value_type* a, size_type n, value_type init);
```

Listing 8.7: Formal specification of `accumulate`

8.2.4. Implementation of `accumulate`

The following listing shows an implementation of the `accumulate` function with corresponding loop annotations.

```
value_type
accumulate(const value_type* a, size_type n, value_type init)
{
  /*@
    loop invariant index:    0 <= i <= n;
    loop invariant partial:  init == Accumulate(a, i, \at(init,Pre));
    loop assigns i, init;
    loop variant n-i;
  */
  for (size_type i = 0u; i < n; ++i) {
    //@ assert rte_help: init + a[i] == Accumulate(a, i+1, \at(init,Pre));
    init = init + a[i];
  }

  return init;
}
```

Listing 8.8: Implementation of `accumulate`

Note that loop invariant `partial` claims that in the i-th iteration step `result` equals the accumulated value of Equation (8.2). This depends on the property `bounds` of `accumulate` [8.7] which expresses that there is no numeric overflow when updating the variable `init`.

8.3. The `inner_product` algorithm

The `inner_product` algorithm in the C++ Standard Library [20, §29.8.4] computes the *inner product*[25] of two ranges. Our version of the original signature reads:

```
value_type
inner_product(const value_type* a, const value_type* b,
              size_type n, value_type init);
```

The result of `inner_product` equals the value

$$\texttt{init} + \sum_{i=0}^{\texttt{n}-1} \mathsf{a}[i] \cdot \mathsf{b}[i]$$

thus, `inner_product` will return `init` for empty ranges.

8.3.1. The logic function `InnerProduct`

As in the case of `accumulate` [8.7] we specify `inner_product` by defining in the following listing the logic function `InnerProduct` that formally expresses the summation of the element-wise product of two arrays.

Predicate `ProductBounds` [8.9] expresses that for $0 \leq i < n$ the products

$$\mathsf{a}[i] \cdot \mathsf{b}[i] \tag{8.4}$$

do not overflow. Predicate `InnerProductBounds` [8.9], on the other hand, states that for $0 \leq i < n$ the following sums do not overflow. cc

$$\texttt{init} + \sum_{k=0}^{\texttt{i}} \mathsf{a}[k] \cdot \mathsf{b}[k] \tag{8.5}$$

Otherwise, one cannot guarantee that the result of our implementation of `inner_product` [8.11] equals the mathematical description of `InnerProduct`. Finally, Lemma `InnerProduct_Unchanged` [8.9] states that the result of the `InnerProduct` only depends on the values of `a[0..n-1]` and `b[0..n-1]`.

[25] Also referred to as *dot product*, see `http://en.wikipedia.org/wiki/Dot_product`

```
/*@
  axiomatic InnerProduct
  {
    logic integer
    InnerProduct{L}(value_type* a, value_type* b, integer n,
                    value_type init) =
      n <= 0 ? init : InnerProduct(a, b, n-1, init) + (a[n-1] * b[n-1]);

    predicate
    ProductBounds(value_type* a, value_type* b, integer n) =
      \forall integer i; 0 <= i < n  ==>
        VALUE_TYPE_MIN <= a[i] * b[i] <= VALUE_TYPE_MAX;

    predicate
    InnerProductBounds(value_type* a, value_type* b, integer n,
                       value_type init) =
      \forall integer i; 0 <= i <= n  ==>
        VALUE_TYPE_MIN <= InnerProduct(a, b, i, init) <= VALUE_TYPE_MAX;

    lemma InnerProduct_Unchanged{K,L}:
      \forall value_type *a, *b, init, integer n;
        Unchanged{K,L}(a, n)  ==>
        Unchanged{K,L}(b, n)  ==>
        InnerProduct{K}(a, b, n, init) == InnerProduct{L}(a, b, n, init);
  }
*/
```

Listing 8.9: The logic definition(s) InnerProduct

8.3.2. Formal specification of inner_product

Using the logic function InnerProduct [8.9], we specify inner_product as shown in the following listing. Note that we needn't require that a and b are separated.

```
/*@
  requires valid:    \valid_read(a + (0..n-1));
  requires valid:    \valid_read(b + (0..n-1));
  requires bounds:   ProductBounds(a, b, n);
  requires bounds:   InnerProductBounds(a, b, n, init);
  assigns            \nothing;
  ensures result:    \result == InnerProduct(a, b, n, init);
  ensures unchanged: Unchanged{Old,Here}(a, n);
  ensures unchanged: Unchanged{Old,Here}(b, n);
*/
value_type
inner_product(const value_type* a, const value_type* b, size_type n,
              value_type init);
```

Listing 8.10: Formal specification of inner_product

8.3.3. Implementation of inner_product

The following listing shows an implementation of inner_product with corresponding loop annotations.

```
value_type
inner_product(const value_type* a, const value_type* b, size_type n,
              value_type init)
{
  /*@
     loop invariant index: 0 <= i <= n;
     loop invariant inner: init == InnerProduct(a, b, i, \at(init,Pre));
     loop assigns i, init;
     loop variant n-i;
  */
  for (size_type i = 0u; i < n; ++i) {
    /*@
      assert rte_help: init + a[i] * b[i] ==
                       InnerProduct(a, b, i+1, \at(init,Pre));
    */
    init = init + a[i] * b[i];
  }

  return init;
}
```

Listing 8.11: Implementation of inner_product

Note that the loop invariant inner claims that in the *i*-th iteration step the current value of init equals the accumulated value of Equation (8.5). This depends of course on the properties bounds in the contract of inner_product [8.10], which express that there is no arithmetic overflow when computing the updates of the variable init.

8.4. The `partial_sum` algorithm

The `partial_sum` algorithm in the C++ Standard Library [20, §29.8.6] computes the sum of a given initial value and the elements in a range. Our version of the original signature reads:

```
size_type
partial_sum(const value_type* a, size_type n, value_type* b);
```

After executing the function `partial_sum` the array `b[0..n-1]` holds the following values

$$\mathsf{b}[i] = \sum_{k=0}^{\mathtt{i}} \mathsf{a}[k] \tag{8.6}$$

for $0 \le i < n$. Equations (8.6) and (8.3) suggest that we define in the following listing the ACSL predicate `PartialSum` by using the logic function `AccumulateDefault` [8.6].

```
/*@
  axiomatic PartialSum
  {
    predicate
    PartialSum{L}(value_type* a, integer n, value_type* b) =
      \forall integer i; 0 <= i < n  ==>  b[i] == AccumulateDefault(a, i+1);

    lemma PartialSum_Section{K}:
      \forall value_type *a, *b, integer m, n;
      0 <= m <= n               ==>
      PartialSum{K}(a, n, b)   ==>
      PartialSum{K}(a, m, b);

    lemma PartialSum_Step{L}:
      \forall value_type *a, *b, integer n;
        0 <= n                              ==>
        PartialSum(a, n, b)                 ==>
        b[n] == AccumulateDefault(a, n+1)   ==>
        PartialSum(a, n+1, b);

    lemma PartialSum_Unchanged{K,L}:
      \forall value_type *a, *b, integer n;
        0 <= n  ==>
        PartialSum{K}(a, n, b)   ==>
        Unchanged{K, L}(a, n)    ==>
        Unchanged{K, L}(b, n)    ==>
        PartialSum{L}(a, n, b);

    lemma PartialSum_One{L}:
      \forall value_type *a, *b, integer n;
        b[0] == AccumulateDefault(a, 1)  ==>  PartialSum(a, 1, b);
  }
*/
```

Listing 8.12: The logic definition(s) `PartialSum`

8.4.1. Formal specification of `partial_sum`

The specification of `partial_sum` [8.13] demands that the arrays `a[0..n-1]` and `b[0..n-1]` are separated, that is, they do not overlap. Note that is a stricter requirement than in the case of the original C++ version of `partial_sum`, which allows that `a` equals `b`, thus allowing the computation of partial sums *in place*.

```
/*@
  requires valid:      \valid_read(a + (0..n-1));
  requires valid:      \valid(b + (0..n-1));
  requires sep:        \separated(a + (0..n-1), b + (0..n-1));
  requires bounds:     AccumulateDefaultBounds(a, n);
  assigns              b[0..n-1];
  ensures result:      \result == n;
  ensures partialsum: PartialSum(a, n, b);
  ensures unchanged:  Unchanged{Old,Here}(a, n);
*/
size_type
partial_sum(const value_type* a, size_type n, value_type* b);
```

Listing 8.13: Formal specification of `partial_sum`

8.4.2. Implementation of partial_sum

The following listing shows an implementation of partial_sum with corresponding loop annotations.

```
size_type
partial_sum(const value_type* a, size_type n, value_type* b)
{
  if (0u < n) {
    //@ assert limits:      AccumulateDefaultBounds(a, n);
    b[0u] = a[0u];
    //@ assert unchanged:   Unchanged{Pre,Here}(a, n);
    //@ assert limits:      AccumulateDefaultBounds(a, n);
    //@ assert accumulate:  b[0] == AccumulateDefault(a, 1);
    //@ assert partialsum:  PartialSum(a, 1, b);

    /*@
       loop invariant bound:      1 <= i <= n;
       loop invariant unchanged:  Unchanged{Pre,Here}(a, n);
       loop invariant accumulate: b[i-1] == AccumulateDefault(a, i);
       loop invariant limits:     AccumulateDefaultBounds(a, n);
       loop invariant partialsum: PartialSum(a, i, b);
       loop assigns i, b[1..n-1];
       loop variant n - i;
    */
    for (size_type i = 1u; i < n; ++i) {
      b[i] = b[i - 1u] + a[i];
      //@ assert unchanged:  Unchanged{LoopCurrent,Here}(b, i);
      //@ assert unchanged:  Unchanged{LoopCurrent,Here}(a, n);
      //@ assert partialsum: b[i] == AccumulateDefault(a, i+1);
      //@ assert limits:     AccumulateDefaultBounds(a, n);
    }
  }

  return n;
}
```

Listing 8.14: Implementation of partial_sum

8.5. The adjacent_difference algorithm

The `adjacent_difference` algorithm in the C++ Standard Library [20, §29.8.11] computes the differences of adjacent elements in a range. Our version of the original signature reads:

```
size_type
adjacent_difference(const value_type* a, size_type n, value_type* b);
```

After executing the function `adjacent_difference` the array `b[0..n-1]` holds the following values

$$\begin{aligned} \mathsf{b}[0] &= \mathsf{a}[0] \\ \mathsf{b}[1] &= \mathsf{a}[1] - \mathsf{a}[0] \\ &\vdots \\ \mathsf{b}[n-1] &= \mathsf{a}[n-1] - \mathsf{a}[n-2] \end{aligned} \tag{8.7}$$

8.5.1. The predicate AdjacentDifference

We start with the definition of the logic function `Difference` whose definition is shown in the following listing.

```
/*@
  axiomatic Difference
  {
    logic integer
    Difference{L}(value_type* a, integer n) =
      n <= 0 ? a[0] : a[n] - a[n-1];

    lemma Difference_Zero{L}:
      \forall value_type *a; Difference(a, 0) == a[0];

    lemma Difference_Next{L}:
      \forall value_type *a, integer n;
        0 < n  ==>  Difference(a, n) == a[n] - a[n-1];

    lemma Difference_Unchanged{K,L}:
      \forall value_type *a, integer n;
        0 <= n  ==>  Unchanged{K,L}(a, n+1)  ==>
        Difference{K}(a, n) == Difference{L}(a, n);
  }
*/
```

Listing 8.15: The logic definition(s) `Difference`

Building on top of Difference we now introduce the predicate AdjacentDifference. We also provide the predicate AdjacentDifferenceBounds that captures conditions that prevent numeric overflows while computing differences of the form a[i] - a[i-1].

```
/*@
  axiomatic AdjacentDifference
  {
    predicate
    AdjacentDifference{L}(value_type* a, integer n, value_type* b) =
      \forall integer i; 0 <= i < n  ==>  b[i] == Difference(a, i);

    predicate
    AdjacentDifferenceBounds(value_type* a, integer n) =
      \forall integer i; 1 <= i < n  ==>
        VALUE_TYPE_MIN <= Difference(a, i) <= VALUE_TYPE_MAX;

    lemma AdjacentDifference_Step{K,L}:
      \forall value_type *a, *b, integer n;
        AdjacentDifference{K}(a, n, b)      ==>
        Unchanged{K,L}(b, n)                ==>
        Unchanged{K,L}(a, n+1)              ==>
        \at(b[n],L) == Difference{L}(a, n)  ==>
        AdjacentDifference{L}(a, n+1, b);

    lemma AdjacentDifference_Section{K}:
      \forall value_type *a, *b, integer m, n;
        0 <= m <= n                     ==>
        AdjacentDifference{K}(a, n, b)  ==>
        AdjacentDifference{K}(a, m, b);
  }
*/
```

Listing 8.16: The logic definition(s) AdjacentDifference

Lemmas AdjacentDifference_Step [8.16] and AdjacentDifference_Section [8.16] will help us later in the verification of adjacent_difference_inv [8.22].

8.5.2. Formal specification of `adjacent_difference`

Using the predicates `AdjacentDifference` [8.16] and `AdjacentDifferenceBounds` [8.16] we can provide in the following listing a concise formal specification of `adjacent_difference`. As in the case of the specification of `partial_sum` [8.13] we require that the arrays `a[0..n-1]` and `b[0..n-1]` are separated.

```
/*@
   requires valid:      \valid_read(a + (0..n-1));
   requires valid:      \valid(b + (0..n-1));
   requires sep:        \separated(a + (0..n-1), b + (0..n-1));
   requires bounds:     AdjacentDifferenceBounds(a, n);
   assigns              b[0..n-1];
   ensures result:      \result == n;
   ensures difference:  AdjacentDifference(a, n, b);
   ensures unchanged:   Unchanged{Old,Here}(a, n);
*/
size_type
adjacent_difference(const value_type* a, size_type n, value_type* b);
```

Listing 8.17: Formal specification of `adjacent_difference`

8.5.3. Implementation of adjacent_difference

The following listing shows an implementation of adjacent_difference with corresponding loop annotations. In order to achieve the verification of the loop invariant difference we rely on

- the assertions bound and difference
- the lemmas AdjacentDifference_Step [8.16] and AdjacentDifference_Section [8.16]
- a statement contract with the two postconditions labeled as step

```
size_type
adjacent_difference(const value_type* a, size_type n, value_type* b)
{
  if (0u < n) {
    b[0u] = a[0u];

    /*@
       loop invariant index:      1 <= i <= n;
       loop invariant unchanged:  Unchanged{Pre,Here}(a, n);
       loop invariant difference: AdjacentDifference(a, i, b);
       loop assigns i, b[1..n-1];
       loop variant n - i;
    */
    for (size_type i = 1u; i < n; ++i) {
      //@ assert bound: VALUE_TYPE_MIN <= Difference(a, i) <= VALUE_TYPE_MAX;
      /*@
        assigns b[i];
        ensures step: Unchanged{Old,Here}(b, i);
        ensures step: b[i] == Difference(a, i);
      */
      b[i] = a[i] - a[i - 1u];
      //@ assert difference: AdjacentDifference(a, i+1, b);
    }
  }

  return n;
}
```

Listing 8.18: Implementation of adjacent_difference

8.6. Inverting partial_sum and adjacent_difference

In this section we show that under appropriate preconditions the algorithms partial_sum and adjacent_difference are inverse to each other.

8.6.1. Inverting partial_sum

Let a[0..n-1] and b[0..n-1] be the respective input and output of partial_sum. We have in other words

$$
\begin{aligned}
\mathsf{b}[0] &= \mathsf{a}[0] \\
\mathsf{b}[1] &= \mathsf{a}[0] + \mathsf{a}[1] \\
&\vdots \\
\mathsf{b}[n-1] &= \mathsf{a}[0] + \mathsf{a}[1] + \ldots + \mathsf{a}[n-1]
\end{aligned}
$$

If we apply now the algorithm adjacent_difference to b[0..n-1], then we find for its output a'[0..n-1]

$$
\begin{aligned}
\mathsf{a}'[0] &= \mathsf{b}[0] & &= \mathsf{a}[0] \\
\mathsf{a}'[1] &= \mathsf{b}[1] - \mathsf{b}[0] & &= \mathsf{a}[1] \\
&\vdots \\
\mathsf{a}'[n-1] &= \mathsf{b}[n-1] - \mathsf{b}[n-2] & &= \mathsf{a}[n-1]
\end{aligned}
$$

Before we start show the ACSL lemmas of our claim, we present the predicate DefaultBounds [8.19] in order to express that the values in the input (and output!) array a[0..n-1] do not overflow.

```
/*@
  axiomatic DefaultBounds
  {
    predicate
    DefaultBounds{L}(value_type* a, integer n) =
      \forall integer i; 0 <= i < n  ==>
        VALUE_TYPE_MIN <= a[i] <= VALUE_TYPE_MAX;
  }
*/
```

Listing 8.19: The logic definition(s) DefaultBounds

Lemma PartialSum_Inverse from the following listing expresses as ACSL lemmas that the algorithms partial_sum and adjacent_difference are inverse to each other.

```
/*@
  axiomatic NumericInverse
  {
    lemma PartialSum_Inverse:
      \forall value_type *a, *b, integer n;
        0 <= n              ==>
        PartialSum(a, n, b)  ==>
        AdjacentDifference(b, n, a);

    lemma AdjacentDifference_Inverse:
      \forall value_type *a, *b, integer n;
        0 <= n                      ==>
        AdjacentDifference(a, n, b)  ==>
        PartialSum(b, n, a);

    lemma AdjacentDifference_InverseBounds:
      \forall value_type *a, *b, integer n;
        0 <= n                      ==>
        DefaultBounds(a, n)          ==>
        AdjacentDifference(a, n, b)  ==>
        AccumulateDefaultBounds(b, n);
  }
*/
```

Listing 8.20: The logic definition(s) NumericInverse

The following listing now shows C function `partial_sum_inv` (both the contract and the implementation). This function calls first `partial_sum` and then `adjacent_difference`.

```
/*@
  requires valid:     \valid(a + (0..n-1));
  requires valid:     \valid(b + (0..n-1));
  requires sep:       \separated(a + (0..n-1), b + (0..n-1));
  requires bounds:    AccumulateDefaultBounds(a, n);
  requires bounds:    DefaultBounds(a, n);
  assigns             a[0..n-1], b[0..n-1];
  ensures unchanged:  Unchanged{Pre,Here}(a, n);
*/
void
partial_sum_inv(value_type* a, size_type n, value_type* b)
{
  partial_sum(a, n, b);
  adjacent_difference(b, n, a);
}
```

Listing 8.21: Implementation of partial_sum_inv

The contract of `partial_sum_inv` formulates preconditions that shall guarantee that during the computation neither arithmetic overflows (property `bounds`) nor unintended aliasing of arrays (property `sep`) occur. Under these precondition, Frama-C shall verify that the final call to `adjacent_difference` [8.17] just restores the original contents of `a[0..n-1]` that we supplied for the initial call to `partial_sum` [8.13].

8.6.2. Inverting adjacent_difference

After executing the function adjacent_difference [8.17] on the input array a[0..n-1] the output array b[0..n-1] holds the following values

$$
\begin{aligned}
b[0] &= a[0] \\
b[1] &= a[1] - a[0] \\
&\vdots \\
b[n-1] &= a[n-1] - a[n-2]
\end{aligned}
$$

If we call now partial_sum with the array b[0..n-1] as input, then we obtain for its output a'[0..n-1]

$$
\begin{aligned}
a'[0] &= b[0] &&= a[0] \\
a'[1] &= b[0] + b[1] &&= a[1] \\
&\vdots \\
a'[n-1] &= b[0] + b[1] + \ldots + b[n-1] &&= a[n-1]
\end{aligned}
$$

which means that applying partial_sum [8.13] on the output of adjacent_difference produces the original input. Lemma AdjacentDifference_Inverse [8.20] expresses this property as a lemma.

The function adjacent_difference_inv [8.22] first calls adjacent_difference and then partial_sum. The contract of this function formulates preconditions that shall guarantee that during the computation neither arithmetic overflows (property bound) nor unintended aliasing of arrays (property sep) occur. In order to improve the automatic verification of adjacent_difference_inv we also use lemma Unchanged_Transitive [7.3]. Lemma AdjacentDifference_InverseBounds [8.20] simplifies the verification of the precondition bounds of partial_sum.

```
/*@
  requires size:      0 <= n;
  requires valid:     \valid(a + (0..n-1));
  requires valid:     \valid(b + (0..n-1));
  requires sep:       \separated(a + (0..n-1), b + (0..n-1));
  requires bounds:    DefaultBounds(a, n);
  requires bounds:    AdjacentDifferenceBounds(a, n);
  assigns             a[0..n-1], b[0..n-1];
  ensures unchanged:  Unchanged{Old,Here}(a, n);
*/
void
adjacent_difference_inv(value_type* a, size_type n, value_type* b)
{
  adjacent_difference(a, n, b);
  partial_sum(b, n, a);
}
```

Listing 8.22: Implementation of adjacent_difference_inv

Part IV.

Sorting algorithms

9. Heap Algorithms

The heap algorithms of the C++ Standard Library [20, 28.7.7] were already part of *ACSL by Example* from 2010–2012. In this chapter we re-introduce them and discuss—based on the bachelor thesis of one of the authors—the verification efforts in some detail [24].

The C++ standard[26] introduces the concept of a *heap* as follows:

1. A *heap* is a particular organization of elements in a range between two random access iterators `[a,b)`. Its two key properties are:
   a) There is no element greater than `*a` in the range and
   b) `*a` may be removed by `pop_heap()`, or a new element added by `push_heap()`, in $O(\log(N))$ time.
2. These properties make heaps useful as priority queues.
3. `make_heap()` converts a range into a heap and `sort_heap()` turns a heap into an increasing sequence.

Figure 9.1 gives an overview on the five heap algorithms by means of an example. Algorithms, which in a typical implementation are in a caller-callee relation, have the same color.

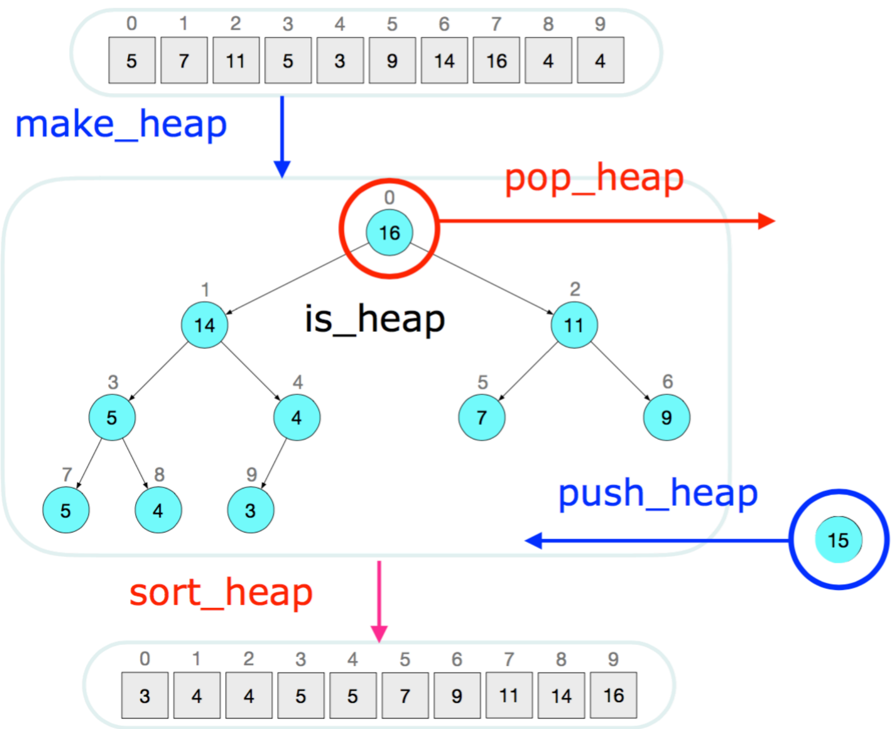

Figure 9.1.: Overview on heap algorithms

[26] See `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf`

Roughly speaking, the algorithms from Figure 9.1 have the following behavior.

- In §9.1 we briefly recapitulate basic heap concepts.
- In §9.2 we show how these heap concepts can be described in ACSL.
- In §9.3 we verify two auxiliary heap functions.
- The algorithms `is_heap_until` and `is_heap` from §9.4 and §9.5 allow to test at run time whether a given array is arranged as a heap
- The algorithm `push_heap` from §9.7 *adds* an element to a given heap in such a way that resulting array is again a heap
- The algorithm `pop_heap` from §9.8, on the other hand, *removes* an element from a given heap in such a way that the resulting array is again a heap
- The algorithm `make_heap` from §9.9 rearranges a given array into a heap.
- Finally, the algorithm `sort_heap` from §9.10 sorts a heap into an increasing range.

In §9.1 we present in more detail how heaps are defined. The ACSL logic functions and predicate that formalize the basic heap properties of heaps are introduced in §9.2.

9.1. Basic heap concepts

The description of heaps at the beginning of this chapter is of course fairly vague. It outlines only the most important properties of various operations but does not clearly state what specific and verifiable properties a range must satisfy such that it may be called a heap.

A more detailed description can be found in the Apache C++ Standard Library User's Guide:[27]

> A heap is a binary tree in which every node is larger than the values associated with either child. A heap and a binary tree, for that matter, can be very efficiently stored in a vector, by placing the children of node i at positions $2i + 1$ and $2i + 2$.

We have, in other words, the following basic relations between indices of a heap:

$$\begin{aligned} &\text{left child for index } i && \mathrm{child_l} : i \mapsto 2i + 1 && (9.1)\\ &\text{right child for index } i && \mathrm{child_r} : i \mapsto 2i + 2 && (9.2) \end{aligned}$$

and

$$\text{parent index for index } i \qquad \mathrm{parent} : i \mapsto \frac{i-1}{2} \qquad (9.3)$$

These function are related through the following two equations that hold for all integers i. Note that in ACSL integer division rounds towards zero (cf. [15, §2.2.4]).

$$\begin{aligned} \mathrm{parent}(\mathrm{child_l}(i)) &= i && (9.4)\\ \mathrm{parent}(\mathrm{child_r}(i)) &= i && (9.5) \end{aligned}$$

In order to given an example for the usefulness of heaps we consider the following multiset of integers X.

$$X = \{2, 3, 3, 3, 6, 7, 8, 8, 9, 11, 13, 14\} \qquad (9.6)$$

[27] See `http://stdcxx.apache.org/doc/stdlibug/14-7.html`

Figure 9.2 shows how the multiset from Equation (9.6) can, according to the parent-child relations of a heap, be represented as a tree.

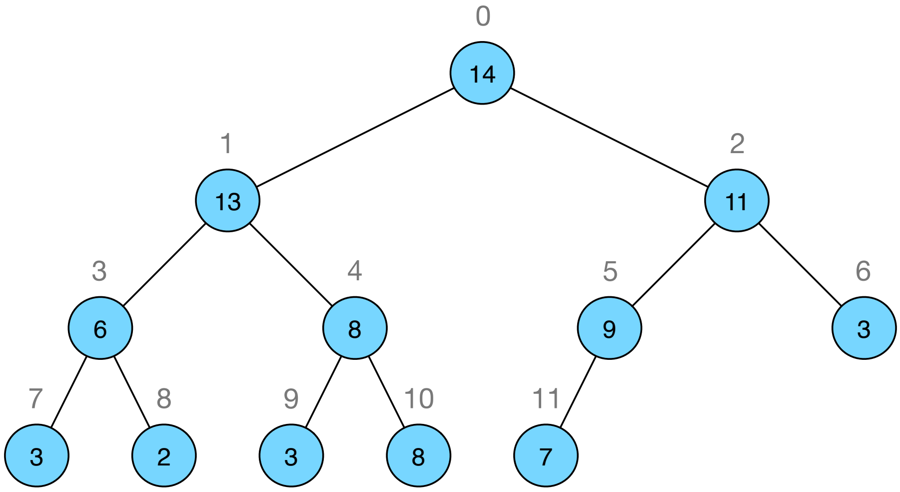

Figure 9.2.: Tree representation of the multiset X

The numbers outside the nodes in Figure 9.2 are the indices at which the respective node value is stored in the underlying array of a heap (cf. Figure 9.3).

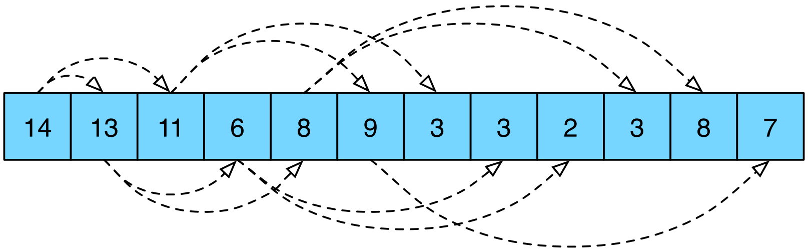

Figure 9.3.: Underlying array of a heap

It is important to understand that there can be various representations of a multiset as a heap. Figure 9.4, for example, arranges the elements of the multiset X as a heap in a different tree.

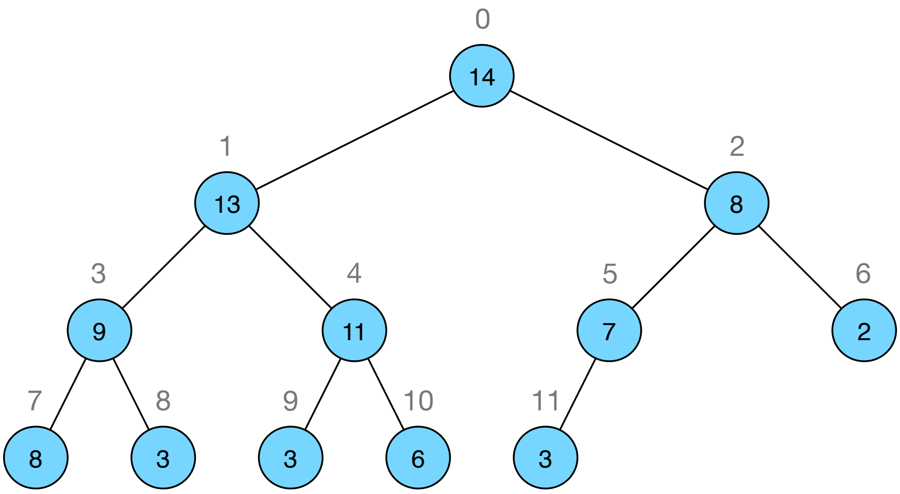

Figure 9.4.: An alternative representation of the multiset X

Figure 9.5 then shows the underlying array that corresponds to the tree in Figure 9.4.

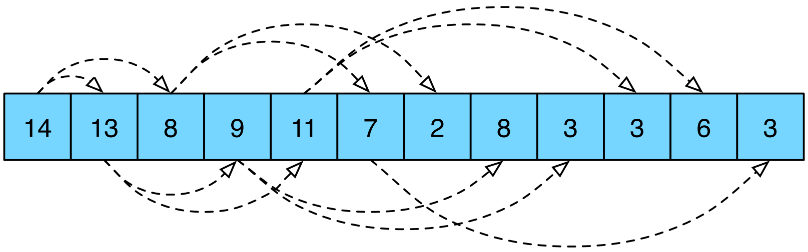

Figure 9.5.: Underlying array of the alternative representation

9.2. Representation of heap concepts in ACSL

The following listing shows three logic functions `HeapLeft`, `HeapRight` and `HeapParent` that correspond to the definitions (9.1), (9.2) and (9.3), respectively. This listing also contains a number of ACSL lemma that state among other things that

- the `HeapParent` function satisfies the equations (9.4) and (9.5) and
- the function `HeapParent` is the *left inverse* to the `HeapLeft` and `HeapRight` functions.[28]

```
/*@
  axiomatic HeapNodes
  {
    logic integer HeapLeft(integer i) = 2*i + 1;

    logic integer HeapRight(integer i) = 2*i + 2;

    logic integer HeapParent(integer i) = (i-1) / 2;

    lemma HeapParent_Zero{L}: HeapParent(0) == 0;

    lemma Heap_ParentLeft:
      \forall integer p; 0 <= p  ==>  HeapParent(HeapLeft(p)) == p;

    lemma Heap_ParentRight:
      \forall integer p; 0 <= p  ==>  HeapParent(HeapRight(p)) == p;

    lemma Heap_ParentChild:
      \forall integer c, p;
        0 < c  ==>  HeapParent(c) == p  ==>
        (c == HeapLeft(p) || c == HeapRight(p));

    lemma Heap_Childs:
      \forall integer a, b;
        0 < a  ==>  0 < b                ==>
        HeapParent(a) == HeapParent(b)  ==>
        (a == b || a+1 == b || a == b+1);

    lemma Heap_ParentBounds:
      \forall integer c; 0 < c  ==>  0 <= HeapParent(c) < c;

    lemma Heap_ChildBounds:
      \forall integer p; 0 <= p  ==>  p < HeapLeft(p) < HeapRight(p);
  }
*/
```

Listing 9.6: The logic definition(s) `HeapNodes`

[28] See Section *Left and right inverses* at `http://en.wikipedia.org/wiki/Inverse_function`

On top of these basic definitions we introduce the predicate `Heap` [9.7]. The fact that element at index 0 of a (maximum) heap, is always the largest element of the heap is express by Lemma `Heap_Maximum` [9.7] using the predicate `MaxElement` [5.2].

```
/*@
  axiomatic Heap
  {
    predicate
    Heap{L}(value_type* a, integer n) =
      \forall integer i; 0 < i < n  ==>  a[i] <= a[HeapParent(i)];

    lemma Heap_Maximum{L} :
      \forall value_type* a, integer n;
        0 < n  ==>  Heap(a, n)  ==>  MaxElement(a, n, 0);

   lemma Heap_Shrink{L}:
     \forall value_type *a, integer m, n;
       0 <= m <= n  ==>  Heap(a, n)   ==>  Heap(a, m);

   lemma Heap_Unchanged{K,L}:
     \forall value_type *a, integer n;
       0 <= n  ==>  Unchanged{K,L}(a, n)  ==>  Heap{K}(a, n)  ==>  Heap{L}(a, n);

   predicate
   HeapCompatible{L}(value_type* a, integer n, integer m, value_type v) =
     (0 <= m < n)                                       &&
     (0 <= HeapParent(m)  ==>  v <= a[HeapParent(m)])   &&
     (HeapLeft(m)  <  n   ==>  a[HeapLeft(m)]  <= v)    &&
     (HeapRight(m) <  n   ==>  a[HeapRight(m)] <= v);

    lemma HeapCompatible_Update{K,L}:
      \forall value_type *a, v, integer m, n;
         0 <= m < n                         ==>
         Heap{K}(a, n)                      ==>
         HeapCompatible{K}(a, n, m, v)      ==>
         ArrayUpdate{K,L}(a, n, m, v)       ==>
         Heap{L}(a, n);

  }
*/
```

Listing 9.7: The logic definition(s) `Heap`

The lemmas `Heap_Shrink` and `Heap_Unchanged` formulate simple rules to “transfer” the heap property from an array to a related (sub-)array.

The predicate `HeapCompatible` expresses under which conditions the changing of an individual heap element does maintain the heap property. This predicate together with lemma `HeapCompatible_Update` will be useful in the verification of the algorithms `push_heap` [9.22] and `pop_heap` [9.29].

9.3. The auxiliary functions heap_parent and heap_child

This section features the two auxiliary heap functions We start with the function heap_parent [9.8] which is in principle the C counterpart of the ACSL function HeapParent [9.6]. We say *in principle* because our definition avoids the border case of the parent node of 0.

```
/*@
   assigns            \nothing;
   ensures   parent:  \result == HeapParent(child);
 */
static inline size_type
heap_parent(size_type child)
{
  return (0u < child) ?  (child - 1u) / 2u : 0u;
}
```

Listing 9.8: Formal specification of heap_parent

Neither do we provide exact C-counterparts for the logic functions HeapLeft [9.6] and HeapRight [9.6]. In fact, we have encountered only one situation (in the implementation of pop_heap [9.29]), where such functions would have been useful. However, what we really need in pop_heap is to determine for a given index p a child index c where the maximum of the respective values a[HeapLeft(p)] and a[HeapRight(p)] resides. This computation is performed by the function heap_child [9.9].

```
/*@
   requires bounds:  0 <= p < n;
   requires valid:   \valid(a + (0..n-1));
   assigns           \nothing;
   ensures  bounds:  p < \result <= n;
   ensures  parent:  \result < n       ==>  p == HeapParent(\result);
   ensures  parent:  \result < n-1     ==>  HeapLeft(p)  < n-1;
   ensures  parent:  \result < n-1     ==>  HeapRight(p) < n;
   ensures  left:    HeapLeft(p)  < n  ==>  \result < n;
   ensures  right:   HeapRight(p) < n  ==>  \result < n;
   ensures  max:     HeapLeft(p)  < n  ==>  a[HeapLeft(p)]  <= a[\result];
   ensures  max:     HeapRight(p) < n  ==>  a[HeapRight(p)] <= a[\result];
   ensures  none:    \result == n      ==>  n <= HeapLeft(p);
   ensures  none:    \result == n      ==>  n <= HeapRight(p);
*/
size_type
heap_child(const value_type* a, size_type n, size_type p);
```

Listing 9.9: Formal specification of heap_child

Note that in the implementation of heap_child [9.10] we explicitly handle the case that the computation of child indices could overflow. If this occurs, the function heap_child returns n.

```
size_type
heap_child(const value_type* a, size_type n, size_type p)
{
  if (p + 1u <= n - p - 1u) {
    const size_type left  = 2u * p + 1u;
    const size_type right = left + 1u;

    if (right < n) {
      // case of two children: select child with maximum value
      return a[right] <= a[left] ? left : right;
    }
    else {
      // at most one child that comes before n-1 can exist
      return left;
    }
  }
  else {
    return n;
  }
}
```

Listing 9.10: Implementation of heap_child

9.4. The is_heap_until algorithm

The is_heap_until algorithm of the C++ Standard Library [20, §28.7.7.5] works on generic sequences. For our purposes we have modified the generic implementation to that of an array of type value_type. The signature now reads:

```
size_type is_heap_until(const value_type* a, int n);
```

The algorithm is_heap_until returns the largest range of an array, beginning at the first position, where it still satisfies the heap properties we have semi-formally described in the beginning of this chapter. In particular, is_heap_until will return the size of the array, called with the array argument from Figure 9.3.

9.4.1. Formal specification of `is_heap_until`

The specification of `is_heap_until` is shown in the following listing. The index `\result` returned by `is_heap_until` indicates that the array `a[0..\result-1]` is a heap. In addition the postcondition `last` states, that for all indices greater than or equal to `i` the predicate `Heap` [9.7] is not satisfied.

```
/*@
   requires valid: \valid_read(a + (0..n-1));
   assigns         \nothing;
   ensures bound:  0 <= \result <= n;
   ensures heap:   Heap(a, \result);
   ensures last:   \forall integer i; \result < i <= n ==> !Heap(a, i);
*/
size_type
is_heap_until(const value_type* a, size_type n);
```

Listing 9.11: Formal specification of `is_heap_until`

9.4.2. Implementation of `is_heap_until`

The following listing shows one way to implement the function `is_heap_until`.

```
size_type
is_heap_until(const value_type* a, size_type n)
{
  size_type parent = 0u;

  /*@
    loop invariant bound:    0 <= parent < child <= n+1;
    loop invariant parent:   parent == HeapParent(child);
    loop invariant heap:     Heap(a, child);
    loop invariant not_heap: a[parent] < a[child] ==> \forall integer i; child < i
        <= n ==> !Heap(a, i);

    loop assigns child, parent;
    loop variant n - child;
  */
  for (size_type child = 1u; child < n; ++child) {
    if (a[parent] < a[child]) {
      return child;
    }

    if ((child % 2u) == 0u) {
      ++parent;
    }
  }

  return n;
}
```

Listing 9.12: Implementation of `is_heap_until`

The algorithms starts at the index 1, which is the smallest index, where a child node of the heap might reside. The algorithms checks for each (child) index whether the value at the corresponding parent index is greater than or equal to the value at the child index. If the value at a parent index is smaller than the value at a (child) index, is_heap_until returns the (child) index. Otherwise, if the algorithm iterates through the whole array, the size of the array is returned.

9.5. The is_heap algorithm

The is_heap algorithm of the C++ Standard Library [20, §28.7.7.5] works on generic sequences. For our purposes we have modified the generic implementation to that of an array of type value_type. The signature now reads:

```
bool is_heap(const value_type* a, int n);
```

The algorithm is_heap checks whether a given array satisfies the heap properties we have semi-formally described in the beginning of this chapter. In particular, is_heap will return **true** called with the array argument from Figure 9.3.

9.5.1. Formal specification of is_heap

The specification of is_heap is shown in the following listing. The function returns **true** if and only if the input array satisfies the predicate Heap [9.7].

```
/*@
   requires valid: \valid_read(a +(0..n-1));
   assigns         \nothing;
   ensures heap:   \result <==> Heap(a, n);
*/
bool
is_heap(const value_type* a, size_type n);
```

Listing 9.13: Formal specification of is_heap

9.5.2. Implementation of is_heap

Our implementation of is_heap in the following listing utilizes the function is_heap_until [9.11].

```
bool
is_heap(const value_type* a, size_type n)
{
  return is_heap_until(a, n) == n;
}
```

Listing 9.14: Implementation of is_heap

9.6. Reorderings and fluctuations

One particular challenge posed by heap algorithms is that while temporarily causing *small fluctuations* in the number of values within an array they essentially only *reorder* it, that is they leave the multiset of its values unchanged. In this section we will introduce various predicates that will help us mastering this challenge.

9.6.1. Formalizing small fluctuations

The predicate `MultisetAdd` in the following listing expresses that the number of occurrences of a specific element in an array has increased by one between two program points `K` and `L`.

```
/*@
  axiomatic MultisetOperations
  {
    predicate
    MultisetAdd{K,L}(value_type* a, integer n, value_type v) =
      Count{L}(a, 0, n, v) == Count{K}(a, 0, n, v) + 1;

    predicate
    MultisetMinus{K,L}(value_type* a, integer n, value_type v) =
      MultisetAdd{L,K}(a, n, v);

    predicate
    MultisetRetain{K,L}(value_type* a, integer n, value_type v) =
      Count{K}(a, 0, n, v) == Count{L}(a, 0, n, v);

    lemma MultisetAdd_Distinct{K,L}:
      \forall value_type *a, v, integer m, n;
        0 <= m < n                        ==>
        At{K}(a, m) != v                  ==>
        At{L}(a, m) == v                  ==>
        MultisetReorder{K,L}(a, 0, m)     ==>
        MultisetReorder{K,L}(a, m+1, n)   ==>
        MultisetAdd{K,L}(a, n, v);

    lemma MultisetMinus_Distinct{K,L}:
      \forall value_type *a, v, integer m, n;
        0 <= m < n                        ==>
        At{K}(a, m) == v                  ==>
        At{L}(a, m) != v                  ==>
        MultisetReorder{K,L}(a, 0, m)     ==>
        MultisetReorder{K,L}(a, m+1, n)   ==>
        MultisetMinus{K,L}(a, n, v);

    lemma MultisetRetain_Distinct{K,L}:
      \forall value_type *a, v, integer m, n;
        0 <= m < n                        ==>
        At{K}(a, m) != v                  ==>
        At{L}(a, m) != v                  ==>
        MultisetReorder{K,L}(a, 0, m)     ==>
        MultisetReorder{K,L}(a, m+1, n)   ==>
        MultisetRetain{K,L}(a, n, v);
  }
*/
```

Listing 9.15: The logic definition(s) `MultisetOperations`

The predicate MultisetMinus, on the other hand, expresses that the number of occurrences of a specific element in an array has decreased by one between two program points K and L. Note that we have defined MultisetMinus by calling MultisetAdd with the labels reversed. Finally, the predicate MultisetRetain expresses that a the number of occurrences of a given value does not change between two program points. In order to guide the automatic provers, we also provide some lemmas that formalize conditions under which the respective predicates hold.

Using the predicate MultisetReorder [7.55] and the logic function At [7.49] we also formulate a few simple lemmas that describe when the predicates from Listing MultisetOperations [9.15] hold.

9.6.2. Simple properties of fluctuations

The predicate MultisetRetainRest [9.16] uses MultisetRetain [9.15] in order to express that all values of an array, except the two given values u and v, occur as often in program point K and program point L.

The lemmas in this listing express conditions under which small fluctuations—expressed by the predicates MultisetAdd [9.15] and MultisetMinus [9.15]—in the number of occurrences between three program points even with each other.

```
/*@
  axiomatic MultisetRetainRest
  {
    predicate
    MultisetRetainRest{K,L}(value_type* a, integer n, value_type  v, value_type w) =
      \forall value_type x;
        x != v  ==>  x != w  ==>  MultisetRetain{K,L}(a, n, x);

    lemma Multiset_AddMinusRetain{K,L,M}:
      \forall value_type *a, u, integer n;
        MultisetAdd{K,L}(a, n, u)    ==>
        MultisetMinus{L,M}(a, n, u)  ==>
        MultisetRetain{K,M}(a, n, u);

    lemma Multiset_MinusAddRetain{K,L,M}:
      \forall value_type *a, u, integer n;
        MultisetMinus{K,L}(a, n, u) ==>
        MultisetAdd{L,M}(a, n, u)   ==>
        MultisetRetain{K,M}(a, n, u);

    lemma Multiset_AddMinusRetainReorder{K,L,M}:
      \forall value_type *a, u, v, integer n;
        u != v                              ==>
        MultisetAdd{K,L}(a, n, u)           ==>
        MultisetMinus{K,L}(a, n, v)         ==>
        MultisetRetainRest{K,L}(a, n, u, v)  ==>
        MultisetAdd{L,M}(a, n, v)           ==>
        MultisetMinus{L,M}(a, n, u)         ==>
        MultisetRetainRest{L,M}(a, n, v, u)  ==>
        MultisetReorder{K,M}(a, n);
  }
*/
```

Listing 9.16: The logic definition(s) MultisetRetainRest

9.6.3. Combining fluctuations

Small fluctuations are so prevalent in the central heap algorithms `push_heap` [9.22] and `pop_heap` [9.29] that it is worthwhile to introduce another predicate to concisely capture this feature. We refer to this predicate as `MultisetParity` [9.17] because it describes the situation where the number of occurrences

- of the first of two given values increases by one
- while that of the second value decreases by one
- and the remaining values retain their respective number of occurrences.

With this predicate we can formulate several lemmas that describe useful combinations of reorderings and fluctuations. For example, lemma `MultisetParity_MultisetReorder` [9.17] describes the situation where two fluctuation cancel each other and consequently establish a reordering of an array.

```
/*@
  axiomatic MultisetParity
  {
    predicate
      MultisetParity{K,L}(value_type* a, integer n, value_type u, value_type v) =
        MultisetAdd{K,L}(a, n, u)     &&
        MultisetMinus{K,L}(a, n, v)   &&
        MultisetRetainRest{K,L}(a, n, u, v);


    lemma MultisetParity_UnchangedFirst{K,L,M}:
      \forall value_type *a, u, v, integer n;
        u != v                          ==>
        Unchanged{K,L}(a, n)            ==>
        MultisetParity{L,M}(a, n, u, v)  ==>
        MultisetParity{K,M}(a, n, u, v);

    lemma MultisetParity_UnchangedSecond{K,L,M}:
      \forall value_type *a, u, v, integer n;
        u != v                          ==>
        MultisetParity{K,L}(a, n, u, v)  ==>
        Unchanged{L,M}(a, n)            ==>
        MultisetParity{K,M}(a, n, u, v);

    lemma MultisetParity_MultisetReorder{K,L,M}:
      \forall value_type *a, u, v, integer n;
        u != v                          ==>
        MultisetParity{K,L}(a, n, u, v)  ==>
        MultisetParity{L,M}(a, n, v, u)  ==>
        MultisetReorder{K,M}(a, n);

    lemma MultisetParity_Combined{K,L,M}:
      \forall value_type *a, u, v, w, integer n;
        u != v                          ==>
        u != w                          ==>
        v != w                          ==>
        MultisetParity{K,L}(a, n, u, v)  ==>
        MultisetParity{L,M}(a, n, w, u)  ==>
        MultisetParity{K,M}(a, n, w, v);
  }
*/
```

Listing 9.17: The logic definition(s) `MultisetParity`

9.6.4. How do fluctuations arise?

The simplest way to creation a small fluctuation is to update an array element with a different value. Thus, similar to the predicate `ArrayUpdate` [7.2] we introduce predicate `MultisetUpdate` [9.18] which in turn relies on `MultisetParity` [9.17]. Lemma `ArrayUpdate_MultisetUpdate` [9.18] then formalizes the claim that updating an array element with a different value creates a small fluctuation.

```
/*@
  axiomatic MultisetUpdate
  {
    predicate
    MultisetUpdate{K,L}(value_type* a, integer n, integer i, value_type v) =
      \let u = At{K}(a, i);
      u != v                            &&
      0 <= i < n                        &&
      MultisetReorder{K,L}(a, 0, i)     &&
      MultisetReorder{K,L}(a, i+1, n)   &&
      MultisetParity{K,L}(a, n, v, u);

    lemma ArrayUpdate_MultisetUpdate{K,L}:
      \forall value_type *a, v, integer n, i;
        ArrayUpdate{K,L}(a, n, i, v)  ==>  MultisetUpdate{K,L}(a, n, i, v);
  }
*/
```

Listing 9.18: The logic definition(s) `MultisetUpdate`

9.7. The push_heap algorithm

The push_heap algorithm assumes that the first $n - 1$ elements of an array of length n form already a heap and adds to it the element a[n-1].

Whereas in the C++ Standard Library [20, §28.7.7.1] push_heap works on a range of random access iterators, our version operates on an array of value_type. We therefore use the following signature for push_heap

```
void push_heap(value_type* a, size_type n);
```

The push_heap algorithm expects that n is greater or equal than 1. It also assumes that the array a[0..n-2] forms a heap. The algorithms then *rearranges* the array a[0..n-1] such that the resulting array is a heap. In this sense the algorithm *pushes* the element a[n-1] on the given heap.

9.7.1. Formal specification of push_heap

The following listing shows our specification of push_heap. Note that the post condition reorder states that push_heap is not allowed to change the number of occurrences of an array element. Without this post condition, an implementation that assigns 0 to each array element would satisfy the post condition heap—surely not what a user of the algorithm has in mind.

```
/*@
   requires nonempty:   0 < n;
   requires valid:      \valid(a + (0..n-1));
   requires heap:       Heap(a, n-1);
   assigns              a[0..n-1];
   ensures  heap:       Heap(a, n);
   ensures  reorder:    MultisetReorder{Old,Here}(a, n);
*/
void
push_heap(value_type* a, size_type n);
```

Listing 9.19: Formal specification of push_heap

Pushing an element on a heap usually *rearranges* several elements of the array (cf. Figures 9.20 and 9.21). We therefore must be able express that push_heap only reorders the elements of the array. We re-use the predicate MultisetReorder [7.55] to formally describe this property.

9.7.2. Implementation of push_heap

The following two figures illustrate how push_heap affects an array, which is shown as a tree with blue and grey nodes, representing heap and non-heap nodes, respectively. Figure 9.20 shows the heap from Figure 9.2 together with the additional element 12 that is to be pushed on the heap. To be quite clear about it: the new element 12 is the last element of the array and not yet part of the heap.

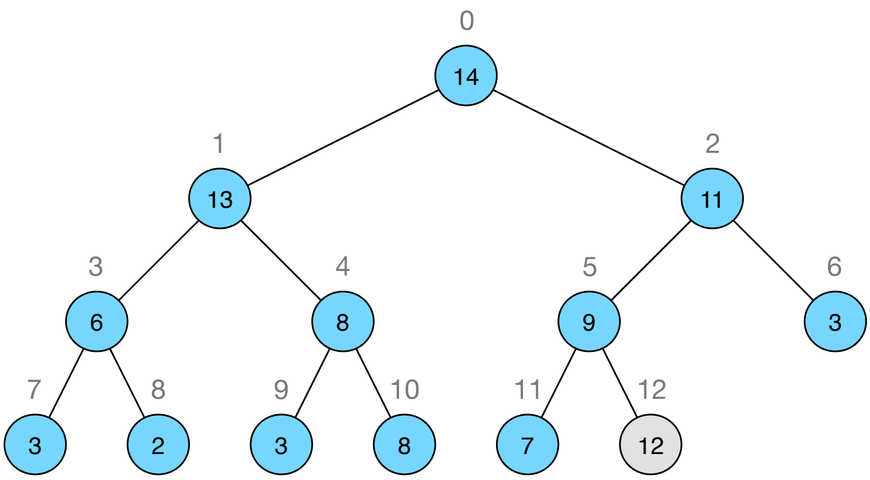

Figure 9.20.: Example heap before the call of push_heap

Figure 9.21 shows the array after the call of push_heap. We can see that now all nodes are colored in blue, i.e., they are part of the heap. The dashed nodes highlight which heap nodes have changed during the function call. The element to be pushed into the heap is now at its correct position. The arrows indicate the *cyclic reordering* of array elements to achieve the desired result.

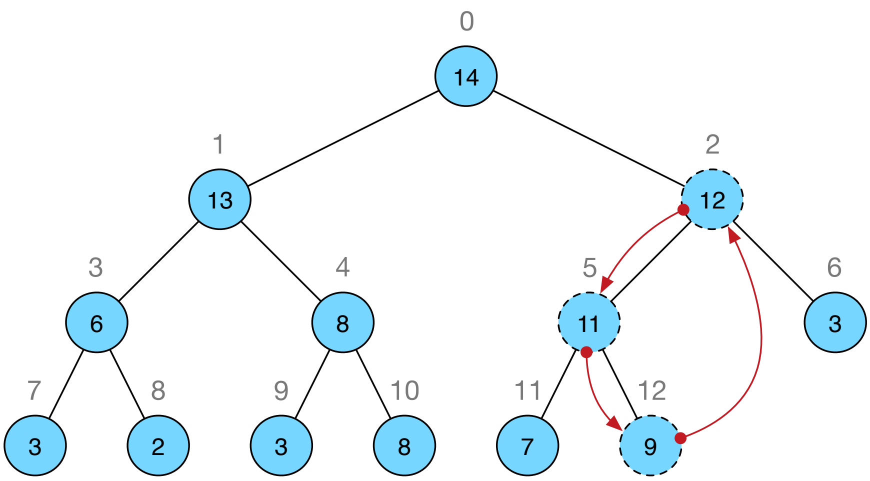

Figure 9.21.: Example heap after the call of push_heap

Verifying our implementation of push_heap [9.22] is a non-trivial undertaking. In order to better structure our discussion we refer to the central loop of the algorithm as the *main act* and the parts before and after it as *prologue* and *epilogue*.

We can establish the heap property of push_heap [9.19] already in the prologue. The reorder property, however, only holds at the function boundaries and is violated while push_heap manipulates the array. To be more precise: We loose the reorder property in the prologue and formally capture and maintain a slightly more general property in the main act. From this we will recover the reorder property in the epilogue.

We will illustrate the changes to the underlying array after each stage by figures of the array in tree form, based on the push_heap example from Figure 9.20.

```
void
push_heap(value_type* a, size_type n)
{
  if (1u < n) { // otherwise nothings needs to be done
    size_type c = n - 1u;
    size_type p = heap_parent(c);
    //@ assert parent: p == HeapParent(c);

    if (a[p] < a[c]) {
      const value_type v  = a[c];
      a[c] = a[p];
      //@ assert update:   ArrayUpdate{Pre,Here}(a, n, c, a[p]);
      //@ assert heap:     Heap(a, n);
      //@ assert reorder:  MultisetParity{Pre,Here}(a, n, a[p], v);

      /*@
        loop invariant bound:      0 <= c < n-1;
        loop invariant heap:       Heap(a, n);
        loop invariant less:       a[c] < v;
        loop invariant parent:     p == HeapParent(c);
        loop invariant reorder:    MultisetParity{Pre,Here}(a, n, a[c], v);
        loop invariant unchanged:  Unchanged{Pre,Here}(a, c);
        loop assigns               c, p, a[0..n-1];
        loop variant               c;
      */
      for (c = p, p = heap_parent(c); 0u < c && a[p] < v;
           c = p, p = heap_parent(c)) {
        //@ ghost value_type ac =  a[c];
        if (a[c] < a[p]) {
          a[c] = a[p];
          //@ assert update:     ArrayUpdate{LoopCurrent,Here}(a, n, c, a[p]);
          //@ assert update:     MultisetUpdate{LoopCurrent,Here}(a, n, c, a[p]);
          //@ assert bound:      0 <= c < n;
          //@ assert less:       ac < a[c] < v;
          //@ assert reorder:    MultisetParity{Pre,Here}(a, n, a[c], v);
        }
      }

      //@ ghost Epilogue: ;
      //@ assert heap: 0 == c || v <= a[HeapParent(c)];
      //@ ghost  value_type  ac = a[c];
      //@ assert update:     ac == At{Epilogue}(a, c) < v;
      //@ assert reorder:    MultisetParity{Pre,Here}(a, n, ac, v);
      a[c] = v;
      //@ assert update:   ArrayUpdate{Epilogue,Here}(a, n, c, v);
      //@ assert heap:     HeapCompatible(a, n, c, v);
      //@ assert heap:     Heap(a, n);
      //@ assert update:   MultisetUpdate{Epilogue,Here}(a, n, c, v);
      //@ assert reorder:  MultisetParity{Epilogue,Here}(a, n, v, ac);
      //@ assert reorder:  MultisetReorder{Pre,Here}(a, n);
    }
  }
}
```

Listing 9.22: Implementation of push_heap

Prologue

In the prologue we check whether the initial heap is nonempty, initialize some variables, *and* also check by comparing with the parent node whether `a[c]`, which is the value to be pushed on the heap and which is the last element of the array, is by chance already at the right place. If not we set aside this value in the variable `v` and assign the parent value `a[p]` to `a[c]`. Note that this assignment only occurs if the respective values differ. This allows us to formally describe the effect of the assignment using the predicate `ArrayUpdate` [7.2]. Figure 9.23 highlights the main effects of the prologue. Here and in the following figures we highlight the currently active node.

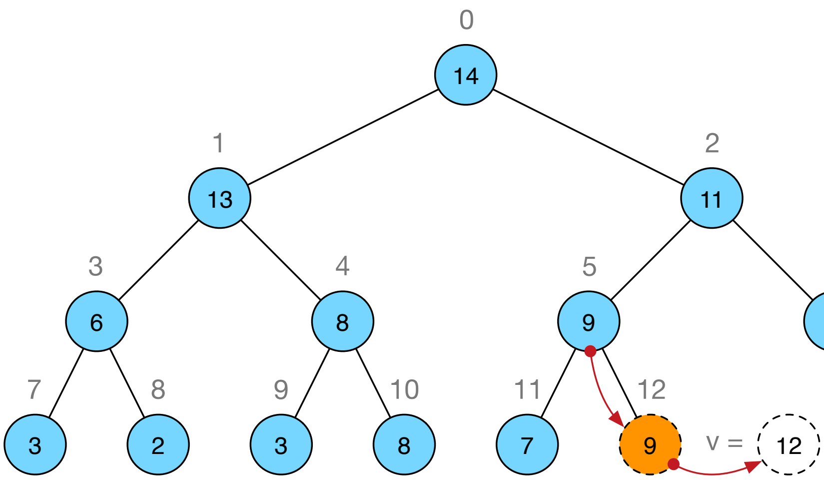

Figure 9.23.: Heap after the prologue of `push_heap`

At this point we have achieved several things.

1. The array `a[0..n-1]` is now a heap.
2. Regarding their respective number of occurences in the array `a[0..n-1]`
   - the original value `a[c]` occurs one time less
   - the original value `a[p]` occurs one time more
   - whereas all other values have not changed their number of occurences.

The first observation is expressed in the assertion `heap` whereas the small fluctuation of array elements described in the second observation is expressed by using the predicate `MultisetParity` [9.17] in the assertion `reorder`.

Main act

In the main act, we start at the parent location, which is now stored in the variable `c` (*child*). Compared to the pre-state of `push_heap` at the beginning of the main act the array `a[0..n-1]`

- contains the value `v` one time less
- contains the value `a[c]` one time more
- whereas all other values have not changed their number of occurences.

Now, as long as the index `c` is not yet the root of the heap and its consequently existing parent value `a[p]` is less than `v`, we haven't found yet an index `c` where we could insert `v` without violating the heap property.

In the loop body we proceed as follows.

- If `a[c]` is less than `a[p]` we copy the latter value on the former. Note that this assignment preserves the heap property of the array. The value `a[p]` now occurs one time more than in the pre-state whereas the now overwritten value `a[c]` occurs as often as in the pre-state. The value `v` continues to occur one time less. We then proceed to the next iteration by setting `c` to `p`.

  The verification of tracking the number of occurences happens in smaller steps than just described. It relies on the predicates `ArrayUpdate` [7.2] and `MultisetUpdate` [9.18] which we can apply in this guarded assignment. Lemma `MultisetParity_Combined` [9.17] also plays an important role here.

- Otherwise, since `c` is a child of `p`, we can conclude that `a[c]` equals `a[p]` and we continue with the next iteration after setting `c` to `p`.

This means that at the begin of the next iteration again the following conditions hold. Compared to the pre-state of `push_heap` the array `a[0..n-1]`

- contains the value `v` one time less
- contains the value `a[c]` one time more
- whereas all other values have not changed their number of occurences.

Figure 9.24 shows the our example heap after the main act. For this particular heap, only one iteration is performed until a node is reached whose parent value `a[p]` is greater or equal than `v`. Note that assignments which have previously occurred are marked with dashed arrows.

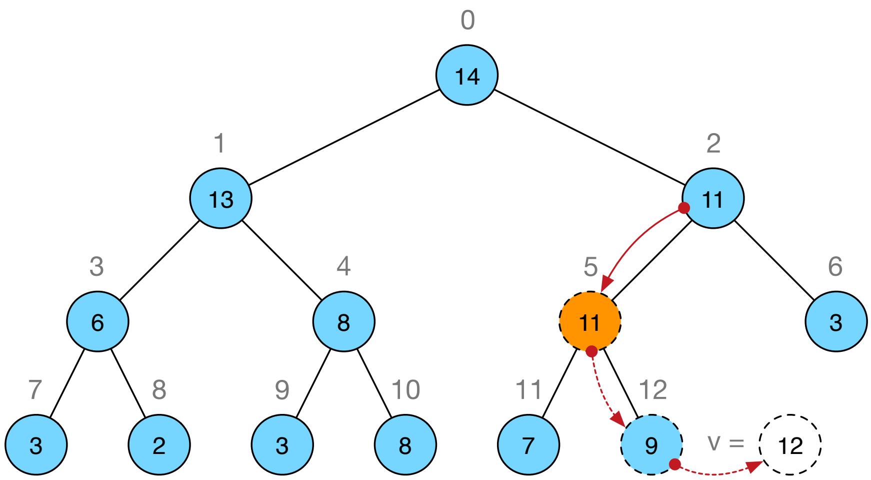

Figure 9.24.: Heap after the main act of `push_heap`

Epilogue

At this point, we have arrived at an index `c` where the assignment of the value `v` preserves the heap property. We express this formally using the predicate `HeapCompatible` [9.7].

Moreover, this assignment also corrects the imbalance in the number of occurences of the values `a[c]` and `v` and consequently establishes the desired property `reorder` of `push_heap`. The verification that this correction leads to a proper reordering relies on lemma `MultisetParity_MultisetReorder` [9.17].

Figure 9.25 shows the final assignment and highlights the completion of the cycle depicted in Figure 9.23. The figure also makes clear that the value `v` acts like an additional element in this assignment cycle.

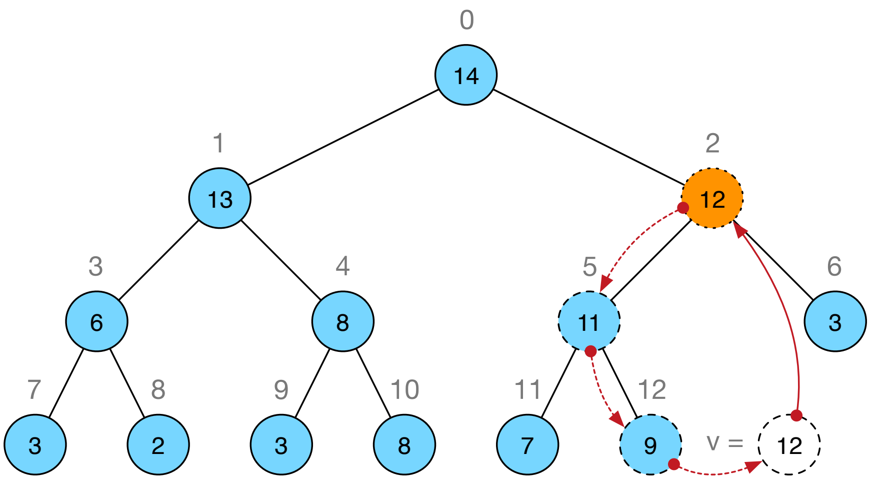

Figure 9.25.: Heap after the epilogue of `push_heap`

9.8. The pop_heap algorithm

The algorithm pop_heap moves the first element of the heap, which holds the heap's largest value, and places it at the the end of the underlying sequence. Whereas in the C++ Standard Library [20, §28.7.7.2] pop_heap works on a range of random access iterators, our version operates on an array of value_type. We therefore use the following signature for pop_heap

```
void pop_heap(value_type* a, size_type n);
```

The pop_heap algorithm expects that n is greater or equal than 1 and that the array a[0..n-1] forms a heap. The algorithms then *rearranges* the array a[0..n-1] such that the resulting array satisfies the following properties.

- a[n-1] = \old(a[0]), that is, the largest element of the original heap is transferred to the end of the array.
- the subarray a[0..n-2] is a heap

In this sense the algorithm *pops* the largest element from a heap.

9.8.1. Formal specification of pop_heap

Based on the above semi-formal description we propose the following function contract for pop_heap [9.26].

```
/*@
   requires bounds:     0 < n;
   requires valid:      \valid(a + (0..n-1));
   requires heap:       Heap(a, n);
   assigns              a[0..n-1];
   ensures heap:        Heap(a, n-1);
   ensures result:      a[n-1] == \old(a[0]);
   ensures max:         MaxElement(a, n, n-1);
   ensures reorder:     MultisetReorder{Old,Here}(a, n);
*/
void
pop_heap(value_type* a, size_type n);
```

Listing 9.26: Formal specification of pop_heap

9.8.2. Implementation of pop_heap

In an abstract sense pop_heap is quite similar to push_heap. In push_heap we started at the last array element and climbed from there up the tree until we would find a node where to insert the new value into the heap. Every time we had reached the next parent node we moved its value down to where we had just come from.

With pop_heap its the other way round. We start at the root of the tree and descend from there by selecting an appropriate child. Every time we lift the value of the selected child to the node where just are. We repeat this process until we find a node where we can insert the last array element into the heap. Once this is done, we can safely place the maximum element (that is the the original root node) at the last element of the array.

The following two figures illustrate how `pop_heap` affects an array, which is shown again as a tree with blue and grey nodes, representing heap and non-heap nodes, respectively. Figure 9.27 is in fact the same figure as Figure 9.2.

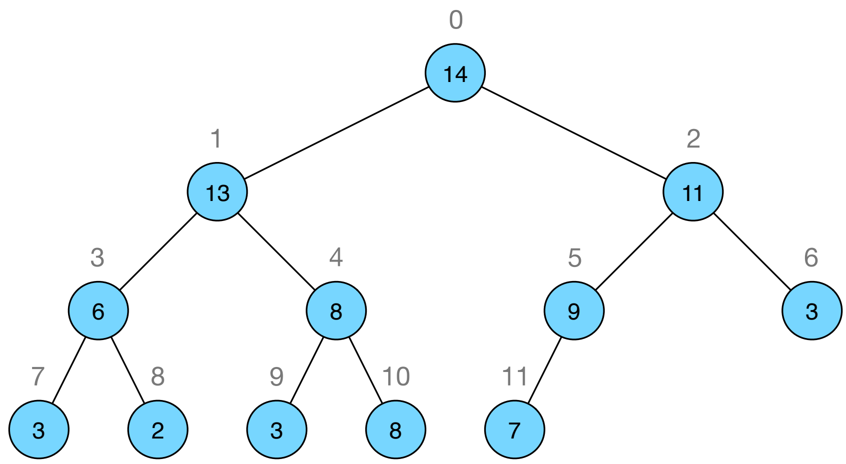

Figure 9.27.: Heap before the call of `pop_heap`

Figure 9.28, on the other hand, shows the heap after the call of `pop_heap` together with arrows that indicate how our implementation moves around elements in the underlying array. We can see that the first element of the original array, where the maximum of the heap resides, is now the last element of the array. Furthermore, the last array element is not part of the heap anymore. The dashed nodes highlight which heap nodes have changed during the call to `pop_heap`. The arrows indicate the *cyclic reordering* of array elements to achieve the desired result.

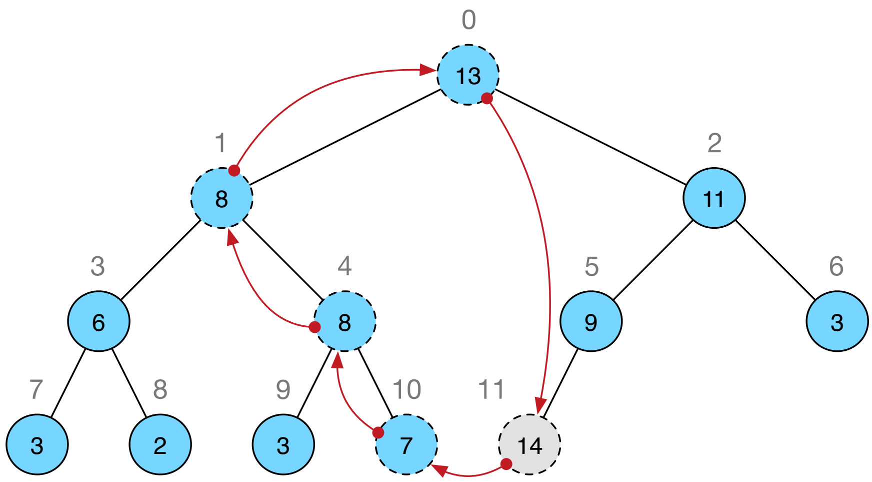

Figure 9.28.: Heap after the call of `pop_heap`

As in the case of `push_heap` [9.22] we will subdivide the discussion of the implementation of `pop_heap` [9.29] into a prologue, main act, and epilogue.

```
void
pop_heap(value_type* a, size_type n)
{
  if (1u < n) {
    //@ assert max:  MaxElement(a, n, 0);
    if (a[n - 1u] < a[0u]) { // otherwise a[0] == a[n-1] and nothing to be done
      size_type p = 0u;
      const value_type v = a[n - 1u];
      a[n - 1u] = a[p];
      //@ assert max:       a[n-1] == a[p];
      //@ assert update:    ArrayUpdate{Pre,Here}(a, n, n-1, a[p]);
      //@ assert heap:      Heap(a, n-1);
      //@ assert reorder:   MultisetParity{Pre,Here}(a, n, a[p], v);

      size_type c = heap_child(a, n - 1u, p);

      /*@
        loop invariant bounds:     0 <= p < c <= n-1;
        loop invariant parent:     c <  n-1 ==> p == HeapParent(c);
        loop invariant child:      c <  n-1 ==> HeapLeft(p) < n-1;
        loop invariant child:      c == n-1 ==> n-1 <= HeapLeft(p);
        loop invariant child:      HeapLeft(p)  < n-1 ==> a[HeapLeft(p)]  <= a[c];
        loop invariant child:      HeapRight(p) < n-1 ==> a[HeapRight(p)] <= a[c];
        loop invariant unchanged:  Unchanged{LoopEntry,Here}(a, p, n);
        loop invariant update:     a[p] == a[HeapParent(p)];
        loop invariant max:        UpperBound(a, n, a[n-1]);
        loop invariant reorder:    MultisetParity{Pre,Here}(a, n, a[p], v);
        loop invariant heap:       v < a[p];
        loop invariant heap:       Heap(a, n-1);
        loop assigns               p, c, a[0..n-2];
        loop variant               n - p;
       */
      for (; c < n - 1u && v < a[c];  p = c, c = heap_child(a, n - 1u, p)) {
        //@ assert max:           a[p] <= a[n-1];
        //@ assert heap:          a[c] <= a[p];
        if (a[c] < a[p]) {
          a[p] = a[c];
          //@ assert update:     ArrayUpdate{LoopCurrent,Here}(a, n, p, a[c]);
          //@ assert update:     ArrayUpdate{LoopCurrent,Here}(a, n-1, p, a[c]);
          //@ assert update:     MultisetUpdate{LoopCurrent,Here}(a, n, p, a[c]);
          //@ assert update:     a[c] == At{LoopCurrent}(a, c);
          //@ assert unchanged:  Unchanged{LoopEntry,Here}(a, p+1, n);
          //@ assert compatible: HeapCompatible(a, n-1, p, a[p]);
          //@ assert reorder:    MultisetParity{Pre,Here}(a, n, a[c], v);
        }
      }

      //@ ghost Epilogue: ;
      //@ assert child:      c == n-1   ||  a[c] <= v;
      //@ assert parent:     p <  n-1   &&  v < a[p];
      //@ assert compatible: HeapCompatible(a, n-1, p, v);
      a[p] = v;
      //@ assert update:     ArrayUpdate{Epilogue,Here}(a, n, p, v);
      //@ assert update:     MultisetUpdate{Epilogue,Here}(a, n, p, v);
      //@ assert reorder:    MultisetReorder{Pre,Here}(a, n);
      //@ assert heap:       Heap(a, n-1);
      //@ assert max:        UpperBound(a, n, a[n-1]);
    }
  }
}
```

Listing 9.29: Implementation of `pop_heap`

9.8.3. Prologue

In the prologue we check whether the initial heap contains at least two elements, initialize some variables, and also check whether the last array element is by chance equal to the maximum element of the heap, which resides at the index `p == 0` of the array. If this is not the case, then we set aside for future reference the last array element in the variable `v`. Finally we copy the value `a[p]` to its final destination at the end of the array. Note that this assignment only occurs if the respective values differ. This allows us, as in the case of `push_heap` [9.22], to formally describe the effect of the assignment using the predicate `ArrayUpdate` [7.2].

Figure 9.30 highlights the main effects of the prologue at the hand of our exemplary heap. Note that we have highlighted the root of the heap as the currently active node.

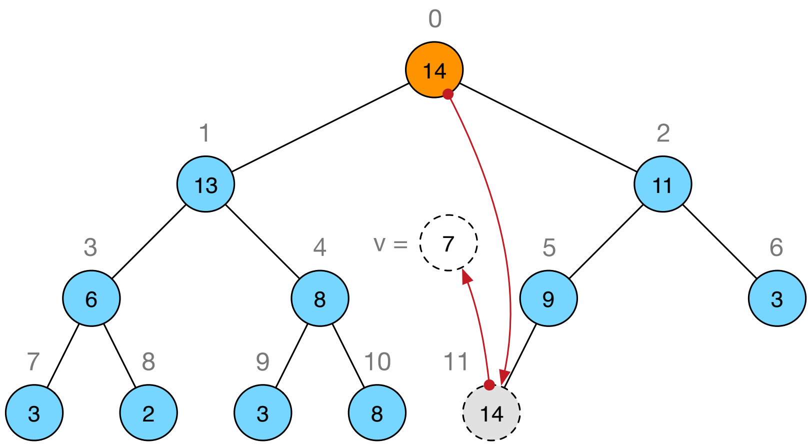

Figure 9.30.: Heap after the prologue of `pop_heap`

9.8.4. Main act

In the main act, we start at a child node `c` of the prologue's index `p`. This means that compared to the pre-state of `pop_heap` at the beginning of the main act the array `a[0..n-1]`

- contains the value `v` one time less
- contains the value `a[p]` one time more
- whereas all other values have not change their number of occurences.

Moreover, the maximum element of the original heap is now at the end of the array and we can only guarantee that the first $n - 1$ array elements got a heap. These observations are necessary reason for our loop invariants.

To be more precise, when we talk in the context of `pop_heap` of a *child node* we usually mean one of the possibly two children where the maximum of the values resides. We do this because copying that larger value to its parent node guarantees that the resulting tree is still a heap. We compute the maximum child of a node using the function `heap_child` [9.9].

Now, as long as the index `c` is not yet the index of the last array element of the heap and its value `a[c]` is less than `v`, we haven't found yet an index where we could insert `v` without violating the heap property.

In the loop body we proceed as follows.

- If `a[c]` is less than `a[p]` we copy the former value on the latter. As mentioned above, using the index `c` of the maximum child maintains heap property of the array. We use here the predicate `HeapCompatible` [9.7] to express that the insertion of the new value `a[p]` maintains the heap property of the array.

  The value `a[c]` now occurs one time more than in the pre-state whereas the now overwritten value `a[p]` occurs as often as in the pre-state of `pop_heap`. The value `v` continues to occur one time less than in the pre-state. We then proceed to the next iteration by setting `p` to `c` and computing the next maximum child node.

  As in the case of `push_heap` [9.22] the verification of the correct number of occurences of the involved values relies on the predicates `ArrayUpdate` [7.2] and `MultisetUpdate` [9.18] and on lemma `MultisetParity_Combined` [9.17].

- Otherwise, the array being a heap, we can conclude that `a[c]` equals `a[p]` and we continue with the next iteration after setting `p` to `c` and computing the corresponding new maximum child node.

The following three figures depict how the main act of `pop_heap` modifies step by step our example heap. In each step we highlight the currently active node `c`.

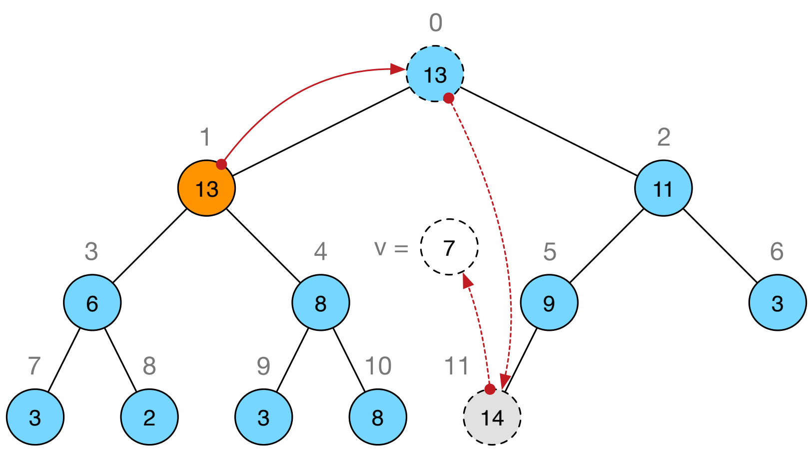

Figure 9.31.: Heap after the first iteration of of `pop_heap`

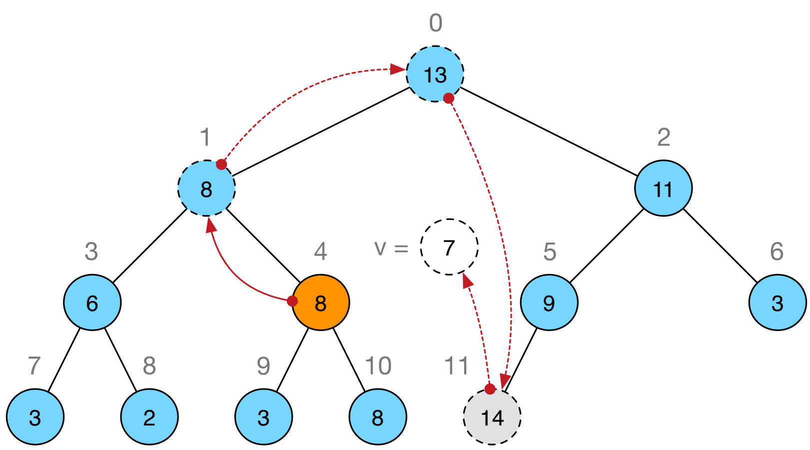

Figure 9.32.: Heap after the second iteration of `pop_heap`

Note that in the final step no value is actually copied as the involved nodes hold the same value.

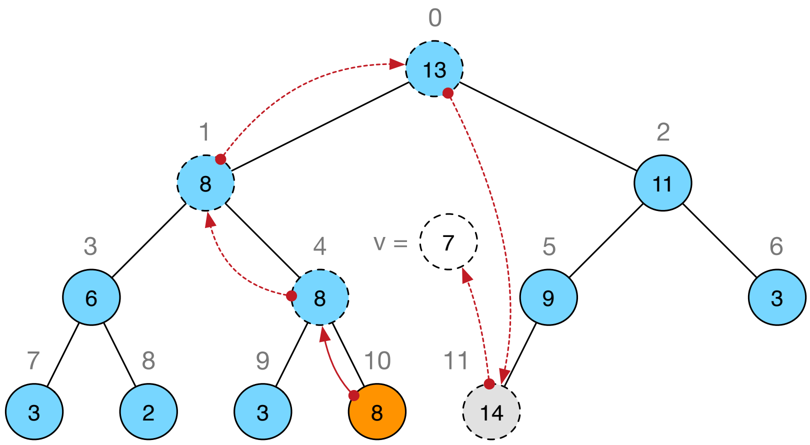

Figure 9.33.: Heap after the third iteration of `pop_heap`

We finally remark that in the main act the the last array element is never modified. Thus, the root element of the original element is still safely stored there.

9.8.5. Epilogue

After leaving the loop, we know that value `v` can be the inserted in the array at the index `p` without violating the heap property of the first $n - 1$ elements. Moreover, compared to the pre-state of `pop_heap` the array `a[0..n-1]` still

- contains the value `v` one time less
- contains the value `a[p]` one time more
- whereas all other values have not change their number of occurences.

In other words, assigning the value `v` to `a[p]` cancels this imbalance and establishes that `pop_heap` only reorders the array elements.

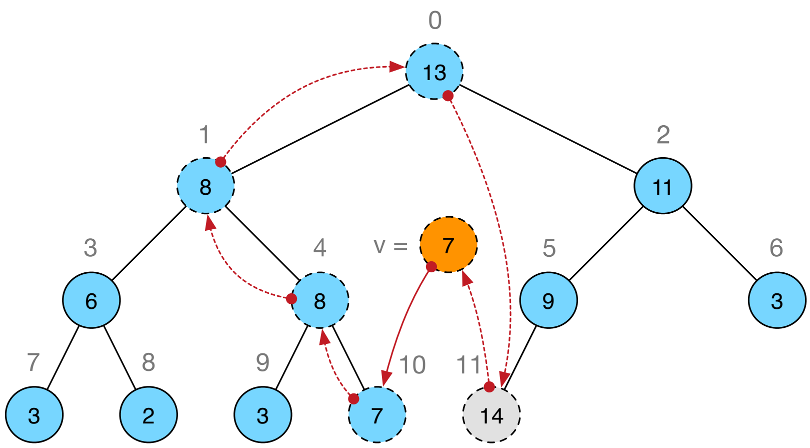

Figure 9.34.: Heap after the epilogue of `pop_heap`

In Figure 9.34 we have marked the value `v` as the currently active node despite not being an array element.

9.9. The make_heap algorithm

Whereas in the C++ Standard Library [20, §28.7.7.3] make_heap works on a pair of generic random access iterators, our version operators on a range of value_type. Thus the signature of make_heap reads

```
void make_heap(value_type* a, size_type n);
```

The function make_heap rearranges the elements of the given array a[0..n-1] such that they form a heap.

As an examples we look at the array in Figure 9.35. The elements of this array do not form a heap, as indicated by the grey colouring. Executing the make_heap algorithm on this array rearranges its elements so that they form a heap as shown in Figure 9.3.

8	8	7	3	14	11	3	6	2	3	13	9

Figure 9.35.: Array before the call of make_heap

9.9.1. Formal specification of make_heap

The following listing shows the specification of make_heap.

```
/*@
   requires valid:  \valid(a + (0..n-1));
   assigns          a[0..n-1];
   ensures heap:    Heap(a, n);
   ensures reorder: MultisetReorder{Old,Here}(a, n);
*/
void
make_heap(value_type* a, size_type n);
```

Listing 9.36: Formal specification of make_heap

Like with push_heap the formal specification of make_heap must ensure that the resulting array is a heap of size n and contains the same multiset of elements as in the pre-state of the function. These properties are expressed by the heap and reorder postconditions respectively. The reorder postcondition uses the predicate MultisetReorder [7.55] to ensure that make_heap only rearranges the array elements.

9.9.2. Implementation of make_heap

The implementation of make_heap, shown in the next listing, is straightforward. From low to high the array's elements are pushed to the growing heap. We used `i < n` as loop condition, rather than the more tempting `i <= n`, in order to admit also `n == SIZE_TYPE_MAX`; as a consequence, we had to call push_heap [9.19] with `i+1`. The iteration starts at `i+1 == 2`, because an array with length one is a heap already.

```
void
make_heap(value_type* a, size_type n)
{
  if (0u < n) {
    /*@
       loop invariant bounds:     1 <= i <= n;
       loop invariant heap:       Heap(a, i);
       loop invariant reorder:    MultisetReorder{Pre,Here}(a, n);
       loop invariant unchanged:  Unchanged{Pre,Here}(a, i+1, n);
       loop assigns   i, a[0..n-1];
       loop    variant n - i;
    */
    for (size_type i = 1u; i < n; ++i) {
      push_heap(a, i + 1u);
      //@ assert reorder:    MultisetReorder{LoopCurrent,Here}(a, i+1);
      //@ assert unchanged:  Unchanged{LoopCurrent,Here}(a, i+1, n);
      //@ assert reorder:    MultisetReorder{LoopCurrent,Here}(a, n);
    }

    //@ assert reorder:    MultisetReorder{Pre,Here}(a, n);
  }

  //@ assert  heap: Heap(a, n);
}
```

Listing 9.37: Implementation of make_heap

Since the loop statement consists just of a call to push_heap [9.19] we obtain the both loop invariants heap and reorder by simply lifting them from the contract of push_heap.

The postcondition of push_heap only specifies the multiset of elements from index 0 to i. We therefore also have to specify that the elements from index i+1 to n−1 are only reordered. This property can be derived from the unchanged property of push_heap.

9.10. The `sort_heap` algorithm

Whereas in the C++ Standard Library [20, §28.7.7.4] `sort_heap` works on a range of random access iterators, our version operates on an array of `value_type`. We therefore use the following signature for `sort_heap`

```
void sort_heap(value_type* a, size_type n);
```

The function `sort_heap` rearranges the elements of a given heap `a[0..n-1]` in increasing order. Thus, applying `sort_heap` to the heap in Figure 9.3 produces the increasing array in Figure 9.38.

2	3	3	3	6	7	8	8	9	11	13	14

Figure 9.38.: Array after the call of `sort_heap`

9.10.1. Formal specification of `sort_heap`

The following listing shows our specification of `sort_heap`. The formal specification of `sort_heap` must ensure that the resulting array is increasing. Furthermore the multiset contained by the array must be the same as in the pre-state of the function. The postconditions `increasing` and `reorder` express these properties, respectively. The specification effort is relatively simple because we can reuse

```
/*@
   requires valid:        \valid(a + (0..n-1));
   requires heap:         Heap(a, n);
   assigns                a[0..n-1];
   ensures reorder:       MultisetReorder{Old,Here}(a, n);
   ensures increasing:    Increasing(a, n);
*/
void
sort_heap(value_type* a, size_type n);
```

Listing 9.39: Formal specification of `sort_heap`

9.10.2. Implementation of sort_heap

The implementation of sort_heap is relatively simple because it relies on pop_heap [9.26] performing essential work. Our implementation of sort_heap repeatedly calls pop_heap to extract the maximum of the shrinking heap and adding it to the part of the array that is already in increasing order. The loop invariants of sort_heap describe the content of the array in two parts. The first i elements form a heap and are described by the heap invariant. The last n-i elements are already arranged in increasing order.

As already mentioned in the introduction of Chapter 6, we use the predicate WeaklyIncreasing [6.2] for the loop annotation increasing. Thus, after leaving the loop we have in fact “only” shown that WeaklyIncreasing(a, n) holds. In order to derive from this fact the final assertion increasing that uses the predicate Increasing [6.1] we rely on lemma WeaklyIncreasing_Increasing [6.3].

```
void
sort_heap(value_type* a, size_type n)
{
  /*@
     loop invariant bound:        0 <= i <= n;
     loop invariant heap:         Heap(a, i);
     loop invariant lower:        LowerBound(a, i, n, a[0]);
     loop invariant reorder:      MultisetReorder{Pre,Here}(a, 0, n);
     loop invariant increasing:   WeaklyIncreasing(a, i, n);
     loop assigns i, a[0..n-1];
     loop variant i;
  */
  for (size_type i = n; i > 1u; --i) {
    /*@
        requires heap:     Heap(a, i);
        assigns  a[0..i-1];
        ensures  heap:     Heap(a, i-1);
        ensures  max:      a[i-1] == \old(a[0]);
        ensures  max:      MaxElement(a, i, i-1);
        ensures  reorder:  MultisetReorder{Old,Here}(a, 0, i);
        ensures  reorder:  Unchanged{Old,Here}(a, i, n);
    */
    pop_heap(a, i);
    //@ assert lower:  LowerBound(a, i, n, a[i-1]);
  }

  //@ assert increasing: Increasing(a, n);
}
```

Listing 9.40: Implementation of sort_heap

To verify the property reorder we rely on the lemmas MultisetReorder [7.55] that express that the properties

- MultisetReorder{K,L}(a, 0, i) and
- Unchanged{Old,Here}(a, i, n)

imply the desired loop invariant MultisetReorder{K,L}(a, 0, n).

10. Sorting Algorithms

Many issues in computer science can be exemplified in the field of sorting algorithms; see e.g. [25] for a famous textbook. Therefore we arrange some of the most common classic sorting algorithms. In this chapter, we present algorithms of the C++ Standard Library [20, §28.7.1] that are related to the task of sorting a linear array.

Following [26], we have also used (C rephrasings of) functions from the C++ Standard Library as far as possible to implement the different algorithmic approaches.

- `is_sorted` in §10.1 is an algorithm that checks if a given array is already in increasing order.
- `partial_sort` in §10.2 rearranges a given array into two parts. All elements in the first part are less or equal than those of the second part. Moreover, while the first part is sorted, the order of elements in the second part is unspecified.
- `bubble_sort` in §10.3 describes a simple, well-known and sorting algorithm.[29]
- `selection_sort` in §10.4 presents the classic *selection sort* algorithm.[30]
- `insertion_sort` in §10.5 the also well-known *insertion sort* algorithm.[31]
- `heap_sort` in §10.6 describes the quite efficient *heap sort*, which relies on the algorithms presented in Chapter 9.[32]
- `merge` in §10.7 the *merge* algorithm from *merge sort*.[33]

While `heap_sort` achieves a run-time complexity upper bound of $O(n \cdot \log(n))$ due to the efficiency of the heap data structure, both `selection_sort` and `insertion_sort` need $O(n^2)$ in the average case, and also in the worst case.

Note that the `sort` algorithm from the C++ Standard Library is not handled here because it typically relies on *introspection sort* which is sophisticated mix of various classic algorithms.[34] In future releases we plan to handle the more algorithms related sorting.

---

[29] See `https://en.wikipedia.org/wiki/Bubble_sort`
[30] See `https://en.wikipedia.org/wiki/Selection_sort`
[31] See `https://en.wikipedia.org/wiki/Insertion_sort`
[32] See `https://en.wikipedia.org/wiki/Heapsort`
[33] See `https://en.wikipedia.org/wiki/Merge_sort`
[34] See `https://en.wikipedia.org/wiki/Introsort`

The sorting algorithms in this chapter essentially share the following contract; it is their implementations that differ fundamentally.

```
/*@
   requires valid: \valid(a + (0..n-1));

   assigns  a[0..n-1];

   ensures increasing:  Increasing(a, n);
   ensures reorder:     MultisetReorder{Old, Here}(a, n);
*/
void xxx_sort(value_type* a, size_type n);
```

As mentioned in the introduction of Chapter 6, we use the predicate `Increasing` [6.1] in the contracts of our sorting algorithms but often resort to the simpler predicate `WeaklyIncreasing` [6.2] in the loop invariants and assertions. In order to conclude that the desired postcondition `Increasing(a, n)` holds, we rely on lemma `WeaklyIncreasing_Increasing` [6.3].

10.1. The `is_sorted` algorithm

Our version of the `is_sorted` algorithm compared to the C++ Standard Library [20, §28.7.1.5] has the signature

```
bool is_sorted(const value_type* a, size_type n);
```

It returns **`true`** if the given array is in increasing order, and **`false`** otherwise.

10.1.1. Formal specification of `is_sorted`

The following listing shows the acsl specification of `is_sorted`. In the contract, we use the predicate `Increasing` [6.1], which states that any array element is always less or equal to any other element right of it. We'll use an easier-to-handle predicate in the implementation of `is_sorted` [10.2].

```
/*@
  requires valid: \valid_read(a + (0..n-1));
  assigns         \nothing;
  ensures result: \result <==> Increasing(a, n);
*/
bool
is_sorted(const value_type* a, size_type n);
```

Listing 10.1: Formal specification of `is_sorted`

10.1.2. Implementation of is_sorted

The implementation of is_sorted is shown in the next Listing. As usual, is_sorted doesn't compare every array element to all that are right to it, but only to the immediately adjacent one, which is of course more efficient. For this, we use the predicate WeaklyIncreasing [6.2] in the loop invariant of the implementation.

```
bool
is_sorted(const value_type* a, size_type n)
{
  if (0u < n) {
    /*@
        loop invariant increasing: WeaklyIncreasing(a, i+1);
        loop assigns i;
        loop variant n - i;
    */
    for (size_type i = 0u; i < n - 1u; ++i) {
      if (a[i] > a[i + 1u]) {
        return false;
      }
    }
  }

  return true;
}
```

Listing 10.2: Implementation of is_sorted

Since our implementation uses WeaklyIncreasing in its loop invariant, and follows the same principle in its code, its verification is straight-forward—except for the final reasoning that WeaklyIncreasing (a,n) implies Increasing(a,n).

We have the lemma WeaklyIncreasing_Increasing [6.3] for that step, which needs to be proven manually with Coq. The converse lemma Increasing_WeaklyIncreasing [6.3] is proven automatically, but isn't actually needed to verify our is_sorted implementation. Alternatively, we could have dragged the predicate Increasing along the loop, which happens to cause no particular problems in this case.

10.2. The `partial_sort` algorithm

Our version of the `partial_sort` algorithm compared to the C++ Standard Library [20, §28.7.1.3] has the signature

```
void partial_sort(value_type* a, size_type m, size_type n);
```

The algorithm *reorders* the given array `a` in such a way that it represents a *partition*: each member of the left part `a[0..m-1]` is less or equal to each member of the right part `a[m..n-1]`. Moreover, the algorithm *sorts* the left part in increasing order. The order of elements in the right part, however, is *unspecified*. Figure 10.3 uses a bar chart to depict a typical result of a call `partial_sort(a, m, n)`. In the post-state, the left and the right part is colored in green and orange, respectively.

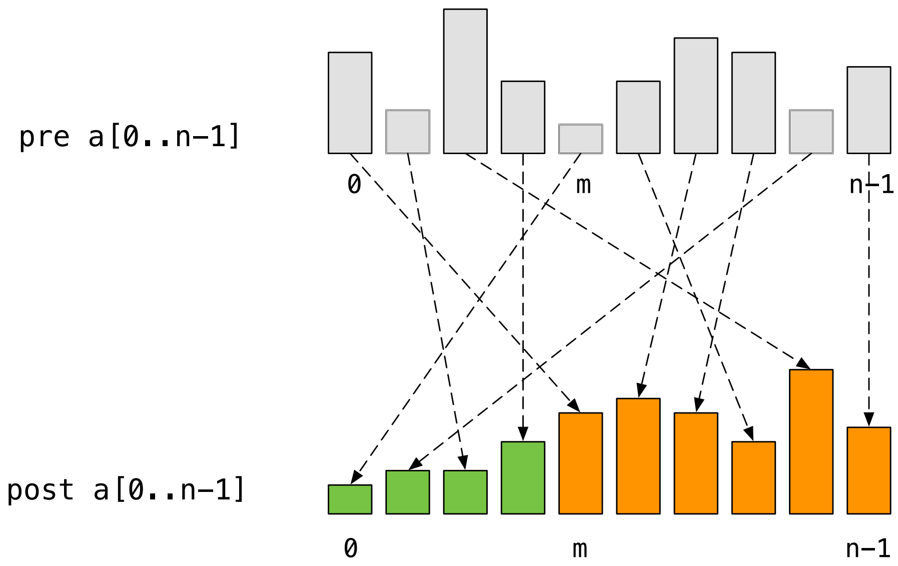

Figure 10.3.: Effects of `partial_sort`

10.2.1. The predicate `Partition`

We start by introducing the new predicate `Partition` [10.4] which formalizes the partitioning property.

```
/*@
  axiomatic Partition
  {
    predicate
    Partition{L}(value_type* a, integer m, integer n) =
      \forall integer i, k; 0 <= i < m <= k < n  ==>  a[i] <= a[k];
  }
*/
```

Listing 10.4: The logic definition(s) `Partition`

The lemmas in the following listing are used in proofs of properties and annotations related to the loop invariants `upper`, `lower`, and `partition` of `partial_sort`.

```
/*@
  axiomatic PartitionLemmas
  {
    lemma MultisetReorder_SomeEqual{K,L}:
      \forall value_type *a, integer n, i;
        0 < n                        ==>
        0 <= i < n                   ==>
        MultisetReorder{K,L}(a, n)   ==>
        SomeEqual{K}(a, n, \at(a[i],L));

    lemma MultisetReorder_LowerBound{K,L}:
      \forall value_type* a, integer n, value_type v;
        0 <= n                       ==>
        MultisetReorder{K,L}(a, n)   ==>
        LowerBound{K}(a, n, v)       ==>
        LowerBound{L}(a, n, v);

    lemma MultisetReorder_UpperBound{K,L}:
      \forall value_type* a, integer n, value_type v;
        0 <= n                         ==>
        MultisetReorder{K,L}(a, n)     ==>
        UpperBound{K}(a, n, v)         ==>
        UpperBound{L}(a, n, v);

    lemma MultisetReorder_PartitionLowerBound{K,L}:
      \forall value_type* a, integer m, n;
        0 < m <= n                        ==>
        MultisetReorder{K,L}(a, 0, m)     ==>
        Partition{K}(a, m, n)             ==>
        Unchanged{K,L}(a, m, n)           ==>
        LowerBound{L}(a, m, n, \at(a[0],L));
  }
*/
```

Listing 10.5: The logic definition(s) `PartitionLemmas`

- Lemma `MultisetReorder_SomeEqual` states that a value `a[i]` taken from a range `a[0..n-1]` after some reordering must have been in that range already before reordering. It is used to prove the subsequent lemmas.
- Lemma `MultisetReorder_LowerBound` informally says that a lower bound `v` of a range `a[0..n-1]` keeps its property even after the range is reordered.
- Dually, lemma `MultisetReorder_UpperBound` says that reordering a range doesn't affect any of its upper bounds.
- Lemma `MultisetReorder_PartitionLowerBound` describes a more particular situation: if each element in `a[0..m-1]` is known to be a less or equal than element `a[m..n-1]` and the former range is reordered while the latter is kept untouched, then `a[0]` will still be a lower bound of `a[m..n-1]`. We employ this lemma to infer that, after `push_heap` [9.19] was called, the new heap maximum `a[0]`, is a lower bound of `a[m..i]`,

The proof of `MultisetReorder_SomeEqual` [10.5] relies on the lemma `Count_SomeEqual` [4.46]. We also rely on the lemma `MultisetSwap_Middle` [7.59] in order to verify that the loop invariant `reorder` is preserved.

10.2.2. Formal specification of `partial_sort`

The formal specification of the `partial_sort` function is shown in the following listing. It uses the just introduced predicate `Partition` and reuses the previously defined predicates `Increasing` [6.1] and `MultisetReorder` [7.55].

```
/*@
  requires valid:        \valid(a + (0..n-1));
  requires split:        0 <= m <= n;
  assigns                a[0..n-1];
  ensures reorder:       MultisetReorder{Old,Here}(a, n);
  ensures partition:     Partition(a, m, n);
  ensures increasing:    Increasing(a, m);
*/
void
partial_sort(value_type* a, size_type m, size_type n);
```

Listing 10.6: Formal specification of `partial_sort`

10.2.3. Implementation of `partial_sort`

Our implementation of `partial_sort` is shown the next listing. It initially calls `make_heap` [9.36] to rearrange the left part `a[0..m-1]` into a heap. After that, it scans the right part, from left to right, for elements that are too small; each such element is exchanged for the left part's maximum, by applying `pop_heap` [9.26] and `push_heap` [9.19] appropriately. When the scan is done, the smallest elements are collected in the left part. We finally convert it from a heap into an increasingly ordered range, by `sort_heap` (9.10).

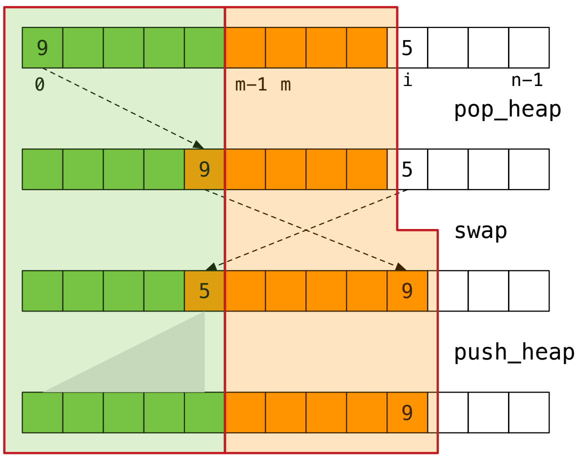

Figure 10.7.: An iteration of `partial_sort`

In the scan loop, we maintain as invariants

- that the left part is a heap (invariant `heap`);
- that its maximal element, `a[0]`, is a "separating element" between the left part `a[0..m-1]` and the right part `a[m..i-1]`, i.e., an upper bound of the left (invariant `upper`) and a lower bound of the right part (invariant `lower`), respectively;
- that `a[i..m-1]` is yet unchanged (invariant `unchanged`); and
- that only permutation operations have been applied to `a[0..i-1]` (invariant `reorder`).

In order to preserve the loop invariants after `i` is incremented, nothing has to be done if `a[0]` happens to be also a lower bound for `a[i]`. Otherwise, let us follow the algorithm through the `then` part code, depicting the intermediate states in Figure 10.7. The elements considered so far are shown colored similar to Figure 10.3; in particular the heap part is shown in green.

The overlaid transparent red shape indicates the ranges to which `Partition` applies, in each state. The figure assumes the initial contents of `a[0]` and `a[i]` to be 9 and 5, for sake of generality, let us call them p and q, respectively.

After `pop_heap` and `swap`, we have p at `a[i]`, and q at `a[m-1]`. At that point we know

1. $q < p \leq$ `a[k]` for each $m \leq k < i$, since p was a lower bound for `a[m..i-1]`;
2. $q < p =$ `a[i]`;
3. `a[j]` $\leq p \leq$ `a[k]` for each $0 \leq j < m-1$ and each $m \leq k < i$, since this held on loop entry, and we didn't more than reordering inside the parts; and
4. `a[j]` $\leq p =$ `a[i]` since p was the heap maximum on loop entry.

```
void
partial_sort(value_type* a, size_type m, size_type n)
{
  if (m > 0u) {
    make_heap(a, m);

    //@ assert reorder: Unchanged{Pre,Here}(a, m, n);
    /*@
      loop invariant bound:     m <= i <= n;
      loop invariant heap:      Heap(a, m);
      loop invariant upper:     UpperBound(a, 0, m, a[0]);
      loop invariant lower:     LowerBound(a, m, i, a[0]);
      loop invariant reorder:   MultisetReorder{Pre,Here}(a, i);
      loop invariant unchanged: Unchanged{Pre,Here}(a, i, n);
      loop assigns              i, a[0..n-1];
      loop variant              n-i;
    */
    for (size_type i = m; i < n; ++i) {
      if (a[i] < a[0u]) {
        /*@
          assigns              a[0..m-1];
          ensures heap:        Heap(a, m-1);
          ensures max:         a[m-1] == \old(a[0]);
          ensures max:         MaxElement(a, m, m-1);
          ensures reorder:     MultisetReorder{Old,Here}(a, m);
          ensures unchanged:   Unchanged{Old,Here}(a, m, i);
          ensures unchanged:   Unchanged{Old,Here}(a, m, n);
        */
        pop_heap(a, m);
        //@ assert lower:      a[0] <= a[m-1];
        //@ assert lower:      a[i] <  a[m-1];
        //@ assert lower:      LowerBound(a, m, i, a[m-1]);
        //@ assert upper:      UpperBound(a, 0, m-1, a[0]);
        //@ assert upper:      UpperBound(a, 0, m,   a[m-1]);
        //@ assert partition:  Partition(a, m, i);
        //@ assert reorder:    MultisetReorder{Pre,Here}(a, i);
        //@ assert unchanged:  Unchanged{Pre,Here}(a, i, n);
```

Listing 10.8: Implementation of `partial_sort` (1)

Altogether, we have `a[j]` $\leq p \leq$ `a[k]` for each $0 \leq j < m$ and each $m \leq k < i + 1$. That is, `Partition (a,m,i+1)` holds, although we cannot name a separating element of `a` here.

After calling `push_heap`, which just performs some more reorderings of the left part, this property is preserved. We can't and we needn't tell which position q is moved to; the former is indicated in Figure 10.3 by the vague grey triangle. Moreover, we now know again that `a[0]` has become an upper bound of the left part, and hence a separating element between `a[0..m-1]` and `a[m..i]`; that is, the loop invariants `upper` and `lower` have been re-established. These two invariants together are eventually used to prove the property `partition` of the contract.

Compared to its size, the algorithm makes a lot of procedure calls; in this respect it is closer to real-life software than most other algorithms of this tutorial. Therefore, we use it to illustrate a methodical point: For almost every procedure call, we give the callee's contract, tailored to its actual parameters, as a statement contract of the call. For example, everything we know from the `pop_heap` contract, instantiated to the particular situation, is documented in the first statement contract. In contrast, we use `assert` clauses to indicate intermediate reasoning to obtain subsequently needed properties.

```
        //@ ghost Before: ;
        //@ assert reorder:    MultisetReorder{Pre,Here}(a, i+1);
        swap(a + m - 1u, a + i);
        //@ assert swapped:    ArraySwap{Before,Here}(a, m-1, i, n);
        //@ assert unchanged:  Unchanged{Before,Here}(a, m-1);
        //@ assert reorder:    MultisetReorder{Before,Here}(a, m-1, i+1);
        //@ assert reorder:    MultisetReorder{Before,Here}(a, i+1);
        //@ assert reorder:    MultisetReorder{Pre,Here}(a, i+1);
        //@ assert unchanged:  Unchanged{Pre,Here}(a, i+1, n);
        //@ assert lower:      a[m-1] < a[i];
        //@ assert partition:  Partition(a, m, i+1);
        //@ assert upper:      UpperBound(a, 0, m-1, a[0]);

        /*@
          assigns              a[0..m-1];
          ensures heap:        Heap(a, m);
          ensures reorder:     MultisetReorder{Old,Here}(a, m);
          ensures unchanged:   Unchanged{Old,Here}(a, m, i+1);
          ensures unchanged:   Unchanged{Old,Here}(a, i+1, n);
        */
        push_heap(a, m);
        //@ assert reorder:    MultisetReorder{Pre,Here}(a, i+1);
        //@ assert upper:      UpperBound(a, 0, m,   a[0]);
        //@ assert lower:      LowerBound(a, m, i+1, a[0]);
      }
    }

    //@ assert partition: Partition(a, m, n);
    /*@
      assigns                 a[0..m-1];
      ensures reorder:        MultisetReorder{Old,Here}(a, m);
      ensures unchanged:      Unchanged{Old,Here}(a, m, n);
      ensures increasing:     Increasing(a, m);
    */
    sort_heap(a, m);
    //@ assert reorder:   MultisetReorder{Pre,Here}(a, n);
    //@ assert partition: Partition(a, m, n);
  }
}
```

Listing 10.9: The Implementation of partial_sort (2)

Our implementation has a worst-case time complexity of $O((n + m) \cdot \log m)$. On the other hand, an implementation that ignores m and just sorts a[0..n-1] also satisfies the contract of partial_sort [10.6], and may have $O(n \cdot \log n)$ complexity. Some arithmetic shows that partial_sort performs better than plain sort if, and only if, $\log m < \frac{n}{m} \cdot \log\left(\frac{n}{m}\right)$, that is, if n is sufficiently larger than m.

10.3. The `bubble_sort` algorithm

The `bubble_sort` algorithm traverses the given array `a[0..n-1]` from left to right, maintaining a right-adjusted, constantly growing range `a[n-i..n-1]` that is already in increasing order. We achieve this range by iterating through the array and swapping two adjacent elements, if their respective value are in the wrong order.

10.3.1. Formal specification of `bubble_sort`

The following listing shows our (generic sorting) contract for `bubble_sort`.

```
/*@
  requires valid:       \valid(a + (0..n-1));
  assigns               a[0..n-1];
  ensures increasing:   Increasing(a, n);
  ensures reorder:      MultisetReorder{Old,Here}(a, n);
 */
void
bubble_sort(value_type* a, size_type n);
```

Listing 10.10: Formal specification of `bubble_sort`

10.3.2. Implementation of `bubble_sort`

Our implementation of `bubble_sort` is shown in the next listing. As it is typical for `bubble_sort`, the implementation uses two nested loops.

We first discuss the verification of the fact that `bubble_sort` produces an increasing array. For this we introduce for the *outer loop* the invariant `increasing`. This loop annotation states that the subrange `a[n-i+1..n-1]` is in increasing order. An important ingredient on the verification of the `increasing` property is the claim that the first element `a[n-i+1]` of the already sorted subrange is an upper bound of *all* elements left of it. This claim is encoded in the loop invariant `upper` of the outer bound. In order to support this claim up we exploit the fact that the index `j` of the `inner loop` points to the maximum element of the subrange `a[0..j]`. We formalize this last property in the loop invariant `max`.

Note that the loop invariants `increasing` and `upper` occur also in the inner loop. This shall "assure" the outer loop that the inner loop really preserves these properties.

```
void
bubble_sort(value_type* a, size_type n)
{
  if (0 < n) {
    /*@
      loop invariant bound:          1 <= i <= n;
      loop invariant increasing:     WeaklyIncreasing(a, n-i+1, n);
      loop invariant upper:          1 < i ==> UpperBound(a, n-i+1, a[n-i+1]);
      loop invariant reorder:        MultisetReorder{Pre,Here}(a, n);
      loop assigns i, a[0..n-1];
      loop variant n-i;
    */
    for (size_type i = 1u; i < n; ++i) {
      /*@
        loop invariant bound:        0 <= j <= n-i;
        loop invariant increasing:   WeaklyIncreasing(a, n-i+1, n);
        loop invariant upper:        1 < i ==> UpperBound(a, n-i+1, a[n-i+1]);
        loop invariant max:          MaxElement(a, j+1, j);
        loop invariant reorder:      MultisetReorder{LoopEntry,Here}(a, j+1);
        loop invariant reorder:      Unchanged{LoopEntry,Here}(a, j+1, n);
        loop assigns                 j, a[0..n-1];
        loop variant n-j;
      */
      for (size_type j = 0u; j < n - i; ++j) {
        if (a[j] > a[j + 1u]) {
          //@ assert max:       MaxElement(a, j+1, j);
          //@ assert reorder:   MultisetReorder{LoopEntry,Here}(a, 0, j+1);
          //@ assert reorder:   MultisetReorder{LoopEntry,Here}(a, 0, j+2);
          swap(&a[j], &a[j + 1u]);
          //@ assert max:       MaxElement(a, j+2, j+1);
          //@ assert swap:      ArraySwap{LoopCurrent,Here}(a, j, j+1, n);
          //@ assert unchanged: Unchanged{LoopCurrent,Here}(a, j);
          //@ assert reorder:   a[j+1] == At{LoopCurrent}(a, j);
          //@ assert reorder:   a[j]   == At{LoopCurrent}(a, j+1);
          //@ assert reorder:   MultisetReorder{LoopCurrent,Here}(a, j, j+2);
        }
      }
    }
  }

  //@ assert increasing: Increasing(a, n);
}
```

Listing 10.11: Implementation of `bubble_sort`

We now discuss briefly the verification of the postcondition `reorder`. In each iteration of the outer loop various elements of the not yet sorted subrange `a[0..n-1]` are swapped with their respective neighbour. More specifically, we know for the iteration `j` of the *inner loop* that while subrange `a[0..j]` has been rearranged, the subrange `a[j+1..n-1]` has not been modified yet. Together this ensures that the loop invariant `reorder` holds for the *outer loop*.

10.4. The `selection_sort` algorithm

Our version of the `selection_sort` algorithm has the signature

```
void selection_sort(value_type* a, size_type n);
```

The `selection_sort` algorithm sorts an array in increasing order, left to right, by selecting in each step the minimum element of the remaining segment and *swaps* it with its first element. This implies that each member of the increasingly ordered initial segment is less or equal than each member of the remaining segment.

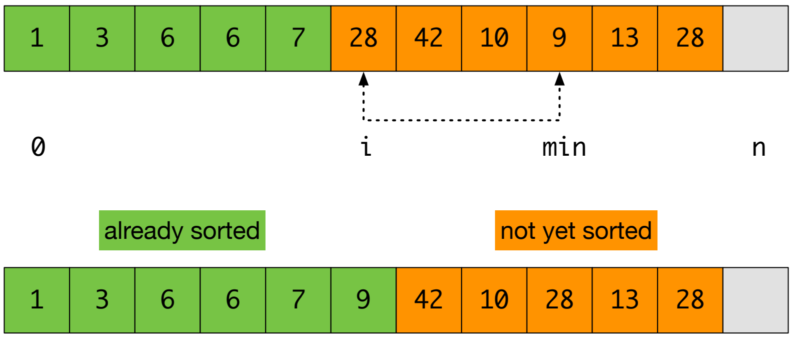

Figure 10.12.: An iteration of `selection_sort`

Figure 10.12 shows a typical situation in an example run. The algorithm will swap the `28` at position `i` with the `9` at position `min` to extend the increasingly ordered initial segment one field to the right.

10.4.1. Formal specification of `selection_sort`

The following listing shows the specification of `selection_sort`.

```
/*@
   requires valid:       \valid(a + (0..n-1));
   assigns               a[0..n-1];
   ensures reorder:      MultisetReorder{Old,Here}(a, n);
   ensures increasing:   Increasing(a, n);
*/
void
selection_sort(value_type* a, size_type n);
```

Listing 10.13: Formal specification of `selection_sort`

10.4.2. Implementation of `selection_sort`

The implementation of `selection_sort` is shown in the next listing. We use `min_element` [5.14] to find the minimum element of the remaining array segment.

```
void
selection_sort(value_type* a, size_type n)
{
  /*@
    loop invariant bound:       0 <= i <= n;
    loop invariant reorder:     MultisetReorder{Pre,Here}(a, n);
    loop invariant increasing:  WeaklyIncreasing(a, i);
    loop invariant increasing:  0 < i  ==> LowerBound(a, i, n, a[i-1]);
    loop assigns   i, a[0..n-1];
    loop variant   n - i;
  */
  for (size_type i = 0u; i < n; ++i) {
    const size_type sel = i + min_element(a + i, n - i);

    if (i < sel) {
      /*@
         assigns          a[sel], a[i];
         ensures swapped: ArraySwap{Old,Here}(a, i, sel, n);
      */
      swap(a + sel, a + i);
    }

    //@ assert reorder: MultisetReorder{LoopCurrent,Here}(a, n);
    //@ assert reorder: MultisetReorder{Pre,Here}(a, n);
  }

  //@ assert increasing: Increasing(a, n);
}
```

Listing 10.14: Implementation of `selection_sort`

The loop invariants `increasing` and `lower` establish that the initial segment `a[0..i-1]` is in increasing order and, respectively, state that `a[i-1]` is a lower bound of the remaining segment `a[i..n-1]`. Since the `min_element` call uses an address offset, we had to employ again the *shift lemmas* from the collection `ArrayBoundsShift` [6.14].

The loop invariant `reorder`, on the other hand, states that the multiset of values in the array `a` are only *rearranged* during the algorithm. While this is intuitively most obvious (as the call to the `swap` [7.6] routine, is the only code that modifies `a`), it took considerable effort to prove it formally; including a statement contract that captures the effects of calling `swap`.

The main reason for introducing the statement contract is that it *transforms* the postcondition of the call to `swap` [7.6] into the hypotheses for the lemma `MultisetSwap_Middle` [7.59]. This lemma, which relies on the lemmas about `MultisetReorder` [7.55], captures the fact that *swapping two elements of an array* is a *reordering*.

10.5. The `insertion_sort` algorithm

Like `selection_sort`, the algorithm `insertion_sort` traverses the given array `a[0..n-1]` left to right, maintaining a left-adjusted, constantly increasing range `a[0..i-1]` that is already in increasing order.

Unlike `selection_sort`, however, `insertion_sort` adds `a[i]` to the initial segment in the `i`th step (see Figure 10.15). It determines the (rightmost) appropriate position to insert `a[i]` by a call to `upper_bound` [6.8] and then uses `rotate` [7.27] to perform a *circular shift* to establish the insertion.

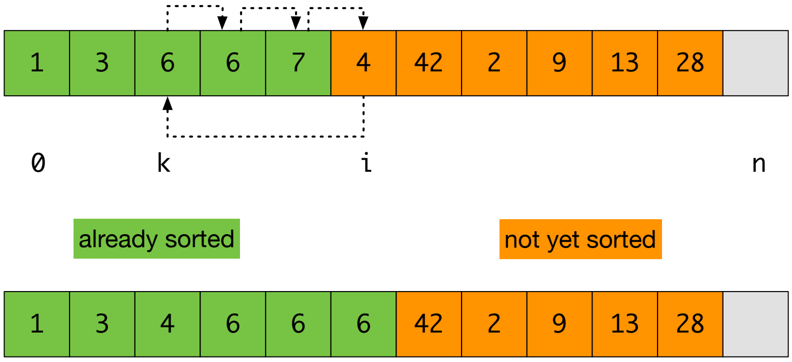

Figure 10.15.: An iteration of `insertion_sort`

10.5.1. Formal specification of `insertion_sort`

The following listing shows our (generic sorting) contract for `insertion_sort`.

```
/*@
   requires valid:        \valid(a + (0..n-1));
   assigns                a[0..n-1];
   ensures reorder:       MultisetReorder{Old,Here}(a, n);
   ensures increasing:    Increasing(a, n);
*/
void
insertion_sort(value_type* a, size_type n);
```

Listing 10.16: Formal specification of `insertion_sort`

10.5.2. Implementation of insertion_sort

The implementation of insertion_sort is shown in the next listing. We used an ACSL statement contract to specify those aspects of the rotate contract that are needed here. Properties related to the result of insertion_sort being in increasing order are labelled increasing. Properties related to the rearrangement of elements are labelled reorder and, whenever their order isn't changed, unchanged.

```
void
insertion_sort(value_type* a, size_type n)
{
  /*@
    loop invariant bound:         0 <= i <= n;
    loop invariant reorder:       MultisetReorder{Pre,Here}(a, 0, i);
    loop invariant unchanged:     Unchanged{Pre,Here}(a, i, n);
    loop invariant increasing:    WeaklyIncreasing(a, i);
    loop assigns    i, a[0..n-1];
    loop variant    n - i;
  */
  for (size_type i = 0u; i < n; ++i) {
    const size_type k = upper_bound(a, i, a[i]);
    //@ assert bound:   0 <= k <= i;
    /*@
       requires increasing: UpperBound(a, k, a[i]);
       requires increasing: StrictLowerBound(a, k, i, a[i]);
       requires increasing: WeaklyIncreasing(a, k, i);
       assigns              a[k..i];
       ensures unchanged:   Unchanged{Old,Here}(a, 0, k);
       ensures unchanged:   Unchanged{Old,Here}(a, i+1, n);
       ensures reorder:     Equal{Old,Here}(a, k, i, k+1);
       ensures reorder:     Equal{Old,Here}(a, i, i+1, k);
       ensures increasing:  WeaklyIncreasing(a, 0, k);
    */
    //@ assert increasing:  UpperBound(a, k, a[i]);
    rotate(a + k, i - k, i - k + 1u);
    //@ assert increasing:  UpperBound(a, k, a[k]);
    //@ assert increasing:  StrictLowerBound(a, k+1, i+1, a[k]);
    //@ assert increasing:  WeaklyIncreasing(a, k+1, i+1);
    //@ assert increasing:  WeaklyIncreasing(a, i+1);
    //@ assert reorder:     MultisetReorder{LoopCurrent,Here}(a, 0, k);
    //@ assert reorder:     MultisetReorder{LoopCurrent,Here}(a, k, i+1);
    //@ assert reorder:     MultisetReorder{LoopCurrent,Here}(a, 0, i+1);
    //@ assert reorder:     MultisetReorder{Pre,Here}(a, i+1);
  }

  //@ assert increasing: Increasing(a, n);
}
```

Listing 10.17: Implementation of insertion_sort

When we originally implemented and verified rotate, we hadn't yet in mind to use that function inside of insertion_sort. Consequently, the properties needed for the latter aren't directly provided by the former. One approach to solve this problem is to add the new properties to the contract of rotate [7.27] and repeat its verification proof. However, if rotate is assumed to be part of a pre-verified library, this approach isn't feasible, since rotate's implementation may not be available for re-verification. Therefore, we used another approach, viz. to prove that rotate's original specification *implies* all the properties we need in insertion_sort. This is another use of the Hoare calculus' implication rule (§3.3). We used several lemmas, shown below, to make the necessary implications explicit, and to help the provers to establish them. Some of them needed manual proofs by induction.

Lemma `Increasing_Equal` [6.3] in the following listing assumes an ordered range `a[m..n-1]` and claims that every (elementwise) equal range range `a[m+p..n+p-1]` is ordered, too. It is needed to establish that the call to `rotate` [7.27] preserves the order of those elements that are shifted upwards (cf. Figure 10.15).

Similarly, lemma `Count_Equal` [4.44] says that two elementwise equal ranges `a[m..n-1]` and `a[p..p+n-m-1]` will result in the same occurrence count, for each value `v`. This lemma is useful in the proof of the lemma `CircularShift_MultisetReorder` [10.18] (discussed below), since the predicate `MultisetReorder` [7.55] is defined via the logic function `Count` [4.44].

Lemma `CircularShift_StrictLowerBound` [10.18] in the next listing is used to prove that the range `a[k..i-1]` having `a[i]` as strict lower bound before our call to `rotate` ensures that it has `a[k]` as such a bound after the call. Note that this lemma reflects that `rotate` is uses as a *circular shift* at the call site. Similarly, lemma `CircularShift_MultisetReorder` establishes that a circular shift just reorders the range it is applied to.

```
/*@
   axiomatic CircularShiftLemmas
   {
     lemma CircularShift_StrictLowerBound{K,L}:
       \forall value_type* a, integer m, n;
         StrictLowerBound{K}(a, m, n, \at(a[n],K))   ==>
         Equal{K,L}(a, m, n, m+1)                    ==>
         Equal{K,L}(a, n, n+1, m)                    ==>
         StrictLowerBound{L}(a, m+1, n+1, \at(a[m],L));

     lemma CircularShift_MultisetReorder{K,L}:
       \forall value_type* a, integer m, n;
         0 <= m <= n                 ==>
         Equal{K,L}(a, m, n, m+1)   ==>
         Equal{K,L}(a, n, n+1, m)   ==>
         MultisetReorder{K,L}(a, m, n+1);
   }
*/
```

Listing 10.18: The logic definition(s) `CircularShiftLemmas`

10.6. The heap_sort algorithm

The heap_sort algorithm has the signature

```
void heap_sort(value_type* a, size_type n);
```

It relies upon the heap algorithms discussed in Chapter 9 to efficiently transform the array into increasing order.

10.6.1. Formal specification of heap_sort

The following Listing shows the specification of heap_sort.

```
/*@
   requires valid:        \valid(a + (0..n-1));
   assigns                a[0..n-1];
   ensures reorder:       MultisetReorder{Old,Here}(a, n);
   ensures increasing:    Increasing(a, n);
*/
void
heap_sort(value_type* a, size_type n);
```

Listing 10.19: Formal specification of heap_sort

10.6.2. Implementation of heap_sort

The implementation of heap_sort, shown in the next listing is straightforward. Given the input array a[0..n-1], we use make_heap [9.36] to arrange it into a heap; after that, we call sort_heap [9.39] to sort this heap into increasing order.

```
void
heap_sort(value_type* a, size_type n)
{
  make_heap(a, n);
  sort_heap(a, n);
}
```

Listing 10.20: Implementation of heap_sort

10.7. The `merge` algorithm

Our version of the `merge` algorithm from the C++ standard library [20, 28.7.5] has the following signature.

```
void
merge(const value_type* a, size_type n,
      const value_type* b, size_type m,
      value_type* result);
```

The `merge` algorithm is a part of the *merge sort* algorithm. It operates on the second step to merge two increasingly ordered sub-arrays into a new one. The algorithm merges two increasingly ordered arrays `a[0..n-1]` and `b[0..m-1]`, respectively. The merged values are stored in the output array that starts at `result` which must be able to hold $m + n$ values of both input arrays.

10.7.1. Formal specification of `merge`

The following listing 10.21 shows the specification of `merge`. The specification expects the input arrays of the proper size and in increasing order and the output array of enough size to contain all the input elements. The input arrays should not overlap with the output array. In the current edition of this guide, we prove only that the resulting array is in increasing order. Future editions will contain additional postconditions stating that the result array consists of reordered input elements and the stability of the algorithm, i.e., the same elements of the input arrays preserve their order in the output array.

```
/*@
  requires bound:        m + n <= SIZE_TYPE_MAX;
  requires valid:        \valid_read(a + (0..m-1));
  requires valid:        \valid_read(b + (0..n-1));
  requires valid:        \valid(c + (0..m+n-1));
  requires sep:          \separated(a + (0..m-1), c + (0..m+n-1));
  requires sep:          \separated(b + (0..n-1), c + (0..m+n-1));
  requires increasing:   Increasing(a, m);
  requires increasing:   Increasing(b, n);
  assigns                c[0 .. m+n-1];
  ensures  increasing:   Increasing(c, m + n);
  ensures  unchanged:    Unchanged{Old,Here}(a, m);
  ensures  unchanged:    Unchanged{Old,Here}(b, n);
 */
void
merge(const value_type* a, size_type m,
      const value_type* b, size_type n, value_type* c);
```

Listing 10.21: Formal specification of `merge`

10.7.2. More Lemmas on `WeaklyIncreasing`

We introduce in the following listing several lemmas about `WeaklyIncreasing` [6.2] that are helpful for the verification of `merge`.

- Lemma `WeaklyIncreasing_Shrink` [10.22] allows to restrict the property *weakly increasing* onto a sub-array.
- Lemma `WeaklyIncreasing_AddElement` [10.22] defines the way a weakly increasing array can be constructed.

- Lemma `WeaklyIncreasing_Shift` [10.22] is used to handle pointer arithmetic with respect to the `WeaklyIncreasing` property.

- Lemmas `WeaklyIncreasing_Unchanged` [10.22] and `WeaklyIncreasing_Equal` [10.22] state that if an array is weakly increasing, then another array (or the same array at another program point), whose elements are in a one-to-one correspondence with the original array, is also weakly increasing.

- Lemma `WeaklyIncreasing_Join` [10.22] defines the conditions that two consequent weakly increasing ranges can be viewed as merged weakly increasing range.

```
/*@
  axiomatic WeaklyIncreasingLemmas
  {
    lemma WeaklyIncreasing_Shrink{L}:
      \forall value_type *a, integer m, n, p, q;
        m <= p <= q <= n             ==>
        WeaklyIncreasing(a, m, n)   ==>
        WeaklyIncreasing(a, p, q);

    lemma WeaklyIncreasing_AddElement{L}:
      \forall value_type *a, integer n;
        1 < n                       ==>
        a[n-2] <= a[n-1]            ==>
        WeaklyIncreasing(a, n-1)    ==>
        WeaklyIncreasing(a, n);

    lemma WeaklyIncreasing_Shift{L}:
      \forall value_type *a, integer m, n;
        WeaklyIncreasing(a + m, 0, n)   <==>
        WeaklyIncreasing(a, m, n + m);

    lemma WeaklyIncreasing_Equal{K,L}:
      \forall value_type *a, *b, integer m, n;
        Equal{K,L}(a, m, n, b)          ==>
        WeaklyIncreasing{K}(a, m, n)    ==>
        WeaklyIncreasing{L}(b, m, n);

    lemma WeaklyIncreasing_Unchanged{K,L}:
      \forall value_type *a, integer m, n;
        WeaklyIncreasing{K}(a, m, n)    ==>
        Unchanged{K,L}(a, m, n)         ==>
        WeaklyIncreasing{L}(a, m, n);

    lemma WeaklyIncreasing_Join{L}:
      \forall value_type *a, integer m, n;
        0 < m < n                    ==>
        WeaklyIncreasing(a, m)       ==>
        WeaklyIncreasing(a, m, n)    ==>
        a[m-1] <= a[m]               ==>
        WeaklyIncreasing(a, n);
  }
*/
```

Listing 10.22: The logic definition(s) `WeaklyIncreasingLemmas`

10.7.3. Implementation of merge

The implementation of merge, shown in the next listings is straightforward. The algorithm operates by traversing both input arrays. On each iteration it writes the smaller of both elements into the result array, thus constructing an increasingly ordered array. If the algorithm reaches the end of one of the input arrays, it just copies the rest elements of the other array to the result array. The listing contains a number of assertions to trigger an application of lemmas by the provers. The **while** loop traverses the input arrays and constructs, in accordance with WeaklyIncreasing_AddElement [10.22], the resulting weakly increasing array. After the loop, the algorithm copies the remaining elements to the resulting array.

```
void
merge(const value_type* a, size_type m,
      const value_type* b, size_type n, value_type* c)
{
  //@ assert increasing:  WeaklyIncreasing(a, 0, m);
  size_type i = 0;
  size_type j = 0;
  size_type x = 0;

  if (0 < m || 0 < n) {
    /*@ loop invariant index:      0 <= i <= m;
        loop invariant index:      0 <= j <= n;
        loop invariant index:      x == i+j;
        loop invariant index:      0 <= x <= m+n-1;
        loop invariant upper:      i < m  ==>  UpperBound(c, 0, x, a[i]);
        loop invariant upper:      j < n  ==>  UpperBound(c, 0, x, b[j]);
        loop invariant increasing: WeaklyIncreasing(c, x);
        loop assigns i, j, x, c[0 .. m+n-1];
        loop variant (m+n) - (i+j);
     */
    while (i < m && j < n) {
      if (a[i] < b[j]) {
        c[x++] = a[i++];
        //@ assert increasing:  WeaklyIncreasing(c, 0, x);
        //@ assert upper:       i < m  ==>  UpperBound(c, 0, x, a[i]);
      }
      else {
        c[x++] = b[j++];
        //@ assert increasing:  WeaklyIncreasing(c, 0, x);
        //@ assert upper:       j < n  ==>  UpperBound(c, 0, x, b[j]);
      }

      //@ assert increasing:  WeaklyIncreasing(c, 0, x);
```

Listing 10.23: Implementation of merge (1)

We also use the following lemmas to support the verification of several properties.

- Lemma WeaklyIncreasing_Equal [10.22] is used to show that the copied elements from one of the input arrays preserve the WeaklyIncreasing property.
- Lemma WeaklyIncreasing_Join [10.22] is used to extend the WeaklyIncreasing property of the two sub-ranges of the resulting array over the whole range. In order to deal with pointer arithmetic we employ Lemma WeaklyIncreasing_Shift.
- Finally, Lemma WeaklyIncreasing_Increasing [6.3] is used to prove the output array is in increasing order.

```
    }

    //@ ghost  Epilogue: ;
    //@ assert index:       x == i+j;
    //@ assert index:       i == m  ^^  j == n;
    //@ assert index:       i <  m  ^^  j <  n;
    //@ assert increasing:  WeaklyIncreasing(c, 0, x);
    //@ assert unchanged:   Unchanged{Pre,Here}(a, 0, m);
    //@ assert increasing:  WeaklyIncreasing(a, 0, m);

    if (i < m) {
      //@ assert upper:       0 < x  ==>  c[x-1] <= a[i];
      //@ assert increasing:  WeaklyIncreasing(a, i, m);
      //@ assert increasing:  WeaklyIncreasing(a+i, 0, m-i);

      /*@
          assigns             c[x..x+m-i-1];
          ensures  equal:     Equal{Epilogue,Here}(a+i, m-i, c+x);
      */
      copy(a + i, m - i, c + x);
      //@ assert equal:       c[x] == At{Epilogue}(a, i);
      //@ assert equal:       a[i] == At{Epilogue}(a, i);
      //@ assert equal:       c[x] == a[i];
      //@ assert equal:       Equal{Epilogue,Here}(a+i, m-i, c+x);
      //@ assert increasing:  WeaklyIncreasing(c+x, 0, m-i);
      //@ assert index:       m-i+x == m+n;
      //@ assert increasing:  WeaklyIncreasing(c, x, m+n);
    }
    else {
      //@ assert upper:       0 < x  ==>  c[x-1] <= b[j];
      //@ assert unchanged:   Unchanged{Pre,Here}(b, 0, n);
      //@ assert increasing:  WeaklyIncreasing(b, 0, n);
      //@ assert increasing:  WeaklyIncreasing(b+j, 0, n-j);

      /*@
          assigns             c[x..x+n-j-1];
          ensures  equal:     Equal{Epilogue,Here}(b+j, n-j, c+x);
      */
      copy(b + j, n - j, c + x);
      //@ assert equal:       c[x] == At{Epilogue}(b, j);
      //@ assert equal:       b[j] == At{Epilogue}(b, j);
      //@ assert equal:       c[x] == b[j];
      //@ assert equal:       Equal{Epilogue,Here}(b+j, n-j, c+x);
      //@ assert increasing:  WeaklyIncreasing(c+x, 0, n-j);
      //@ assert index:       n-j+x == m+n;
      //@ assert increasing:  WeaklyIncreasing(c, x, m+n);
    }

    //@ assert unchanged:   Unchanged{Epilogue,Here}(c, 0, x);
    //@ assert increasing:  WeaklyIncreasing(c, 0, x);
    //@ assert increasing:  WeaklyIncreasing(c, x, m+n);
    //@ assert increasing:  0 < x  ==>  c[x-1] <= c[x];
    //@ assert increasing:  0 < x  ==>  WeaklyIncreasing(c, x-1, m+n);
    //@ assert increasing:  WeaklyIncreasing(c, 0, m+n);
  }
}
```

Listing 10.24: The Implementation of `merge` (2)

Part V.

Verification of data structures

11. The `stack` data type

So far we have used the ACSL specification language for the task of specifying and verifying one single C function at a time. However, in practice we are also faced with the task to implement a family of functions, usually around some sophisticated data structure, which have to obey certain rules of interdependence. In this kind of task, we are not only interested in the properties of a single function but also in properties describing how several function play together.

The C++ Standard Library provides a generic container adaptor `stack` [20, §26.6.6] whose signature and behavior we try to follow as far as our C implementation it allows. For a more detailed discussion of our approach to the formal verification of `stack` we refer to Kim Völlinger's thesis [27].

A *stack* is a data type that can hold objects and has the property that, if an object *a* is *pushed* on a stack *before* object *b*, then *a* can only be removed (*popped*) after *b*. A stack is, in other words, a *first-in, last-out* data type (see Figure 11.1). The *top* function of a stack returns the last element that has been pushed on a stack.

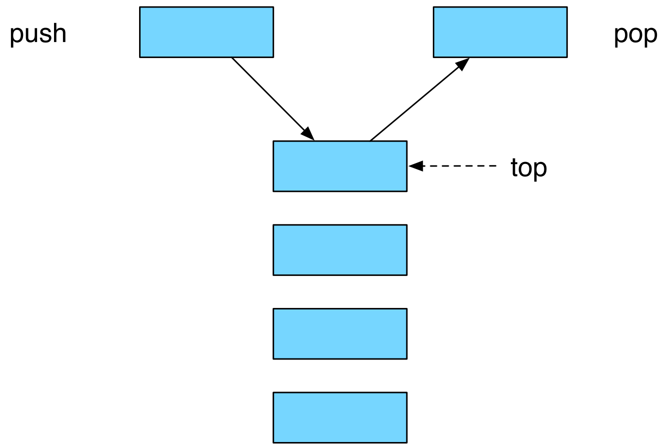

Figure 11.1.: Push and pop on a stack

We consider only stacks that have a finite *capacity*, that is, that can only hold a maximum number *c* of elements that is constant throughout their lifetime. This restriction allows us to define a stack without relying on dynamic memory allocation. When a stack is *created* or *initialized*, it contains no elements, i.e., its *size* is 0. The function *push* and *pop* increases and decreases the size of a stack by at most one, respectively.

11.1. Methodology overview

Figure 11.2 gives an overview of our methodology to specify and verify abstract data types (verification of one axiom shown only).

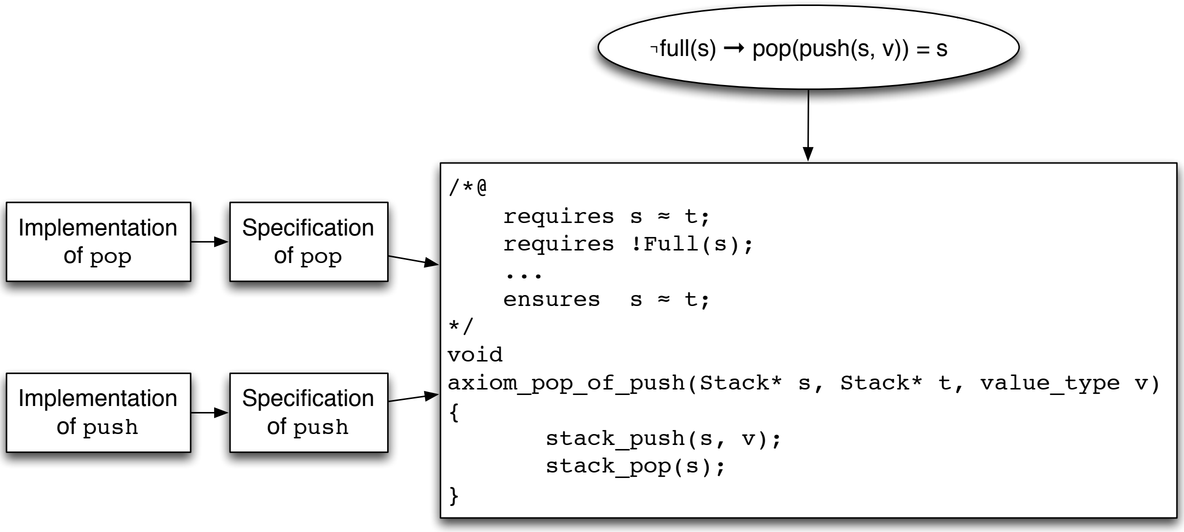

Figure 11.2.: Methodology Overview

What we will basically do is:

1. specify axioms about how the stack functions should interact with each other (§11.2),
2. define a basic implementation of C data structures (only one in our example, viz. `struct Stack`; see §11.3) and some invariants the instances of them have to obey (§11.4),
3. provide for each stack function an ACSL contract and a C implementation (§11.6),
4. verify each function against its contract (§11.6),
5. transform the axioms into ACSL-annotated C code (§11.7), and
6. verify that code, using access function contracts and data-type invariants as necessary (§11.7).

§11.5 provides an ACSL-predicate deciding whether two instances of a `struct Stack` are considered to be equal (indication by "≈" in Figure 11.2), while §11.6.1 gives a corresponding C implementation. The issue of an appropriate definition of equality of data instances is familiar to any C programmer who had to replace a faulty comparison `if(s1 == s2)` by the correct `if(strcmp(s1,s2)== 0)` to compare two strings `char *s1,*s2` for equality.

11.2. Stack axioms

To specify the interplay of the stack access functions, we use a set of axioms[35], all but one of them having the form of a conditional equation.

Let V denote an arbitrary type. We denote by S_c the type of stacks with capacity $c > 0$ of elements of type V. The aforementioned functions then have the following signatures.

$$
\begin{aligned}
\text{init} &: S_c \to S_c, \\
\text{push} &: S_c \times V \to S_c, \\
\text{pop} &: S_c \to S_c, \\
\text{top} &: S_c \to V, \\
\text{size} &: S_c \to \mathbb{N}.
\end{aligned}
$$

With $\mathbb{B}$ denoting the *boolean* type we will also define two auxiliary functions

$$
\begin{aligned}
\text{empty} &: S_c \to \mathbb{B}, \\
\text{full} &: S_c \to \mathbb{B}.
\end{aligned}
$$

To qualify as a stack these functions must satisfy the following rules which are also referred to as *stack axioms*.

11.2.1. Stack initialization

After a stack has been initialized its size is 0.

$$\text{size}(\text{init}(s)) = 0. \tag{11.1}$$

The auxiliary functions empty and full are defined as follows

$$\text{empty}(s), \qquad \text{iff} \qquad \text{size(s)} = 0, \tag{11.2}$$
$$\text{full}(s), \qquad \text{iff} \qquad \text{size(s)} = c. \tag{11.3}$$

We expect that for every stack s the following condition holds

$$0 \le \text{size}(s) \le c. \tag{11.4}$$

[35] There is an analogy in geometry: Euclid (e.g. [28]) invented the use of axioms there, but still kept definitions of *point*, *line*, *plane*, etc. Hilbert [29] recognized that the latter are not only unformalizable, but also unnecessary, and dropped them, keeping only the formal descriptions of relations between them.

11.2.2. Adding an element to a stack

To push an element v on a stack the stack must not be full. If an element has been pushed on an eligible stack, its size increases by 1

$$\text{size}(\text{push}(s, v)) = \text{size}(s) + 1, \qquad \text{if} \quad \neg\text{full}(s). \tag{11.5}$$

Moreover, the element pushed on a stack is the top element of the resulting stack

$$\text{top}(\text{push}(s, v)) = v, \qquad \text{if} \quad \neg\text{full}(s). \tag{11.6}$$

11.2.3. Removing an element from a stack

An element can only be removed from a non-empty stack. If an element has been removed from an eligible stack the stack size decreases by 1

$$\text{size}(\text{pop}(s)) = \text{size}(s) - 1, \qquad \text{if} \quad \neg\text{empty}(s). \tag{11.7}$$

If an element is pushed on a stack and immediately afterwards an element is removed from the resulting stack then the final stack is equal to the original stack

$$\text{pop}(\text{push}(s, v)) = s, \qquad \text{if} \quad \neg\text{full}(s). \tag{11.8}$$

Conversely, if an element is removed from a non-empty stack and if afterwards the top element of the original stack is pushed on the new stack then the resulting stack is equal to the original stack.

$$\text{push}(\text{pop}(s), \text{top}(s)) = s, \qquad \text{if} \quad \neg\text{empty}(s). \tag{11.9}$$

11.2.4. A note on exception handling

We don't impose a requirement on `push(s, v)` if `s` is a full stack, nor on `pop(s)` or `top(s)` if `s` is an empty stack. Specifying the behavior in such *exceptional* situations is a problem by its own; a variety of approaches is discussed in the literature. We won't elaborate further on this issue, but only give an example to warn about "innocent-looking" exception specifications that may lead to undesired results.

If we'd introduce an additional error value `err` in the element type V and require `top(s)= err` if `s` is empty, we'd be faced with the problem of specifying the behavior of `push(s, err)`. At first glance, it would seem a good idea to have `err` just been ignored by `push`, i.e. to require

$$\text{push}(s, \text{err}) = s. \tag{11.10}$$

However, we then could derive for any non-full and non-empty stack `s`, that

$$\begin{aligned} \text{size}(s) &= \text{size}(\text{pop}(\text{push}(s, \text{err}))) && \text{by 11.8} \\ &= \text{size}(\text{pop}(s)) && \text{as assumed in 11.10} \\ &= \text{size}(s) - 1 && \text{by 11.7} \end{aligned}$$

i.e. no such stacks could exist, or all **`int`** values would be equal.

11.3. The structure `stack` and its associated functions

We now introduce one possible C implementation of the above axioms. It is centred around the C structure `stack` shown in the following listing.

```
struct Stack
{
  value_type* obj;

  size_type   capacity;

  size_type   size;
};

typedef struct Stack Stack;
```

Listing 11.3: Definition of type `stack`

This struct holds an array `obj` of positive length called `capacity`. The capacity of a stack is the maximum number of elements this stack can hold. The field `size` indicates the number elements that are currently in the stack. See also Figure 11.4 which attempts to interpret this definition according to Figure 11.1.

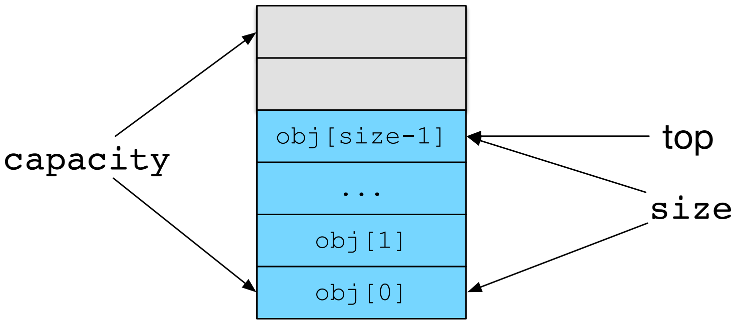

Figure 11.4.: Interpreting the data structure `stack`

Based on the stack functions from §11.2, we declare in the next listing the following functions as part of our `stack` data type.

```
void         stack_init(Stack* s, value_type* a, size_type n);

bool         stack_equal(const Stack* s, const Stack* t);

size_type    stack_size(const Stack* s);

bool         stack_empty(const Stack* s);

bool         stack_full(const Stack* s);

value_type   stack_top(const Stack* s);

void         stack_push(Stack* s, value_type v);

void         stack_pop(Stack* s);
```

Listing 11.5: Declaration of functions of type `stack`

Most of these functions directly correspond to methods of the C++ `std::stack` template class [20, §26.6.6.1]. The function `stack_equal` corresponds to the comparison operator ==, whereas one use of `stack_init` is to bring a stack into a well-defined initial state. The function `stack_full` has no counterpart in `std::stack`. This reflects the fact that we avoid dynamic memory allocation, while `std::stack` does not.

11.4. Stack invariants

Not every possible instance of type `stack` is considered a valid one, e.g., with our definition of `stack` in Listing 11.3, `Stack s = {{0,0,0,0},4,5}` is not. In the following listing, we present basic logic functions and predicates that we will use throughout this chapter In particular, we define the predicate `StackInvariant` [11.6] that discriminates valid and invalid instances.

```
/*@
  axiomatic StackInvariant
  {
    logic integer
    StackCapacity{L}(Stack* s) = s->capacity;

    logic integer
    StackSize{L}(Stack* s) = s->size;

    logic value_type*
    StackStorage{L}(Stack* s) = s->obj;

    logic integer
    StackTop{L}(Stack* s) = s->obj[s->size-1];

    predicate
    StackEmpty{L}(Stack* s) =  StackSize(s) == 0;

    predicate
    StackFull{L}(Stack* s)  =  StackSize(s) == StackCapacity(s);

    predicate
    StackInvariant{L}(Stack* s) =
      0 < StackCapacity(s) &&
      0 <= StackSize(s) <= StackCapacity(s) &&
      \valid(StackStorage(s) + (0..StackCapacity(s)-1)) &&
      \separated(s, StackStorage(s) + (0..StackCapacity(s)-1));
  }
*/
```

Listing 11.6: The logic definition(s) `StackInvariant`

We start, with the auxiliary logic function `StackCapacity`, `StackSize` and `StackStorage` which we can use in specifications to refer to the fields `capacity`, `size` and `obj` of `stack`, respectively. This listing also contains the logic function `StackTop` which defines the array element with index `size - 1` as the top place of a stack.

The reader can consider this as an attempt to hide implementation details from the specification. We intentionally use here integer as a return value of these logic functions. Inaccurate use of logic functions with bounded types in axioms with arithmetic operations may lead to inconsistencies.

We also introduce the predicates `StackEmpty` [11.6] and `StackFull` [11.6] that express the concepts of empty and full stacks by referring to a stack's size and capacity (see Equations (11.2) and (11.3)).

There are some obvious invariants that must be fulfilled by every valid object of type `stack`:

- The stack capacity shall be strictly greater than zero (an empty stack is ok but a stack that cannot hold anything is not useful).
- The pointer `obj` shall refer to an array of length `capacity`.
- The number of elements `size` of a stack the must be non-negative and not greater than its capacity.

These invariants are all formalized in the predicate `StackInvariant` [11.6].

Note how the use of the previously defined logic functions and predicates allows us to define the stack invariant without directly referring to the fields of `stack`.

We sometimes wish to express that there is no *memory aliasing* between two stacks. If there were aliasing, then modifying one stack could modify the other stack in unexpected ways. In order to express that there is no aliasing between two stacks, we define the predicate `StackSeparated` in the next listing.

```
/*@
  axiomatic StackUtility
  {
    predicate
    StackSeparated(Stack* s, Stack* t) =
      \separated(s, s->obj + (0..s->capacity-1),
                 t, t->obj + (0..t->capacity-1));

    predicate
    StackUnchanged{K,L}(Stack* s) =
      StackSize{K}(s)     == StackSize{L}(s)      &&
      StackStorage{K}(s)  == StackStorage{L}(s)   &&
      StackCapacity{K}(s) == StackCapacity{L}(s)  &&
      Unchanged{K,L}(StackStorage{K}(s), StackSize{K}(s));
  }
*/
```

Listing 11.7: The logic definition(s) `StackUtility`

This listing also contains the predicate `StackUnchanged` [11.7] that we will use to describe cases that the contents of a stack hasn't changed.

11.5. Equality of stacks

Defining equality of instances of non-trivial data types, in particular in object-oriented languages, is not an easy task. The book *Programming in Scala* [30, Chapter 28] devotes to this topic a whole chapter of more than twenty pages. In the following two sections we give a few hints how ACSL and Frama-C can help to correctly define equality for a simple data type.

We consider two stacks as equal if they have the same size and if they contain the same objects. To be more precise, let `s` and `t` two pointers of type `stack`, then we define the predicate `StackEqual` as in the following listing.

```
/*@
  axiomatic StackEquality
  {
    predicate
    StackEqual{S,T}(Stack* s, Stack* t) =
      StackSize{S}(s) == StackSize{T}(t) &&
      Equal{S,T}(StackStorage{S}(s), StackSize{S}(s), StackStorage{T}(t));

    lemma StackEqual_Reflexive{S} :
      \forall Stack* s; StackEqual{S,S}(s, s);

    lemma StackEqual_Symmetric{S,T} :
      \forall Stack *s, *t;
        StackEqual{S,T}(s, t)  ==>  StackEqual{T,S}(t, s);

    lemma StackEqual_Transitive{S,T,U}:
      \forall Stack *s, *t, *u;
        StackEqual{S,T}(s, t)  ==>
        StackEqual{T,U}(t, u)  ==>
        StackEqual{S,U}(s, u);
  }
*/
```

Listing 11.8: The logic definition(s) `StackEquality`

Our use of labels in this listing makes the specification somewhat hard to read (in particular in the last line where we reuse the predicate `Equal` [4.28]. However, this definition of `StackEqual` will allow us later to compare the same stack object at different points of a program. The logical expression `StackEqual {A,B}(s,t)` reads informally as: The stack object `*s` at program point `A` equals the stack object `*t` at program point `B`.

The reader might wonder why we exclude the capacity of a stack into the definition of stack equality. This approach can be motivated with the behavior of the method `capacity` of the class `std::vector<T>`. There, equal instances of type `std::vector<T>` may very well have different capacities.[36]

If equal stacks can have different capacities then, according to our definition of the predicate `StackFull` [11.6], we can have to equal stacks where one is full and the other one is not.

A finer, but very important point in our specification of equality of stacks is that the elements of the arrays `s->obj` and `t->obj` are compared only up to `s->size` and *not* up to `s->capacity`. Thus the two stacks `s` and `t` in Figure 11.9 are considered equal although there is are obvious differences in their internal arrays.

[36]See `http://www.cplusplus.com/reference/vector/vector/capacity`

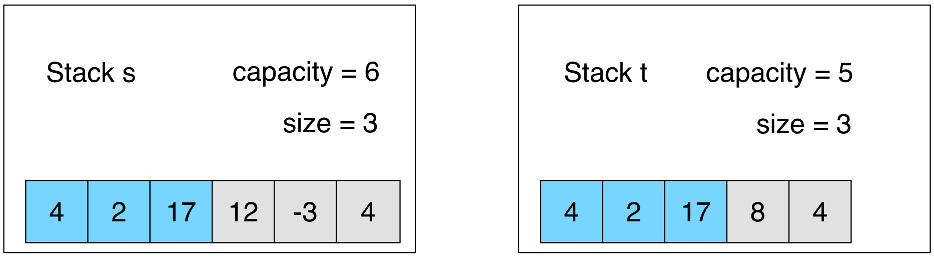

Figure 11.9.: Example of two equal stacks

If we define an equality relation (=) of objects for a data type such as `stack`, we have to make sure that the following rules hold.

$$\begin{aligned}
&\text{reflexivity} && \forall s \in S : s = s, && (11.11a)\\
&\text{symmetry} && \forall s, t \in S : s = t \implies t = s, && (11.11b)\\
&\text{transitivity} && \forall s, t, u \in S : s = t \wedge t = u \implies s = u. && (11.11c)
\end{aligned}$$

Any relation that satisfies the conditions (11.11) is referred to as an *equivalence relation*. The mathematical set of all instances that are considered equal to some given instance `s` is called the equivalence class of `s` with respect to that relation.

Our formalization of `StackEquality` [11.8] shows these three rules for the relation `StackEqual`; it can be automatically verified that they are a consequence of the definition of `StackEqual`.

The two stacks in Figure 11.9 show that an equivalence class of `StackEqual` can contain more than one element.[37] The stacks `s` and `t` in Figure 11.9 are also referred to as two *representatives* of the same equivalence class. In such a situation, the question arises whether a function that is defined on a set with an equivalence relation can be defined in such a way that its definition is *independent of the chosen representatives*.[38] We ask, in other words, whether the function is *well-defined* on the set of all equivalence classes of the relation `StackEqual`.[39] The question of well-definition will play an important role when verifying the functions of the `stack` (see §11.6).

[37] This is a common situation in mathematics. For example, the equivalence class of the rational number $\frac{1}{2}$ contains infinitely many elements, viz. $\frac{1}{2}, \frac{2}{4}, \frac{7}{14}, \ldots$.

[38] This is why mathematicians know that $\frac{1}{2} + \frac{3}{5}$ equals $\frac{7}{14} + \frac{3}{5}$.

[39] See `http://en.wikipedia.org/wiki/Well-definition`.

11.6. Verification of stack functions

In this section we verify the functions

- `stack_equal` (§11.6.1)
- `stack_init` (§11.6.2)
- `stack_size` (§11.6.3)
- `stack_full` (§11.6.4)
- `stack_empty` (§11.6.5)
- `stack_top` (§11.6.6)
- `stack_push` (§11.6.7)
- `stack_pop` (§11.6.8)

of the data type `stack`. To be more precise, we provide for each of function `stack_foo`:

- an ACSL specification of `stack_foo`
- a C implementation of `stack_foo`
- a C function `stack_foo_wd`[40] accompanied by a an ACSL contract that expresses that the implementation of `stack_foo` is well-defined. Figure 11.10 shows our methodology for the verification of well-definition in the `pop` example, (≈) again indicating the user-defined `stack` equality.

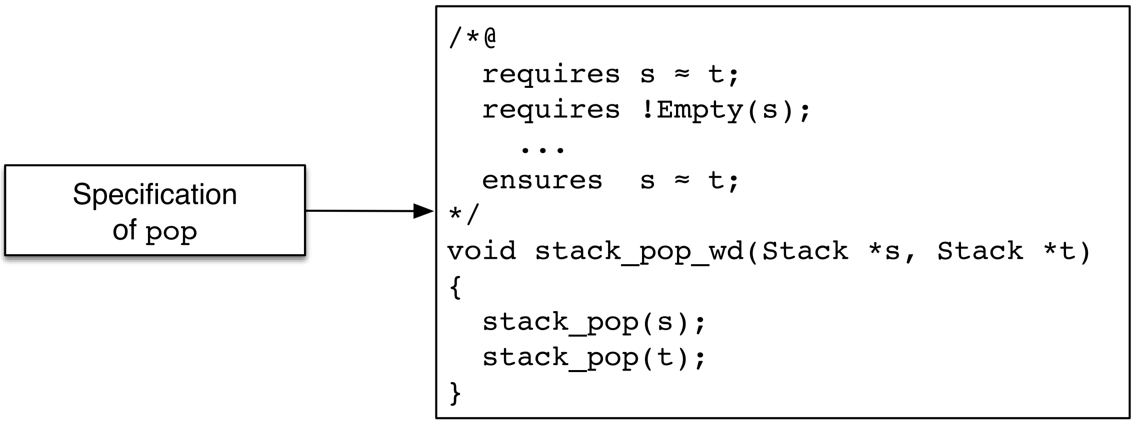

Figure 11.10.: Methodology for the verification of well-definition

Note that the specifications of the various functions will explicitly refer to the *internal state* of `stack`. In §11.7 we will show that the *interplay* of these functions satisfy the stack axioms from §11.2.

[40]The suffix `_wd` stands for *well definition*

11.6.1. The function `stack_equal`

The function `stack_equal` in the following listing is the runtime counterpart for the `StackEqual` [11.8] predicate. Note that this specifications explicitly refers to valid stacks.

```
/*@
    requires valid:     \valid(s)  && StackInvariant(s);
    requires valid:     \valid(t)  && StackInvariant(t);
    assigns             \nothing;
    ensures  equal:     \result == 1  <==>  StackEqual{Here,Here}(s, t);
    ensures  not_equal: \result == 0  <==> !StackEqual{Here,Here}(s, t);
*/
bool
stack_equal(const Stack* s, const Stack* t);
```

Listing 11.11: Formal specification of `stack_equal`

The implementation of `stack_equal` in the next listing compares two stacks according to the same rules of predicate `StackEqual`.

```
bool
stack_equal(const Stack* s, const Stack* t)
{
  return (s->size == t->size) && equal(s->obj, s->size, t->obj);
}
```

Listing 11.12: Implementation of `stack_equal`

11.6.2. The function `stack_init`

The following listing shows the specification of `stack_init`. Note that our specification of the post-conditions contains a redundancy because a stack is empty if and only if its size is zero.

```
/*@
  requires valid:     \valid(s);
  requires capacity:  0 < capacity;
  requires storage:   \valid(storage + (0..capacity-1));
  requires sep:       \separated(s, storage + (0..capacity-1));
  assigns             s->obj, s->capacity, s->size;
  ensures  valid:     \valid(s);
  ensures  capacity:  StackCapacity(s) == capacity;
  ensures  storage:   StackStorage(s) == storage;
  ensures  invariant: StackInvariant(s);
  ensures  empty:     StackEmpty(s);
*/
void
stack_init(Stack* s, value_type* storage, size_type capacity);
```

Listing 11.13: Formal specification of `stack_init`

The next listing shows the implementation of `stack_init`. It simply initializes `obj` and `capacity` with the respective value of the array and sets the field `size` to zero.

```
void
stack_init(Stack* s, value_type* storage, size_type capacity)
{
  s->obj      = storage;
  s->capacity = capacity;
  s->size     = 0u;
}
```

Listing 11.14: Implementation of `stack_init`

11.6.3. The function `stack_size`

The function `stack_size` is the runtime version of the logic function `StackSize` [11.6]. The specification of `stack_size` in the following listing simply states that `stack_size` produces the same result as `StackSize`.

```
/*@
    requires valid: \valid(s) && StackInvariant(s);
    assigns         \nothing;
    ensures   size: \result == StackSize(s);
*/
size_type
stack_size(const Stack* s);
```

Listing 11.15: Formal specification of `stack_size`

As in the definition of the logic function `StackSize` the implementation of `stack_size` in the next listing simply returns the field `size`.

```
size_type
stack_size(const Stack* s)
{
  return s->size;
}
```

Listing 11.16: Implementation of `stack_size`

The next listing shows our check whether `stack_size` is well-defined. Since `stack_size` neither modifies the state of its `stack` argument nor that of any global variable we only check whether it produces the same result for equal stacks. Note that we simply may use operator == to compare integers since we didn't introduce a nontrivial equivalence relation on that data type.

```
/*@
  requires valid:  \valid(s) && StackInvariant(s);
  requires valid:  \valid(t) && StackInvariant(t);
  requires equal:  StackEqual{Here,Here}(s, t);
  assigns          \nothing;
  ensures  equal:  \result;
*/
bool
stack_size_wd(const Stack* s, const Stack* t)
{
  return stack_size(s) == stack_size(t);
}
```

Listing 11.17: Implementation of `stack_size_wd`

11.6.4. The function `stack_full`

The function `stack_full` is the runtime version of the predicate `StackFull` [11.6].

```
/*@
    requires valid:    \valid(s) && StackInvariant(s);
    assigns            \nothing;
    ensures  full:     \result == 1  <==>  StackFull(s);
    ensures  not_full: \result == 0  <==> !StackFull(s);
*/
bool
stack_full(const Stack* s);
```

Listing 11.18: Formal specification of `stack_full`

As in the definition of the predicate `StackFull` the implementation of `stack_full` in the next listing simply checks whether the size of the stack equals its capacity.

```
bool
stack_full(const Stack* s)
{
  return stack_size(s) == s->capacity;
}
```

Listing 11.19: Implementation of `stack_full`

Note that with our definition of stack equality (§11.5) there can be equal stack with different capacities. As a consequence, there can are equal stacks where one is full while the other is not. In other words, `stack_full` is not well-defined!

11.6.5. The function `stack_empty`

The function `stack_empty` is the runtime version of the predicate `StackEmpty` [11.6].

```
/*@
    requires valid:    \valid(s) && StackInvariant(s);
    assigns            \nothing;
    ensures empty:     \result == 1  <==>  StackEmpty(s);
    ensures not_empty: \result == 0  <==> !StackEmpty(s);
*/
bool
stack_empty(const Stack* s);
```

Listing 11.20: Formal specification of `stack_empty`

As in the definition of the predicate `StackEmpty` the implementation of `stack_empty` in the next listing simply checks whether the size of the stack is zero.

```
bool
stack_empty(const Stack* s)
{
  return stack_size(s) == 0u;
}
```

Listing 11.21: Implementation of `stack_empty`

The following listing shows our check whether `stack_empty` is well-defined.

```
/*@
  requires valid:  \valid(s) && StackInvariant(s);
  requires valid:  \valid(t) && StackInvariant(t);
  requires equal:  StackEqual{Here,Here}(s, t);
  assigns          \nothing;
  ensures  equal:  \result;
*/
bool
stack_empty_wd(const Stack* s, const Stack* t)
{
  return stack_empty(s) == stack_empty(t);
}
```

Listing 11.22: Implementation of `stack_empty_wd`

11.6.6. The function stack_top

The function stack_top is the runtime version of the logic function StackTop [11.6]. The specification of stack_top in the following listing simply states that for non-empty stacks stack_top produces the same result as StackTop which in turn just returns the element obj[size-1] of stack.

```
/*@
    requires valid: \valid(s) && StackInvariant(s);
    assigns         \nothing;
    ensures  top:   !StackEmpty(s) ==> \result == StackTop(s);
*/
value_type
stack_top(const Stack* s);
```

Listing 11.23: Formal specification of stack_top

For a non-empty stack the implementation of stack_top in the next listing simply returns the element obj[size-1]. Note that our implementation of stack_top does not crash when it is applied to an empty stack. In this case we return the first element of the internal, non-empty array obj. This is consistent with our specification of stack_top which only refers to non-empty stacks.

```
value_type
stack_top(const Stack* s)
{
  if (!stack_empty(s)) {
    return s->obj[s->size - 1u];
  }
  else {
    return s->obj[0u];
  }
}
```

Listing 11.24: Implementation of stack_top

The next listing shows our check whether stack_top is well-defined. Since our axioms in §11.2 did not impose any behavior on the behavior of stack_top for empty stacks, we prove the well-definition of stack_top only for nonempty stacks.

```
/*@
  requires valid:  \valid(s) && StackInvariant(s) && !StackEmpty(s);
  requires valid:  \valid(t) && StackInvariant(t) && !StackEmpty(t);
  requires equal:  StackEqual{Here,Here}(s, t);
  assigns          \nothing;
  ensures  equal:  \result;
*/
bool
stack_top_wd(const Stack* s, const Stack* t)
{
  return stack_top(s) == stack_top(t);
}
```

Listing 11.25: Implementation of stack_top_wd

11.6.7. The function stack_push

The following listing shows the specification of the function stack_push. In accordance with Axiom (11.5), stack_push is supposed to increase the number of elements of a non-full stack by one. The specification also demands that the value that is pushed on a non-full stack becomes the top element of the resulting stack (see Axiom (11.6)).

```
/*@
  requires valid:      \valid(s) && StackInvariant(s);
  assigns              s->size, s->obj[s->size];

  behavior full:
    assumes            StackFull(s);
    assigns            \nothing;
    ensures valid:     \valid(s) && StackInvariant(s);
    ensures full:      StackFull(s);
    ensures unchanged: StackUnchanged{Old,Here}(s);

  behavior not_full:
    assumes            !StackFull(s);
    assigns            s->size;
    assigns            s->obj[s->size];
    ensures valid:     \valid(s) && StackInvariant(s);
    ensures size:      StackSize(s) == StackSize{Old}(s) + 1;
    ensures top:       StackTop(s) == v;
    ensures storage:   StackStorage(s) == StackStorage{Old}(s);
    ensures capacity:  StackCapacity(s) == StackCapacity{Old}(s);
    ensures not_empty: !StackEmpty(s);
    ensures unchanged: Unchanged{Old,Here}(StackStorage(s), StackSize{Old}(s));

  complete behaviors;
  disjoint behaviors;
*/
void
stack_push(Stack* s, value_type v);
```

Listing 11.26: Formal specification of stack_push

The implementation of stack_push is shown in the next listing. It checks whether its argument is a non-full stack in which case it increases the field size by one but only after it has assigned the function argument to the element obj[size].

```
void
stack_push(Stack* s, value_type v)
{
  if (!stack_full(s)) {
    //@ assert not_full: s->size < s->capacity;
    s->obj[s->size++] = v;
  }
}
```

Listing 11.27: Implementation of stack_push

The following listing shows our formalization of the well-definition for stack_push. The function stack_push does not return a value but rather modifies its argument. For the well-definition of stack_push we therefore check whether it turns equal stacks into equal stacks.

```
/*@
  requires valid:     \valid(s) && StackInvariant(s);
  requires valid:     \valid(t) && StackInvariant(t);
  requires equal:     StackEqual{Here,Here}(s, t);
  requires not_full:  !StackFull(s) && !StackFull(t);
  requires sep:       StackSeparated(s, t);
  assigns             s->size, s->obj[s->size];
  assigns             t->size, t->obj[t->size];
  ensures  valid:     StackInvariant(s) && StackInvariant(t);
  ensures  equal:     StackEqual{Here,Here}(s, t);
*/
void
stack_push_wd(Stack* s, Stack* t, value_type v)
{
  stack_push(s, v);
  stack_push(t, v);
  //@ assert top:   StackTop(s) == v;
  //@ assert top:   StackTop(t) == v;
  //@ assert equal: Equal{Here,Here}(StackStorage(s), StackSize{Pre}(s),
      StackStorage(t));
}
```

Listing 11.28: Implementation of `stack_push_wd`

However, equality of the stack arguments is not sufficient for a proof that `stack_push` is well-defined. We must also ensure that there is no *aliasing* between the two stacks. Otherwise modifying one stack could modify the other stack in unexpected ways. In order to express that there is no aliasing between two stacks, we use the predicate `StackSeparated` [11.7].

In order to achieve an automatic verification of `stack_push_wd` [11.28] we have added the assertions `top` and `equal` and introduced the lemma `StackPush_Equal` [11.29] in the following listing.

```
/*@
  axiomatic StackLemmas
  {
    lemma StackPush_Equal{K,L}:
      \forall Stack *s, *t;
        StackEqual{K,K}(s,t)                    ==>
        StackSize{L}(s) == StackSize{K}(s) + 1  ==>
        StackSize{L}(s) == StackSize{L}(t)      ==>
        StackTop{L}(s)  == StackTop{L}(t)       ==>
        Equal{L,L}(StackStorage{L}(s),
                   StackSize{K}(s),
                   StackStorage{L}(t))          ==>
        StackEqual{L,L}(s,t);
  }
*/
```

Listing 11.29: The logic definition(s) `StackLemmas`

11.6.8. The function stack_pop

The following listing shows the specification of the function stack_pop. In accordance with Axiom (11.7), stack_pop is supposed to reduce the number of elements in a non-empty stack by one. In addition to the requirements imposed by the axioms, our specification demands that stack_pop changes no memory location if it is applied to an empty stack.

```
/*@
  requires valid: \valid(s) && StackInvariant(s);
  assigns         s->size;
  ensures  valid: \valid(s) && StackInvariant(s);

  behavior empty:
    assumes            StackEmpty(s);
    assigns            \nothing;
    ensures empty:     StackEmpty(s);
    ensures unchanged: StackUnchanged{Old,Here}(s);

  behavior not_empty:
    assumes            !StackEmpty(s);
    assigns            s->size;
    ensures size:      StackSize(s) == StackSize{Old}(s) - 1;
    ensures full:      !StackFull(s);
    ensures storage:   StackStorage(s) == StackStorage{Old}(s);
    ensures capacity:  StackCapacity(s) == StackCapacity{Old}(s);
    ensures unchanged: Unchanged{Old,Here}(StackStorage(s), StackSize(s));

  complete behaviors;
  disjoint behaviors;
*/
void
stack_pop(Stack* s);
```

Listing 11.30: Formal specification of stack_pop

The implementation of stack_pop is shown in the next listing. It checks whether its argument is a non-empty stack in which case it decreases the field size by one.

```
void
stack_pop(Stack* s)
{
  if (!stack_empty(s)) {
    --s->size;
  }
}
```

Listing 11.31: Implementation of stack_pop

The next listing shows our check whether `stack_pop` is well-defined. As in the case of `stack_push` we use the predicate `StackSeparated` [11.7] in order to express that there is no aliasing between the two stack arguments.

```
/*@
  requires valid:  \valid(s) && StackInvariant(s);
  requires valid:  \valid(t) && StackInvariant(t);
  requires equal:  StackEqual{Here,Here}(s, t);
  requires sep:    StackSeparated(s, t);
  assigns          s->size;
  assigns          t->size;
  ensures valid:   StackInvariant(s);
  ensures valid:   StackInvariant(t);
  ensures equal:   StackEqual{Here,Here}(s, t);
*/
void
stack_pop_wd(Stack* s, Stack* t)
{
  stack_pop(s);
  stack_pop(t);
}
```

Listing 11.32: Implementation of `stack_pop_wd`

11.7. Verification of stack axioms

In this section we show that the stack functions defined in §11.6 satisfy the stack Axioms of §11.2.

The annotated code has been obtained from the axioms in a fully systematical way. In order to transform a condition equation $p \to s = t$:

- Generate a clause `requires p`.
- Generate a clause `requires x1 == ... == xn` for each variable `x` with n occurrences in s and t.
- Change the i-th occurrence of `x` to `xi` in s and t.
- Translate both terms s and t to reversed polish notation.
- Generate a clause `ensures y1 == y2`, where `y1` and `y2` denote the value corresponding to the translated s and t, respectively.

This makes it easy to implement a tool that does the translation automatically, but yields a slightly longer contract in our example.

11.7.1. Resetting a stack

Our formulation in ACSL /C of the axiom in Equation (11.1) is shown in the following listing.

```
/*@
  requires valid:  \valid(s);
  requires valid:  \valid(a + (0..n-1));
  requires sep:    \separated(s, a + (0..n-1));
  requires pos:    0 < n;
  assigns          s->obj, s->capacity, s->size;
  ensures  size:   \result == 0;
  ensures  valid:  StackInvariant(s);
*/
size_type
axiom_size_of_init(Stack* s, value_type* a, size_type n)
{
  stack_init(s, a, n);
  return stack_size(s);
}
```

Listing 11.33: Implementation of `axiom_size_of_init`

11.7.2. Adding an element to a stack

Axioms (11.5) and (11.6) describe the behavior of a stack when an element is added.

```
/*@
  requires valid:    \valid(s) && StackInvariant(s);
  requires not_full: !StackFull(s);
  assigns            s->size, s->obj[s->size];
  ensures size:      \result == StackSize{Old}(s) + 1;
  ensures valid:     StackInvariant(s);
*/
size_type
axiom_size_of_push(Stack* s, value_type v)
{
  stack_push(s, v);
  return stack_size(s);
}
```

Listing 11.34: Implementation of `axiom_size_of_push`

Except for the `assigns` clauses, the ACSL specification refers only to encapsulating logic functions and predicates defined in §11.4. If ACSL would provide a means to define encapsulating logic functions returning also sets of memory locations, the expressions in `assigns` clauses would not need to refer to the details of our `stack` implementation.[41] As an alternative, `assigns` clauses could be omitted, as long as the proofs are only used to convince a human reader.

```
/*@
  requires valid:     \valid(s) && StackInvariant(s);
  requires not_full:  !StackFull(s);
  assigns             s->size, s->obj[s->size];
  ensures  top:       \result == v;
*/
value_type
axiom_top_of_push(Stack* s, value_type v)
{
  stack_push(s, v);
  return stack_top(s);
}
```

Listing 11.35: Implementation of `axiom_top_of_push`

[41] In [15, §2.3.4], a powerful sublanguage to build memory location set expressions is defined. We will explore its capabilities in a later version.

11.7.3. Removing an element from a stack

This section shows the Listings for Axioms 11.7, 11.8 and 11.9 which describe the behavior of a stack when an element is removed.

```
/*@
  requires valid:  \valid(s) && StackInvariant(s);
  requires empty:  !StackEmpty(s);
  assigns          s->size;
  ensures   size:  \result == StackSize{Old}(s) - 1;
*/
size_type
axiom_size_of_pop(Stack* s)
{
  stack_pop(s);
  return stack_size(s);
}
```

Listing 11.36: Implementation of `axiom_size_of_pop`

```
/*@
  requires  valid:    \valid(s) && StackInvariant(s);
  requires  not_full: !StackFull(s);
  assigns             s->size, s->obj[s->size];
  ensures   equal:    StackEqual{Old,Here}(s, s);
*/
void
axiom_pop_of_push(Stack* s, value_type v)
{
  stack_push(s, v);
  stack_pop(s);
}
```

Listing 11.37: Implementation of `axiom_pop_of_push`

```
/*@
  requires  valid:     \valid(s) && StackInvariant(s);
  requires  not_empty: !StackEmpty(s);
  assigns              s->size, s->obj[s->size-1];
  ensures   equal:     StackEqual{Old,Here}(s, s);
*/
void
axiom_push_of_pop_top(Stack* s)
{
  const value_type v = stack_top(s);
  stack_pop(s);
  stack_push(s, v);
}
```

Listing 11.38: Implementation of `axiom_push_of_pop_top`

Part VI.

Appendices

A. Results of formal verification with Frama-C

In this chapter we introduce the formal verification tools used in this tutorial. We will afterwards present to what extent the examples from Chapters 4–11 could be deductively verified.

Within Frama-C, the Frama-C/WP plug-in [1] enables deductive verification of C programs that have been annotated with the ANSI/ISO-C Specification Language (ACSL) [9]. The Frama-C/WP plug-in uses weakest precondition computations to generate proof obligations. To formally prove the ACSL properties, these proof obligations can be submitted to external automatic theorem provers or interactive proof assistants. For the precise settings for Frama-C/WP we employed in this release we refer to Chapter 1.

In §A.2 and §A.3 we show detailed verification results for different scenarios how the provers are called.

A.1. Verification settings

Here are the most important options of Frama-C that we used in for almost all functions.[42]

```
-pp-annot
-no-unicode
-wp
-wp-rte
-wp-model Typed
-warn-unsigned-overflow
-warn-unsigned-downcast
-wp-timeout 1
-wp-coq-timeout 5
```

Note that we use a relative small timeout value for the provers. For a couple of algorithms, however, we had to use a larger timeout.

For the precise versions of the employed provers we refer to Table 1.1 on Page 3.

[42]For the `my_lrand48()` function in `shuffle`, the option `-warn-unsigned-overflow` is disabled as explained in §7.18.

A.2. Verification results (sequential)

In the *sequential verification scenario* each proof obligation is processed by a set of automatic and interactive theorem provers that are arranged as a *pipe*.[43] This means that each prover passes on to the next prover only those proof obligations that it could not verify. This *verification pipeline* is shown in Figure A.1.

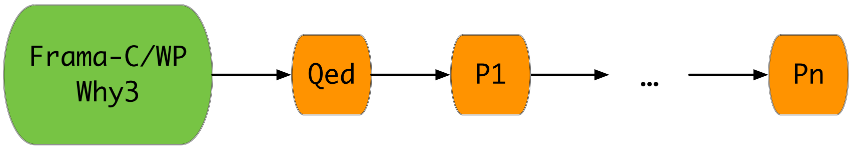

Figure A.1.: Verification pipeline of automatic and interactive theorem provers

For each algorithm we list in the following tables the number of generated verification conditions (VC), the percentage of proven verification conditions, and the number of VC proven by each prover. The value zero is indicated by an empty cell. The tables show that all verification conditions could be verified. Please note that the number of proven verification conditions do *not* reflect on the quality/strength of the individual provers. The reason for that is that we "pipe" each verification condition sequentially through a list of provers (see Figure A.1).

Algorithm		**Verification Conditions**		**Individual Provers** QD	AE	Z3	C4	CQ
`find`	§4.1	25/25	(100%)	16	9	·	·	·
`find2`	§4.2	27/27	(100%)	14	13	·	·	·
`find3`	§4.3	31/31	(100%)	8	19	·	·	4
`find4`	§4.3.4	33/33	(100%)	11	18	·	·	4
`find5`	§4.3.4	22/22	(100%)	5	13	·	·	4
`find_if_not`	§4.4	37/37	(100%)	8	23	·	·	6
`find_first_of`	§4.5	41/41	(100%)	30	11	·	·	·
`adjacent_find`	§4.6	28/28	(100%)	16	12	·	·	·
`mismatch`	§4.7	26/26	(100%)	16	10	·	·	·
`equal`	§4.7	7/ 7	(100%)	6	1	·	·	·
`search`	§4.8	44/44	(100%)	32	12	·	·	·
`search_n`	§4.9	93/93	(100%)	62	31	·	·	·
`find_end`	§4.10	34/34	(100%)	21	13	·	·	·
`count`	§4.11	34/34	(100%)	7	20	·	·	7
`count2`	§4.12	42/42	(100%)	7	25	·	·	10

Table A.2.: Results for non-mutating algorithms

[43] Sequential processing is achieved by passing the option `-wp-par 1` to Frama-C/WP.

Algorithm		Verification Conditions		Individual Provers QD	AE	Z3	C4	CQ
`clamp`	§5.3	28/28	(100%)	22	6	·	·	·
`make_pair`	§5.4	4/ 4	(100%)	4	·	·	·	·
`max_element`	§5.5	30/30	(100%)	19	11	·	·	·
`max_element2`	§5.6	30/30	(100%)	18	12	·	·	·
`max_seq`	§5.7	8/ 8	(100%)	5	3	·	·	·
`min_element`	§5.8	30/30	(100%)	18	12	·	·	·
`minmax_element`	§5.9	60/60	(100%)	43	17	·	·	·

Table A.3.: Results for maximum and minimum algorithms

Algorithm		Verification Conditions		Individual Provers QD	AE	Z3	C4	CQ
`lower_bound`	§6.1	19/19	(100%)	5	14	·	·	·
`upper_bound`	§6.2	19/19	(100%)	7	12	·	·	·
`equal_range`	§6.3	22/22	(100%)	17	5	·	·	·
`equal_range2`	§6.3	70/70	(100%)	24	39	·	·	7
`binary_search`	§6.4	10/10	(100%)	8	2	·	·	·
`binary_search2`	§6.4	12/12	(100%)	8	4	·	·	·

Table A.4.: Results for binary search algorithms

Algorithm		Verification Conditions		Individual Provers QD	AE	Z3	C4	CQ
`fill`	§7.2	14/ 14	(100%)	4	10	·	·	·
`swap`	§7.3	8/ 8	(100%)	5	3	·	·	·
`swap_ranges`	§7.4	22/ 22	(100%)	5	17	·	·	·
`copy`	§7.5	15/ 15	(100%)	4	11	·	·	·
`copy_backward`	§7.6	17/ 17	(100%)	7	10	·	·	·
`reverse_copy`	§7.7	17/ 17	(100%)	4	13	·	·	·
`reverse`	§7.8	24/ 24	(100%)	5	19	·	·	·
`rotate_copy`	§7.9	17/ 17	(100%)	5	12	·	·	·
`rotate`	§7.10	24/ 24	(100%)	10	14	·	·	·
`replace_copy`	§7.11	19/ 19	(100%)	7	12	·	·	·
`replace`	§7.12	15/ 15	(100%)	4	11	·	·	·
`remove_copy`	§7.13	23/ 23	(100%)	9	14	·	·	·
`remove_copy2`	§7.14	74/ 74	(100%)	9	46	1	·	18
`remove_copy3`	§7.15	108/108	(100%)	13	70	1	·	24
`remove`	§7.16	103/103	(100%)	10	69	·	·	24
`shuffle`	§7.17	55/ 55	(100%)	12	34	·	·	9
`random_number`	§7.18	33/ 33	(100%)	19	14	·	·	·

Table A.5.: Results for mutating algorithms

Algorithm		Verification Conditions		Individual Provers				
				QD	AE	Z3	C4	CQ
iota	§8.1	16/16	(100%)	7	9	·	·	·
accumulate	§8.2	23/23	(100%)	6	15	·	·	2
inner_product	§8.3	25/25	(100%)	6	17	·	·	2
partial_sum	§8.4	56/56	(100%)	16	38	·	·	2
adjacent_difference	§8.5	35/35	(100%)	11	24	·	·	·
partial_sum_inv	§8.6	39/39	(100%)	8	28	·	·	3
adjacent_difference_inv	§8.6	39/39	(100%)	8	28	·	·	3

Table A.6.: Results for numeric algorithms

Algorithm		Verification Conditions		Individual Provers				
				QD	AE	Z3	C4	CQ
heap_parent	§9.3	11/ 11	(100%)	3	8	·	·	·
heap_child	§9.3	38/ 38	(100%)	7	30	·	·	1
is_heap_until	§9.4	34/ 34	(100%)	6	27	·	·	1
is_heap	§9.5	19/ 19	(100%)	5	13	·	·	1
push_heap	§9.7	117/117	(100%)	32	72	2	·	11
pop_heap	§9.8	134/134	(100%)	35	86	2	·	11
make_heap	§9.9	64/ 64	(100%)	17	38	·	·	9
sort_heap	§9.10	73/ 73	(100%)	17	44	·	·	12

Table A.7.: Results for heap algorithms

Algorithm		Verification Conditions		Individual Provers				
				QD	AE	Z3	C4	CQ
is_sorted	§10.1	18/ 18	(100%)	7	8	·	·	3
partial_sort	§10.2	146/146	(100%)	39	88	·	·	19
bubble_sort	§10.3	85/ 85	(100%)	22	51	·	·	12
selection_sort	§10.4	67/ 67	(100%)	15	36	·	·	16
insertion_sort	§10.5	81/ 81	(100%)	18	50	·	·	13
heap_sort	§10.6	45/ 45	(100%)	8	28	·	·	9
merge	§10.7	111/111	(100%)	30	72	4	·	5

Table A.8.: Results for algorithms related to sorting

Algorithm		Verification Conditions		Individual Provers QD	AE	Z3	C4	CQ
stack_equal	§11.6.1	18/18	(100%)	7	11	·	·	·
stack_init	§11.6.2	14/14	(100%)	4	10	·	·	·
stack_size	§11.6.3	6/ 6	(100%)	1	5	·	·	·
stack_full	§11.6.4	11/11	(100%)	5	6	·	·	·
stack_empty	§11.6.5	10/10	(100%)	5	5	·	·	·
stack_top	§11.6.6	16/16	(100%)	6	10	·	·	·
stack_push	§11.6.7	41/41	(100%)	25	16	·	·	·
stack_pop	§11.6.8	29/29	(100%)	17	12	·	·	·

Table A.9.: Results for `stack` functions

Algorithm		Verification Conditions		Individual Provers QD	AE	Z3	C4	CQ
stack_size_wd	§11.6.3	12/12	(100%)	8	4	·	·	·
stack_empty_wd	§11.6.5	12/12	(100%)	8	4	·	·	·
stack_top_wd	§11.6.6	12/12	(100%)	8	4	·	·	·
stack_push_wd	§11.6.7	15/15	(100%)	3	12	·	·	·
stack_pop_wd	§11.6.8	12/12	(100%)	6	6	·	·	·

Table A.10.: Results for the well-definition of the `stack` functions

Algorithm		Verification Conditions		Individual Provers QD	AE	Z3	C4	CQ
axiom_size_of_init	§11.7.1	15/15	(100%)	11	4	·	·	·
axiom_size_of_push	§11.7.2	12/12	(100%)	9	3	·	·	·
axiom_top_of_push	§11.7.2	11/11	(100%)	8	3	·	·	·
axiom_size_of_pop	§11.7.3	11/11	(100%)	8	3	·	·	·
axiom_pop_of_push	§11.7.3	10/10	(100%)	6	4	·	·	·
axiom_push_of_pop_top	§11.7.3	15/15	(100%)	9	6	·	·	·

Table A.11.: Results for `stack` axioms

A.3. Verification results (parallel)

In the *parallel verification scenario* each proof obligation is first passed to Frama-C/WP's built-in simplifier Qed. If Qed cannot discharge a proof obligation it is submitted in parallel to *all* the other provers from Table 1.1.[44] Figure A.12 depicts this arrangement of provers. This arrangement of theorem provers makes it a little bit easier to quantify their strength.

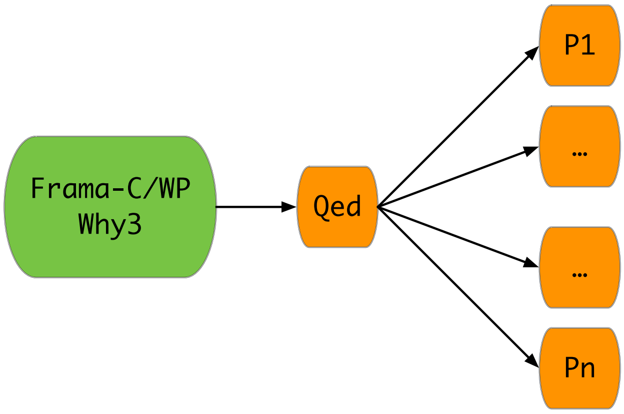

Figure A.12.: Parallel execution of theorem provers

Note that in this scenario we used Frama-C/WP only for the generation and simplification of the proof obligations. For the parallel execution we developed our own (shell) scripts that pass the proof obligations directly through Why3 to the individual provers.

Algorithm		**Verification Conditions**		**Individual Provers**				
				QD	AE	Z3	C4	CQ
`find`	§4.1	25/25	(100%)	16	9	9	5	1
`find2`	§4.2	27/27	(100%)	14	13	6	9	1
`find3`	§4.3	31/31	(100%)	8	19	13	13	5
`find4`	§4.3.4	33/33	(100%)	11	18	7	15	4
`find5`	§4.3.4	22/22	(100%)	5	13	7	11	4
`find_if_not`	§4.4	37/37	(100%)	8	23	14	13	7
`find_first_of`	§4.5	41/41	(100%)	30	11	5	7	1
`adjacent_find`	§4.6	28/28	(100%)	16	12	8	5	2
`mismatch`	§4.7	26/26	(100%)	16	10	7	5	1
`equal`	§4.7	7/ 7	(100%)	6	1	1	1	·
`search`	§4.8	44/44	(100%)	32	12	9	4	1
`search_n`	§4.9	93/93	(100%)	62	31	23	18	4
`find_end`	§4.10	34/34	(100%)	21	13	7	4	1
`count`	§4.11	34/34	(100%)	7	20	15	9	8
`count2`	§4.12	42/42	(100%)	7	25	21	11	11

Table A.13.: Results for non-mutating algorithms

[44]We did not include the interactive theorem prover Coq in this setting.

Algorithm		Verification Conditions		Individual Provers QD	AE	Z3	C4	CQ
clamp	§5.3	28/28	(100%)	22	6	6	6	3
make_pair	§5.4	4/ 4	(100%)	4	·	·	·	·
max_element	§5.5	30/30	(100%)	19	11	11	6	3
max_element2	§5.6	30/30	(100%)	18	12	9	7	1
max_seq	§5.7	8/ 8	(100%)	5	3	3	1	·
min_element	§5.8	30/30	(100%)	18	12	9	7	1
minmax_element	§5.9	60/60	(100%)	43	17	12	9	1

Table A.14.: Results for maximum and minimum algorithms

Algorithm		Verification Conditions		Individual Provers QD	AE	Z3	C4	CQ
lower_bound	§6.1	19/19	(100%)	5	14	11	5	1
upper_bound	§6.2	19/19	(100%)	7	12	10	3	·
equal_range	§6.3	22/22	(100%)	17	5	2	2	·
equal_range2	§6.3	70/70	(100%)	24	39	23	11	12
binary_search	§6.4	10/10	(100%)	8	2	1	·	·
binary_search2	§6.4	12/12	(100%)	8	4	1	2	·

Table A.15.: Results for binary search algorithms

Algorithm		Verification Conditions		Individual Provers QD	AE	Z3	C4	CQ
fill	§7.2	14/ 14	(100%)	4	10	6	6	1
swap	§7.3	8/ 8	(100%)	5	3	3	3	·
swap_ranges	§7.4	22/ 22	(100%)	5	17	11	8	1
copy	§7.5	15/ 15	(100%)	4	11	8	5	1
copy_backward	§7.6	17/ 17	(100%)	7	10	7	5	1
reverse_copy	§7.7	17/ 17	(100%)	4	13	11	5	2
reverse	§7.8	24/ 24	(100%)	5	19	15	6	2
rotate_copy	§7.9	17/ 17	(100%)	5	12	10	3	·
rotate	§7.10	24/ 24	(100%)	10	14	7	3	·
replace_copy	§7.11	19/ 19	(100%)	7	12	10	4	1
replace	§7.12	15/ 15	(100%)	4	11	8	4	1
remove_copy	§7.13	23/ 23	(100%)	9	14	10	6	·
remove_copy2	§7.14	74/ 74	(100%)	9	46	33	26	18
remove_copy3	§7.15	108/108	(100%)	13	70	49	31	24
remove	§7.16	103/103	(100%)	10	69	47	29	25
shuffle	§7.17	55/ 55	(100%)	12	34	24	12	9
random_number	§7.18	33/ 33	(100%)	19	14	13	6	1

Table A.16.: Results for mutating algorithms

Algorithm		Verification Conditions		Individual Provers				
				QD	AE	Z3	C4	CQ
iota	§8.1	16/16	(100%)	7	9	7	5	1
accumulate	§8.2	23/23	(100%)	6	15	10	6	4
inner_product	§8.3	25/25	(100%)	6	17	14	7	3
partial_sum	§8.4	56/56	(100%)	16	38	23	15	4
adjacent_difference	§8.5	35/35	(100%)	11	24	21	6	1
partial_sum_inv	§8.6	39/39	(100%)	8	28	15	12	6
adjacent_difference_inv	§8.6	39/39	(100%)	8	28	16	12	6

Table A.17.: Results for numeric algorithms

Algorithm		Verification Conditions		Individual Provers				
				QD	AE	Z3	C4	CQ
heap_parent	§9.3	11/ 11	(100%)	3	8	8	3	1
heap_child	§9.3	38/ 38	(100%)	7	30	27	12	2
is_heap_until	§9.4	34/ 34	(100%)	6	27	23	9	5
is_heap	§9.5	19/ 19	(100%)	5	13	10	4	2
push_heap	§9.7	117/117	(100%)	32	72	50	21	13
pop_heap	§9.8	134/134	(100%)	35	86	59	28	14
make_heap	§9.9	64/ 64	(100%)	17	38	29	15	10
sort_heap	§9.10	73/ 73	(100%)	17	44	31	18	18

Table A.18.: Results for heap algorithms

Algorithm		Verification Conditions		Individual Provers				
				QD	AE	Z3	C4	CQ
is_sorted	§10.1	18/ 18	(100%)	7	8	5	2	4
partial_sort	§10.2	146/146	(100%)	39	88	54	26	22
bubble_sort	§10.3	85/ 85	(100%)	22	51	38	17	18
selection_sort	§10.4	67/ 67	(100%)	15	36	32	12	17
insertion_sort	§10.5	81/ 81	(100%)	18	50	31	18	16
heap_sort	§10.6	45/ 45	(100%)	8	28	21	8	10
merge	§10.7	111/111	(100%)	30	72	43	26	16

Table A.19.: Results for algorithms related to sorting

Algorithm		**Verification Conditions**		**Individual Provers**				
				QD	AE	Z3	C4	CQ
`stack_equal`	§11.6.1	18/18	(100%)	7	11	7	·	·
`stack_init`	§11.6.2	14/14	(100%)	4	10	8	1	·
`stack_size`	§11.6.3	6/ 6	(100%)	1	5	3	2	1
`stack_full`	§11.6.4	11/11	(100%)	5	6	4	1	·
`stack_empty`	§11.6.5	10/10	(100%)	5	5	3	2	·
`stack_top`	§11.6.6	16/16	(100%)	6	10	8	2	1
`stack_push`	§11.6.7	41/41	(100%)	25	16	14	·	·
`stack_pop`	§11.6.8	29/29	(100%)	17	12	10	·	·

Table A.20.: Results for `stack` functions

Algorithm		**Verification Conditions**		**Individual Provers**				
				QD	AE	Z3	C4	CQ
`stack_size_wd`	§11.6.3	12/12	(100%)	8	4	2	1	·
`stack_empty_wd`	§11.6.5	12/12	(100%)	8	4	2	1	·
`stack_top_wd`	§11.6.6	12/12	(100%)	8	4	1	·	·
`stack_push_wd`	§11.6.7	15/15	(100%)	3	12	6	·	·
`stack_pop_wd`	§11.6.8	12/12	(100%)	6	6	3	·	·

Table A.21.: Results for the well-definition of the `stack` functions

Algorithm		**Verification Conditions**		**Individual Provers**				
				QD	AE	Z3	C4	CQ
`axiom_size_of_init`	§11.7.1	15/15	(100%)	11	4	2	1	1
`axiom_size_of_push`	§11.7.2	12/12	(100%)	9	3	1	·	·
`axiom_top_of_push`	§11.7.2	11/11	(100%)	8	3	1	·	·
`axiom_size_of_pop`	§11.7.3	11/11	(100%)	8	3	1	·	·
`axiom_pop_of_push`	§11.7.3	10/10	(100%)	6	4	2	·	·
`axiom_push_of_pop_top`	§11.7.3	15/15	(100%)	9	6	4	2	·

Table A.22.: Results for `stack` axioms

B. Changes in previous releases

This chapter describes the changes in previous versions of this document. For the most recent changes we refer to Chapter 1.

The version numbers of this document are related to the versioning of Frama-C [2]. The versions of Frama-C are named consecutively after the elements of the periodic table. Therefore, our version numbering (X.Y.Z) are constructed as follows:

X the major number of our tutorial is the atomic number[45] of the chemical element after which Frama-C is named.

Y the Frama-C subrelease number

Z the subrelease number of this tutorial

B.1. New in Version 21.1.0 (Scandium, July 2020)

This release is intended for Frama-C [2, v21.1] issued in June 2020. We are also using for this release the Why3 platform [3, v1.3.1] and the provers listed in the following table.

Prover	Type	Version	Reference
Alt-Ergo	automatic	2.3.2	[4]
CVC4	automatic	1.7	[5]
CVC3	automatic	2.4.1	[6]
Z3	automatic	4.8.8	[7]
Coq	interactive	8.9.1	[8]

Table B.1.: Information on automatic and interactive theorem provers

Note that all automatic provers use the Why3 interface. However, the interactive prover Coq still relies on the native interface provided by Frama-C/WP.

New examples

None.

Improvements

- Improve many code annotations in order to maintain the verification rate and to reduce timeout values.
- Use predicate `WeaklyIncreasing` instead of `Increasing` in the assertions and invariants of sorting algorithms. This allows the removal of lemma `IncreasingUpperBound`.

[45] See `http://en.wikipedia.org/wiki/Atomic_number`

- Add Coq to parallel verification
- Replace an ACSL lemma on integer division by a Coq lemma in driver.
- `remove_copy` and `remove`
  - Remove logic helper function `NextNotEqual` for `RemovePartition` which served as a workaround in Frama-C 20.
  - Remove lemma `RemovePartition_StrictlyIncreasing`.
  - Make the definition of predicate `Remove` more flexible.
  - Remove lemma `Remove_Update`.
- heap algorithms
  - Simplify definition of predicates `MultisetRetainRest` and `MultisetMinus`.
  - Add predicate `PushHeapAdjust`.
  - Add lemmas `PushHeapAdjust_Init` and `PushHeapAdjust_Finish`.
  - Rename lemma `MultisetPushHeapRetain` to `PushHeapAdjust_Retain`.
  - Add predicate `HeapCompatible` and lemmas `Heap_Shrink`, `Heap_Unchanged` and `Heap_Update`.
  - Improve annotations of `push_heap` and `pop_heap`.
  - Remove predicate `HeapChildMax` and simplify the contract of `heap_child_max`.
- Remove lemma `SwappedInside_Preserve` which was of limited usefulness.
- Add lemmas `Accumulate_Init`, `AccumulateBounds_Read` and `Accumulate_Read_Shrink` , which were suggested by Allan Blanchard, in to simplify the verification of `partial_sum` and `adjacent_difference_inv`.
- Add lemmas `Unchanged_Symmetric` and `MultisetUnchanged_Symmetric`.

Open issues The following algorithms and/or lemmas are not completely verified

- `pop_heap`
- `merge`

B.2. New in Version 20.0.2 (Calcium, April 2020)

This release is intended for Frama-C [2, v20.0] issued in December 2019. We are also using for this release the Why3 platform [3, v1.2.1] and the provers listed in the following table.

Prover	Type	Version	Reference
Alt-Ergo	automatic	2.3.2	[4]
CVC4	automatic	1.6	[5]
CVC3	automatic	2.4.1	[6]
Z3	automatic	4.8.6	[7]
Coq	interactive	8.9.1	[8]

Table B.2.: Information on automatic and interactive theorem provers

Note that all automatic provers use the Why3 interface. However, the interactive prover Coq still relies on the native interface provided by Frama-C/WP.

New examples

- Add examples `find4` and `find5` that verify the equivalence of the contracts of `find2` and `find3`.
- Add example `find_if_not`.

Improvements

- Add indices for examples and logic definitions.
- Re-add results of running all provers in parallel. Thanks to Allan Blanchard for explaining how Frama-C/WP's *session* mechanism can be used in the implementation.
- Fix a ghost label in `partial_sort`. Thanks to Virgile Prevosto for pointing out stricter checks in upcoming releases of Frama-C.
- Reduce very long verification times of several examples.
  - Add assertion `unchanged` to empty else branch of `remove_copy3`.
  - Add assertion `reorder` to empty else branch of `shuffle`.
  - Rewrite assertion `update` of `remove`.
  - Add another assertion `heap` to `push_heap`.
- Remove chapter on `unique_copy` because on its reliance on axioms. Moreover, the main ideas are already extensively discussed in the sections on `remove_copy` and `remove`.
- Verify properties of operator < within example `clamp`.
- Improve admitted Coq proof of `Reorder_Match`.
- Fix misplaced arrow in figure of `equal_range` algorithm

Open issues The following algorithms and/or lemmas are not completely verified

- `pop_heap` (property `reorder`)
- `merge` (property `reorder`)
- `Reorder_Match`

B.3. New in Version 20.0.1 (Calcium, March 2020)

This release is intended for Frama-C [2, v20.0] issued in December 2019. We are also using for this release the Why3 platform [3, v1.2.1] and the provers listed in the following table.

Prover	Type	Version	Reference
Alt-Ergo	automatic	2.3.1	[4]
CVC4	automatic	1.6	[5]
CVC3	automatic	2.4.1	[6]
Z3	automatic	4.8.6	[7]
Coq	interactive	8.9.1	[8]

Table B.3.: Information on automatic and interactive theorem provers

Note that all automatic provers use the Why3 interface. However, the interactive prover Coq still relies on the native interface provided by Frama-C/WP.

New examples

- add a third version of `find` that is specified using the new logic function `Find`

Improvements

- improve text in many places
- improve specification of `remove_copy` and `remove`
  - provide an explicit definition of `RemovePartition` that allows to replace axioms by lemmas
  - rename predicate `ConstantRange` to `AllEqual` and add its negation `SomeNotEqual`
  - add logic functions `CountNotEqual` and `FindNotEqual`
- place all logic definitions in `axiomatic` blocks to better control generated names
- make names of ACSL predicates, functions and lemmas more uniform and place them together in files where appropriate
- among the renamed ACSL entities are
  - rename predicate `HasValue` to `SomeEqual` and add its negation `NoneEqual`
  - rename lemma `HasValueImpliesPositiveCount` to `SomeEqualCount`
  - rename lemma `PositiveCountImpliesHasValue` to `Count_SomeEqual`
  - rename `RotatePreservesStrictLowerBound` to `CircularShift_StrictLowerBound`
  - rename `RotateImpliesMultisetUnchanged` to `CircularShiftMultisetUnchanged`

Open issues

The following algorithms and/or lemmas are not completely verified

- pop_heap
- Reorder_Match

B.4. New in Version 20.0.0 (Calcium, December 2019)

Aside from the above-mentioned version of Frama-C we are using for this release the Why3 platform [3, v1.2.1] and the provers listed in the following table. Note that all automatic provers are use the Why3 interface. In other words, we do not use anymore the native interface for Alt-Ergo.

Prover	Type	Version	Reference
Alt-Ergo	automatic	2.3.0	[4]
CVC4	automatic	1.6	[5]
CVC3	automatic	2.4.1	[6]
Z3	automatic	4.8.6	[7]
Coq	interactive	8.9.1	[8]

Table B.4.: Information on automatic and interactive theorem provers

New examples

- add bubble_sort

Improvements

- remove Why3 and Alt-Ergo lemmas from driver
- switch from memory model 'Typed+Ref' to 'Typed'
- the E theorem prover is not yet supported by this version of Frama-C
- no results on parallel verification are reported in this release
- rewrite random_shuffle to shuffle
  - adapt signature of random_number
  - add auxiliary function random_init
- replace, where applicable, ghost labels by loop labels or statement labels
- remove lemma SwapImpliesMultisetUnchanged by using predicate SwappedInside and its related lemmas
- improve specification and verification rate of numeric algorithms
  - resolve overloaded version of Accumulate into AccumulateDefault
  - resolve overloaded version of AccumulateBounds into AccumulateDefaultBounds
  - improve definition of predicate PartialSum
  - add lemmas Difference_Zero and Difference_Next
  - add predicate DefaultBounds

- add assigns in behaviors of maxmin and non-mutating algorithms
  - find, find2, find_first_of, adjacent_find, mismatch, search, find_end
  - max_element, max_element2, min_element, minmax_element
- rename predicate Sorted to Increasing; also rename related logic names
  - rename EqualRangesPreservesSorted ↦ EqualRangesPreservesIncreasing
  - rename SortedUpperBound ↦ IncreasingUpperBound
  - rename WeaklySortedAddElement ↦ WeaklyIncreasingAddElement
  - rename WeaklySortedShift ↦ WeaklyIncreasingShift
  - rename EqualRangesWeaklySorted ↦ EqualRangesWeaklyIncreasing
  - rename WeaklySortedJoin ↦ WeaklyIncreasingJoin
  - rename WeaklySortedLemmas ↦ WeaklyIncreasingLemmas
  - rename SortedIFFWeaklySorted ↦ IncreasingIFFWeaklyIncreasing
  - rename SortedImpliesWeaklySorted ↦ IncreasingImpliesWeaklyIncreasing
  - rename WeaklySortedImpliesSorted ↦ WeaklyIncreasingImpliesIncreasing
  - rename WeaklySorted ↦ WeaklyIncreasing
  - rename SortedShift ↦ Increasing_Shift
- remove lemma SortedDownIsHeap

Open issues

The following algorithms and/or lemmas are not completely verified

- adjacent_difference_inv
- pop_heap
- random_number
- ReorderImpliesMatch

B.5. New in Version 19.1.0 (Potassium, October 2019)

This release is intended for Frama-C 19.1 (*Potassium*), issued in September 2019. [2]

Aside from the above-mentioned version of Frama-C we are using for this release the Why3 platform [3, v1.2.0] and the provers listed in the following table. Note that all automatic provers are use the Why3 interface. In other words, we do not use anymore the native interface for Alt-Ergo.

Prover	Type	Version	Reference
Alt-Ergo	automatic	2.3.0	[4]
CVC4	automatic	1.6	[5]
CVC3	automatic	2.4.1	[6]
Z3	automatic	4.8.6	[7]
E	automatic	2.3	[31]
Coq	interactive	8.9.1	[8]

Table B.5.: Information on automatic and interactive theorem provers

Improvements

- Rename arguments of `search` and `find_end` and improve also the description of these algorithms.
- Rename and reorder arguments of `search_n`, make the verification more robust and improve its description.
- Make verification of property `size` of `remove_copy2` more robust.
- Explain role of lemma `RemoveImpliesNotHasValue` in `remove_copy3` and `remove`.
- Simplify definition of `RemoveSize` and `RemovePartition`.
- Make verification of property `reorder` of `partial_sort` more robust.
- Strengthen precondition of `replace_copy`.
- Rename lemma `random_number_modulo` into `RandomNumberModulo`.
- Differentiate between properties `unique` and `solitary` for `unique_copy` examples.
- Simplify the implementation of `is_heap` by calling the new function `is_heap_until`.
- Replace remaining instances of label `Pre` in contracts by `Old`.
- Unify use of `Unchanged` predicate for mutating algorithms.

New examples

- Add the algorithm `clamp` which “clips” a value between a pair of boundary values.
- Add the algorithm `minmax_element` and improve description of other algorithms related to finding minimum and maximum values.
- Add new example `is_heap_until` that generalizes `is_heap`.
- The following examples are not new since they were implicitly used as helper functions for other examples. They are now explicitly listed as examples.
  - `make_pair`
  - `random_number`
  - `heap_parent`
  - `heap_child_max` (formerly known as `heap_maximum_child`

Open issues

The following algorithms and/or lemmas are not completely verified

- adjacent_difference_inv
- partial_sum_inv
- pop_heap
- ReorderImpliesMatch

B.6. New in Version 19.0.0 (Potassium, June 2019)

- Structure of document
  - The document is now structured into several parts.
  - The chapter on classic sorting algorithms has been merged into the chapter on sorting.
  - The various variants of unique_copy are now grouped into a separate chapter.
- Fix various inconsistencies
  - Change the return types of the logic functions Accumulate, Difference, Capacity, Size, Top from bounded one (e.g., value_type, size_type) to integer. A combination of bounded type for a logic function with an arithmetic operations in the logical definitions may lead to inconsistency. This fixes the inconsistencies in the accumulate, stack and stack_wd examples.
  - Fix an inconsistency in DifferenceRead axiom: restriction on the array size added to premises.
- Various improvements
  - An important change is the rewriting of the implicit, *axiomatic* definitions of Accumulate, Count, Difference, InnerProduct and UniqueSize logic functions to explicit, *recursive* ones. Accordingly, all axioms in the respective examples have been rewritten as lemmas.
  - Generalize CountSectionMonotonic, UnchangedSection lemmas: remove restriction on lower bound for the range.
  - Fix typo in postcondition of find.
  - Rewrite specifications of remove_copy and remove examples.
  - Rename predicate RemoveCount to RemoveSize.
  - Gather all versions of MultisetRetainRest in section on push_heap.
  - Add another figure to highlight simple contract for unique_copy.
  - Adapt Coq proofs to the fact that the Z scope is not available by default.
- New examples
  - Add count2 example with an inductive predicate instead of a logic function in count.
  - Add merge example.
- Infrastructure

- Travis-CI configuration for the GitHub repository added as an illustrative example of how the verification results could be reproduced.
- Add support for Frama-C/AstraVer plugin.

B.7. New in Version 18.0.0 (Argon, December 2018)

- Replace the links to the (now abandoned) original site of *Standard Template Library* (STL) by references to the C++ standard.
- Add new algorithm `unique_copy` (two versions).
- Add another assertion `half` for `reverse`.
- Add two overloaded versions of predicate `ConstantRange` and use them for the algorithms `fill` and `unique_copy`, respectively.

B.8. New in Version 17.1.0 (Chlorine, July 2018)

The exact version number of Frama-C originally was Chlorine-20180502. This version number was changed in October 2018 to 17.1

- Slightly change the definition of predicate `HasEqualNeighbors` and its use in the specification of `adjacent_find`.
- Remove the algorithm `remove` and the more elaborate version of `remove_copy`. We are currently working on new specifications of these algorithms.
- Adapt some Coq proofs related to the logic function `Count` in order to reflect changes in output of Frama-C/WP.
- Remove table on ACSL lemmas that had to be proved by Coq.

B.9. New in Version 16.1.1 (Sulfur, March 2018)

- fix several errors reported by Aaron Rocha, including,
  - fix an error in figure for `upper_bound` algorithms
- fix merging of contracts in second version of `binary_search`
- improve and justify the `retain` annotations of in the implementation of `remove`
- Alt-Ergo is now directly called in the parallel setting (instead of going through Why3) to be compatible with the sequential setting
- add a third assertion `reorder` in the `random_shuffle` body to keep verification rate at 100% after prover upgrade

B.10. New in Version 16.1.0 (Sulfur, December 2017)

- special thanks to Aaron Rocha who provided various improvements for Chapters 4, 5, and 6
- improve some mutating algorithms
  - add more assertions to `reverse` to reduce reliance on CVC3
  - improve structure and ACSL annotations of `remove_copy` and `remove`
    - add overloaded version of predicate `MultisetRetainRest`
    - add lemma `HasValueImpliesPositiveCount`
    - add lemma `PositiveCountImpliesHasValue`
    - remove lemma `HasValueShiftInversion`
    - remove lemma `HasValueCountInversion`
  - add custom lemma `random_number_modulo` for `random_shuffle`
- add new Chapter 10 with more algorithms related to sorting
  - add algorithm `is_sorted` including predicate `WeaklyIncreasing`
    - add lemma `IncreasingImpliesWeaklyIncreasing`
    - add lemma `WeaklyIncreasingImpliesIncreasing`
  - add algorithm `partial_sort` including predicate `Partition`
    - add lemma `ReorderImpliesMatch`
    - add lemma `ReorderPreservesUpperBound`
    - add lemma `ReorderPreservesLowerBound`
    - add lemma `PartialReorderPreservesLowerBounds`
    - add lemma `SwappedInside`
    - add lemma `SwappedInsideMultisetUnchanged`
    - add lemma `SwappedInsidePreservesMultisetUnchanged`
- improve various lemmas
  - rename lemma `SortedUp` to `IncreasingUpperBound`
  - generalize lemma `UnchangedSection`
  - refactor lemma `HeapBounds` into `C_Division_Two`

B.11. New in Version 15.1.2 (Phosphorus, October 2017)

- fix several typos reported by `seniorlackey@github` (thanks a lot!)
- add a new chapter on classic sorting algorithms which comprises
  - `selection_sort` including lemma `SwapImpliesMultisetUnchanged`

  - insertion_sort including lemmas
    * RotatePreservesStrictLowerBound
    * RotateImpliesMultisetUnchanged
    * EqualRangesPreservesIncreasing
    * EqualRangesPreservesCount
  - heap_sort
- heap algorithms
  - remove length requirements in pop_heap, sort_heap, make_heap, and heap_sort
    * introduce SIZE_TYPE_MAX to catch border cases in ACSL and C
  - improve description of pop_heap
    * add predicate HeapChildMax
    * provide the auxiliary function heap_child_max
    * the postcondition reorder is still not verified
  - improve description of push_heap
  - other, minor improvements
    * add auxiliary function heap_parent
    * add predicate SortedDown and lemma SortedDownIsHeap
    * add lemmas HeapParentChild and HeapChilds
    * add lemmas HeapParentBounds and HeapChildBounds

B.12. New in Version 15.1.1 (Phosphorus, September 2017)

- add ensures clause to default behavior of the following algorithms
  - find, find_first_of, adjacent_find, mismatch, search, search_n, find_end
  - max_element, min_element
- rewrite axiomatic definitions to ensure disjoint guards which is better suited for E-ACSL
  - concerns the axiomatic definitions of Count, Accumulate, InnerProduct and Difference
  - some Coq proofs related to Count had to be adapted as well
- shorten names of some auxiliary algorithms
  - adjacent_difference_inverse $\mapsto$ adjacent_difference_inv
  - partial_sum_inverse $\mapsto$ partial_sum_inv
- heap algorithms
  - fix a typo in Figure 9.3

- fix a typo in Figure 9.38
- explain that there can be multiple representations of an array as a heap
- add a version of `pop_heap` that is, however, not completely verified

B.13. New in Version 15.1.0 (Phosphorus, June 2017)

- The verification results are now part of the appendix.
- Fix an error in the specification of the well-definition of `stack_size`.
- This release of Frama-C/WP could not discharge some of our assertions of `push_heap`. We therefore have completely rewritten the annotations and also tweaked the implementation of `push_heap`. We also added some new predicates and lemmas to maintain a concise specification that can easily be verified by automatic provers.
  - add predicate `MultisetAdd` and lemma `MultisetAdd_Distinct`
  - add predicate `MultisetMinus` and lemma `MultisetMinus_Distinct`
  - add predicate `MultisetRetain` and lemma `MultisetPushHeapRetain`
  - provide an additional version of predicate `MultisetRetainRest`
  - and lemma `MultisetPushHeapClosure`

B.14. New in Version 14.1.1 (Silicon, April 2017)

- changes in verification infrastructure
  - add verification results for the case where each proof obligation is submitted to all automatic theorem provers
- changes in algorithms
  - simplify loop invariants of `search_n` and improve description
  - rename predicate `CountOneHit` to `CountHit`
  - rename predicate `CountOneMiss` to `CountMiss`
  - rewrite predicates `EqualRanges` and `Reverse` in order to simplify the task for automatic theorem provers
  - remove lemmas on `Reverse` that were necessary for `rotate` but are not needed anymore
  - rename predicate `Valid(Stack*)` to `Invariant(Stack*)` and
    remove `\valid` from `Invariant(Stack*)`
  - add a simple random number generator to `random_shuffle` and verify it
- fix an inconsistency in the axioms for `Count` (thanks to Denis Efremov for reporting this issue)
  - add more guards to axioms `CountSectionHit` and `CountSectionMiss`
  - add corresponding guards to lemmas
    * `CountSectionOne`, `CountHit`, `CountMiss` and `CountOne`

        * RemoveCountHit and RemoveCountMiss

  - add lemma Unchanged_Shift and add more assertions to remove in order to simplify the task for automatic theorem provers

B.15. New in Version 14.1.0 (Silicon, January 2017)

- use label Old instead of Pre in function contracts
- add algorithm rotate
- rewrite definition of predicates EqualRanges and Reverse and provide more overloaded versions
- add figures for algorithms rotate and replace_copy
- update figure for predicate Reverse
- update Coq proofs and add a table with more information on the ACSL lemmas that had to be verified with Coq

B.16. New in Version 13.1.1 (Aluminium, November 2016)

- improve layout of tables of verification results
- use two additional automatic theorem provers (CVC3 and E)
- non-mutating algorithms
  - add algorithm find_end
  - add definition of predicate HasSubRange on subranges
  - add definition of predicate EqualRanges on subranges
  - rename lemma HasSubRange_fit_size to HasSubRangeSize
  - rename lemma HasConstantSubRange_fit_size to HasSubRangeSize
  - rename logic function CountSection to Count (using overloading in ACSL)
  - add lemma HasValueCountInversion
  - add lemma HasValueShiftInversion
  - add lemma Count_Shift
- mutating algorithms
  - add algorithm copy_backward
  - relax precondition on separation of copy, replace_copy and remove_copy
  - provide a more sophisticated implementation of remove
  - re-introduce a second version of remove_copy that also specifies the *stability* of the algorithm
  - add algorithm random_shuffle

B.17. New in Version 13.1.0 (Aluminium, August 2016)

The most notable changes of this version are the re-introduction of heap algorithms in Chapter 9. This new description of heap algorithms is based to a large extend on the bachelor thesis of one of the authors [24].

- provide names ("labels") for more ACSL annotations
- non-mutating algorithms
  - reorder and improve description in chapter on non-mutating algorithms
  - add more figures to describe algorithms
  - add non-mutating algorithm `search_n`
  - rewrite logic function `Count` with new logic function `CountSection`
  - move lemmas `Count_Bounds` and `CountMonotonic` to separate files
  - use `integer` instead of `size_type` in `HasSubRange`
  - change index computation in `HasEqualNeighbors`
- maximum and minimum algorithms
  - isolate predicate `ConstantRange` from predicates on lower and upper bounds
  - fix typo in precondition of first version of `max_element`
- binary search algorithms
  - add version `Sorted` for subranges
  - add second (more efficient) version of `equal_range`
    * add lemmas `SortedShift`, `LowerBound_Shift`, `StrictLowerBound_Shift`, `UpperBound_Shift` and `StrictUpperBound_Shift` to support the automatic verification of this version of `equal_range`
  - add figures to binary search algorithms and improve description
- mutating algorithms
  - greatly reduce the number of assertions needed to verify the first version `remove_copy`
  - temporarily remove the second version of `remove_copy` which also specified the *stability* of the algorithm
  - add `remove`, an in-place variant of `remove_copy`
  - rename predicate `RetainAllButOne` to `MultisetRetainRest`
- re-introduce chapter on heap algorithms
  - includes the heap algorithms `is_heap`, `push_heap`, `make_heap` and `sort_heap`
  - for `pop_heap` only a function contract is provided in this version
  - add lemma `SortedUp` to support verification of `sort_heap`
  - add several lemmas to combine the predicates `Unchanged` and `MultisetUnchanged`

B.18. New in Version 12.1.0 (Magnesium, February 2016)

A main goal of this release is to reduce the number of proof obligations that cannot be verified automatically and therefore must be tackled by an interactive theorem prover such as Coq. To this end, we analyzed the proof obligations (often using Coq) and devised additional assertions or ACSL lemmas to guide the automatic provers. Often we succeeded in enabling automatic provers to discharge the concerned obligations. Specifically, whereas the previous version 11.1.1 of *ACSL by Example* listed *nine* proof obligations that could only be discharged with Coq, the document at hand (version 12.1.0) only counts *five* such obligations. Moreover, all these remaining proof obligations are associated to ACSL lemmas, which are usually easier to tackle with Coq than proof obligations directly related to the C code. The reason for this is that ACSL lemmas usually have a much smaller set of hypotheses.

Adding assertions and lemmas also helps to alleviate a problem in Frama-C/WP Magnesium and Sodium where prover processes are not properly terminated.[46] Left-over "zombie processes" lead to a deterioration of machine performance which sometimes results in unpredictable verification results.

- mutating algorithms
  - simplify annotations of `replace_copy` and add new algorithm `replace`
    - add predicate `Replace` to write more compact post conditions and loops invariants
  - add several lemmas for predicate `Unchanged` and use predicate `Unchanged` in postconditions of mutating and numeric algorithms
  - simplify annotations of `reverse`
    - rename `Reversed` to `Reverse` (again) and provide another overloaded version
    - add figure to support description of the `Reverse` predicate
  - changes regarding `remove_copy`
    - rename `PreserveCount` to `RetainAllButOne`
    - rename `StableRemove` to `RemoveMapping`
    - add statement contracts for both versions of `remove_copy` such that only ACSL lemmas require Coq proofs
- numeric algorithms
  - define limits `VALUE_TYPE_MIN` and `VALUE_TYPE_MAX`
  - simplify specification of `iota` by using new logic function `Iota`
  - simplify implementation of `accumulate`
    - add overloaded predicates `AccumulateBounds`
    - add lemmas `AccumulateDefault0`, `AccumulateDefault1`, `AccumulateDefaultNext`, and `AccumulateDefault_Read`
  - simplify implementation of `inner_product`
    - add predicates `ProductBounds` and `InnerProductBounds`
  - enable automatic verification of `partial_sum`

[46] See `https://bts.frama-c.com/view.php?id=2154`

  * add lemmas PartialSumSection, PartialSumUnchanged, PartialSum_Step
    ,
    and PartialSumStep2 to automatically discharge loop invariants
- enable automatic verification of adjacent_difference
  * add logic function Difference and predicate AdjacentDifference
  * add predicate AdjacentDifferenceBounds
  * add lemmas AdjacentDifference_Step and AdjacentDifference_Section to automatically discharge proof obligation
- add two auxiliary functions partial_sum_inverse and adjacent_difference_inverse in order to verify that partial_sum and adjacent_difference are inverse to each other
  * add lemmas PartialSumInverse and AdjacentDifferenceInverse to support the automatic verification of the auxiliary functions

- stack functions
  - add lemma StackPush_Equal to enable the automatic verification of the well-definition of stack_push

B.19. New in Version 11.1.1 (Sodium, June 2015)

- add Chapter on numeric algorithms
  - move iota algorithm to numeric algorithms (§8.1)
  - add accumulate algorithm (§8.2)
  - add inner_product algorithm (§8.3)
  - add partial_sum algorithm (§8.4)
  - add adjacent_difference algorithm (§8.5)

B.20. New in Version 11.1.0 (Sodium, March 2015)

- Use built-in predicates \valid and \valid_read instead of valid_range.
- Simplify loop invariants of find_first_of.
- Replace two loop invariants of remove_copy by ACSL lemmas.
- Rename several predicates
  - IsEqual ↦ EqualRanges.
  - IsMaximum ↦ MaxElement.
  - IsMinimum ↦ MinElement.
  - Reverse ↦ Reversed.
  - IsSorted ↦ Sorted.

- Several changes for `stack`:
  - Rename `stack` functions from *`foo`*`_stack` to `stack_`*`foo`*.
  - Equality of stacks now ignores the `capacity` field. This is similar to how equality for objects of type `std::vector<T>` is defined. As a consequence `stack_full` is not well-defined any more. Other stack functions are not effected.
  - Remove all assertions from stack functions (including in axioms).
  - Describe predicate `Separated` in text.

B.21. New in Version 10.1.1 (Neon, January 2015)

- use option `-wp-split` to create simpler (but more) proof obligations
- simplify definition of predicate `Count`
- add new predicates for lower and upper bounds of ranges and use it in
  - `max_element`
  - `min_element`
  - `lower_bound`
  - `upper_bound`
  - `equal_range`
  - `fill`
- use a new auxiliary assertion in `equal_range` to enable the complete *automatic* verification of this algorithm
- add predicate `Unchanged` and use it to simplify the specification of several algorithms
  - `swap_ranges`
  - `reverse`
  - `remove_copy`
  - `stack_push` and `stack_push_wd`
  - `stack_pop` and `stack_pop_wd`
- add predicate `Reverse` and use it for more concise specifications of
  - `reverse_copy`
  - `reverse`
- several changes in the two versions of `remove_copy`
  - use predicate `HasValue` instead of logic function `Count`
  - add predicate `PreserveCount`
  - reformulate logic function `RemoveCount`
  - add predicate `StableRemove`

- add predicate RemoveCountMonotonic
- add predicate RemoveCountJump

- use overloading in ACSL to create shorter logic names for stack
- remove unnecessary labels in several stack functions

B.22. New in Version 10.1.0 (Neon, September 2014)

- remove additional labels in the assumes clauses of some stack function that were necessary due to an error in Oxygen
- provide a second version of remove_copy in order to explain the specification of the *stability* of the algorithms
- coarsen loop assigns of mutating algorithms
- temporarily remove the unique_copy algorithm

B.23. New in Version 9.3.1 (Fluorine, not published)

- specify bounds of the return value of count and fix reads clause of Count predicate
- use an auxiliary function make_pair in the implementation of equal_range
- provide more precise loop assigns clauses for the mutating algorithms
  - simplify implementation of fill
  - removed the ensures \valid(p) clause in specification of swap
  - simplify implementation of swap_ranges
  - simplify implementation of copy
  - fix implementation of reverse_copy after discovering an undefined behavior
  - new implementation of reverse that uses a simple **for**-loop
  - simplify implementation of replace_copy
  - refactor specification and simplify implementation of remove_copy
- remove work-around with Pre-label in assumes clauses of stack_push and stack_pop

B.24. New in Version 9.3.0 (Fluorine, December 2013)

- adjustments for *Fluorine* release of Frama-C
- swap now ensures that its pointer arguments are valid after the function has been called
- change definition of size_type to **unsigned int**
- change implementation of the iota algorithm . The content of the field a is calculated by increasing the value val instead of sum val+i.

- change implementation of `fill`.
- The specification/implementation of `stack` has been revised by Kim Völlinger [27] and now has a much better verification rate.

B.25. New in Version 8.1.0 (Oxygen, not published)

- simplified specification and loop annotations of `replace_copy`
- add binary search variant `equal_range`
- greatly simplified specification of `remove_copy` by using the logic function `Count`
- remove chapter on heap operations

B.26. New in Version 7.1.1 (Nitrogen, August 2012)

- improvements with respect to several suggestions and comments of Yannick Moy, e.g., specification refinements of `remove_copy`, `reverse_copy` and `iota`
- restricted verification of algorithms to Frama-C/WP with Alt-Ergo
- replaced deprecated `\valid_range` by `\valid`
- fixed inconsistencies in the description of the `stack` data type
- binary search algorithms can now be proven without additional axioms for integer division
- changed axioms into lemmas to document that provability is expected, even if not currently granted
- adopted new Fraunhofer logo and contact email

B.27. New in Version 7.1.0 (Nitrogen, December 2011)

- changed to Frama-C Nitrogen
- changed to Why 2.30
- discussed both plug-ins Frama-C/WP and Jessie
- removed `swap_values` algorithm

B.28. New in Version 6.1.0 (Carbon, not published)

- changed definition of `stack`
- renamed `reset_stack` to `init_stack`

B.29. New in Version 5.1.1 (Boron, February 2011)

- prepared algorithms for checking by the new Frama-C/WP plug-in of Frama-C

- changed to Alt-Ergo Version 0.92, Z3 Version 2.11 and Why 2.27
- added List of user-defined predicates and logic functions
- added remarks on the relation of logical values in C and ACSL
- rewrote section on `equal` and `mismatch`
- used a simpler logical function to count elements in an array
- added `search` algorithm
- added chapter to unite the maximum/minimum algorithms
- added chapter for the new `lower_bound`, `upper_bound` and `binary_search` algorithms
- added `swap_values` algorithm
- used `IsEqual` predicate for `swap_ranges` and `copy`
- added `reverse_copy` and `reverse` algorithms
- added `rotate_copy` algorithm
- added `unique_copy` algorithm
- added chapter on specification of the data type `stack`

B.30. New in Version 5.1.0 (Boron, May 2010)

- adaption to Frama-C Boron and Why 2.26 releases
- changed from the `-jessie-no-regions` command-line option to using the pragma `SeparationPolicy(`*`value`*`)`

B.31. New in Version 4.2.2 (Beryllium, May 2010)

- changed to latest version of CVC3 2.2
- added additional remarks to our implementation of `find_first_of`
- changed `size_type` (**`int`**) to `integer` in all specifications
- removed casts in `fill` and `iota`
- renamed `is_valid_range` as `IsValidRange`
- renamed `has_value` as `HasValue`
- renamed predicate `all_equal` as `IsEqual`
- extended timeout to 30 sec.

B.32. New in Version 4.2.1 (Beryllium, April 2010)

- added alternative specification of `remove_copy` algorithm that uses `ghost` variables

- added Chapter on heap operations
- added mismatch algorithm
- moved algorithms adjacent_find and min_element from the appendix to chapter on non-mutating algorithms
- added typedefs size_type and value_type and used them in all algorithms
- renamed is_valid_int_range as is_valid_range

B.33. New in Version 4.2.0 (Beryllium, January 2010)

- complete rewrite of pre-release
- adaption to Frama-C Beryllium 2 release

Bibliography

[1] WP Plug-in. http://frama-c.com/wp.html.

[2] Frama-C Software Analyzers. http://frama-c.com, 2018.

[3] Why – Where Programs Meet Provers. http://why3.lri.fr, 2018.

[4] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. The Alt-Ergo SMT Solver. http://alt-ergo.lri.fr, 2018.

[5] Clark Barrett and Cesare Tinelli. Homepage of CVC4. http://cvc4.cs.stanford.edu/web/, 2018.

[6] Clark Barrett and Cesare Tinelli. Homepage of CVC3. http://www.cs.nyu.edu/acsys/cvc3/, 2010.

[7] Microsoft Research. The Z3 Theorem Prover. https://github.com/Z3Prover/z3, 2018.

[8] The Coq Consortium. The Coq Proof Assistant. https://coq.inria.fr, 2018.

[9] ANSI/ISO C Specification Language. http://frama-c.com/acsl.html, 2018.

[10] CEA LIST, Laboratory of Applied Research on Software-Intensive Technologies. http://www-list.cea.fr/gb/index_gb.htm.

[11] INRIA-Saclay, French National Institute for Research in Computer Science and Control . http://www.inria.fr/saclay/.

[12] LRI, Laboratory for Computer Science at Université Paris-Sud. http://www.lri.fr/.

[13] Fraunhofer-Institut für Offene Kommunikationssysteme (FOKUS). http://www.fokus.fraunhofer.de.

[14] Virgile Prevosto. ACSL Mini-Tutorial. http://frama-c.com/download/acsl-tutorial.pdf.

[15] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL 1.13 Implementation in Argon 18.0. https://frama-c.com/download/acsl-implementation-18.0-Argon.pdf, 2018.

[16] Allan Blanchard. Introduction to C Program Proof using Frama-C and its wp plugin. http://allan-blanchard.fr/publis/frama-c-wp-tutorial-en.pdf, December 2017.

[17] Programming languages – C, Committee Draft. http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1362.pdf, 2009.

[18] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.

[19] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symposium on Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 19–32, Providence, RI, 1967. American Mathematical Society.

[20] Richard Smith. Working Draft, Standard for Programming Language C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf, 2017. publicly available draft of C++ 17 standard.

[21] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J Comput*, 6(2):323–350, Jun 1977.

[22] Lincoln E. Moses and Robert V. Oakford. *Tables of Randon Permutations*. Stanford University Press, 1963.

[23] Ming Li and Paul Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Graduate texts in computer science. Springer, New York, 1997.

[24] Timon Lapawczyk. Formale Verifikation von Heap-Algorithmen mit Frama-C. bachelor thesis, Humboldt-Universität zu Berlin, July 2016.

[25] D.E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[26] Sellibitze. How to Implement Classic Sorting Algorithms in Modern C++. https://stackoverflow.com/questions/24650626/how-to-implement-classic-sorting-algorithms-in-modern-c, Aug 2014.

[27] Kim Völlinger. Einsatz des Beweisassistenten Coq zur deduktiven Programmverifikation. Diplomarbeit, Humboldt-Universität zu Berlin, August 2013. https://www2.informatik.hu-berlin.de/top/_media/www/mitarbeiter/diplomarbeit-kim-voellinger.pdf.

[28] Richard Fitzpatrick J.L. Heiberg. *Euclid's Elements of Geometry*. http://farside.ph.utexas.edu/euclid.html, Austin/TX, 2008.

[29] David Hilbert. *Grundlagen der Geometrie*. B.G.Teubner, Stuttgart, 1968.

[30] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2008.

[31] Stephan Schulz. The E Theorem Prover. https://wwwlehre.dhbw-stuttgart.de/~sschulz/E/E.html, 2018.

Index of ACSL definitions

Caveat: This index has been automatically generated from the ACSL /C sources. For the time being it mentions only the page where an ACSL definition is first included in the text. Moreover, if a listing had to be split, then the page number refers to the first part of the listing even if the specific ACSL definition appears in the second part.

E

F

G

H

I

L

M

N

P

R

S

U

W

Index of examples

Caveat: This index has been automatically generated from the ACSL /C sources. For the time being it mentions only a few of the pages where a C function occurs in the text. Moreover, if a listing had to be split, then a page number might refer only to the first part of the listing.

U